

A.A. 2021/22

FONDAMENTI DI INGEGNERIA DEL SOFTWARE

PROF. MARIANO CECCATO

FABS :)

NOTA

Questi appunti/sbobinatura/versione “discorsiva” delle slides sono per mia utilità personale, quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 🌸 ✨

**Questo file fa parte della mia collezione di sbobinature,
che è disponibile (e modificabile!) insieme ad altre in questa repo:**
<https://github.com/fabfabretti/sboninamento-seriale-uniVR>

INDICE

NOTA	1
Indice	2
0 – Introduzione	4
1– Software process	6
2 - Agile Software Development	11
3 – Ingegneria dei requisiti	17
4 –UML	24
5 – Architectural design	28
6 - Testing	32
7 – Refactoring.....	36
8 – Project management.....	39

0 - INTRODUZIONE

Software

Il software è ormai una componente **fondamentale** della vita, poiché aziende e intere nazioni vi dipendono. Il problema è che è astratto, quindi si fa fatica a identificarne i contorni e le qualità.

Le qualità sono percepite in maniera astratta non solo dagli utilizzatori del software, ma anche dagli sviluppatori - che avendo una conoscenza molto dettagliata del singolo pezzo perdono di vista tutto il resto. Questo causa tipicamente grossi problemi, che tendono a far **degenerare molto velocemente il software** in qualcosa di molto difficile da gestire e mantenere.

Il software engineering contiene le **teorie**, i **metodi** e gli **strumenti** per sviluppare in via professionale, e permette di risolvere questi problemi.

Costo

Il costo di sviluppo del software è di ordini di grandezza superiore al costo dell'hardware che lo fa girare. Il concetto di hardware è ancora più evanescente ora che si fa tutto in cloud 😊

Inoltre, il costo di manutenzione del software è diverse volte superiore di quello dello sviluppo iniziale, perché ci vorranno aggiornamenti per stare dietro all'evoluzione della normativa (es. GDPR), oppure alcuni sono troppo critici per essere sostituiti (banche con software di anni 60-70) e la manutenzione, protrandosi per decenni, arriva ad avere un costo enooorme.

Percentuali di successo

Qualche dato da uno studio su un campione di 50'000 progetti:

- Solo il **29%** dei progetti è stato completato on time nel 2017!
- 50% è andato fuori dai tempi/costi
- 20% è direttamente stato cancellato.

Problemi tipici

- **Costi effettivi maggiori** dei costi stimati.
→ La stima è difficile, dato che il software è impalpabile
- Sistema finale **non ha tutte le features**
- Consegna **DOPO la deadline** da contratto
- Software di **bassa qualità**
- Utente finale **non del tutto soddisfatto**.
- Software **#cancellato**

<i>Problemi "assunti"</i>	<i>Problemi reali</i>
<ul style="list-style-type: none">○ Intrinseca complessità del software e dei problemi intrinseci di questo dominio○ Problemi tecnologici (linguaggio di programmazione sbagliato..)	<ul style="list-style-type: none">○ Problemi di comunicazione fra SW engineers e stakeholders○ Difficoltà a capire i problemi degli utenti○ Difficoltà a capire il dominio(es. gergo specifico del settore, o dare per scontate procedure che tutte le persone di quell'ambito già conoscono)○ Problemi di management○ Problemi di teamworking

Esempio celebre: Ariane 5

È un progetto pensato per viaggi spaziali nel 1996. 37 secondi dopo il lancio il razzo è esploso. L'errore fu che si cercò di scrivere un valore a 64 bit su un registro a 16 bit; andò in buffer overflow e interpretato come numero negativo. Questo disastro causa la perdita di 7 mld di dollari e 10 anni di lavoro. Il problema sorge perché, normalmente, si fanno dei controlli per vedere che il numero è più o meno lungo di 16 bit. Questo si è reso necessario perché il software originario era per Ariane 4, con meno bit; lo hanno riasato aggiungendo dei controlli. Quando si calcola il valore successivo, però, non si fanno questi controlli. Prima c'erano, ma un programmatore li ha **valutati come ridondanti** e ha **migliorato le performance**. Non se ne sono accorti poiché gli scenari nei quali ha girato il software erano diversi dallo scenario dell'Ariane 4, e **non avevano i casi che avrebbero beccato la presenza di questo errore**.

È solo parzialmente un errore di programmazione: in realtà è dato ad una **gestione sbagliata del progetto**, che non ha messo in campo tutte le verifiche di correttezza del progetto.

Per porre rimedio a questi problemi nasce la **Nato conference** nel **1968**, che ha la missione di dare vita a metodologie rigorose per gestire i progetti software e non ripetere questi errori. È questa conferenza a coniare il termine Software Engineering ed è oggi nota come **International Conference of Software Engineering**; è l'evento principale dell'ingegneria del software, dove si presentano prodotti industriali e accademici.

Software Engineering

L'idea è di tradurre le pratiche di altri ambiti ingegneristici anche in ambito software. Sotto il nome ombrello di ingegneria del software distinguiamo gli aspetti rilevanti per l'ingegneria del software:

- **Raccolta dei requisiti**
- **Managing del progetto**

L'ingegneria vuole ottenere **risultati** della **qualità richiesta**, entro la **schedule prevista** e con il **budget a disposizione**. Questo spesso significa fare dei compromessi: gli ingegneri non possono essere perfezionisti, a differenza di chi scrive software per sé stesso.

Come professionisti attivi in questo ambito, vogliamo essere in grado di produrre sistemi funzionali ed affidabili velocemente e in modo economico. Ovviamente questi goals sono contrastanti fra loro; quindi vogliamo trovare un compromesso accettabile sia per chi sviluppa che per chi utilizzerà il software.

Etica dello sviluppo software

Lavorare come sviluppatori software può produrre anche delle questioni etiche: per esempio,, mi aspetto che chi lavora in questo ambito abbia un comportametno etico, e questo si declina in tanti aspetti.

- **Confidenzialità**: un ingegnere è tenuto a vincoli di confidenzialità anche quando non viene richiesto esplicitamente un NDA.
- **Competenza**: non bisogna vantare competenze che non possediamo
- **Leggi sulla proprietà intellettuale**: siamo tenuti a rispettare le leggi locali e globali
- **Misuse**: non dobbiamo abusare delle risorse a cui abbiamo l'accesso.

Ci sono associazioni professionali che raccolgono a livello ufficiale i codici di etica: **ACM**: Association for Computer Machinery, **IEEE**: Institute of Electrical and Electronic Engineers, **British Computer Society**. Queste società hanno elaborato una lista abbastanza lunga di concetti che descrivono come bisognerebbe comportarsi rispetto a diverse sfaccettature del lavoro.

Tipicamente, comunque, sono tutti **astratti** e potrebbe ugualmente non essere semplice prendere decisioni. Qualche esempio:

- Sto collaborando a un progetto safety critical e sono sotto NDA, e mi accorgo che per soddisfare i vincoli di tempo stiamo saltando verifiche di sicurezza. Sono tenuto a mantenere la riservatezza, ma sto osservando comportamenti pericolosi; dunque, il code of ethics mi dice sia di denunciare che di mantenere la segretezza.
- Artefatti software che possono essere usati come dual use, sia in ambito civile che militare.
- Sistemi di scommesse che possono essere usati in entertainment ma provocano dipendenza.

1- SOFTWARE PROCESS

In precedenza, il software era sviluppato per via molto manuale, “più arte che scienza”: molto artigianalmente era l'esperto a creare il software senza linee guida condivise con le altre persone.

In analogia con le altre discipline ingegneristiche, anche nel software si vuole adottare un processo che potesse prendere materiale grezzo e trasformarlo in un prodotto di qualità misurabile con un **processo misurabile e ripetibile**.

Un processo di sviluppo software è importante per poter pianificare, mettere ordine e percorrere in maniera razionale le operazioni di produzione, e sapere in ogni momento cosa si vuole e si sta facendo.

Ordinato

Controllato

Ripetibile

L'obiettivo è migliorare la **produttività dei developer** e assicurare la **qualità del prodotto: process quality → product quality**

Processo software:

Una serie strutturata di attività ordinate, controllate e ripetibili finalizzate allo sviluppo di un prodotto software.

Esistono tanti processi diversi, ma tutti hanno in comune delle attività:

1. **Specifica:** capire *bene* quello che si vuole implementare prima di lavorarci
2. **Design e implementazione:** decidere l'organizzazione del sistema e implementare il sistema
3. **Validazione:** verificare che tutto il lavoro svolto è effettivamente ciò che vuole il cliente.
4. **Evoluzione:** una volta prodotto, il software è soggetto a cambiamenti e bisogna garantire la manutenzione del software.

Solitamente il processo è diviso in **attività** condotte per portare avanti lo sviluppo del software. Oltre a questo, abbiamo anche altre entità:

- **Prodotti:** output delle attività
- **Ruoli:** differenti persone possono avere ruoli e responsabilità distinte all'interno del processo
- **Pre-post condizioni:** sono condizioni che devono essere vere all'inizio o alla fine di un processo

Plan-driven vs agile

Ci sono due macro raggruppamenti

A piani	Agile
Struttura rigida, più antichi	Dinamici e più propensi ad accogliere cambiamenti. Nasce perché in alcuni domini gli obiettivi non possono essere chiari fin dall'inizio.
Tutte le attività sono pianificate in anticipo , e per misurare il processo ci si rifà al piano	Pianificazione a grandi linee, e il dettaglio viene definito durante lo sviluppo del processo per via incrementale .

Nella pratica, solitamente si usano dei processi ibridi che includono elementi da entrambi gli approcci.

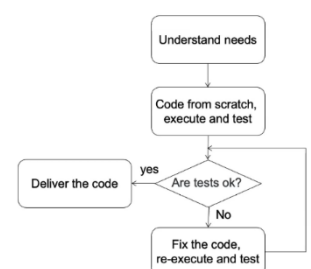
Modelli di processi software

Code and fix [sus]

È probabilmente ciò che seguiamo di default.

- Si inizia subito a implementare seguendo le indicazioni
- No analisi o fasi di design
- Va bene per piccoli progetti – LoC < 1500, NON per progetti più grandi o lavoro di team.

Non è un processo software. È solo un tentativo iterativo di risolvere un problema.



Waterfall

È uno dei **primi processi proposti**, in risposta al problema del software artigianale. È proposto negli anni '70 e si è super diffuso; l'obiettivo è riuscire a trasformare una lavorazione artigianale in una lavorazione seria e industriale. È infatti derivata dal processo di **produzione manifatturiero**, e fa una forte analogia con i processi usati per sviluppare in altri domini (tipo le costruzioni). **Ciascuna fase è l'input delle fasi successive**, quindi ogni fase parte solo alla fine di quella precedente.

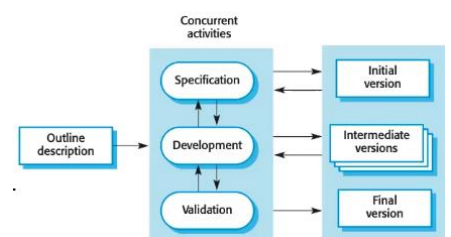
1. Analisi dei requisiti	In questa fase il team di sviluppo parlano con i committenti, gli stakeholders e gli utenti finali per capire bene quali sono le funzionalità da implementare nel sistema. Non è una fase banale, poiché questo è l' aspetto più critico : un singolo errore in questa fase arriverà fino alla fine, perché tutte le altre fasi sono guidate da una comprensione erronea dei requisiti. È molto complicato dato che spesso nemmeno i clienti sanno bene quello che vogliono!
2. Design di sistema e di software	Traduciamo i problemi degli utenti in un' architettura software , delineando le linee guida ad alto livello delle risposte che deve dare il sistema. Descriviamo i vari componenti in generale , come parlano, dove sono deployati.
3. Implementazione e unit testing	Una volta che abbiamo il design di sistema iniziamo effettivamente a realizzare e testare le unità del programma .
4. Integrazione e test di sistema	I programmi singoli vengono integrati e testati come sistema completo per assicurarsi che tutto sia stato meetato.
5. Operazioni di manutenzione	Bisogna garantire il supporto al committente. È la fase più lunga dell'intero ciclo di vita. Le manutenzioni possono essere: <ul style="list-style-type: none"> • Correttive: sistema errori • Miglioramenti: miglioro le prestazioni • Aggiunte: sorgono nuovi requisiti

L'output di ciascuna fase deve essere congelato fino all'inizio della fase successiva

Vantaggi	Svantaggi
<ul style="list-style-type: none"> • Si spende tempo e importanza sui requisiti • Applicabile per requisiti molto chiari • Permette di coordinare molto bene il lavoro, avendo dei milestones chiari e degli artefatti che devono essere prodotti in tempo 	<ul style="list-style-type: none"> • Una fase deve essere completa prima di potersi spostare alla successiva • I cambiamenti ai requisiti sono esternamente difficili da integrare nei processi software. Buono per commesse governative.
<ul style="list-style-type: none"> ○ Sviluppo hardware: non voglio poter cambiare l'hardware a metà strada ○ Sistemi critici: requisiti di legge impongono validazioni e certificazioni molto fisse e stringenti. ○ Software grandi con tanti sistemi uniti e più compagnie. 	Non indicato per software in generale , dove ciascuna fase successiva potrebbe dare del feedback sull'output; inoltre congelare i requisiti all'inizio potrebbe non essere una grande idea.

Incrementale

Se nel processo a cascata le varie fasi sono chiaramente divise nel tempo, qui invece si adotta un approccio trasversale e si cerca di arrivare il prima possibile a una **versione eseguibile** – seppur parziale – per poter fare una **demo** al cliente. Questo permette di esporre ognuna delle fasi intermedie al cliente finale, con numerosi vantaggi:



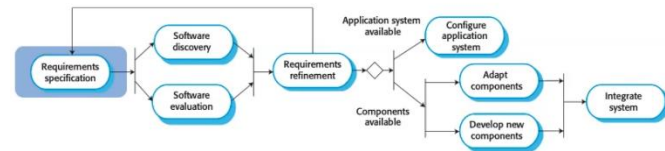
Vantaggi	Svantaggi
<ul style="list-style-type: none"> • Diversamente dall'approccio a piani ha un costo molto più basso per rispondere ai requisiti ed è sempre pronto a riceverli • Consegne molto rapide, e può essere usato (Anche se con versioni incomplete fin dall'inizio). • Feedback per stimolare riflessione e verificare se la comprensione dei requisiti è allineata • Chiarificare i requisiti: magari il cliente si accorge, avendocelo davanti, che avrebbe preferito qualcosa di diverso. • Permette ai requisiti di evolvere naturalmente: l'analisi dei requisiti perdura per tutto lo sviluppo. 	<ul style="list-style-type: none"> • Il management è più difficile, perché non ho sempre a disposizione un piano visibile e non è cost-effective produrre documentazione per ogni singola versione. • La struttura del sistema tende a degradare ad ogni incremento; <ul style="list-style-type: none"> ○ Serve fare refactoring per non incorrere in costi molto maggiori.

Integrazione e configurazione

Si basa sullo sviluppo di software attraverso l'integrazione di parti già pronte. Si parte da componenti disponibili sul mercato, oppure sviluppati in precedenza e appartenenti al mio bagaglio aziendali.

Probabilmente sono componenti generici che devono essere configurati o leggermente modificati per il mio problema specifico.

- Lo **sviluppo è limitato** al minimo; il sistema viene assemblato partendo da una serie di componenti configurabili già esistente.
- Può essere **plan driven o agile**



Il flow delle attività è il seguente:

1. Specifica dei requisiti	Si capiscono i problemi da risolvere per guidare la scelta dei componenti. Non deve essere troppo elaborato. Basta raccogliere i requisiti essenziali ad alto livello.
2. Indagine	a. Cerco quali sono i componenti disponibili per risolvere il problema b. Seleziono quello che più si addice.
3. Raffinazione dei requisiti	Potrebbero essere modificati a causa di precondizioni di componenti particolari, che mi obbligano a rischiare i requisiti. Potrei dover iterare qualche volta.
4. Configurazione	Valuto se i componenti trovati sono un buon compromesso e possono essere utilizzati. Configuro e integro i componenti per realizzare il sistema
5. Integrazione	Se i miei componenti non sono sufficienti dovrò anche adattarli o svilupparne di nuovi.

Vantaggi	Svantaggi
<ul style="list-style-type: none"> • Costa molto poco • Poco rischio, dato che mi rifaccio a componenti testati di cui posso fidarmi • Consegna molto veloce 	<ul style="list-style-type: none"> • Livello di qualità basso, non essendo sviluppato apposta • Potrei dover aver fatto dei compromessi sui requisiti • Perdo il controllo sul software: non sono io a sviluppare i componenti, quindi devo affidarmi e sperare che i componenti vengano mantenuti. Bisogna valutare che il tempo di vita del software sia compatibile con il tempo di vita del componente.

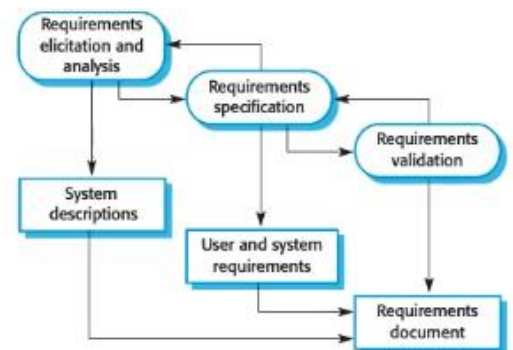
Attività di processi software

1. Specifica e ingegneria dei requisiti

È critica: un errore qui causa sicuramente problemi nelle parti successive. Può essere preceduta da uno studio di fattibilità, ovvero un mini progetto che consiste nell'ideare un prototipo e decidere in base all'esperienza se continuare o meno.

- **Obiettivo**: capire e definire che cosa serve e identificare i constraints
- **Output**: documento dei requisiti che descrive problemi e features del mio software.

È composta da tre fasi:



Fase	Output
1. Elicitazione e analisi dei requisiti : descrizione del sistema ottenuta sia osservando sistemi precedenti, sia discutendo con i clienti. Potrebbe coinvolgere anche dei modelli o dei prototipi per aiutare a capire cosa implementare alla fine.	Descrizione del sistema
2. Specifica dei requisiti : traduciamo l'analisi in un documento. Tipicamente dividiamo i requisiti in: <ul style="list-style-type: none"> • User requirements: astratti, definiscono i requisiti dell'utente • System requirements: dettagliata descrizione della funzionalità da dare. 	Requisiti utente e di sistema
3. Validazione : si riprendono i requisiti e si verifica se sono realistici, consistenti e completi. Si identificano gli errori.	Documento finale dei requisiti

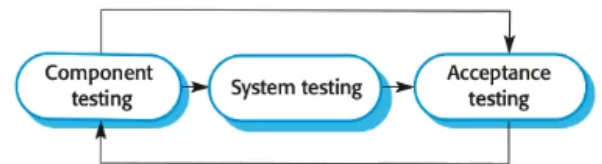
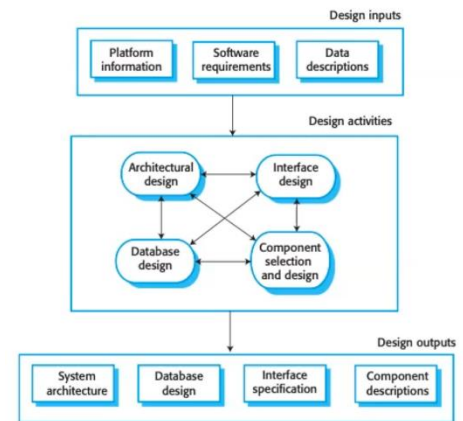
2. Design e implementazione

Per design e implementazione si intende la progettazione e scrittura del codice. L'obiettivo di questa task è tradurre le informazioni estratte attraverso l'analisi dei requisiti.

- **Design:** consiste nella scrittura di un modello, per definire ad esempio:
 - a. le strutture dati, il database
 - b. le interfacce fra i componenti
 - c. gli algoritmi usati.
- **Implementazione:** tradurre il design in un programma

Le due fasi cambiano in base al tipo di sistema da implementare. Le fasi di questa task sono:

1. **Design architetturale:** decidere che architettura avrà il sistema.
2. **Design del database:** decido quali tabelle, colonne e strutture dati usare, in quanto sono solitamente condivise fra vari modelli.
3. **Design dell'interfaccia:** design di come le varie componenti possono comunicare e offrire servizi fra di loro. Se rispetto l'interfaccia, poi posso sviluppare i componenti in maniera molto indipendente.
4. **Selezione e design dei componenti:** definisco dei dettagli dei componenti. Posso sviluppare o cercare componenti già implementati. Potrei anche realizzare un modello in UML e poi generare un'implementazione automaticamente.



3. Testing e validazione

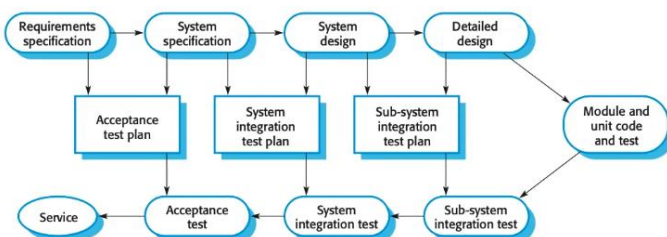
Verifichiamo che il lavoro è stato portato avanti in maniera corretta. Abbiamo tipicamente due tipi di validazioni:

- **Checking:** è un'ispezione del codice, in cui il verificatore osserva il codice per verificare che non ci siano errori. È detto anche manual inspection o code review.
- **Program testing:** il programma viene eseguito usando dei dati simulati e degli scenari di test.
System testing: mette in esecuzione il componente secondo gli scenari derivanti da dati reali.

Tipicamente non è condotto solo a fine sviluppo, ma in parallelo. Ci sono tre momenti in cui si testa il software:

- **Component/unit testing:** è condotto durante lo sviluppo e ciascun componente è sviluppato in maniera indipendente dagli altri. Gli scenari sono scritti dallo stesso sviluppatore che ha fatto il componente dato che bisogna conoscerlo in profondità; esistono strumenti come JUnit per scriverli e automatizzarne l'esecuzione, generando un report.
- **System testing:** integro insieme i componenti e faccio del testing per cercare i problemi che potrebbero sorgere da interazioni inaspettate. Per grandi progetti, potrei anche avere più livelli, tipo testare sottoparti. Tipicamente no.
- **Acceptance testing (aka customer testing):** è lo stadio finale del sistema e viene condotto portando il sistema software nelle condizioni operative finali – similmente a come girerà dal cliente. Viene testato con dati realistici, magari simili ai dati di produzione. Questo tipo di test potrebbe rivelare alcune riconducibili alla fase di raccolta dei requisiti. Passare questi test può essere la condizione di pagamento.

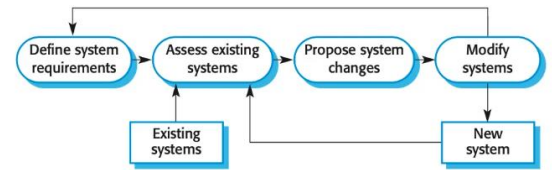
Esempio: plan driven + testing



Una volta che i requisiti sono disponibili posso già scrivere un piano dagli acceptance test, dato che sono indipendenti dall'implementazione! La validazione avviene ad ogni fase del piano → V model

4. Evoluzione

La vita del software continua anche dopo la produzione per garantire la manutenzione. Si inizia con la valutazione del sistema esistente. Si valuta una proposta di cambiamento. Integro i nuovi requisiti e la implemento.



Il problema, quindi, è gestire il cambiamento. Ho vari modi:

- **Anticipare/prevedere i cambiamenti:** È importante avere in mente che il sistema potrà essere modificato già in fase di progettazione del sistema; il sistema così sarà più facile da modificare.
- **Prototype:** è opportuno usare dei prototipi per avere un feedback su potenziali cambiamenti prima che questi vengano richiesti. Ci sono anche prototipi non funzionanti con una interfaccia grafica simile a un prodotto finale per valutare le alternative anche in ambito di requirement engineering.
- **Consegne incrementali:** permette di avere feedback molto presto e consegnare per prime le funzionalità critiche.

Vantaggi	Svantaggi
<ul style="list-style-type: none"> • Con le versioni parzialmente funzionali portiamo già da subito valore al cliente • I primi incrementi ci aiutano a estrarre ulteriori requisiti • Diminuiamo il rischio di fallimento del progetto • Le parti più importanti sono testate più a lungo. 	<ul style="list-style-type: none"> • Spesso c'è un core di funzionalità che è necessario per tutte le altre. Questo nucleo tipicamente è il motore, e deve essere sviluppato per primo per poterci attaccare le altre parti; la conoscenza completa di come deve essere fatto, però, potrei avercela solo alla fine. Questo core quindi dovrà essere modificato pesantemente durante lo sviluppo per attaccare le altre funzionalità • Le specifiche e i requisiti tipicamente sono sviluppati insieme al software; quindi saprò i prossimi componenti da fare solo a quel punto. Non sapere le specifiche all'inizio potrebbe essere incompatibile con commesse governative o requisiti di validazione in sistemi critici

2 - AGILE SOFTWARE DEVELOPMENT

Contesto storico

1980s-1990s: plan driven

Sono caratterizzati da un attento project planning, e un rigoroso processo di sviluppo aiutato dai tool.

Inizialmente erano commissionati soprattutto da **enti militari**. In genere per **programmi complessi** (pensa agli aerei) ci vogliono tempi molto lunghi, tipo 10 anni, e infatti c'è il forte problema di dover prevedere che hardware ci sarà fra 10 anni!

Tipicamente ci sono **tanti teams** diversi che lavorano sopra (hardware, software, data scientists...) e ci vuole il coordinamento fra università, con sono sforzi di collaborazione internazionale. Esempi:

- **LHC**: ha fin troppi dati acquisibili, hanno dovuto stimare la qtà di dati memorizzabile fra 10 anni.
- **Esplorazione spaziale**. In questo caso ci vogliono piani dettagliati per collaborare.

Fine 90s

A fine anni 90 iniziano anche commesse non governative, come industrie che vogliono supportare il proprio business. Questo secondo tipo di software ha caratteristiche molto diverse, e anche la produzione si modifica e adatta:

- Definizione di nuovi tipi di sviluppo che potesse essere in grado di sviluppare software **più rapidamente e con consegne intermedie – versioni intermedie che potevano già essere usate dal cliente**. Si arriva a preferire la velocità alla qualità.
- In software più piccoli, **l'overhead del processo rigoroso non dà abbastanza vantaggi**: il processo di documentazione dei requisiti, del design per tutto il ciclo doveva essere limitato.
- La cosa più importante, inoltre, è **essere in grado di modificare le risposte ai requisiti**: se prima facevano addirittura parte del contratto, ora i requisiti diventano **soggetti a evoluzione**: i prodotti e i cicli di produzione cambiano, e il software deve adattarsi. Sviluppare così tanta documentazione risulta troppo costoso: dovrei riscrivere tonnellate di documentazione ogni volta.

Di conseguenza nascono nuove tecniche di sviluppo in grado di recepire questi cambiamenti molto velocemente: AGILE.

Caratteristiche dello sviluppo agile

1. **Specifica, design e implementazione avvengono contemporaneamente.**
La prospettiva è ribaltata: le fasi sono in parallelo, questo è essenziale perché i requisiti cambiano e bisogna portare avanti l'attività di valutazione dei requisiti per tutto il processo.
2. **Il sistema è sviluppato per incrementi successivi con il coinvolgimento degli stakeholders.**
L'incremento è stabilito all'inizio e va dalle 2-4 settimane (es. ogni 2 settimane si rilascia una nuova versione del software). Questi incrementi devono essere sviluppati con l'intervento degli stakeholders, ovvero committenti o utenti, che devono essere coinvolti nel team di sviluppo.
 - **La documentazione è tenuta al minimo**, ed è bene avere comunicazioni informali fra i team di sviluppo; per esempio team meetings molto veloci cercando di liberarsi dai formalismi.
3. **Uso di strumenti automatizzati.**
Servono per alleggerire lo sviluppatore da task manuali e noiosi che potrebbero introdurre errori:
 - Automatizzazione dei test
 - Configuration management
 - Continuous integration
 - Generazione automatica dell'interfaccia grafica a partire da un design dichiarativo; un cambiamento di interfaccia è generato automaticamente.**! Non sono degli aiuti: sono fondamentali** nell'Agile, perché servono ad alleggerire lo sviluppatore per farlo concentrare su task di alto livello (scrittura di parti critiche e decisioni).

Manifesto

Insieme all'Agile è stato scritto anche un manifesto.

Individui e interazioni	meglio di	Processi e strumenti
Software funzionante	meglio di	Documentazione completa
Collaborazione con il cliente	meglio di	Negoziare un contratto
Prontezza/risposta al cambiamento	meglio di	Seguire un piano

Principi:

- **Coinvolgimento del cliente:** importante nel team di sviluppo; il cliente ci permette di comprendere i requisiti in maniera accurata e capire le priorità nei vari cicli di sviluppo.
- **Consegna incrementale:** ogni step è una versione usabile e installabile
- **Persone over processo:** è molto importante che tutte le persone siano *skillate*, dato che gli sviluppatori sono lasciati liberi di prendere decisioni e lavorare come desiderano.
- **Accogliere il cambiamento:** ci si aspetta che i requisiti cambino, quindi è importante che l'architettura sia in grado di recepire i cambiamenti.
- **Mantenere le cose semplici:** le cose semplici sono quelle che funzionano meglio, quindi è importante che il software sia semplice; le soluzioni troppo artificiose rischiano di dare problemi, e si rischia di dover scendere a compromessi più avanti.

Domini di applicabilità delle tecniche agili

- Software di **piccole o medie dimensioni**.
- Il **cliente** deve essere **disposto a fare parte del team** di sviluppo.
- **Poche entità regolatrici esterne** a imporre vincoli sul software
 - Ad esempio il software medicale ha forti vincoli sul design e richiede che i requisiti siano congelati e certificati prima dello sviluppo. No buono.
- Contestuale a **tecniche agili di project management**
 - Processo di sviluppo (codice) agile + gestione del progetto agili (es. scrum), che possono essere applicati anche ad altri progetti e non solo allo sviluppo software.

Extreme programming

L'agile ha poi avuto una declinazione estrema dove vengono portate all'estremo le sue caratteristiche, ovvero l'extreme programming:

- Numerose versioni rilasciate ogni giorno
- Consegne ogni 2 settimane
- Ciascun test deve andare su ciascuna versione e la build è accettata solamente se tutti vanno.
- Design semplice, prima scrivo i test poi il codice, pair programming.

Tecniche Agile

Requisiti in Agile

User stories

I requisiti possono cambiare, e la loro estrazione è integrata allo sviluppo. I requisiti possono cambiare, e la loro estrazione è integrata allo sviluppo.

Le **user stories** sono delle **descrizioni di scenari** in cui il software finale sarà usato.

- Sono **narrazioni brevi** in quanto è comprensibile sia allo sviluppatore che agli stakeholders.
- Sono dal punto di vista dell'utente.

Mentcare: Prescribing medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select 'current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose. If she wants to change the dose, she enters the new dose then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose then confirms the prescription.

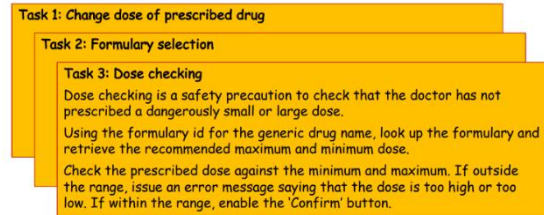
The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose. After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Si usano al posto dei requisiti. Le user stories sono raccolte in quella che si definisce **user card**. (*comic sans momento*)

Task cards

A partire dalle user stories, si spezzano le storie in varie tasks. Ciascuna di queste tasks ha una descrizione ripresa dalle user stories.

- Questo lavoro di **spaccare la user story** è fatto dal team di sviluppo, che di fatto è supportato dallo stakeholder.
- Una volta che le tasks sono pronte, il team deve **stimare quanto effort** ciascuna delle tasks necessiterà (**effort point**) in funzione del tempo e della difficoltà.
- A questo punto si **torna dal cliente** per capire quali sono più importanti e vanno sviluppate prima; questo è importante per disambiguare e poter consegnare prima.



C'è un problema di completezza: è difficile capire se i task hanno catturato tutte le user stories; quindi è molto importante collaborare con il cliente.

Agile software development

Test-driven development

È una delle modalità di sviluppo. Tradizionalmente il software veniva scritto dopo, invece qui invertiamo questa tradizione. Questo significa che:

- I tests **possono essere eseguiti mentre si sviluppa il codice**, e non si può andare avanti con lo sviluppo se i tests non sono successful.
→ Si scoprono subito i problemi, durante lo sviluppo. 😊👉
- I **test incrementali vanno di pari passo con lo sviluppo incrementale dei tasks**
→ !! Per scrivere un test chiaro devo avere ben chiaro il requisito !! È anche una verifica: se non so scrivere il test probabilmente non ho ben capito nemmeno il requisito.
- **L'utente** deve essere coinvolto, e può aiutare a scrivere i tests e **dirmi se è corretto**.
- Mi servono strumenti **automatizzati per l'esecuzione dei tests**.
→ Il numero dei casi di test inizierà ad essere molto consistente: questo significa che mi serve eseguire tutti i tests in batch eseguendo un solo bottone, velocizzando lo sviluppo. Questo mi aiuta anche a controllare velocemente di non aver sminchiato implementazioni precedenti quando cambio qualcosa.

Refactoring

Essendoci molti cambiamenti in corsa, alcune funzionalità possono essere sviluppate come patch, e potrebbero esserci dei compromessi. Questo significa che la struttura del codice potrebbe degradare: codice duplicato o uso inappropriato di codice usato per altro ("piego il codice per le nuove esigenze")

È dunque importante fare delle **attività esplicitamente mirate a riportare la qualità del codice a un livello ottimale**. È super costoso se viene fatto alla fine, quando ormai è tutto sminchiato; anticipando invece costa poco e mantiene il codice facile da cambiare ed evolvere.

Pair programming

Consiste nell'avere **due sviluppatori a un solo computer**. I dettami dell'agile prevedono anche di alternare chi scrive alla tastiera, e anche le coppie che lavorano insieme.

Porta numerosi vantaggi:

- **Collective ownership**: il codice non è di un singolo sviluppatore, ma tutti lavorano a tutto; il software è un bene comune del team, e tutti vogliono che abbia successo e sia di qualità. È un traguardo del team nella sua interezza.
- Il secondo sviluppatore fa **controllo del codice**, fornendo un punto di vista alternativo. È molto più economico di una revisione formale del software.
- **Incoraggiamento del refactoring**: quando c'è un problema che potrebbe far decadere la qualità viene sistemato subito, in quanto è percepito come un'inefficienza per tutti
- Si **condivide conoscenza**, e questo permette dei vantaggi (come il fatto che più persone sanno come/dove modificare e ci si para il culo da gente che abbandona il progetto)
- **Non è necessariamente un'inefficienza**: la q.tà di codice sviluppato NON dimezza, ma è paragonabile a quella prodotta da due software developers separati – le linee scritte sono la metà ma richiederanno molta meno revisione e quindi si va avanti alla stessa velocità o più velocemente.

Agile project management: SCRUM

Può essere applicato in vari contesti; noi vediamo il software.

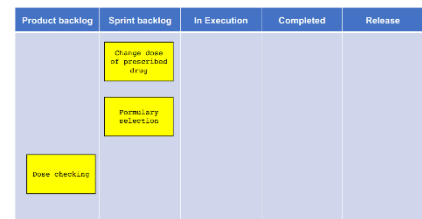
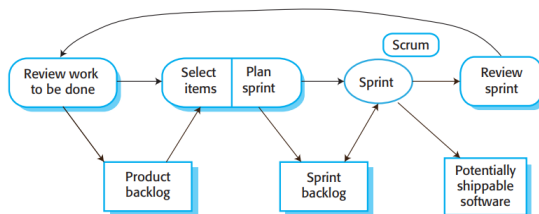
Ci sono alcune fasi che caratterizzano lo sviluppo:

1. **Fase iniziale:** si avvia il progetto e si decide come procedere. È importante dare obiettivi generali, e abbozzare già dall'inizio l'architettura, poiché su di essa si innesteranno tutti i componenti.
2. Cicli di avanzamento di 2-4 settimane, detti **sprint**.
3. **Chiusura del progetto:** si fa un wrap up, si valuta la completezza, si scrive la documentazione. 😊

Terminologia

- **Team:** gruppo di persone di massimo 7-8 persone
- **Potentially shippable product increment:** alla fine di ogni ciclo c'è un progetto già consegnabile (potenzialmente).
- **Product backlog:** è semplicemente la todo list.
- **Product owner:** è il committente o gli stakeholders e ci aiuta nel team identificare funzionalità del prodotto, priorità, e verificare se il product backlog è corretto.
- **Scrum:** meeting giornalieri in cui ci si aggiorna per coordinare lo sviluppo e prioritizzare le attività del giorno. Tipicamente è la mattina, tipicamente è in piedi davanti a un caffè. ☕🧘
- **ScrumMaster:** è la persona nel team che si assicura che i dettami dello scrum siano implementati correttamente.
- **Sprint:** un ciclo di sviluppo incrementale di 2-4 settimane (rimane fisso).
- **Velocity:** è la quantità di backlog che il team di sviluppo può portare a compimento in una settimana.

Ciclo di sprint



1. Backlog	Contiene tutto ciò che c'è da fare. Inizialmente viene riempito dalle stories o da altre descrizioni .
2. Selezione dal backlog	Il product owner seleziona quali funzionalità vanno inserite nel prossimo sprint, in quanto è lui a definire le priorità.
3. Programmazione primo sprint	<p>a. Si selezionano elementi nel backlog con più alta di priorità, e si inseriscono nello sprint corrente.</p> <p>b. Si stima quanta roba mettere con la velocity. 🧑🧑🧑 La si stima mano a mano con i feedback degli sprint.</p>
4. Scrum	<p>La mattina di ogni giornata si fa lo scrum, parlando di cosa si è fatto il giorno prima, cosa si farà oggi e eventuali problemi riscontrati. Insomma si fa il punto. 🍷</p> <p>a. La durata dello sprint è fissa; se una task è troppo difficile non posso allungarlo! Se mi accorgo che il tempo non è sufficiente, viene direttamente tolta dal backlog in quanto la scadenza deve essere rispettata.</p> <p>b. Durante lo sprint il team è isolato dal mondo, e le comunicazioni con azienda/cliente sono gestite dallo scrum master che propaga info solo se necessario – il team è lasciato lavorare agli obiettivi.</p> <p>c. Lo sviluppatore decide a che task contribuire, e le sposta in esecuzione.</p> <p>d. Se una task viene completata la si mette nel completed.</p>
5. Sprint review	<p>È fatta a fine sprint, e si cerca di migliorare il modo in cui il team lavora.</p> <p>a. Si pensa se la stima della velocity andava bene e cosa è andato bene/male in vista degli sprint successivi.</p> <p>b. In questa review viene spesso fatta una demo e c'è anche l'owner, che conferma se le sue aspettative sono soddisfatte o no; quello che va bene viene messo in release.</p>

Poi si ricomincia.

Benefici

1. Prodotto **diviso in parti gestibili** e comprensibili
2. Lo sviluppo **non viene ritardato** da eventuali decisioni sui requisiti
3. Il team ha la **visione su tutto** e di fatto la comunicazione viene migliorata grazie agli scrum; c'è una condivisione del software e degli obiettivi.
4. Il cliente vede **consegne puntuali** e può fornire **feedback**
5. Relazione di fiducia fra il team di sviluppo e il committente. Questo è buono e facilita il successo 😊

Scaling agile methods

Lo sviluppo agile era pensato per teams di piccole dimensioni. Per software grandi, progetti di lunga durata o teams multipli o teams distribuiti si definiscono delle estensioni per permettere di integrare queste caratteristiche in teoria incompatibili con le prime versioni.

Posso scalare in due direzioni:

Scaling up: Scala di dimensioni del software

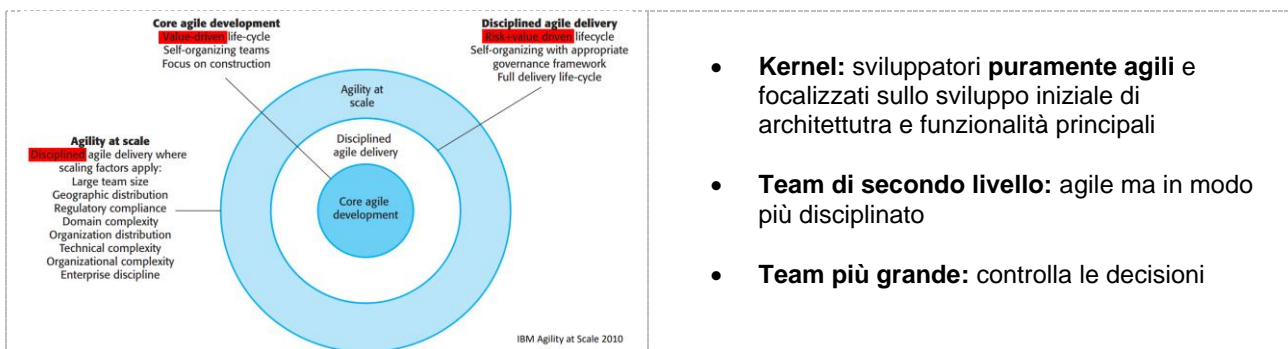
Scaling out: Scala dell'organizzazione

Tutte le proporte hanno sempre una serie di caratteristiche:

- Organizzazione flessibile
- Releases frequenti (firefox!) 🦊
- Integrazione continua dei contributi
- Test scritti prima del software
- Comunicazione 🗣️🗣️🗣️🗣️

IBM Scrum

Si è tentato di usare Agile anche su grandi sistemi, per esempio IBM ha proposto una metodologia per scalare agile, con una struttura a cipolla:



Distributed Scrum

Durante la vita di Scrum, si è anche cercato di applicarlo a contesti per i quali non era pensato - come quelli distribuiti.

- Scrum prevede di avere tutti nella stessa collocazione fisica per poter comunicare e fare i **meeting**; se questo non succede (es. geografia) è comunque importante che:
 - Il product owner faccia visita al team di sviluppo durante le ore di meeting perché è più facile discurrete.
 - Gli sviluppatori distribuiti nel mondo mantengano mantenere **comunicazione informale**, per esempio evitando le mail in favore di videocall e instant messaging – sluck , discord.
- È importante avere un **sistema di integrazione continua** per sapere sempre come è il sistema finale
- È importante avere tutti lo stesso **ambiente di sviluppo**.

Multi team scrum

si prova a unire più team scrum usando una metodologia scrum.

- **Si replicano le figure chiave** (scrum master, product owner) per ogni team
- Nuova figura detta **software architect** in ogni team: collaborano per definire la struttura in generale del sistema. Hanno la **visione overall** e **portano la conoscenza nei singoli team**.
- Le **date di rilascio sincronizzate** fra tutti i team, che devono consegnare insieme per integrare tutto in un prodotto finale.
- **Scrum of scrums:** un rappresentante porta il lavoro del team portando le informazioni agli altri team. È il punto di sincronizzazione quotidiano tra i vari teams di sviluppo.

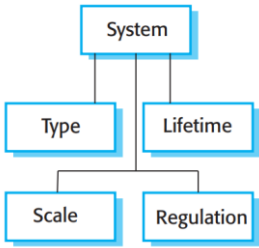
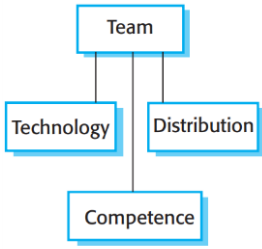
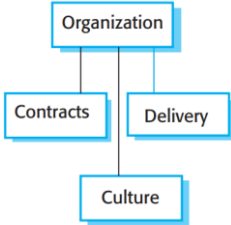
Manutenzione di Agile

È un **problema tipico** delle tecniche agili: hanno **effetti collaterali** sulla gestione dell'evoluzione del software una volta scritto.

Il software scritto in agile è difficile da mantenere per diversi motivi:

- Manca la **documentazione**: se un membro del team originale lascia il progetto porta con sé conoscenza sulle decisioni, sulle motivazioni delle decisioni e delle ripercussioni di eventuali revisioni: problema grosso sui progetti di lunga durata.
- L'**owner** dovrebbe essere **coinvolto** anche nella manutenzione. Ma può essere difficile convincerlo... 😞

Problemi di sistema di Agile (anche nel best case scenario)

Sistema	Team	Organizzazione
		
<ul style="list-style-type: none">• Quanto è grande il sistema?• Che tipo di sistema? Regolazioni esterne? → tipi diversi potrebbero richiedere che alcune decisioni siano validate da organismi di controllo.• Vita attesa del sistema? → se deve durare molti anni garantire la manutenzione con agile è difficile	<ul style="list-style-type: none">• Che supporto tecnologico è disponibile? → Alcuni linguaggi hanno cose utili tipo generazione automatica di testing e refactoring, ma non tutti li hanno.• Livello di skill dei componenti del team → In Agile il team deve essere in grado di fare decisioni <i>skillate</i> autonomamente.• Come è organizzato lo sviluppo? Se si è distribuiti nel mondo <i>aiuto</i>. 🌐 📖	<ul style="list-style-type: none">• Organizzazione e contratto: richiede tanta documentazione?• Cultura aziendale abituata a lavorare diversamente?• Il cliente è disponibile a far parte del team di sviluppo?

Resistenza ai metodi agile

Ci sono numerose resistenze aziendali e non solo a questi metodi:

- Le metodologie sono in **contrapposizione con quelle tradizionali**
- I **project manager** a volte **non hanno esperienza** di questi metodi e sono riluttanti all'introduzione delle novità.
- Spesso le grandi organizzazioni hanno **procedure interne di gestione della verifica della qualità burocratici e incompatibili con agile**.
- Ci vuole un **team molto skillato**.
- Possono esserci **resistenze culturali in organizzazioni esperte di approcci a piani**.

Agile contro plan driven

Considerazioni:

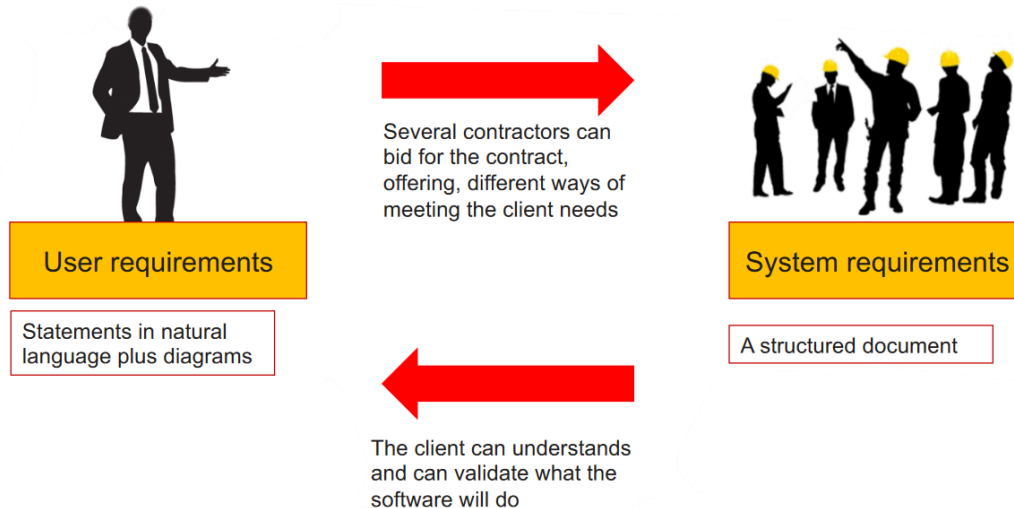
- E' più importante avere **specifici/progetto prima di iniziare**? Se è molto importante (es. richiesto da leggi), meglio plan driven
- Se un approccio incrementale è realistico (=va bene al cliente), allora possiamo optare per agile.
- Quanto è **grosso** il sistema da sviluppare?
→ Se serve un piccolo team locale va bene agile
→ Se è un grosso sistema con un grande team va bene plan-driven.

3 - INGEGNERIA DEI REQUISITI

Requisiti

Requisiti è un termine generico; in SW engineering intendiamo la descrizione di quei servizi che il sistema dovrà fornire agli utenti finali durante la sua normale operatività; riflette i bisogni del cliente.

Possiamo dividere i requisiti in requisiti utente e sistema.



Requisiti utente	Requisiti di sistema
<ul style="list-style-type: none"> Sono espressi in linguaggio naturale ed esprimono le necessità che il sistema deve risolvere. Sono tipicamente molto astratti, e ammettono molte interpretazioni e soluzioni diverse fra loro. Sono scritti per permettere a vari fornitori di formulare un preventivo 	<ul style="list-style-type: none"> Non definiscono i bisogni dell'utente, ma definiscono il modo in cui gli addetti ai lavori comprendono i requisiti dell'utente, e come il sistema riceverà queste necessità per implementare il sistema software. Sono un po' più strutturati. Permettono di: <ul style="list-style-type: none"> Progettare in maniera consapevole Validare se l'implementazione è corretta; quello di utente sono troppo generici per essere usati in validazione. Inoltre si specificano le cose "non dette" per capire se l'interpretazione è corretta. È necessario perché l'utente potrebbe non avere le idee chiare, e così riusciamo a capire se è bene.
<p><i>Esempio:</i> The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month</p>	<p><i>Esempio:</i> 1. On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated. 2. The system shall generate the report for printing after 17.30 on the last working day of the month. ...</p>

Stakeholders

Sono tutti coloro che hanno un qualche interesse per il sistema, o la cui attività viene influenzata dal sistema:

- Affected dal sistema
- Interessati al sistema
- Quelli che controllano i requisiti
- I clienti che saranno inseriti nel sistema
- ...

Requisiti funzionali

I requisiti funzionali sono quelli già presentati; descrivono le funzionalità che dovrebbe fornire il sistema. Come reagisce il sistema agli input? Come si comporta?

I suoi obiettivi sono:

- **Completezza:** tutte le funzionalità devono essere menzionate
- **Consistenza:** non devono mai essere contraddittori 😊

[!] Gli **stakeholders** potrebbero essere **in contrasto**, ergo produrre indicazioni in contraddizione

[!] **Imprecisioni**, e questo conduce a controversie fra committente e sviluppatore – queste potrebbero essere identificate solo durante lo sviluppo 😞

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

Requisiti non funzionali

Sono dei vincoli sul sistema e sul servizio; tipicamente sono vincoli sul **sistema nella sua interezza**; ergo potrebbero mettere vincoli sull'architettura.

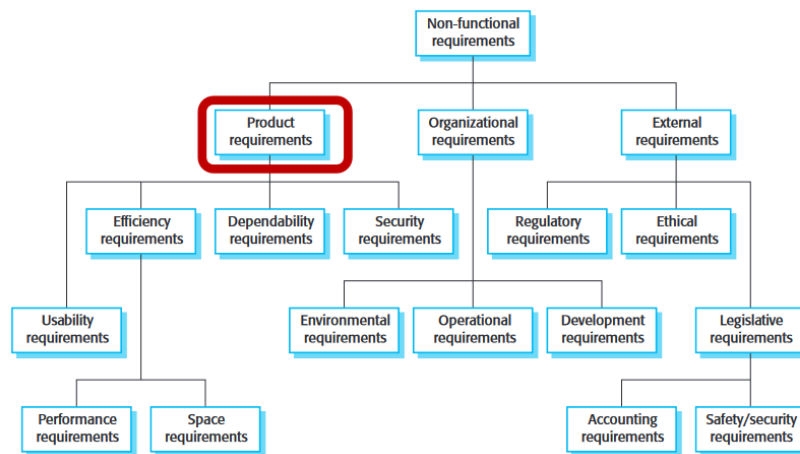
[!] **Un singolo requisito non funzionale (che dà un vincolo) potrebbero discendere numerosi requisiti funzionali**; in un certo senso sono più critici e ricevere più attenzione. Il problema è che, tipicamente, l'utente ci dà solo quelli funzionali :')

Esempi:

- Requisiti su vincoli posti al sistema software, come i **tempi di risposta** (magari devono essere certificati entro un certo tempo). Non ha a che fare con l'effettiva funzione ma solo un vincolo
- Processo da adottare: es. magari ci è imposto un certo **linguaggio di programmazione**
- Standards da utilizzare in quanto prescritti da **normativa** o committente.

Tassonomia dei requisiti non funzionali

- **Requisiti di prodotto:** specificano che il prodotto deve funzionare in un certo modo
 - › **Accessibilità:** ad esempio, prevedere modalità per daltonici e ipovedenti
 - › **Efficienza**
 - › **Affidabilità:** ad esempio, si potrebbe richiedere che ci sia un uptime del 99% ****contrattuale**** (uua ⚠)
 - › **Sicurezza:** come gestire l'autenticazione e a quali utenti grantare l'accesso? Ad esempio, in EU i sistemi di pagamento DEVONO avere la 2FA.
- **Organizzativi:**
 - › **Linguaggio di programmazione:** magari deve interfacciarsi con altri moduli
 - › **Ambiente operativo:** ambienti realtime o HW specializzato. Es. nei treni ci vuole un software che si interfaccia con HW normato, quindi è tutto molto normato.
- **Esterni:** provengono da entità governative tipo banche/medicali
 - › **Normativi:** in ambito ferroviario, ad esempio, ho forti vincoli sul software e sul processo di sviluppo!
 - › **Etici**



Esempi

Product requirement

- The Mentcare system shall be available to all clinics during normal working hours (Mon-Fri, 08:30-17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

External requirement

- The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Organizational requirement

- Users of the Mentcare system shall identify themselves using their health authority identity card.

Concretizzazione dei requisiti

Devono essere quantificabili, perché altrimenti è un po' difficile testarli oggettivamente...

Alcuni esempi di metriche:

- Velocità:** transazioni processate al secondo, response time, screen refresh time
- Dimensione:** mbytes, ROM chips
- Facilità d'uso:** tempo di training, frames di aiuto
- Affidabilità:** mean time of failure, tempo di availability, rate of failure
- Robustness:** tempo di ripartenza, percentuale di fallimento, probabilità di corruzione dei dati
- Portabilità:** % di statements con dipendenza, # sistemi target

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized



Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use

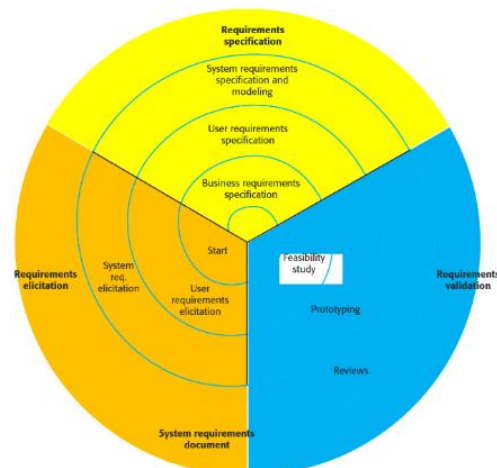
Processo di ingegneria dei requisiti

È un processo molto complicato ed è facile commettere errori, ergo tendiamo a usare metodi fissi nell'ottica di trasformare un metodo artigianale in un **approccio ingegneristico e ripetibile**.

1. Requirement elicitation (cattura dei requisiti)

L'obiettivo è capire cosa i nostri utenti debbano fare e che funzionalità dovranno esporre; come il SW deve supportare l'operatività degli utenti. Alcuni punti salienti sono:

- Dominio:** potrebbe non essere del tutto informatico e l'ingegnere potrebbe non conoscerlo
- Vincoli:** hardware, performance, SO, linguaggio, performance



Questa prima fase non è semplicissima: ci sono degli ostacoli che rendono questa fase più complicata e potrebbero inserire errori nei requisiti.

- Tipicamente gli stakeholders **non sanno esattamente cosa vogliono**, e i requisiti qui raccolti saranno parziali e parzialmente corretti; e richieste potrebbero essere impossibili o essere in priorità sballata
- **Esplicitare la conoscenza implicita**: gli stakeholders potrebbero usare della loro terminologia non per forza a noi chiara; es. nel dominio medico non è detto che l'ingegnere sappia *esattamente* cosa sia una prescrizione... oppure, "testing" che è molto diverso fra SW e HW.
- Stakeholders differenti potrebbero avere **differenti punti di vista**! Le contrapposizioni potrebbero generare problemi, e addirittura falsare la raccolta di requisiti per facilitare una categoria, o mettere i bastoni fra le ruote. 🦊🦊
- **Fattori politici e organizzativi**: es. darsi più visibilità, roba che al committente non interesserebbe ma il manager se li inventa per comodo suo. Ye.
- I requisiti potrebbero anche **mutare**! Nuovi stakeholders, cambi di business.



Intervista

È uno dei metodi più usati per trovare i requisiti. Si dividono in:

- **Domande chiuse**: domande chiuse a cui rispondono e ci si segna la risposta.
 - > Rischio di influenzare lo stakeholder.
- **Domande aperte**: no lista di domande; ha una traccia degli argomenti ma è molto dinamica.
- **Approccio misto**: obv the best.

In questa fase non ci interessano i dettagli: gli stakeholders, comunque, tendono a facilitare ciò dato che a meno di dimostrazioni e prototipi rimangono molto generici.

Problemi tipici	Soluzioni
<ul style="list-style-type: none"> • Gergo tecnico che l'ING non conosce, quindi problemi alle prime interviste e possibili non detti / incomprensioni. • Lo stakeholder potrebbe non voler andare nei dettagli politico aziendali 	<ul style="list-style-type: none"> • No bias, no preconcetti, be open minded. Uaaa. 😊 • Cercare di aiutare lo stakeholder evitando cose troppo aperte.

Studi etnografici

People know their work, but not the relation with the rest of the work in their organization

Aim: try to understand the social and organizational issues that affect the use of the system

L'obiettivo è capire bene **operatività e relazioni** all'interno della organizzazione.

L'ingegnere del software, anziché intervistare gli utenti, li **osserva** per un certo periodo. È uno **studio immersivo** durante il quale non si fanno domande, ma gli utenti vengono osservati nell'operatività quotidiana.

Si è dimostrato **estremamente utile** per raccogliere requisiti che normalmente sarebbero stati impliciti e sarebbero sfuggiti alle interviste.

Sono **molto potenti** ma anche **molto costosi**; inoltre, possono essere condotti quando ci sono dei sistemi o comunque un'operatività da rimpiazzare (es. torre di controllo c'è già e va rimpiazzata). Invece se il software è per una nuova organizzazione non ho niente da osservare.

Storie e scenari

Storie e scenari sono rappresentazioni molto concrete dell'operatività. Questo ci aiuta a parlare con gli stakeholders.

Storia	Scenario
<ul style="list-style-type: none"> • Testo in forma narrativa • Descrive ad alto livello • Generico; audience vasta • Chiunque nel sistema può avere un ruolo 	<ul style="list-style-type: none"> • Strutturato • Informazione più specificata (es. input/output) • È un'interazione fra due o più utenti

→ Tipicamente la storia è più generica e posso estrarre più scenari da una storia.

Storia:

<p>Photo sharing in the classroom</p> <p>Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused on the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this project, pupils are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site because he wants pupils to take and comment on each other's photos and to upload scans of old photographs that they may have in their families.</p> <p>Jack sends an email to a primary school teachers' group, which he is a member of, to see if anyone can recommend an appropriate system. Two teachers reply, and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.</p>	<p>Elementi concreti:</p> <ul style="list-style-type: none"> • Attori: jack, pupils, teacher's group, two teachers. • Verde: è la funzione che possiamo ampliare con uno scenario.
--	--

Scenario:

<p>Uploading photos to KidstakePics</p> <p>Initial assumption: A user can upload or delete their own or more digital photographs to be uploaded to a laptop computer. Users have successfully logged on to KidsTakePics.</p> <p>Normal: The user chooses to upload photos to the computer and to select the project name. A description of what the system and users expect when the scenario starts. Also be given the option of inputting keywords that should be associated with each uploaded photo. Unloaded photos are named by creating a conjunction of the user name with the filename of the photo.</p> <p>A description of what can go wrong and how resulting problems can be handled</p> <p>What can go wrong: No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed of a possible delay in making their photos visible.</p> <p>Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wants to rename the photos, or cancel the upload. If the user chooses to rename the photos, the system should automatically rename the photos by adding a number to the existing filename.</p> <p>Other activities: The moderator may be logged on to the system when the user uploads photos. A description of the system state when the scenario ends.</p> <p>System state on completion: User is logged on. The selected photos have been uploaded and assigned a status "awaiting moderation." Photos are visible to the moderator and to the user who uploaded them.</p>	<p>Strutturato:</p> <ul style="list-style-type: none"> • Titolo • Assunzione iniziale • Descrizione del flow normale • Descrizione dei problemi possibili • Informazioni su altre attività condotte in contemporanea • Stato del sistema al completamento.
---	---

2. Specifica dei requisiti

Consiste nello scrivere i requisiti. Possono far parte del contratto, quindi devono essere il più **precisi possibili**.

- Requisiti **utente**: devono essere comprensibili dagli utenti finali, anche non avendo background tecnico.
- Requisiti di **sistema**: sono requisiti più dettagliati e potrebbero avere dettagli tecnici.

In teoria dovrebbero descrivere il *cosa* e non il *come*; comunque queste due cose possono a volte **non essere separati**. Ad esempio:

- Per strutturare i requisiti potrebbe servire una certa architettura
- Il sistema potrebbe interoperare con altri sistemi che generano requisiti
- I requisiti potrebbero essere soggetti a leggi particolari che definiscono le modalità.
- Requisiti non funzionali che pongono limiti all'architettura, ergo parlano anche del *come*

Specificazione dei requisiti in linguaggio naturale

Il linguaggio naturale è espressivo intuitivo e universale, ma può essere anche **ambiguo**. Il problema è che diverse persone potrebbero interpretare diversamente un'ambiguità.

Linee guida:

- Definire un **formato standard** da mantenere consistentemente per ragionare in modo **schematico**
- Usare un **linguaggio consistente**; ad esempio "shall" è per mandatory e "should" per cose che si vorrebbe fare ma non sono obbligatorie.
- Utile usare il **text highlighting** per evidenziare in modo **consistente** (es. grassetto o italics?)
- **Evitare termini informatici**; sì dominio di operazione (es. medico)
- Aggiungere **spiegazione** di perché un requisito è

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

necessario

Specifica dei requisiti **strutturata**

Dà meno libertà in modo da avere meno incertezza.

Funziona bene per alcuni ambiti, come ad esempio embedded, ma è troppo rigida per ambiti business.

Insulin Pump/Control Software/SRS/3.3.2		
Function	Compute insulin dose: Safe sugar level	
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units	
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)	
Source	Current sugar reading from sensor. Other readings from memory	
Outputs	CompDose—the dose in insulin to be delivered	
Destination	Main control	
Action	Condition	Action
	Sugar level falling ($r2 < r1$)	Register patient
	Sugar level stable ($r2 = r1$)	View personal info
	Sugar level increasing and decreasing ($(r2 - r1) < (r1 - r0)$)	View record
Requires	Two previous readings	
Precondition	The insulin reservoir contains insulin	
Postcondition	r0 is replaced by r1 then r1 is replaced by r2	
Side effects	None	

Setup consultation allows two or more doctors, working in different offices, to view the same patient record at the same time. One doctor initiates the consultation by choosing the people involved from a dropdown menu of doctors who are online. The patient record is then displayed on their screens, but only the initiating doctor can edit the record. In addition, a text chat window is created to help coordinate actions. It is assumed that a phone call for voice communication can be separately arranged.

Use-cases

Sono un tipo di scenario incluso in UML.

Identificano gli attori in ciascuna interazione, e consistono in un modello “astratto”-grafico e una tabella più dettagliata. Possono essere integrati con i sequenc diagrams epr aggiungere più dettaglio.

2. Documento dei requisiti

Raccoglie i **requisiti ufficiali**, e include sia requisiti utente che di sistema. Non è un documento di design: racconta cosa fa il sistema, non il *come*.

Nel plan driven rappresenta la milestone dei requisiti e viene passato al team di sviluppo.

È molto variabile, e le informazioni contenute dipendono dal tipo di approccio usato; esistono comunque anche degli standard ufficiali, come quello di IEEE.

Utenti

- **Clienti di sistema:** si verifica che i requisiti siano soddisfatti. I customers possono specificare anche cambiamenti di requisiti
- **Managers:** usano i requisiti per documentare e plannare lo sviluppo
- **System engineers:** usano i requisiti per capire che sistema sviluppare
- **System tests engineers:** usano i requisiti per sviluppare i test
- **System maintenance engineers:** usano i requisiti per capire il sistema e le relazioni fra le sue componenti.

Standard IEEE

1. Prefazione: ciclo di vita, aspettative
2. Introduzione
3. Glossario: in collaborazione con gli stakeholders! Tipicamente si definiscono anche le abbreviazioni.
4. User requirements
5. System architecture
6. System requirements
7. Modelli di sistema
8. Evoluzione del sistema: iniziamo già a dar conto su come funzionano i cambiamenti, con che conseguenze, chi se ne occupa...
9. Appendice
10. Indice

3. Validazione dei requisiti

L'obiettivo è essere sicuri che sia del tutto valido; in quanto correggerlo in seguito ha costi esorbitanti – anche x100.

Cose da fare per valutare la correttezza:

- **Validità del documento:** il sistema fornisce le funzionalità nel modo migliore per il cliente?

- **Consistenza:** ci sono conflitti nei requisiti? Tipicamente potrebbe accadere in sistemi di grandi dimensioni. Ci son tool che lo fanno! 😊
- **Completezza:** tutte le funzioni sono descritte?
- **Realismo:** è fattibile, dati i vincoli di budget o tecnologia?
- **Verificabilità:** posso controllare i requisiti? È legato alla definizione numerica e/o dettagliata.

Tecniche di ricontrollo

- **Ricontrollo dei requisiti** in modo manuale e sistematico
- **Prototyping:** con pochi click ci sono strumenti che mi fanno fare un mock con un sottoinsieme delle funzionalità; per esempio ci sono strumenti commerciali per le app
- **Scrittura di scenari di test:** scriviamo scenari di test prima ancora di scrivere il sistema. Scrivere lo scenario ci impone di farci delle domande, innanzitutto sulla verificabilità: ci impone di pensare in concreto alla condizione da controllare

4. Cambiamento dei requisiti

Tipicamente sono soggetti a cambiamento; quindi cerchiamo di controllarlo e gestirlo, anziché esserne sopraffatti.

In un software di grandi dimensioni ci aspettiamo che cambino costantemente:

- **Problemi** intrinsecamente **difficili** da decidere
- **Evoluzioni tecniche:**
 - › Cambiamenti di hardware
 - › Bisogno di interfacciarmi;
 - › Cambio di leggi: ad esempio adesso le banche devono comunicare tutte le transazioni di notte alla banca centrale per fare cose antiriciclaggio
- **Conflitti fra compratori e utenti finali** del sistema
- **Cambio di priorità** fra stakeholders.

Per aiutare il processo:

- Identificare ciascun requisito con un **id unico**
- Definire le **attività condotte per accettare o meno** un cambiamento
- Regole per tracciare il cambiamento: **relazioni** fra requisiti e anche fra componenti del sistema, per capire dove ho ripercussioni. → Strumenti e tool che tengono i link

Il processo è il seguente:

1. **Analisi del problema:** fermiamoci a capire se è veramente necessario, capire se è valido
2. **Analisi del costo e dell'analisi:** capire l'effetto e il costo; è più facile se ho i link detti prima → in base a questo si decide se attuare o meno il cambiamento
3. **Implementazione del cambiamento:** modifichiamo requisiti e sistema; i cambiamenti devono essere tenuti allineati affinché documentazione dei requisiti e sistema siano sempre allineati.

4-UML

(Cerca una buona ref online.)

UML è una famiglia di notazioni grafiche che supportano la descrizione e il design di sistemi software. L'astrazione descrive sia le entità che le relazioni e interazioni fra di esse, con l'obiettivo di:

- Comprendere il dominio
- Discutere obiettivi e soluzioni
- Fare scelte
- Design
- Documentazione

Storia

Inizialmente si voleva unificare direttamente il linguaggio di programmazione; ovviamente questo ebbe molto poco successo, e OMG – che cercò di definire un linguaggio unificante – ricevette solo critiche in risposta. Infine, in IBM si inizia a portare in avanti la definizione di un linguaggio; era produttrice di software e quindi in buona posizione, e aveva dentro i 3 big men della modellazione.

Così, alla conferenza Oopsla del 95, presentano il primo unified method: è stato preso dalla OMG e alcuni si sono affiliati alla OMG per produrre la prima versione ufficiale. Poi si sono raffinate via via con le varie versioni, fino alla nostra 1.5

Usi

UML può essere usato:

- Come **sketch** per proporre soluzioni
- Più formalmente per decisioni di progetto
 - Esistono tool in grado di sintetizzare lo scheletro del programma da realizzare.
 - Esistono tool in grado di estrarre l'UML partendo dal codice (ingegneria inversa).

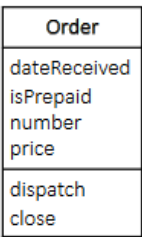



Tipi di modelli

Modelli strutturali	Modelli di interazione	Modelli sul funzionamento del sistema
Class diagram Object diagram	Use case diagram Sequence diagram	Activity diagram (dati) State machine diagram (eventi)

Modelli strutturali

Descrivono l'organizzazione del sistema in termini di componenti e relazioni. Ne esistono **modelli statici** – che descrivono la struttura del sistema – e **dinamici** – che descrivono l'organizzazione in esecuzione.

Class diagram

Classe 	Descrizione degli oggetti con proprietà simili. <ul style="list-style-type: none">• Attributo: [visibility nome: tipo = default {proprietà}]<ul style="list-style-type: none">○ Attributo derivato: /nome: tipo○ Visibility: + public, - private, # protected, - package.• Operazione: [visibility nome(parametri): returntype]• Interfaccia: si aggiunge <<interface>>, e tutto è abstract• Abstract: non può essere istanziata. Nome in corsivo. <p>Ciò che è statico è sottolineato.</p>
Oggetto	Gli oggetti sono istanziazioni delle classi.
Relazione: Associazione 	 Sono collegamenti fra le classi.
Relazione: Dipendenza	 Cambiamenti al supplier potrebbero causare cambiamenti nel client. Normalmente si mostrano solo quelle rilevanti.

	<p>Ce ne sono tanti tipi: call, create, derive, instantiate, permit, realize...</p>
<p>Relazione: Generalizzazione</p>	<p>Tutte le istanze della classe concreta sono anche istanze della classe astratta. La sottoclasse inherits tutti gli attributi e le associazioni di tutti i suoi parenti.</p>
<p>Relazioni: realizzazione</p>	<p>Classe che realizza un'interfaccia.</p>
<p>Comments</p>	
<p>Constraint rules</p>	<p>Possono essere espresse in molti modi diversi, come un linguaggio di programmazione o naturale.</p>
<p>Aggregazione</p>	<p>“parte di”, l'esistenza delle parti è indipendente dal gruppo.</p>
<p>Composizione</p>	<p>Un oggetto non dovrebbe essere parte di più di una composizione. Il lifecycle è dipendente, quindi cancellando la sorgente cancello anche il target.</p>
<p>Interfaccia richiesta</p>	<p>Permette di cambiare l'implementazione della lista senza cambiare l'implementazione del sorting.</p>
<p>Classi associate</p>	<p>Constraint implicito che può esserci una sola istanza di una classe associata con altre due.</p>
<p>Classi parametrizzate</p>	
<p>Enumerazione</p>	

Modelli di interazione

Aiutano a comprendere user requirements e interazione con utenti e sistemi

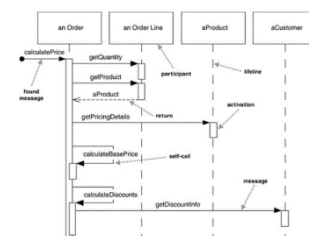
Use cases

Nascono per aiutare a estrarre i requisiti ma ora sono anche usati come documentazione UML rigorosa.

In forma narrativa o visuale, raccontano interazioni tipiche fra utente (attore primario e attori secondari) e sistema.

In base alle necessità sono molto versatili:

- **Sea level:** un attore primario, finiscono al raggiungimento del goal
- **Fish level:** molto dettagliati
- **Kite level:** overview dei ruoli dei casi sea-level.





È utile per capire i requisiti funzionali e per avere una visuale esterna del sistema.

Sequence diagram

Permettono di modellare l'interazione fra gli attori e gli oggetti in un certo use case. Descrivono i messaggi scambiati durante l'use case, ma non spiegano gli algoritmi nel dettaglio.

Per attività o use cases più complicati sono meglio gli activity diagram.

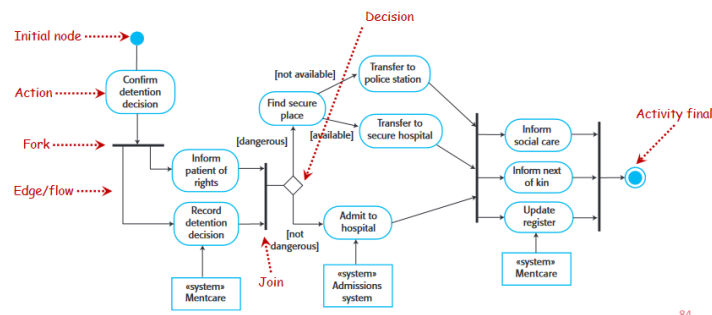
Operatori	alt : alternative multiple fragments; solo quello true viene eseguito opt: il segmento è eseguito solo se true par: ciascun segmento va in parallelo loop: il frammento viene eseguito più volte region: regione critica dove i frammenti hanno un solo thread neg: interazione non valida ref: reference, si riferisce a un'interazione su un altro diagramma sd: racchiude tutto il sequence diagram.
Chiamata sincrona 	Il chiamante aspetta che il messaggio sia finito (es. chiamata di funzione)
Chiamata asincrona 	Il chiamante continua senza aspettare risposta.

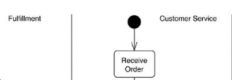

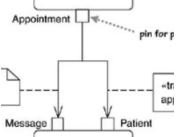
Modelli comportamentali

Mostrano comportamenti dinamici mentre il sistema è in esecuzione. Gli stimoli a cui risponde il sistema possono essere **eventi** o **dati**.

Data driven: activity diagram

Sono modelli che mostrano la sequenza di azioni coinvolte nel processing dei dati in input, e nella generazione dell'output. Sono molto utili per l'analisi dei requisiti, perché mostrano il processing end-to-end.

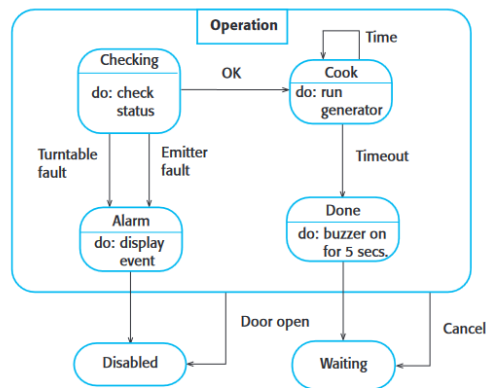


Swimlanes 	Dividono lo schema con linee per mostrare chi fa cosa
Segnali 	Segnali di tempo ricevuti da processi esterni
Pins e trasformazioni 	Mostrano come le risorse sono prodotte e consumate dalle azioni.
Regioni di espansione	Chiamano un'azione molteplici volte, una per ogni elemento della collezione.

<p>The diagram shows a flow starting with 'Choose Topics', which leads to a 'keyword' and a 'list of topics'. These lead into a dashed box labeled 'expansion region'. Inside this region, there are two concurrent activities: 'Write Article' and 'Review Article'. The flow then exits the region and leads to a 'list box pin'.</p>	
<p>Final flow</p> <p>The symbol is a circle with a cross inside, representing a final flow. It is labeled with '[reject]'.</p>	<p>Terminano un flow, senza terminare tutta l'attività. Se siamo in una regione di espansione, fa da filtro diminuendo il numero di elementi.</p>
<p>Join</p> <p>The diagram shows two paths, A and B, merging into a single path. Below the join symbol, the text reads: '{joinSpec = A and B and value of inserted coins >= €}'.</p>	<p>Normalmente, la join emette il token solo dopo aver ricevuto tutti i token, Possiamo specificare anche altri comportamenti.</p>

Event driven: state machine diagram

Modellano il sistema in risposta a eventi interni o esterni. Gli stati sono nodi, e gli eventi sono archi fra i nodi.



5 - ARCHITECTURAL DESIGN

L'obiettivo della progettazione dell'architettura è avere una fase intermedia fra i requisiti e la progettazione in dettaglio.

È bene avere questa fase esplicita per:

- **Comunicazione con lo stakeholder**
- **Analisi di sistema:** l'architettura potrebbe influenzare requisiti non funzionali.
- Possibilità di **riuso del software** (se s

La rappresentazione avviene con una vista ad alto livello che mostra dei blocchi per le entità. Nasconde info ma permette di fare delle valutazioni d'insieme.

Usi:

- **Facilitare la discussione** del design del sistema e su dove allocare i singoli componenti
- **Documentare** design architetturale

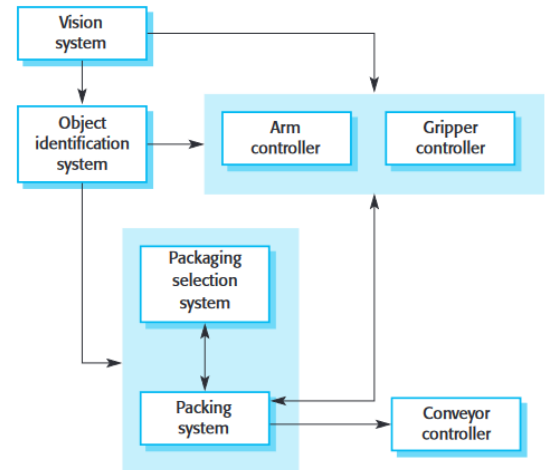
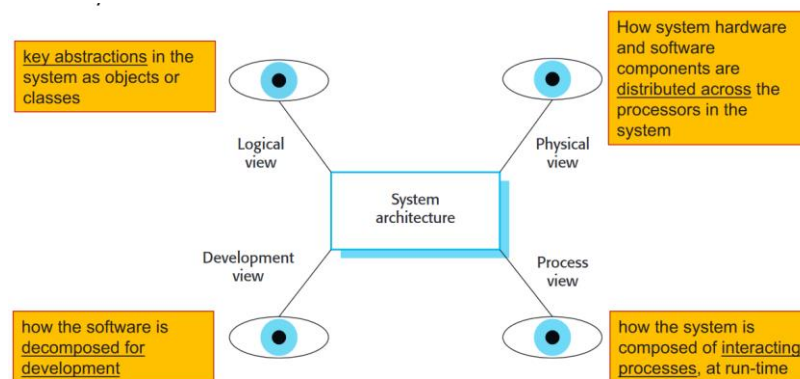
Non è solo rappresentazione: vengono prese molte decisioni,

- Come distribuire l'hardware nel software
- Ci sono best practices/ architetture note da sfruttare? Dovrò farvi modifiche?
- Saremo in grado di rispondere ai requisiti non funzionali?
- I componenti hanno responsabilità precise o ambigue (es. a più componenti == bad)

Le conseguenze di questa scelta sono molteplici:

- **Performance**
- **Sicurezza** : con alcune architetture ho un controllo più dettagliato
- **Safety** : se le funzionalità più critiche possono essere arginate a pochi componenti il controllo meglio
- **Availability/affidabilità** : potrei voler assegnare una funzionalità a più componenti per garantire la stabilità.
- **Mantenibilità**

Ciascuna vista ci permette di esporre caratteristiche diverse.

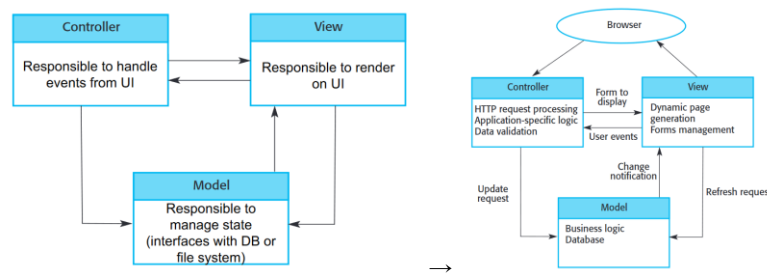


Pattern architetturali

Avere dei pattern ben definiti ha una serie di lati positivi:

- Sono un modo comodo di **comunicare informazioni con nomenclatura/simbologia conosciuta**.
- **Evito di reinventare la ruota** ogni volta: ci sono soluzioni preconfezionate che danno risposta a problemi tipici.

Model View Controller



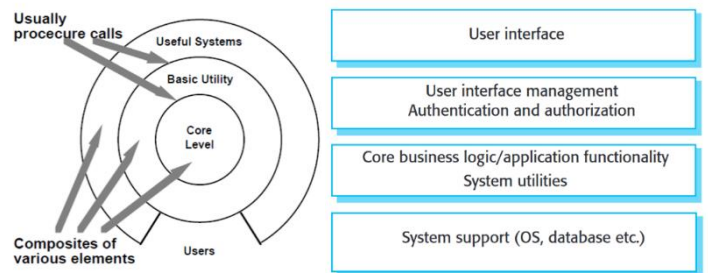
Quando va usato?	<ul style="list-style-type: none"> Quando ho gli stessi dati da visualizzare in tante schermate diverse Potrebero esserci altre interazioni coi dati che non so; con questo modello faccio subito a sviluppare nuove view.
Vantaggi?	<ul style="list-style-type: none"> I dati possono essere cambiati e la presentazione può essere cambiata indipendentemente Tante visualizzazioni su un dato Cambiamenti di una rappresentazioni sono viste su tutte
Svantaggi?	<ul style="list-style-type: none"> Questo scoppimento impone di avere un'architettura più complessa, e l'overhead potrebbe essere significativo su piccole app.

Architettura a strati

Prevede la presenza di più **strati**, uno sopra l'altro. Ciascuno strato **utilizza i servizi messi a disposizione** dallo strato sotto.

Ci permette di sviluppare facilmente in modo **incrementale**: parto dal core e aggiungo livelli.

Inoltre è **portabile**: posso cambiare ciascuno strato fintantoché rispetta le interfacce.

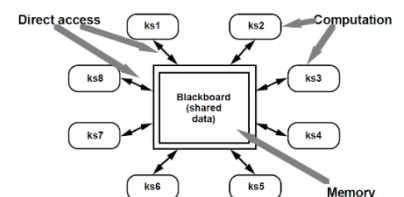


Quando va usato?	<ul style="list-style-type: none"> Sto costruendo un servizio sopra servizi già esistenti (es. ho già il kernel) Sviluppo degli strati è diviso fra team distinti; ho un accordo preliminare sull'interfaccia e poi posso sviluppare in parallelo Multilevel security
Vantaggi?	<ul style="list-style-type: none"> Alto livello di compatibilità Facile rimpiazzare le cose
Svantaggi?	<ul style="list-style-type: none"> È difficile isolare perfettamente ciascun layer; a volte potrei dover permettere dei salti di livello. Performance possono andare male, poiché tipicamente per raggiungere lo strato inferiore dal superiore ci sarà una catena di chiamate.

Esempio: android

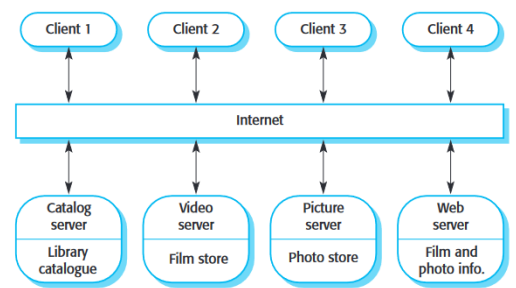
Repository architecture

Prevede che ci sia un componente centralizzato che contiene i dati, e vari componenti satellite che interagiscono SOLO con il componente centrale. Non c'è comunicazione fra i satelliti.



Quando va usato?	<ul style="list-style-type: none"> Large set of data che deve rimanere consistente (es. codebase) Data driven
Vantaggi?	<ul style="list-style-type: none"> Molto indipendenti l'uno con l'altro Cambiamenti immediatamente propagati a tutti gli sviluppatori del progetto Dati consistenti sempre, dato che c'è un solo punto di riferimento.
Svantaggi?	<ul style="list-style-type: none"> Single point of failure: se si rompe il database centrale sono fottuta 🤔😅😓😭 Tutte le comunicazioni passano per i dati: può essere un collo di bottiglia. Quindi devo fare molta attenzione al dimensionamento (in base al # di sviluppatori che devono lavorare)

Esempini: strumenti per automatizzare SW engineering, version control



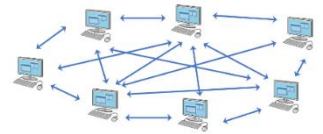
Architettura client-server

Ci sono uno o più componenti **forniscono servizi**; uno o più client li richiedono e consumano. Tipicamente accade via rete, con http o altro.

Quando va usato?	<ul style="list-style-type: none"> Dati messi a disposizione via servizi dislocati sul territorio Prevede la possibilità di replicare servizi in modo da poter prevedere un reindirizzamento per la gestione del carico (sarà un dispositivo di balancing del load a distribuire le richieste sui server)
Vantaggi?	<ul style="list-style-type: none"> I server possono essere distribuiti sulla rete Posso decidere dove implementare quali funzionalità (nel server) in base all'utilizzo.
Svantaggi?	<ul style="list-style-type: none"> Ogni servizio è un single point of failure; se un server muore, quel servizio è tolto a tutti i suoi client! Problemi di denial of service e così. Performance difficili da prevedere, dato che sono basate sulla rete e potrei non avere il controllo totale della rete dove transitano queste informazioni. Gestione: se l'architettura prevede servers gestiti da entità diverse potrei non avere il controllo sui tempi di risposta.

Peer2Peer

Generalizza l'architettura client server facendo sì che ciascun componente sia sia client che server, e nessun componente ha una responsabilità "maggiore" o un ruolo. Communism



Quando va usato?	
Vantaggi?	<ul style="list-style-type: none"> I componenti sono perfettamente intercambiabili, e quindi facilmente intercambiabili. Molto indipendente: se un nodo venisse disconnesso, il registro rimane distribuito nella rete.
Svantaggi?	

Esempini: bittorrent, bitcoin

Pipe-and-filter

Vari componenti partecipano a una pipeline di trasformazione dei dati.



Quando va usato?	<ul style="list-style-type: none"> Analisi dei dati, gestione di processo sui dati che prevede degli steps successivi.
Vantaggi?	<ul style="list-style-type: none"> Molto semplici, c'è un'assegnazione di responsabilità chiara basata sui dati e sui componenti È facile far evolvere queste architetture perché è facile aggiungere altri filtri. Possono essere facilmente implementati in maniera sequenziale o concorrente. Posso mandare avanti i dati mano a mano (anziché aspettare che tutti abbiano passato il filtro)
Svantaggi?	<ul style="list-style-type: none"> Non è appropriato se serve l'operatività dell'utente; essa romperebbe il meccanismo della catena di montaggio. Il formato dei dati deve essere deciso all'inizio; il modo in cui i componenti si scambiano i dati deve essere chiaro fin da subito Ogni componente deve farsi il parsing dei dati prima di poterli elaborare, poiché magari ciascuno usa una sua logica interna. Tipicamente il parsing è un'operazione abbastanza costosa... Se ho formati diversi è molto difficile riutilizzare i componenti.

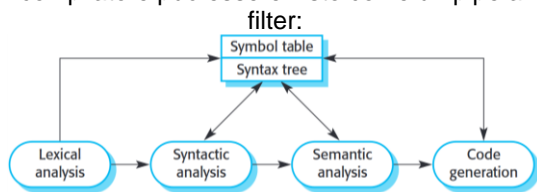
Esempini: grep

Architetture eterogenee

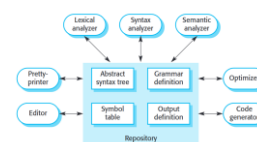
A volte è necessario avere caratteristiche più diverse. Quindi, potrei avere cose tipo divisione gerarchica con alcuni componenti pipe and filter ma che all'interno sono layer, oppure un componente ha più ruoli in più architetture.

Esempio: compilatore

Il compilatore può essere visto come un pipe and



Allo stesso tempo, però, si presta anche all'architettura repository:



6 - TESTING

È nella fase di **validazione**. È una di quelle attività che ci permette di mantenere alta la qualità

L'obiettivo del testing è stato chiarito da Dijkstra:

il software testing non può dimostrare la correttezza del programma; tuttavia, può farci capire se ci sono degli errori.

- **Trovare difetti nel software prima di consegnarlo.** (bug)
- **Implementa correttamente i requisiti** raccolti in precedenza; ci aspettiamo almeno:
 - Uno scenario di test per ogni requisito
 - Un test per ogni funzionalità e combinazione di funzionalità

Non potendo avere la certezza della correttezza, possiamo basarci sull'avere una certa confidenza. Questo livello di confidenza dipende da:

- **Scopo del purpose:** quanto è importante il software nell'organizzazione? Software critico != proof of concept
- **User expectation:** magari in alcuni casi l'utente può tollerare fail
- **Marketing environment:** potrebbe essere più importante uscire subito rather than rilasciare software corretto.

Tipicamente, le fasi sono le seguenti:

1. **Development testing:** viene effettuato durante lo sviluppo, dagli sviluppatori stessi
2. **Release testing:** tipicamente condotto da un team distinto
3. **User testing:** coinvolge gli utenti finali nella realizzazione del software per verificare che tutti funzioni anche nel loro environment

Terminologia

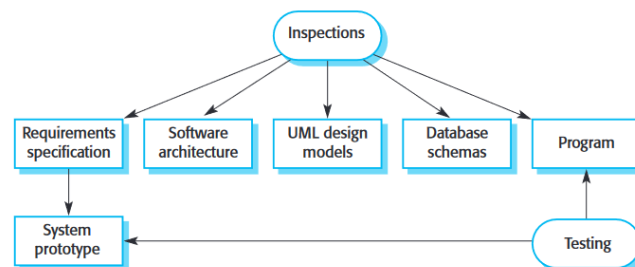
Testando un pezzo di software, il sistema viene generalmente chiamato **SUT**. (Tipicamente, oltre a pass e fail, c'è anche la condizione associata agli errori "gravi" e ai crash: è lo stato di error, dove magari non si arriva nemmeno ad avere un output da "failare".)

Code inspection

Oltre all'esecuzione dei test veri e propri, posso anche sfruttare la code inspection.

Vantaggi:

- **Non soffre del problema che gli errori potrebbero essere mascherati;** magari un primo errore mi interrompe l'esecuzione e mi nasconde errori successivi. (masking)
- Si può fare su **versioni incomplete** o non compilabili
- Posso estendere la review anche ad altri aspetti rispetto al funzionamento: **documentazione, qualità del codice, commenti, efficienza, compliancy to convenzioni** varie per mantenibilità.



Stage 1: Development testing

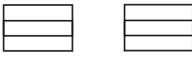
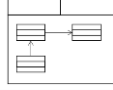
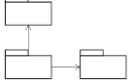
È condotto durante lo sviluppo dagli sviluppatori. L'obiettivo è trovare bug.

Si usano anche debuggers. Una volta che il test fallisce, devo seguire due fasi:

1. **Fault localization**

2. Fault removal


Si compone di diverse fasi:

	Unit testing si concentra sugli oggetti singoli <i>Focus: funzionalità di oggetti e metodi</i>
	Component testing più classi disponibili vengono messe in relazione e si testa l'unione delle classi. <i>Focus: interfacce dei componenti</i>
	System testing quando il sistema è sviluppato posso mettere assieme tutte le componenti e fare tests. <i>Focus: interazioni fra componenti</i>

Unit testing

Obiettivo: verificare il funzionamento di un'unit.

Un'unit è una funzione o una classe. Tipicamente, lo unit test è un pezzo di codice fatto da diverse parti

	<ol style="list-style-type: none">1. Setup: porta il sistema in uno stato testabile.2. Call: Codice che esercita la parte da verificare3. Parte assertiva: verifica se gli output sono coerenti o meno con l'aspettativa. Qui, dunque, si passa o fallisce. <p>Ove possibile, sarebbe caldamente auspicabile che i test di unità siano eseguibili in batch e in automatico.</p>
---	--

Test di classe

È difficile beccare tutto, perché per esempio per inheritance potrei aver bisogno di uscire dalla mia unità (e non sarebbe più unit testing). Devo testare:

- Tutte le operazioni associate all'oggetto
- Setting and querying tutti gli attributi
- Tutti i possibili stati

Solitamente, i tests fa fare sia quelli che riflettono la **normale operazionalità del programma**, sia quelli che somigliano a **casi dove spesso potrebbero esserci problemi**, come input abnormali e tentativi di far crashare.

Mock

Potrebbe succedere che abbiamo bisogno di funzionalità non ancora implementata. Dunque si usano i **mock**, ovvero **sostituti temporanei di altre classi** che simulano molto semplicemente l'unità necessaria. Presenta la stessa interfaccia, ma tornerà valori hardcoded o simili.

Guidelines: equivalence partitioning

Testare qualunque input è impossibile, chiaramente. Quindi si partiziona il dominio dei valori in **classi di equivalenza**. Una partizione degli input è una **divisione in sottoinsiemi disgiunti**. Assumendo che ciascuna partizione abbia valori equivalenti, mi basta scegliere **un solo valore** dalla partizione. Come li scelgo:

- Scelgo valori **di confine** fra domini e un valore centrale
- Se sono liste o array, provo sequenze con **N di elementi diverso, sequenze vuote, 1 elemento**
- Provo anche tutti gli **input sbagliati** per verificare tutti i tipi di errore, tipo input che **generano overflow**, forzare input non validi, provare a fare sì che gli output siano non validi
- Forzare il componente a generare **risultati molto piccoli o molto grandi**.

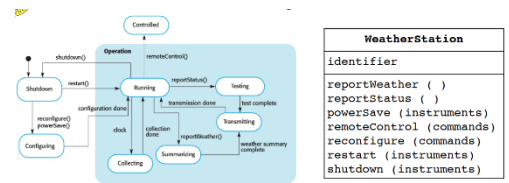
Component testing

Obiettivo: interfaccia fra più classi.

Quindi coinvolge come ciascuna classe invoca servizi dell'altra e comunica.

Verifiche possibili

- **Misuso dell'interfaccia:** ad esempio dati di tipo sbagliato o passati in ordine sbagliato
- **Misunderstanding dell'interfaccia:** ad esempio viene passata una lista di oggetti disordinata ma il chiamato invece assume che sia ordinata



Examples of state sequences that should be tested in the weather station:

- Shutdown → Running → Shutdown
- Configuring → Running → Testing → Transmitting → Running
- Running → Collecting → Running → Summarizing → Transmitting → Running

- **Errori temporali:** se ad esempio viene utilizzata un'area di memoria comune, magari i dati sono prodotti troppo velocemente e la classe client non riesce a leggere i dati in tempo prima che vengano sovrascritti.

Tipi di interfaccia:

- **Interfacce sui parametri:** dati passati da un metodo all'altro
- **Memoria condivisa:** area di memoria su cui entrambe le unità scrivono e leggono.
- **Interfacce procedurali:** sottosistemi che racchiudono un insieme di funzioni da far chiamare ad altri. Ma magari ho un ordine/protocollo con cui far chiamare questo, o vincoli di integrità sui dati passati

Guidelines

- Usare **valori estremi**, dato che tipicamente sono problematici
- Se ci sono **puntatori**, provare a passare il **Null**
- Test fatti apposta per fare fallire i componenti e **testare le condizioni di errore**
- **Stress testing:** fornisco dati a ritmo sempre più crescente per vedere se il componente va in errore.
- Se c'è memoria condivisa, provare ad avviare i componenti in ordine diverso per essere certi non ci siano assunzioni strane

System testing

Focus: interazione fra componenti per realizzare le funzionalità ad alto livello richieste dagli stakeholders.

Metto tutto insieme e guardo se va 😊

Verifiche possibili

Controlla se i sistemi:

- Sono compatibili
- Interagiscono correttamente
- Trasferiscono correttamente i dati utilizzando le interfacce

Tipicamente, uno use case riguarda l'interazione con diversi componenti! Quindi gli **use case** sono un'ottima base per questi tests. Se ci sono anche i **sequence diagram**, questi ci fanno anche vedere i messaggi che ci aspettiamo vengano scambiati in un funzionamento corretto. Il test dovrebbe focalizzarsi sia sul comportamento atteso che sulle eccezioni.

Policies per la completezza

Testare tutte le esecuzioni possibili è impossibile. Ergo si scelgono delle policies:

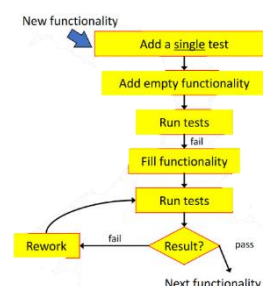
- Tutte le istruzioni di un programma devono essere eseguite da almeno un test
- Tutte le funzioni accessibili dai menu devono essere testate
- Se serve l'input dell'utente, bisogna testare almeno un input corretto e uno scorretto.

Sviluppo test driven

È stata introdotta con Agile. Scrivere il test prima della funzionalità mi obbliga a pensare alla funzionalità in termini di cosa deve fare (input/output) anziché come farlo.

Vantaggi:

- **Code coverage:** se ho uno o più test per ogni funzionalità, sicuramente tutte passano per almeno un caso di test
- **Regression test / test di non regressione:** è sempre possibile runnare tutti i tests ogni volta che voglio aggiungere cose, ed assicurarmi che la qualità del codice non regredisca se devo mettere mano a parti vecchie mentre ne aggiungo di nuove.
- **Simplified debugging:** un test che fallisce è relativo a una sola feature, quindi diventa facile trovare dov'è il bug
- **Documentazione:** posso leggere i test per capire le features del sistema.



Stage 2: Release Testing

Abbiamo una release pronta che è stata consegnata a un team diverso per il testing. Si adotta un **punto di vista black box**. Rispetto al system testing:

- Il release testing è una **forma di system testing**
- Il release testing è dato a un **team distinto**

- Nel system testing si cercano bugs, mentre nella release testing si verifica se i **requisiti sono stati implementati correttamente**.

<p>Scenario testing</p> <p>Uno scenario è una storia che descrive come si deve comportare il sistema. Il vantaggio è proprio che le situazioni sono più veritiere. Nel caso in cui scenari e user story siano disponibili dalla fase di raccolta dei requisiti, abbiamo il lavoro gratis 😊</p>	<p>Performance testing</p> <p>L'obiettivo è verificare che il sistema funziona correttamente quando sottoposto al carico che ci si attende di avere durante la produzione. Vogliamo verificare fin quanto si può sovraccaricare, e se il sovraccarico è gestito correttamente.</p>
---	---

Esempio:

Requirements:	Tests:
If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user	1. Set up a patient record with <u>no known allergies</u> . Prescribe medication for allergies that are known to exist. Check that a <u>warning message is not issued</u> by the system.
If a prescriber chooses to ignore an allergy warning, they shall provide a <u>reason</u> why this has been ignored	2. Set up a patient record with a known <u>allergy</u> . Prescribe the medication to that the patient is allergic to, and check that the <u>warning is issued</u> by the system.
	3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the <u>correct warning</u> for each drug is issued.
	4. Prescribe two drugs that the patient is <u>allergic</u> to. Check that <u>two warnings</u> are correctly issued.
	5. Prescribe a drug that issues a warning and <u>overrule</u> that warning. Check that the system <u>requires</u> the user to provide information <u>explaining</u> why the warning was overruled.

Stage 3: User Testing

Si vuole testare il software direttamente nell'ambiente dove verrà eseguito, dato che questo potrebbe influenzare il sistema software. Ci sono diverse tipologie:

Alpha testing	Beta testing	Test di accettazione
Consegno delle versioni preliminari del software a un numero ristretto di utenti. Lavorano insieme agli sviluppatori direttamente nel loro ambiente (es. sito del developer).	Un gruppo di utenti più grande è coinvolto nel testing del sistema nell'ambiente finale di lavoro.	Si coinvolgono i clienti finali; il cliente decide se accettare questa versione e se è pronta o meno. Tipicamente è dove si paga.
		<pre> graph TD A[Define acceptance criteria] --> B[Plan acceptance testing] B --> C[Derive acceptance tests] C --> D[Run acceptance tests] D --> E[Negotiate test results] E --> F[Accept or reject system] A --- A1[Before the contract is signed] B --- B1[Resources, timing, budget, order] C --- C1[Define tests so that (ideally) all the requirements are covered] E --- E1[Agree on how to fix failing tests] F --- F1[Or conditioned acceptance] A --- TC[Test criteria] B --- TP[Test plan] C --- T[Tests] D --- TR[Test results] E --- TR2[Testing report] </pre>

Nei metodi agile, il cliente è parte del development team. I tests sono definite dall'utente/cliente, e sono integrati con gli altri tests runnati in automatico. Quindi non c'è un processo separato per l'accettazione!

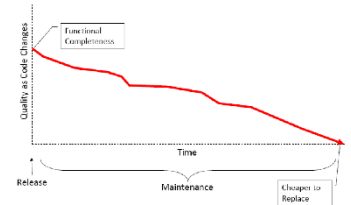
7 - REFACTORING

Refactoring: è un cambiamento **interno (=non osservabile dall'esterno)** alle strutture del programma che lo rendono più **semplice** e più **economico**.

È un processo che permette di **migliorare la struttura del codice** e **rallentare la tipica degradazione del software** che si genera mano a mano che si sviluppa il software. Il refactoring è una specie di **manutenzione preventiva**: prendo precauzioni prima che si presentino problemi al software. Modifica la struttura del codice per aumentare la comprensione del codice e semplificarlo, senza modificare il funzionamento/features del software.

Notiamo che codare senza fare manufacturing porta al decadimento del software. Vogliamo invertire questa curva, per:

- Test di casi più semplici da scrivere
- Semplificare la futura attività di manutenzione



Parentesi: Re-engineering

Il reengineering prende un sistema software mantenuto da molto tempo e mira a **riscrivere le parti** per adottare paradigmi, tecnologie e design più recenti.

È un'attività **puntuale**, che inizia a un punto e produce una nuova versione, il refactoring è invece un'attività continua che viene condotta in parallelo allo sviluppo del sistema software.

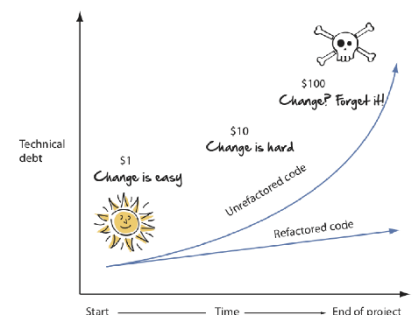
Debito tecnico

È importante calcolare bene come fare il refactoring, perché a mano a mano che il tempo passa senza fare refactoring il costo di aggiungere funzionalità diventa sempre più rilevante (perché dovrei capire le logiche astruse dentro); si arriva a un punto in cui aggiungere funzionalità diventa pressoché impossibile.

Questo sforzo è detto anche **debito tecnico**, ovvero un debito causato da una cattiva implementazione e una mancata comprensibilità del codice scritto. Facendo il refactoring mano a mano, invece, questo debito tecnico rimane basso in quanto sono semplici da comprendere.

Debito tecnico = lacune del software che rendono difficile la manutenibilità.

Ad esempio, potresti scegliere una terminologia non adatta, e poi mantenerla così per non sbatti di cambiarla. Sto generando debito tecnico perché poi un altro developer potrebbe non capire cosa sto facendo e avere difficoltà a implementare cose nuove. Se invece a una certa sistemassi i nomi, facendo refactoring, abbasserei il debito tecnico rendendo il codice più comprensibile.



Quando fare refactoring

È una task che sta allo sviluppatore capire quando fare; tuttavia esistono linee guida. Comunque, è sempre bene farlo il più spesso possibile. Sarebbe bene farlo:

- **Prima di aggiungere nuove funzionalità**, per semplificarsi l'aggiunta della nuova funzionalità e non dover rifactorare anche quella.
- Dopo una **bug fix**
- Quando c'è **code smell**



Code smell ☹️ ☁️ ☹️ ☁️ ☹️

È un identificatore di quando c'è codice di bassa qualità, legato allo stile o alla comprensibilità.

Problemi tipici:

- **Codice duplicato**: Se dovessi modificare poi devo propagare manualmente la modifica! E errori vengono propagati! E la dimensione del software cresce in maniera innaturale 😞
- **Metodi lunghi**: probabilmente questo metodo ha troppe funzionalità e sarebbe appropriato spezzarlo in sottometodi. Tipicamente come regola la dimensione del metodo dovrebbe essere circa quanto una

schermata, oppure se ci sono dentro troppi commenti e si evincono delle “sezioni” che potrebbero essere separate.

- **Switch-case:** attiva una funzionalità diversa a seconda di un valore. Tipicamente questo pattern è ripetuto, e si preferisce togliere lo switch case per rimpiazzarlo con il polimorfismo dei linguaggi object-orientate.
- **Coaguli di dati / data clumping:** intendiamo un insieme di dati di una classe o di una struttura che vengono sempre usati assieme. In tal caso forse ha senso prendere questi dati e metterli in una struttura separata, fornendo dei metodi per accedervi in una botta sola. (=è tipo una classe mancata)
- **Speculative generality :** succede quando lo sviluppatore include cose molto generiche nel codice pensando che nel futuro potrebbe usarlo. Invece è meglio che se non serve la togliamo, in quanto complicazione inutile.
- **Troppo codice:**
 - **Classi troppo grandi (God-class):** meglio spezzarle in più sottoclassi, similmente ai metodi.
 - **Codice morto:** codice che non è mai eseguito. Se non è eseguito mai mai c'è un problema
 - **Long parameter list:** se abbiamo una funzione con troppi parametri formali diventa difficile capire come chiamarla. Forse non stiamo usando l'incapsulamento in maniera opportuna.
- **Troppo poco codice:**
 - **Classi quasi vuote (Data-class):** se abbiamo solo getter/setter le funzionalità di quei dati magari sono in altre classi, quindi stiamo violando la filosofia dell'object oriented
 - **Empty catch clause:** ci piace il rischio, dato che un errore nel try verrebbe catturato e il programma andrebbe avanti senza problemi immediati (ma likely problemi dopo).
- **Outside code:** se ho bisogno di troppi commenti forse il codice non è proprio chiaro

Cloni software

Per codice clonato intendiamo **codice copiaincollato**, o magari quasi copiaincollato con qualche modifica 😊 Il principale problema è che 1. faccio anche **bug propagation** e 2. è molto **difficile fare manutenzione** (dovendo propagarla per ogni copia).

Si stima che 5%-20% del codice sia duplicato – quindi è molto prominente. Esiste software che ci aiuta a tenere traccia del software duplicato.

Esistono più tipi di test duplicato:

- **Clone di tipo 1:** è un clone esatto
- **Clone di tipo 2:** è sempre copia e incolla ma abbiamo cambiato il nome dei parametri locali.
- **Clone di tipo 3:** ci sono dei cambi veri e propri, come per esempio il tipo dei parametri formali. Ma la funzionalità è sempre la stessa!

Cataloghi (classificazione)

Esistono una marea di cataloghi di refactoring: si parte da uno smell e i possibili refactoring da applicare per risolvere il problema. [Catalogo del Fowler \(libro\)](#) [Altro catalogo](#)

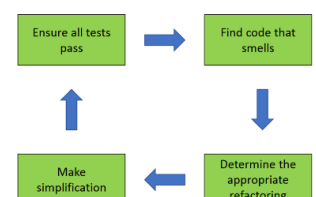
Smell	Description	Refactoring
Incomplete Library Class	Occurs when responsibilities emerge in our code that clearly should be moved to a library class, but we are unable or unwilling to modify the library class to accept these new responsibilities. [F 86]	Introduce Foreign Method [F 162]
		Introduce Local Extension [F 164]

Tipicamente possiamo classificare i refactoring in:

- Refactoring di **basso livello:** rinominare/estrarre metodi, ... Sono spesso automatizzati dall'IDE.
- Refactoring **complessi:** sono una lista di refactoring semplici. Non sono fatti dall'IDE (se non come sequenza di refactoring semplici).

Processo di refactoring

Il processo di refactoring deve essere accompagnato dal testing di regressione per essere certi di non introdurre errori nel codice. Avendo nessuno o pochi test sicuramente mettere mano al codice ci porta dover fare molte modifiche manualmente, ed essere in genere molto meno sicuri dei cambiamenti approtati.



Applicazione del refactoring

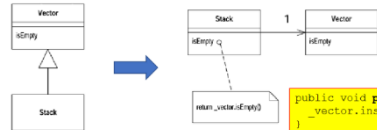
Può essere svolto **a mano** o con **tool automatizzati**; per esempio, IntelliJ ne implementa alcuni (tipo classe rinomina), ed è bene utilizzarli per risparmiare tempo dal dover cambiare tutti i costruttori etc. Tuttavia, le cose automatizzate sono tendenzialmente solo per il refactoring semplice, e ci sono articoli che dimostrano che gli strumenti automatici possono introdurre degli errori.

Esempi: Jdeodorant (LOL), ReShaper...

Refactoring semplici

Extract interface	Voglio disaccoppiare le classi dalla struttura dati , perché così per esempio potrei voler cambiare una array list in una linked list e questo mi aiuterebbe a non dover riscrivere il codice.
Pull up	Ho una serie di classi. Voglio spostare dei metodi nelle super classi per renderli più generici.
Extract method	Se ho codice duplicato lo trasformo in un metodo che chiamo . Mi chiederà il nome del metodo e eventuali parametri da passare
Move method	Sposto un metodo in modo consistente, aggiornando riferimenti a dati e metodi. Nell'esempio, spostiamo il metodo per sapere se una persona partecipa al progetto dalla persona al progetto dato che fa riferimento alle cose di project: <code>participate(person)</code> → <code>participate(project)</code>
Replace temp with query	Se ho una variabile locale calcolata, la sostituisco con dei metodi che me la calcolino lì per lì. Questo è per disincentivare metodi lunghi.
Replace parameter with method	Accorcio il # di parametri : se ho qualche parametro che può essere calcolato con gli altri, lo calcolo ogni volta anziché richiederlo in input.
Extract class (Godclass o Blobclass)	Classi con poca coesione che potrebbero essere spezzate facilmente . Esponiamo comunque il codice di <code>gettelephonenumber</code> per mantenere l'interfaccia, però si limita a chiamare la classe <code>telephonenumber</code> .

Refactoring complesso

Replace inheritance with delegation	<p>Se ho una sottoclasse che usa una porzione della superclasse ma non usa nient'altro, comunque vede tutte le funzionalità esposte. → Sostituisco l'ereditarietà con la delega: <code>stack</code> usa <code>vector</code> come struttura dati, chiamando semplicemente i metodi</p>  <pre>public void push(Object element) { _vector.insertElementAt(element, 0); } public Object pop() { Object result = _vector.removeElementAt(0); return result; }</pre>
Replace conditional with polymorphism	È il caso di switch case visto prima. Per esempio, se ho tante forme, anziché fare un oggetto che decide come calcolare l'area con un bello switch case che mi guarda in che forma sono, creo tanti oggetti forme (ognuno con il suo "calcola area") e uso quello.
Separate domain from presentation	Non unire la classe GUI (che così sarebbe una blob class) con le classi di calcolo :')

Fare refactoring senza testing è come correre al buio si va veloci ma si rischia di sbattere
- Mariano Ceccato 2021

8 – PROJECT MANAGEMENT

Il software management è una parte essenziale per far sì che il progetto soddisfi:

- Constraint **organizzativi**
- Constraint di **budget**
- Constraint di **schedule**

! Good management non garantisce il successo del progetto, ma un bad management risulta nel fallimento del progetto.

In ambito software ci sono anche delle sfide ulteriori:

- È qualcosa di **non tangibile**, è difficile tenere traccia dell'avanzamento del progetto.
- Ciascun progetto software è **unico**: l'organizzazione sarà unica in quanto sta sviluppando qualcosa di nuovo, magari in ambiente nuovo.
- Il processo dipende anche da ciascuna **organizzazione**/azienda.

Ci sono numerosi fattori che influenzano il project management:

- **Dimensione dell'azienda**: azienda piccola: più informale
- **Stakeholders**: customers esterni magari sono più formali
- **Dimensione del progetto**: grandi progetti magari contemplano tanti teams, o distribuzione geografica
- **Software type**: sistemi critici hanno bisogno di più giustificazioni e carte
- **Cultura organizzativa**: essere disposti a prendere dei rischi, più o meno burocrazia
- Processo di sviluppo software: **agile o formale**

Attività di management

1. **Project planning**: i managers sono responsabili della pianificazione, delle stime e dell'assegnazione dei task alle persone
2. **Risk management**: bisogna individuare i rischi che potrebbero causare dei problemi al progetto.
3. **Management** delle persone: i managers dovranno scegliere le persone per il loro team
4. **Reporting**: i managers devono rendere conto del progresso ai clienti e ai managers dell'azienda.
5. **Proposal writing**: progettato il progetto, bisogna scrivere una proposal tipicamente consegnata al decision maker. Più sarà elaborata, più egli potrà fare una decisione informata.

1. Risk management

Consiste nell'identificare i rischi che potrebbero impedire di raggiungere i goals del progetto, e di preparare delle azioni per minimizzare l'effetto di questi rischi.

È cruciale a causa delle incertezze intrinseche dello sviluppo software.


Processo di risk management









1. Identificazione dei rischi

È la più impegnativa perché bisogna cercare di calcolare tutti i rischi in cui potrebbe incappare il progetto. Sono di vario tipo.

In ambito software abbiamo:

Tipo di rischio	Rischi possibili
Stima 	Errori nello stimare quanto tempo o persone servono per scrivere software. Dato che scrivere software è molto intangibile, è facile aver tralasciato feature o aspetti che poi scopro essere importanti <ul style="list-style-type: none">• Il tempo richiesto è sottostimato

	La dimensione del software è sottostimata
Organizzazione 	Sono rischi che emergono dall'ambiente organizzativo. <ul style="list-style-type: none"> L'azienda è costretta a fare una ristrutturazione aziendale e cambiano i manager Il budget diminuisce causa problemi finanziari dell'azienda
Persone 	Provengono dalle persone che lavorano nel team di sviluppo. <ul style="list-style-type: none"> Persone nel team che si mettono contro il progetto stesso. Non si riescono ad assumere persone con le skill necessarie Non posso fare training Sviluppatori malati in un momento critico
Requisiti 	Provengono da cambiamenti nei requisiti da parte dei clienti o simili. <ul style="list-style-type: none"> stakeholders potrebbero voler rivedere i requisiti Arrivano nuove normative...
Tecnologia 	Provengono dalle tecnologie software o hardware (o cloud) usate <ul style="list-style-type: none"> Il database non riesce a soddisfare il rate di richieste. Un componente software che ho deciso di integrare contiene dei difetti che pregiudica l'utilizzo
Tools  	Provengono da sistemi di supporto allo sviluppo. <ul style="list-style-type: none"> Generatori di codici che partono dai diagrammi UML, o cose simili. Magari il codice generato così non è efficiente e creo dei rischi

2. Analisi dei rischi

Rappresentiamo il rischio su due assi:

- Probabilità** che si presenti: molto basso, basso, moderato, alto o molto alto
- Seriousness**: che conseguenze avrei se si manifestasse : catastrofico, serio, tollerabile, insignificante

Tipicamente usiamo delle scale, che quindi sono qualitative e sono molto arbitrarie.

Esempi

<i>Rischio</i>	<i>Probabilità</i>	<i>Effetto</i>
Problemi finanziari in azienda portano a ridurre il budget	Bassa	Catastrofico
Impossibile assumere persone con le skillz necessarie	Alta	Catastrofico
Staff chiave malato in momento critico	Moderato	Serio
Difetti in software riutilizzato che devono essere aggiustati prima di poterlo usare	Moderato	Serio
Codice generato automaticamente è inefficiente	Moderato	Insignificante
Cambio dei requisiti che impongono grossi cambiamenti	Moderato	Serio
Cambio di organizzazione e cambiano i managers	Alto	Serio
Database usato non riesce a star dietro alle transazioni	Moderato	Serio

3. Risk planning

Per ciascun rischio:

- Avoidance - Strategie per evitare il rischio**: riduco la probabilità che il rischio si presenti, o cambio qualcosa nel progetto per evitare il problema altogether. Non sempre è possibile
- Minimization**: riduco l'impatto che questo rischio avrebbe sul progetto.
- Contingency**: scatta quando il rischio si manifesta per cercare di contenere i danni.

Esempi

<i>Rischio</i>	<i>Strategia</i>
Componente difettoso	[Avoidance] Sostituisco i componenti potenzialmente difettosi con componenti più reliable, magari comprati
Impossibilità di avere lo staff (es. malattia)	[Minimization] Non posso evitarlo; riorganizzo il team in modo che ho sempre responsabilità spalmate, e quindi ho sempre qualcuno con un po' di conoscenza su quello che mi serve.
Problemi finanziari	[Contingency] Non dipende da me, quindi posso farci poco; posso avere pronto un report per cercare di convincere il management che questo progetto è estremamente strategico.

4. Monitoring dei rischi

Devo continuare a valutare i rischi, e rendermi conto se un rischio sta aumentando di probabilità. Questo mi permette di non essere sorpreso* e accorgermi mano a mano che un rischio sta diventando più probabile. Posso farlo attraverso indicatori:

<i>Tipo di rischio</i>	<i>Indicatori da monitorare</i>
Stima	Non riesco a meettare una schedule.
Organizzativi	Gossip, mancanza di azione dai managers...
Persone	Monitoro lo stato d'animo degli sviluppatori, gente che abbandona o si licenzia spesso...
Requisiti	Lamentele dal cliente; tante richieste di cambiamento (ce ne erano altri ma li ha balzati

2. Gestione del personale

Tipicamente è difficile e costoso riuscire ad assumere e tenere dei bravi ingegneri del software, perché il mercato ha molte richieste. Il project manager deve trovare degli ingegneri che lavorino produttivamente, ma spesso queste figure non hanno buone soft skillz..

Teamwork

Uno degli obiettivi è avere un team abbastanza **coeso**, che lavori bene insieme. Per ottenere ciò:

- Utilizzare degli standard di qualità
- Far sì che ognuno possa imparare dagli altri
- Condividere la conoscenza, per mantenere la continuità se un membro lascia il lavoro
- Continuo refactoring e miglioramento. I membri lavorano insieme per portare risultati di qualità e aggiustare problemi.

Composizione del gruppo

Per comporre bene un team, bisognerebbe cercare di conoscere i tipi di personalità che possiamo incrociare:

Task oriented 🙌	Persone molto motivate, che trovano motivazione nel lavoro stesso (es. serotonina quando consegnano buon codice). Lavorano bene da sole. La maggior parte sta qui
Self oriented	La motivazione del lavorare è raggiungere degli obiettivi personali esterni al lavoro, tipo diventare ricchi. Il problema è che vorrebbero avere il comando (aka è una buona idea renderli capi, ma problemi se ne ho più di uno)
Interaction oriented	La motivazione per lavorare è essere in gruppo e interagire con gli altri. Stanno bene in compagnia; c'è il rischio che perdano troppo tempo a comunicare e discutere ma tralascino poi il resto del lavoro.

Esistono anche altri nomi per queste personalità, tipo task oriented → rockstar

3. Project planning

Plan driven project planning

Ci riporta un po' al processo plan driven. Definisco dei piani in cui decido cosa fare, come suddividere, a chi assegnare e delle deadline.

I managers useranno il piano per fare **scelte migliori** e per **misurare gli avanzamenti**.

Vantaggi:

- Pianificando dall'inizio si possono considerare molto bene possibili problematiche organizzative (es. disponibilità dello staff)
- Si scoprono problemi e dipendenze prima di iniziare

Spesso però poi bisogna revisionare le scelte a causa di nuove variabili. È quindi necessario rivisitare spesso il piano.

Project scheduling

È il processo di decisione come dividere il lavoro in task e come eseguire le task. Lo scheduling è abbastanza dettagliato, e mi permette per esempio di organizzare le task per esplicitare:

- **Durata:** tempo necessario e calendarizzazione. Dovrebbe durare 6-8 settimane.
- **Sforzo:** stimato, ovviamente; mostra il numero di persone necessari per completare lo sforzo.
- **Deadline**
- **End-point:** può essere un documento, un meeting, aver finito tutti i test, etc
- Chi ci lavora

- Risorse e dipendenze che servono a completare la task (spazio sul server...)
Diminuire le dipendenze permette di evitare ritardi dati dal fatto che una task deve aspettarne un'altra.

È bene che le tasks siano eseguite in concorrenza per una questione di ottimizzazione.

Altri concetti importanti:

Milestones	Deliverables
punti in cui si può <i>assessare il progresso</i> , per esempio il momento in cui passo a fare testing	prodotti che sono <i>consegnabili al cliente</i> , ad esempio un documento dei requisiti

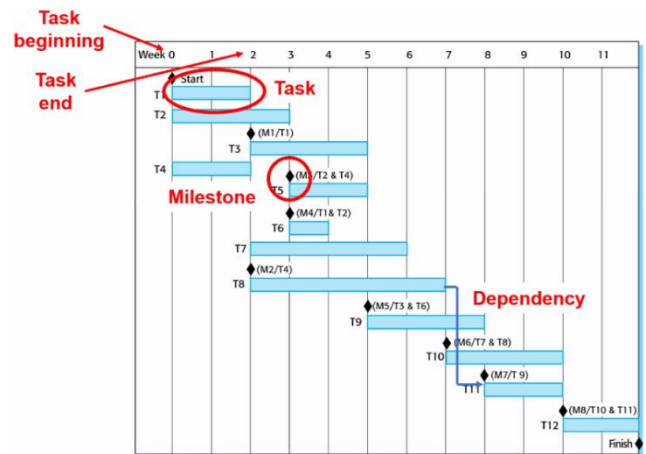
Ad esempio:

Task	Sforzo (persone/giorno)	Durata	Dipendenze
T1	15	10	
T2	8	15	
T3	20	15	T1

Diagramma di Gantt

È un tipico diagramma ed è una tipica domanda di esame :) È un diagramma che rappresenta il **flusso temporale del progetto**.

- Nelle ascisse c'è lo **sviluppo temporale**
- Nelle ordinate ho le varie **task**; ciascuna barra è una task. La barra ha una data di inizio, fine e una durata.
- Le **milestones** sono dei diamanti posti a inizio o fine task.
- Posso rappresentare o meno le **dipendenze**, con delle frecce che terminano nella task che ha la dipendenza. Nell'esempio, la task P8 deve essere completata prima della task P11.
CREDO si possano anche mettere tra parentesi, ma ovviamente è più comunicativa con le versioni delle frecce



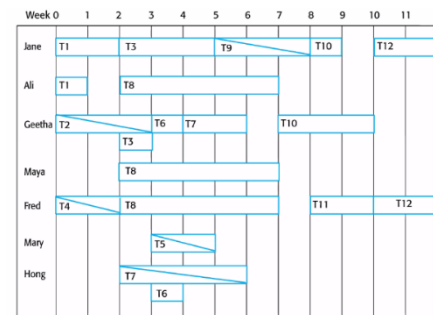
Fa capire a colpo d'occhio come sono posizionate le task e le loro dipendenze. Permette anche di fare alcuni ragionamenti: per esempio, potrei voler spostare avanti o indietro delle tasks per minimizzare l'impatto di eventuali problemi e/o ritardi

Staff allocation chart

Un altro diagramma comune è lo staff allocation chart, che scritto il diagramma di Gantt permette di **allocare facilmente le persone**.

Sulle righe metto le persone, e per ogni persona metto delle righe che mostrano le task a cui è allocata quella persona.

Inoltre, realizzo subito se ci sono persone sovraccariche, o eventuali problemi in caso di task che prendono più tempo del previsto (quindi cerco di far succedere dei gap fra un task e l'altro)



Agile planning

L'alternativa è la gestione agile, che ricalca l'agile visto nel processo software. Sostanzialmente si mette più enfasi sull'iteratività, flessibilità e incremento. È un approccio iterativo dove il software viene sviluppato e consegnato per incrementi.

Troviamo due fasi:

- **Release planning:** look ahead di qualche mese
Si decide cosa rilasciare nella release di sistema
- **Iteration planning:** look ahead di qualche settimana
Si concentra sul pianificare il prossimo incremento.

Per farlo si usa il planning game, originariamente sviluppato in XP ma che ora si può usare anche in generale.



- **Identificazione di storie:** rappresentano le feature. Il team di sviluppo e il cliente lavorano assieme per scrivere le user stories che riflettono le features che dovrebbero essere incluse nel sistema.
 - **Stima iniziale:** Per ciascuna si assegna una prima stima dello sforzo necessario, attraverso gli effort points. Il numero di punti prodotti dipende dalla velocity. E tutte le stime andranno corrette mano mano
- **Release planning:** identifichiamo le release, definendo anche il tempo totale. Selezioniamo e rifiniamo le storie che riflettono le features da implementare, scegliamo l'ordine e usiamo la velocity per stimare cosa produrre. Infine, si sceglie la durata delle iterazioni (2-3 settimane)
- **Iteration:** all'inizio di ciascuna iterazione di sviluppo. Dividiamo le user stories in tasks di sviluppo, dove ciascuna task dovrebbe prendere 4-16 ore.
 - **Allocazione delle task:** ogni membro del team prende le task che ritiene essere più adatte a lui. Tutte le task devono essere completate; a metà iterazioni si fanno piccole review per capire come si sta procedendo.

A metà iterazione si fa una review per correggere il tiro se necessario; la schedule di delivery però non può essere missata.

Vantaggi	Svantaggi
<ul style="list-style-type: none"> • Tutti i membri del team hanno una visione completa degli assignments: <ul style="list-style-type: none"> ◦ Sanno le dipendenze ◦ Sanno a chi chiedere in caso di problemi • I developer decidono le proprie tasks <ul style="list-style-type: none"> ◦ Senso di ownership 😊 	<ul style="list-style-type: none"> • Se i customers non sono disponibili il planning non è per nulla efficiente <ul style="list-style-type: none"> ◦ I customers potrebbero non essere familiari con le tecniche agile • Agile funziona meglio con teams piccoli e stabili. <ul style="list-style-type: none"> ◦ Potrebbero esserci problemi con team più numerosi o con un turnover alto; progetti più grandi di solito si fanno col plan driven