

# Fondamenti di ingegneria del software

## Processi software



**Processo software:** insieme strutturato di attività che sono richieste per sviluppare un sistema.

Ci servono per mantenere lo sviluppo:

- **controllato**
- **ordinato**
- **ripetibile**

L'obiettivo è quello di aumentare la produttività degli sviluppatori e controllare la qualità del prodotto, è ovvio pensare che un processo di qualità porta a un prodotto di qualità.

Ci sono vari tipi di processi software ma tutti includono le attività di:

- **specificazione:** si definisce cosa dovrebbe fare il sistema
- **progettazione e implementazione:** si definisce l'organizzazione del sistema e lo si implementa
- **validazione:** si verifica che il sistema faccia ciò che era richiesto
- **evoluzione:** si cambia il sistema in base alle necessità del cliente

I processi pur essendo un insieme di attività, considerano anche:

- **prodotto:** l'output
- **ruoli:** responsabilità delle persone coinvolte nel processo
- **pre/post condizioni:** dichiarazioni vere prima e dopo un'attività di processo o dopo la messa in produzione di un prodotto

## Categorie di processi Software

**Plan Driven Processes**

**Agile Processes**

Tutte le attività dei processi sono pianificate in maniera dettagliata e il progresso è misurato su questo piano generato

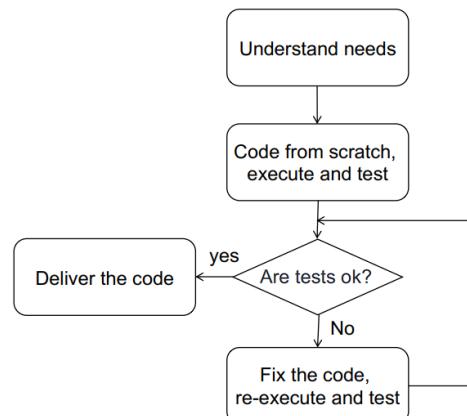
La pianificazione è incrementale, è più facile cambiare i processi per soddisfare cambiamenti da parte del cliente.

## Modelli di processi Software

### Code and Fix (non è un vero processo)

Non è di fatto un processo software, il codice è sviluppato a tentativi, si sviluppa correggendo fintanto che ci sono errori e poi si consegna, non c'è analisi e tantomeno fasi di design, tipicamente usato per progetti di piccole dimensioni e con singoli individui

*Sconsigliato per lavori in Team e progetti che richiedono documentazione  
Non garantisce precisione*



### Modello Waterfall

È un modello plan-driven, separa e distingue le fasi di specifica e sviluppo, inoltre ogni output di una fase è l'input per la successiva.

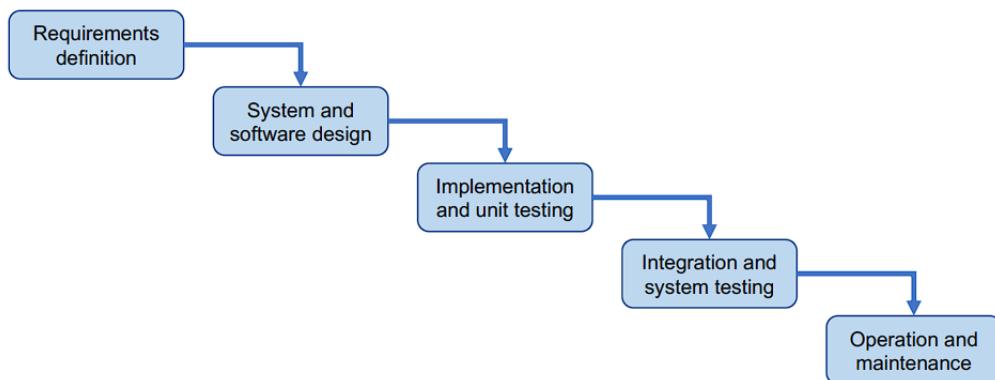
Si parte con la **definizione dei requisiti** come obiettivi, funzionalità, vincoli consultandosi con gli utenti, i requisiti raccolti sono poi definiti in dettaglio e diventano specifiche del sistema, si passa poi alla **progettazione del sistema** dove si fornisce un'architettura generale del sistema e si identificano e descrivono le principali astrazioni e relazioni tra queste ultime.

La fase seguente riguarda l'**implementazione** e lo **unit testing**, il software viene sviluppato come un'insieme di unità che vengono poi testate già durante lo sviluppo del sistema.

Nella penultima fase, ossia quella di **Integrazione e system testing** si assemblano le varie unità arrivando ad avere il sistema completo che viene testato e infine consegnato

al cliente.

Giunti all'ultima fase, quella di **operation and maintenance** il software è quindi messo in produzione ed entra nella fase più lunga del suo ciclo di vita, da qui in poi deve essere manutenuto per: correggere errori, aggiungere nuove features e migliorare l'implementazione delle varie unità.



Tra le caratteristiche principali di questo processo c'è il **congelamento di una fase** prima di passare alla successiva(una volta finita non si torna indietro), è molto utile se si vogliono limitare i costi nello sviluppo hardware, mentre è meno efficace nel contesto software dove le fasi successive potrebbero dare feedback agli output delle precedenti.

È molto utile ~~nello~~ sviluppo ~~di~~:

- **sistemi integrati software-hardware:** in quanto modifiche all'hardware a fine sviluppo non sono realizzabili
- **sistemi critici:** c'è una profonda analisi della sicurezza del software e della protezione, se si procede in un futuro con la risoluzione di problemi di sicurezza si potrebbe incombere in grandi costi.
- **Sistemi molto grandi:** sistemi composti di tanti sistemi sviluppati da diverse aziende.

### Vantaggi

- Sollecita l'analisi dei requisiti e il design del sistema
- Ritarda l'implementazione solo dopo analisi accurate delle necessità

### Svantaggi

- Una fase deve essere totalmente completa prima di procedere alla prossima

↗ SONO AMMESSI  
 salti all'indietro  
 ma il costo aumenta  
 più grande è il  
 salto

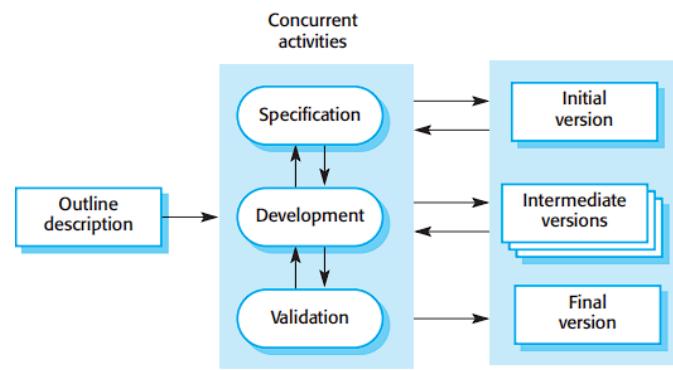
dell'utente

- Introduce pianificazione e sviluppo disciplinato
- Aiuta a coordinare i lavori

- Difficoltà nell'introdurre cambiamenti a processo in corso, va quindi usato solo quando i requisiti sono veramente chiari e i cambiamenti saranno molto limitati

## Sviluppo Incrementale

L'implementazione iniziale è mostrata agli utenti finali in maniera da avere dei feedback, il software si evolve attraverso diverse versioni fino a che il sistema completo non è terminato, è preferibile al waterfall se i requisiti possono evolversi durante lo sviluppo. Per quanto riguardale varie fasi le attività di **specificazione, sviluppo e validazione sono svolte concorrentemente con rapidi feedback tra queste ultime.**



## Vantaggi

- Costi ridotti ai cambiamenti di requisiti, di conseguenza anche meno analisi e documentazione da dover rifare
- È più facile ottenere feedback dagli user (lo user può commentare delle demo e gli incrementi man mano che vengono conclusi)
- È possibile rilasciare anticipatamente funzionalità principali del sistema anche se quest'ultimo non è ancora stato completato

## Svantaggi

- Il processo non è marcato tanto quanto il waterfall, è difficile capire se si è in ritardo o in linea coi tempi, inoltre la documentazione è minimale (non conviene fare una documentazione completa se l'obiettivo è consegnare in fretta).
- La struttura del software tende a degradare con l'introduzione di nuove features, anche nel caso venissero spesi molti soldi e tempo in refactoring.

## Integrazione e configurazione

Basato sul riutilizzo del software, si selezionano e integrano componenti di sistemi già esistenti sul prodotto da consegnare.

Gli elementi utilizzati possono essere configurati per adattarsi alle richieste dell'utente, ad oggi è una tecnica molto diffusa.

Tra i componenti comunemente usati troviamo **sistemi stand-alone(cots)** con molte features configurate per un particolare ambiente, ma anche **framework e web services**.

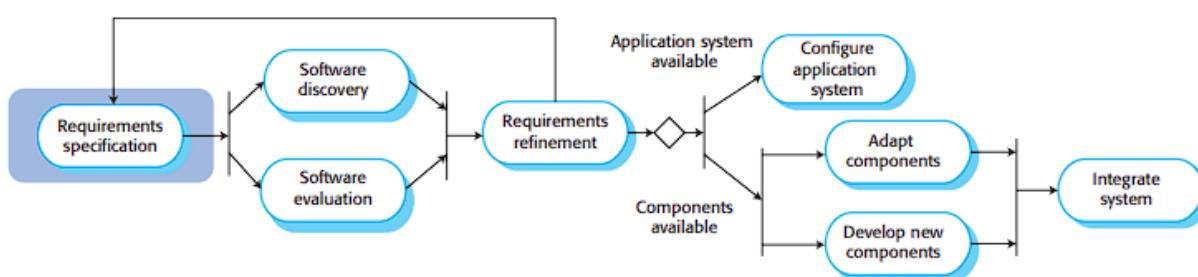
Per quanto riguarda le fasi si ha la **specifiche dei requisiti** che inizialmente è minimale con brevi descrizioni dei componenti essenziali e feature desiderate. Si passa poi alla **ricerca di componenti e sistemi** da riutilizzare che forniscono le date funzionalità, una volta trovati si valutano per capire se rispecchiano le aspettative e se è fattibile utilizzarli.

A questo punto vengono **perfezionati i requisiti** usando le informazioni dei componenti trovati, vengono modificati in maniera da ~~lo~~ pag 27.

A questo punto ci si trova ad un bivio:

essere compatibili  
con i componenti a  
disposizione

- Si ha un **sistema completo** da integrare: viene configurato per il sistema che sta venendo creato
- Si hanno **componenti** da integrare: i componenti riutilizzabili possono essere modificati, possono essere inoltre creati nuovi componenti e infine vengono uniti e integrati per andare a formare il sistema che sta venendo creato



## Vantaggi

- Bassi costi
- Basso rischio
- Poco sviluppo di codice da 0
- Consegna rapida

## Svantaggi

- Bassa qualità
- Tradeoff nei requisiti e rischio che il sistema finale non soddisfi l'utente
- Perdita di controllo dell'evoluzione dei componenti riutilizzati

devo adattarmi ai componenti già implementati

## Attività dei processi Software

- **Specificazione:** definire cosa dovrebbe fare il sistema
- **Validazione:** controllare che il sistema fa quello che lo user chiede
- **Design e implementazione:** definire l'organizzazione del sistema e svilupparlo
- **Evoluzione:** cambiare il sistema in risposta a nuove necessità dell'utente

## Specificazione (Ingegneria dei requisiti)

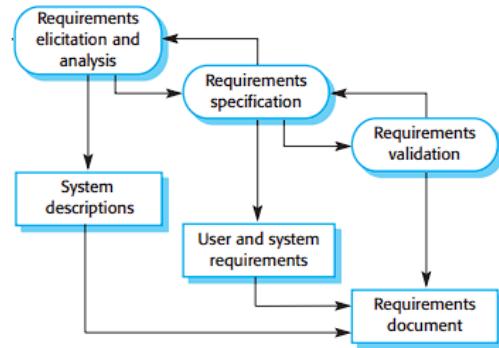
Capisce i servizi richiesti e definisce i vincoli che il sistema deve rispettare, è **critico** in quanto un errore in questa fase **causerà sicuramente** un errore nella fase di design/sviluppo.

Può essere preceduta da uno **studio di fattibilità** per capire se procedere o meno con analisi più dettagliate, il risultato di questa fase è un **documento dei requisiti** concordato che specifica un sistema che soddisfa i requisiti degli stakeholders.

Ci sono varie fasi:

1. **Elicitazione e analisi dei requisiti:** si ottiene una descrizione del sistema attraverso osservazioni di sistemi esistenti, discussioni con gli utenti e stakeholders, analisi delle attività e altro. Può includere lo sviluppo di un modello o prototipo del sistema, in generale quindi aiutano a capire come sarà il sistema.

2. **Specificazione dei requisiti:** traduce l'output della fase precedente in un documento con un insieme di requisiti, che possono essere di due tipi:
- Requisiti dell'utente: dichiarazioni astratte dei requisiti del sistema per gli utenti/clienti.
  - Requisiti di sistema: descrizione più dettagliata di come il sistema fa a implementare quella funzionalità.



3. **Validazione dei requisiti:** controllo della realisticità, consistenza e completezza dei requisiti, gli errori nel documento dei requisiti devono essere trovati in questa fase(se trovati il documento va modificato).

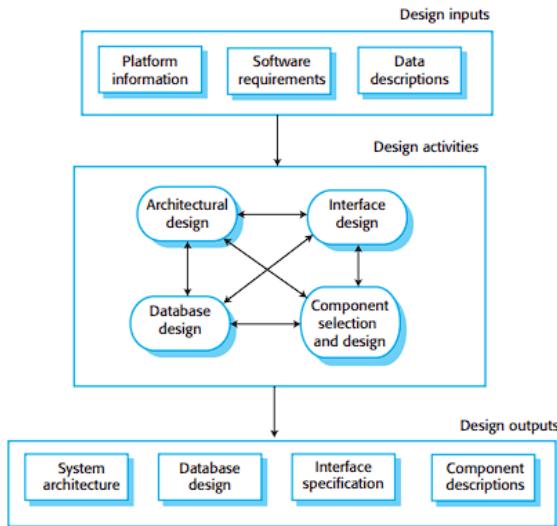
## Software Design e Implementazione

In questa fase si trasformano le specifiche in un sistema eseguibile che può essere consegnato al cliente:

- **Design:** descrizione della struttura del software, dei modelli dei dati e strutture usate dal sistema, delle interfacce tra i componenti e a volte anche degli algoritmi utilizzati
- **Implementazione:** traduzione del design in un eseguibile

Tra le **attività di design** troviamo:

- **Design architetturale:** identificazione di una struttura generale del sistema (componenti principali, relazioni tra questi e come sono distribuiti).
- **Design delle interfacce:** interfacce tra i componenti del sistema (i componenti possono essere progettati e sviluppati in momenti diversi).



- **Design del database:** strutture dati di sistema e come sono rappresentate in un DB
- **Design e selezione dei componenti:** ricerca di componenti riutilizzabili e relativa lista di cambiamenti da fare a questi, progettazione di nuovi componenti, modello uml

## Verifica e Validazione

L'obiettivo è dimostrare che il sistema è conforme alle specifiche e rispetti i requisiti, lo si fa con:

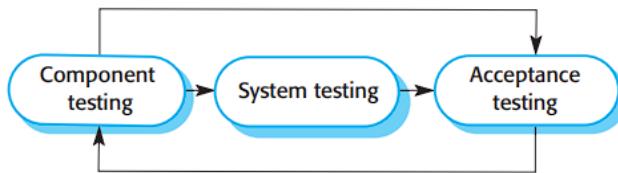
- **Checking:** si fanno ispezioni e verifiche, è un processo manuale.
- **Program Testing:** il sistema è eseguito con dati di test.

Testare il sistema comporta anche testare il sistema con casi di test derivati , **chiedere pag 42.**

Tra le **Fasi del Program Testing** troviamo:

- **Component testing:** ogni componente è testato indipendentemente, i test sono scritti dalla stessa persona che lo ha sviluppato.
- **System testing:** si testa il sistema completo, serve per vedere che il sistema soddisfi i requisiti funzionali e non.

- **Customer testing:** test eseguito dal cliente con dati realistici, può rivelare errori o omissioni nella definizione dei requisiti di sistema



## Evoluzione del software

Il software è intrinsecamente **flessibile** e può cambiare(a differenza dell'hardware) e come cambiano i requisiti del business anche il software che supporta tale business deve riuscire a stare al passo ed evolvere.

Per stare al passo con i cambiamenti bisogna essere in grado di:

- **Anticipare il cambiamento:** anticipare possibili cambiamenti prima che sia necessario un sostanziale carico di lavoro.
- **Tolleranza al cambiamento:** cercare di progettare il sistema in modo che eventuali cambiamenti possano essere introdotti con poche spese.
- **Prototipo:** è una versione iniziale del sistema usata per dimostrarne i concetti, avere un'idea del design e trovare eventuali problematiche, torna utile per non dovere stravolgere il sistema a causa di mal'interpretazioni con il cliente, inoltre durante la fase di **specificazione** aiuta nell'elicitazione e validazione dei requisiti mentre durante la **progettazione del sistema** è utile per esplorare varie soluzioni e sviluppare la Gui.

Una tecnica da poter adottare per rispondere meglio ai cambiamenti è la **Consegna Incrementale**. Si tratta di uno sviluppo del software dove alcuni degli incrementi sviluppati vengono consegnati al cliente, in maniera tale il cliente può sperimentare il sistema, ciò aiuta a chiarire i requisiti per i prossimi incrementi.

### Vantaggi

- Gli incrementi agiscono da prototipo per facilitare l'elicitazione dei requisiti per i prossimi incrementi.

### Svantaggi

- Molti sistemi richiedono un insieme di utilità di base utilizzate da varie parti del sistema, può essere quindi

- Minor rischio di fallimento del progetto
- I servizi di sistema con più priorità subiscono più testing
- Le specifiche sono sviluppate insieme al software, in alcuni casi è necessario avere già chiare tutte le specifiche in quanto parte di un contratto

*Per lo TIngia è l'unica  
possibile risposta alle  
richieste d'eme per lo sviluppo di  
prodotti innovativi*

→ Utilizza git hub

↳ nonostante non siano un principio principale dei metodi agili, gli strumenti sicuramente adattarsi ai loro cambiamenti aiuta nel miglioramento del processo

↳ o sprint

## Sviluppo Agile

Fa ovviamente parte dei processi agili.

Specifica, design e implementazione sono **interconnesse**, qui il sistema è sviluppato a incrementi con un attivo coinvolgimento degli stakeholders, ogni incremento dura tra le 2 e le 4 settimane e la documentazione è minima.

Sono inoltre utilizzati strumenti di supporto per automatizzare delle task, come ad esempio il testing automatizzato o la produzione automatica dell'interfaccia grafica.

## Manifesto Agile

*Dove viene posta l'attenzione*

*Dov'era posta prima*

Individuals and interactions	over	Process and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

## Principi Agili

- **Coinvolgimento del cliente:** il cliente deve essere fortemente incluso lungo tutto il processo di sviluppo, fornisce e da priorità a nuovi requisiti del sistema e valuta le iterazioni del sistema
- **Consegna incrementale:** il software è sviluppato in incrementi con il cliente che dice che requisiti devono esserci in ogni incremento
- **Persone e non processi:** ogni skill del team di sviluppo deve essere riconosciuto e sfruttato, ogni persona deve essere libera di procedere nello sviluppo a suo modo senza forzature dovute ai processi. → lo finge, permette a chiunque di raggiungere nuove opportunità, nuove informazioni e migliorare le proprie capacità. → imparare sempre cose nuove
- **Embrace change:** aspettarsi che i requisiti del sistema cambino, quindi progettare un sistema in maniera che questo sia pronto ad accogliere questi cambiamenti.
- **Mantenere la semplicità:** mantenere sviluppo e processi semplici e se possibile attivarsi per ridurre qualsiasi complessità incontrata.

## Applicabilità del metodo Agile

Bisogna che il prodotto sia di **piccole/medie dimensioni**, inoltre bisogna avere la disponibilità del cliente a partecipare al processo e poche regole esterne (vincoli esterni o regolamenti), lo sviluppo agile inoltre molte volte a braccetto con un **project management agile** (scrum).

Quando si estremizza lo sviluppo agile si può parlare di **extreme programming**, troviamo nuove versioni ogni giorno, incrementi ogni 2 settimane e i test sono eseguiti per ogni build, che ovviamente è accettata solo se ha superato i test.

## User Stories

I requisiti in questo tipo di sviluppo vengono raccolti in maniera informale, non abbiamo una stipulazione di un contratto con tutti i requisiti pre definiti prima dello sviluppo, si preferisce avere una raccolta dei requisiti tramite le **user stories**, che descrivono quello che l'utente necessita in un testo:

- Descrivere cosa l'utente può fare e come il sistema deve rispondere in base all'azione suona

#### 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.

## Task Cards

In base alla user story prodotta il team genera delle **task card**, dove per ognuna di queste card si capisce sforzi e risorse da impiegare.

Lo user ordina le task card per priorità con l'obiettivo di identificare feature importanti da sviluppare in due settimane.

L'unico problema può essere quello di capire se tutti gli aspetti fondamentali sono stati catturati

#### Task 1: Change dose of prescribed drug

#### Task 2: Formulary selection

#### Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, look up the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

## Sviluppo Test-Driven

È un altro tipo di sviluppo che rientra nella categoria dei processi agile.

I test sono scritti **prima** del codice del sistema, possono essere runnati **durante** lo sviluppo così da trovare i problemi già durante lo sviluppo in tale maniera da non continuare lo sviluppo fintantochè i test non vengono superati.

Sviluppo incrementale dei test in base agli scenari, c'è quindi una forte relazione tra requisiti e test, i test vengono scritti solo avendo una chiara visione delle specifiche.

L'utente viene coinvolto nello sviluppo dei test e nella validazione, test che viene automatizzato tramite l'utilizzo di framework e eseguito per ogni nuova funzionalità che non deve introdurre errori.

sviluppata

**Test 4: Dose checking**

**Input:**

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

**Tests:**

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose \* frequency is too high and too low.
4. Test for inputs where single dose \* frequency is in the permitted range.

**Output:**

OK or error message indicating that the dose is outside the safe range.

## Refactoring

Essendo nell'ambito agile, abbiamo uno sviluppo incrementale, ciò porta però i cambiamenti locali a degradare la struttura del software, inoltre questi cambiamenti con il passare del tempo diventano sempre più difficili da introdurre, questo perché più codice introduciamo e più possiamo andare incontro a fenomeni come duplicazione del codice o riutilizzo inappropriate del codice.

Per questo motivo non appena il team nota la possibilità di migliorare il codice, questo deve essere migliorato, questa attività si chiama **refactoring** e coinvolge operazioni come:

- Riconfigurazione delle gerarchie di una classe
- Rimozione del codice duplicato
- Rimpiazzo di porzioni di codice simili con chiamate a metodi
- Riordino e rinominazione di attributi e metodi

Tutto ciò migliora la struttura e aumenta la leggibilità ad esempio.

## Pair Programming

Ci sono due sviluppatori sullo stesso computer, le coppie sono create dinamicamente e tutte le persone del team arrivano a lavorare insieme.

### Vantaggi

- Proprietà e responsabilità collettiva.
- Refactoring è incoraggiato
- Review informale del codice
- Condivisione del sapere

↳ collaborazione tra junior e senior, migliora anche la capacità di lavorare in team per lo junior

## Project Management Agile

I software managers hanno la responsabilità di garantire che il software sia consegnato nei tempi e costi previsti.

Il project management agile si adatta allo sviluppo incrementale e alle pratiche utilizzate nei metodi agili.

### Scrum

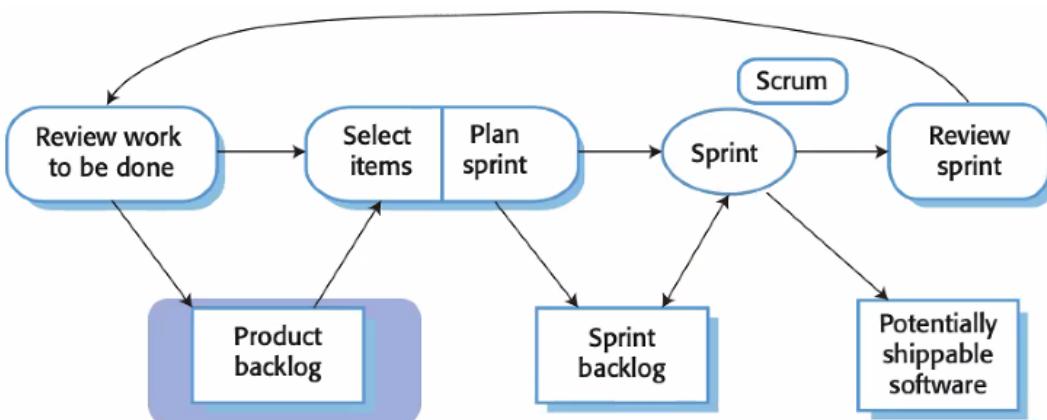
Forse la più nota metodologia, la gestione del progetto è portata avanti a iterazioni che richiamano gli incrementi dello sviluppo, le fasi sono le seguenti:

1. **Fase iniziale:** stabilire gli obiettivi generali del progetto e design dell'architettura del software
2. **Sprint:** ogni sprint sviluppa un incremento del sistema.
3. **Chiusura del progetto:** wrap up del progetto, si completa la documentazione e si stila cosa si è imparato da questo progetto.

### Terminologia dello Scrum

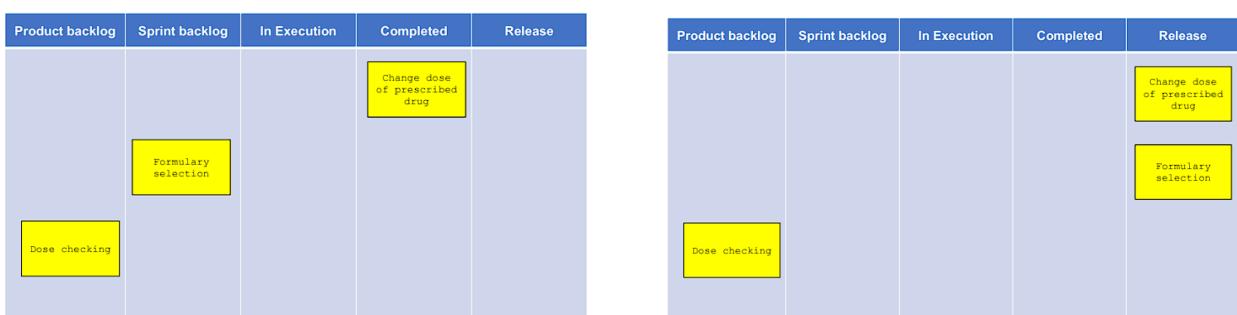
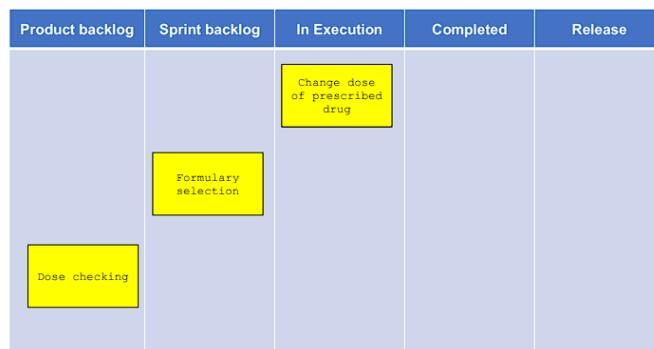
- team di sviluppo
- product owner: → *colui che comprende le necessità del customer e le traduce in obiettivi per il team Scrum*
- scrum: breve meeting giornaliero
- velocity: sforzo che compie il team nel product backlog durante uno sprint  
↳ *quantità di lavoro che il Team riesce a sostenere*  
↳ *In loTing usano gli sprint point che indicano b difficoltà di una task*
- product backlog: todo list
- sprint: 2-4 settimane
- scrum master: a capo dello scrum process

## Struttura di un ciclo di sprint



- Si parte dal **product backlog** dove si identificano feature, requisiti e miglioramenti, il backlog iniziale del prodotto può anche derivare da un documento di requisiti o user stories, inoltre il product owner decide quali funzionalità devono essere sviluppate durante lo sprint.
- Si passa allo **sprint backlog** è essenzialmente la todo list dello sprint, qui vengono messe le funzionalità più ad alta priorità scelte dal product owner, per capire quante di queste inserire nello sprint backlog si utilizza la velocity dei precedenti sprint così da non saturare troppo il team
- Inizia poi la fase di **sprint** che ha una durata prefissata tra le 2-4 settimane, non può essere estesa e il team si isola dal customer (le comunicazioni con questo le fa lo scrum master), in questa fase ci sono gli **scrum** dove si esaminano i progressi e si riorganizzano le priorità, ogni membro del team condivide informazioni, risalta problemi e dice cosa è pianificato per quel giorno.
- Si giunge poi al **review sprint** dove l'obiettivo è migliorare il processo, si capisce cosa è andato bene e cosa si poteva fare meglio così da migliorare per il prossimo sprint, le funzionalità implementate sono consegnabili e quelle non completate ritornano nel product backlog.

Il tutto viene mantenuto nella **Scrum Task Board**



## Benefici dello scrum

- Il prodotto è scomposto in un insieme di chunk gestibili e capibili e requisiti instabili non delayano lo sviluppo.
- L'intero team ha visibilità di tutto e di conseguenza la comunicazione è migliore
- I clienti vede le consegne puntuali(fiducia) e può dare anche feedback
- Si stabilisce fiducia tra cliente e sviluppatori.

*Scrum oggi  
Se lo scrum master non ha  
esperienza nei processi  
agili*

## Distributed Scrum

Un'estensione dello Scrum è quella dello **scrum distribuito**, lo scrum master deve essere nello stesso posto del team di sviluppo così che possa essere a contatto con le dinamiche giornaliere, il product owner deve fare visita agli sviluppatori così da generare una relazione di fiducia.

Inoltre è fondamentale una comunicazione real-time per comunicazioni informali come videocalls o instant messaging.

C'è una continua integrazione così i membri del team conoscono sempre lo stato del prodotto.

Ambiente di sviluppo comune per tutti i <sup>team</sup> ~~team~~ e videoconferenze tra il product owner e il team di sviluppo

## Scaling dei Metodi Agili

I metodi agili sono sicuramente ottimi per progetti medio-piccoli e sviluppati da un piccolo team, il successo ne deriva anche dalle comunicazioni migliorate.

Scalarlo per progetti più grandi, con più persone o con team distribuiti in varie località comporta delle modifiche.

## Scaling Up & Out

- **Scaling Up:** sviluppare software grandi che non possono essere sviluppati da team piccoli
- **Scaling out:** introdotto in grandi organizzazioni con molti anni di esperienza nel software development, in uno sviluppo che è molto più lungo temporalmente rispetto ai tempi che ci sono mediamente quando si usa scrum

In generale comunque in entrambi i casi dobbiamo mantenere quelle che sono le caratteristiche principali dei metodi agili quindi:

- planning flessibile
- continua integrazione
- buona comunicazione di team
- frequenti release
- test-driven development

# Agile e Manutenzione

Mantenere un software esistente è solitamente più costoso di sviluppare un nuovo software, i metodi agili dovrebbero supportare sia sviluppo che manutenzione, ci sono due problemi principali però:

1. Un sistema sviluppato con un approccio agile può essere difficile da manutenere a partire dalla scarna documentazione fino ad arrivare a casi in cui i membri originali del team abbandonano la posizione
2. Il cliente dovrebbe essere continuamente coinvolto anche nella manutenzione

# Agile e Plan Driven

Molti progetti in realtà includono sia elementi di approccio agile che elementi di approccio plan-driven, bisogna comunque ricordare che per scegliere al meglio cosa utilizzare tornano utili domande come:

- è importante avere una specifica dettagliata e un design prima di passare allo sviluppo? → **plan driven**
- la consegna incrementale può avere senso → **agile**
- quanto grande è il sistema?
  - sviluppabile con small team collocato → **agile**  
*comunicazione diretta  
rende più semplice il "cambiare"*
  - sviluppabile con big team distribuito → **plan-driven**  
*più semplice organizzarsi vista la ridotta possibilità  
di comunicazione*

## Tipi di problemi da considerare nella scelta del tipo di processo

### Problemi di sistema

- quanto grande è il sistema?
- che tipo di sistema bisogna sviluppare?
- qual è il lifetime previsto per il sistema?
- il sistema è soggetto a regolamenti esterni?

## Problemi di team

- che tecnologie di supporto sono disponibili?
- quanto bravi sono i designer e sviluppatori del team?
- com'è organizzato il team di sviluppo?

## Problemi di organizzazione

- riescono i rappresentati dei clienti a essere disponibili per fornire feedback dei vari incrementi?
- bo
- bo

## Metodi agili per sistemi grandi

Software su larga scala sono più difficili da capire e gestire rispetto a quelli piccoli.

Ci possono essere sistemi ~~disponibili~~ sviluppati da vari team, può essere inoltre più importante la fase di configurazione di un sistema che non quella di sviluppo.

Si possono avere vincoli regolatori esterni all'organizzazione e diversi stakeholder che non possono ovviamente essere tutti coinvolti.

## Agility at scale model(ASM)

Proposta di ibm per far scalare metodologie agile a grandi sistemi.

Agile utilizzato su fasi successive:

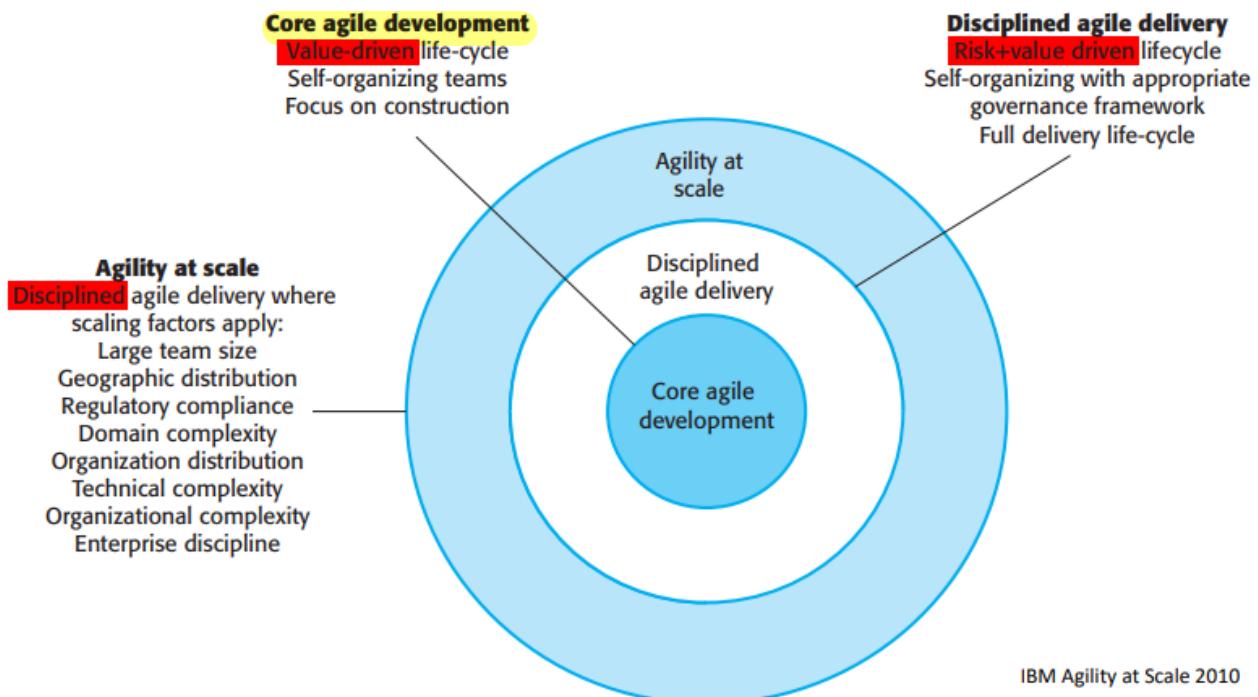
- sviluppo core: value driven
- disciplined agile delivery:
- agility at scale:

Modello di IBM

che propone come  
scalare i metodi  
agili

↪ noti per realtà  
piccole e medie  
↓  
diffusi anche su  
progetti più  
ampi

Sistema domani quando  
sei fresco



## Scaling Up to large systems

Un approccio completamente incrementale sull'ingegneria dei requisiti è impossibile, un'analisi preliminare è obbligatoria

Non può esserci un singolo product owner o un singolo rappresentante, diverse persone sono interessate in differente parti di sistema

Per lo sviluppo di grandi sistemi non è possibile concentrarsi solo sul codice del sistema ma anche su parti critiche e l'architettura

Deve esserci un meccanismo di comunicazione tra vari team e una continua integrazione è praticamente impossibile, tuttavia è essenziale mantenere compilazioni frequenti e regolari per fare la release del sistema.

Una soluzione tipicamente adottata è il **multi-team scrum**:

- i ruoli vengono replicati: ogni team ha un product owner e uno scrum master
- ogni team ha un product architect, tutti i product architect collaborano per progettare e evolvere l'architettura media del sistema
- release alignment: ossia le date della release del prodotto sono allineate per tutti i team in maniera da fornire un sistema completo e dimostrabile

- scrum of scrums: c'è uno scrum giornaliero degli scrum dove i rappresentati di ogni team si incontrano per discutere dei progressi e pianificare il lavoro da fare.

## Specifiche

### Ingegneria dei requisiti

Si tratta della descrizione delle funzionalità che un sistema dovrebbe fornire, inoltre comprende anche i vincoli sulle operazioni di quest'ultimo.

Parliamo di:

- **User requirements:** frasi in linguaggio naturale che esprimono il sistema voluto, sono molto astratte, dicono cosa il sistema dovrebbe fornire non come lo dovrebbe fare, ad esempio:
  - il sistema deve generare un report mensile del costo dei medicinali
- **System requirements:** Un documento strutturato dove si scende nel dettaglio, definiscono cosa e come fa il sistema per implementare le funzioni richieste e sono comunque capibili dal cliente.
  - il sistema deve generare un report nell'ultimo giorno del mese alle 17:00 che contiene costi dei medicinali per dosi e separati per ogni dose

### Stakeholder

Qualsiasi persona che in qualche maniera ha a che fare con il sistema o ha interesse in questo, possono esserne un esempio i pazienti(hanno info registrate), dottori(curano i pazienti), infermieri(si consultano con i dottori) e it staff (mantiene e installa il sistema).

### Requisiti funzionali e non funzionali

#### Requisiti funzionali

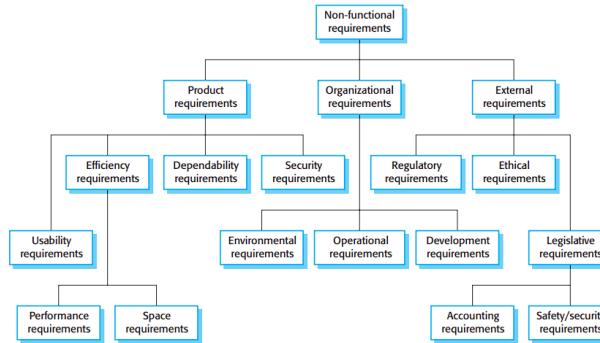
Sono per esempio funzionalità e servizi che il sistema dovrebbe fornire, ma anche come dovrebbe reagire il sistema ad un particolare input o come dovrebbe comportarsi in certe situazioni.

L'obiettivo è avere dei requisiti dove le seguenti proprietà sono soddisfatte, la **completezza** ossia tutti i servizi e info richieste dall'utente devono essere definiti e la **consistenza** ossia i requisiti non devono essere contraddittori.

Fare ciò però non è molto facile da far succedere perchè solitamente gli stakeholders tendono ad avere necessità diverse e succede magari che alcune inconsistenze vengono scovate solo dopo una profonda analisi durante lo sviluppo.

## Requisiti non funzionali

Vincoli sui servizi e funzioni che offre il sistema, spesso addirittura si applicano sul sistema intero piuttosto che nelle singole funzionalità, ad esempio il tempo di risposta del sistema o l'idea da usare per lo sviluppo piuttosto che il linguaggio di programmazione utilizzato.



Un singolo requisito non funzionale può generare più requisiti funzionali

Ci sono vari tipi di requisiti non funzionali:

- **product requirements:** specificano che il prodotto deve comportarsi un una certa maniera
- **organizational requirements:** requisiti derivati dalle policies delle organizzazioni del cliente e dello sviluppatore
- **external requirements:** derivati da fattori esterni, possono essere requisiti regolatori che devono essere comprovati da un'entità regolatrice o requisiti legislativi/etici

#### 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.

#### Organizational requirement

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

#### External requirement

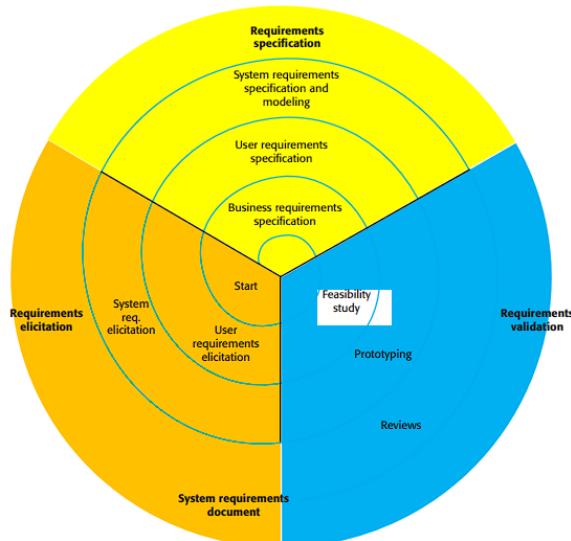
- The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Quando è possibile è meglio scrivere i requisiti non funzionali **quantitativamente** in maniera che siano concretamente testabili

Requisiti quantitativi: si riferiscono alle caratteristiche non funzionali del sistema come l'efficienza, l'affidabilità e l'usabilità; ad esempio la facilità d'uso.  
Requisiti quantitativi: possono essere misurati in modo oggettivo, sono dunque meno ambigui e più facili da implementare.

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

## Processi dell'ingegneria dei requisiti



## Elicitazione dei requisiti

L'obiettivo è quello di capire il lavoro degli stakeholders e come potrebbero usare il sistema per aiutarli in quel lavoro, gli stakeholders e i sw eng lavorano insieme per capire il dominio dell'applicazione, i servizi che dovrebbe fornire, la performance e i vincoli hw ec.

### Ostacoli nel cercare di fare ciò sono:

- requisiti espressi potrebbero essere surreal<sup>o volte nel gergo del proprio lavoro</sup>
- lo stakeholder esprime i requisiti in una terminologia poco chiara
- parlando con diversi stakeholder potrei avere obiettivi contrastanti
- fattori organizzativi o politici che influenzano i requisiti (manager può esprimere in maniera da trarne poi vantaggio)
- i requisiti potrebbero cambiare durante il processo di analisi

Tra i modi per fare l'elicitazione dei requisiti troviamo:

### INTERVISE

Ci sono vari tipi di interviste, **chiuse** ossia basate su domande predefinite e **aperte** se si esplorano i vari problemi con gli stakeholders cercando di trarre informazioni da ciò senza cose predefinite, alternativamente troviamo le **mix** dove ci sono domande che aprono nuovi problemi ma discussi in maniera meno strutturata rispetto alle chiuse.

### Problemi:

- termini e concetti del dominio dello stakeholder non capibili dagli ingegneri
- concetti familiari allo stakeholder che questo non esprime neanche

Per affrontare questi problemi si può utilizzare ad esempio una **springboard question** dove si fanno proposte di requisiti piuttosto che dire "dimmi cosa vuoi".

### ANALISI ETNOGRAFICA

Cerca di capire i problemi sociali e organizzativi che affligono l'utilizzo del sistema.

È una tecnica osservazionale capisce i processi operazionali e aiuta a derivare requisiti per supportare questi processi.

Un'analista si immerge nell'ambiente di lavoro dove il sistema sarà utilizzato, fare ciò aiuta anche a scovare **requisiti impliciti** come ad esempio la reale maniera in cui le persone lavorano, piuttosto che rifarsi agli standard dell'azienda.

L'etnografia può essere utile per capire processi esistenti ma non per identificare nuove features da aggiungere al sistema.

## STORIE E SCENARI

È più semplice affacciarsi ad esempi di real-life rispetto a una descrizione astratta, lo stakeholders ha la possibilità di commentare rispetto a una storia/scenario di utilizzo del sistema.

### Storia

Testo narrativo, si fa una descrizione ad alto livello e generiche

**Photo sharing in the classroom**  
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 SCARAN (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.

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.

### Scenario

Strutturati e specifici (input/output) e sono un esempio di interazione tra utenti

**Uploading photos to KidstakePics**  
**Initial requirement:** A user or a group of users have one or more digital photographs to be uploaded to KidstakePics.  
A description of the normal flow of events in the scenario  
**Normal:** The user chooses to upload photos to KidstakePics. A description of what the system and users expect when the scenario starts  
the computer and to select the project name  
also be given the option of inputting keywords that should be associated with each uploaded photo.  
Uploaded photos are named by creating a conjunction of the user name with the filename of the photo.  
A description of what can go wrong and how resulting problems can be handled  
automatically sends an email to the project moderator, generates an on-screen message to the user that this  
**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.  
Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wants to replace the file, rename the photos, or cancel the upload.  
Information about other activities that might be going on at the same time  
**Other activities:** The moderator may log in during the scenario ends  
**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.

## Specifiche dei requisiti

Il processo di scrittura dei requisiti in un **documento dei requisiti**, qui gli **user requirements** devono essere capibili dagli stakeholders mentre i **system requirements** possono contenere più tecnicità, i requisiti possono far parte di un contratto quindi devono essere più completi possibile.

forse aggiungere ultime righe

## SPECIFICAZIONE DEI REQUISITI IN LINGUAGGIO NATURALE

Il linguaggio naturale per quanto espressivo e intuitivo sia può essere altrettanto ambiguo, è quindi opportuno usare delle linee guida

Maniera strutturata, usi il linguaggio naturale seguendo uno standard  
 ↳ tabelle  
 ↳ usecases

#### Guidelines:

- Invent a standard format and use it for all requirements
- Use language in a consistent way: Use "shall" for mandatory requirements, "should" for desirable requirements.
- Use text highlighting to identify key parts of the requirement (bold, underline, italics) → bold
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary
  - You know whom to consult if the requirement has to be changed

dove

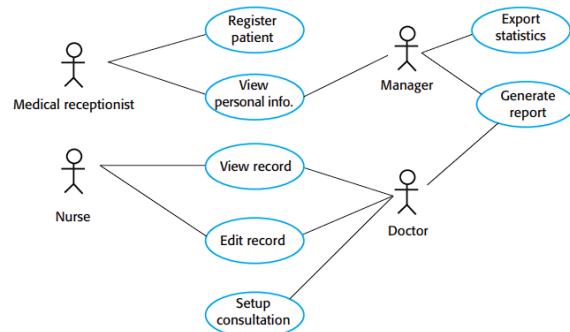
## SPECIFICAZIONE DEI REQUISITI STRUTTURATA

Bisogna impostare un limite nella libertà dello scrittore dei requisiti, devono essere scritti con un certo **standard**

## Use Cases

Sono inclusi nell'uml, identificano attori e interazioni.

Descrivono le operazioni che gli utenti possono svolgere con il sistema



## DOCUMENTO DEI REQUISITI DEL SOFTWARE

Contiene ufficialmente quali devono essere i requisiti del sistema, quindi requisiti di sistema e di utente, non è un documento di design, quindi dice **cosa e non come** fa qualcosa il sistema.

Sistemi sviluppati incrementalmente avranno in generale meno requisiti specificati, ci sono degli standard da rispettare anche qui(glossario, appendice, introduzione,...)

Gli utenti di questi requisiti sono:

- clienti:** specifica e legge i requisiti scritti
- manager:** usa il documento per fare una pianificazione del processo di sviluppo

- **ingegneri del sistema:** capiscono cosa sviluppare
- **ingegneri del test:** sviluppano i validation test
- **ingegneri della manutenzione** *Capiscono come mantenere il sistema*

## Validazione dei requisiti

Si controlla che i requisiti definiscano il sistema che il customer ha chiesto, che siano realistici e che non abbiano errori, in quanto correzioni future relative a un requisito errato potrebbero costare molto

## Cambiamento dei requisiti

Nei grandi sistemi software i requisiti cambiano continuamente, i problemi sono intrinsecamente difficili da definire completamente, possono cambiare le tecnologie, cambiamenti di priorità.

## Management dei requisiti

- **Identificazione dei requisiti:** identificare univocamente ogni requisito così che possa essere *grossamente riferenziato* con altri requisiti *collegati*
- **Processo di gestione del cambiamento:** insieme di attività che stimano l'impatto e il costo dei cambiamenti
- **Policies di tracciabilità:** relazioni tra ogni requisito e tra i requisiti di sistema e il system design
- **Tool support:** strumenti di supporto vari(database, excel, ...)

## Processo di cambiamento dei requisiti

- **Analisi del problema e specificazione del cambiamento:** si valuta se il problema che comporterebbe il cambiamento è valido o meno, ed eventuali revisioni di chi chiede il cambiamento.
- **Analisi del cambiamento e costi:** effetti e costi dell'eventuale cambiamento, operazioni di tracciabilità per stimare il numero di cambiamenti
- **Implementazione del cambiamento:** sono modificati il documento dei requisiti ed eventualmente il system design.

## Tecnical offer

Documento sviluppato in bilingue in cui dopo aver capito l'obiettivo del cliente, definito un possibile design del sistema, decisi i passi per fare ciò, il Solution Architect e il system designer fanno una proposta al cliente.

# Design e Implementazione

## Architectural Design

L'obiettivo del design architetturale è capire come il software deve essere organizzato, progettare la struttura generale e identificare i componenti strutturali principali di un sistema e le loro relazioni.



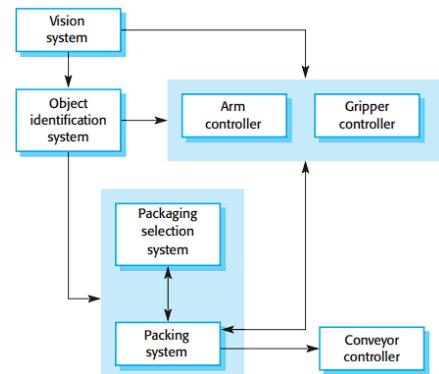
La domanda che potremmo porci è: come mai voglio esplicitare un'architettura?

Le risposte sono svariate tra cui usarla come rappresentazione ad alto livello comunicando con gli stakeholders, aiuta inoltre a fare analisi del sistema dato che certe decisioni sull'architettura potrebbero influenzare requisiti non funzionali del sistema come performance e affidabilità.

Infine può essere usata anche per motivi di riutilizzo tra vari sistemi.

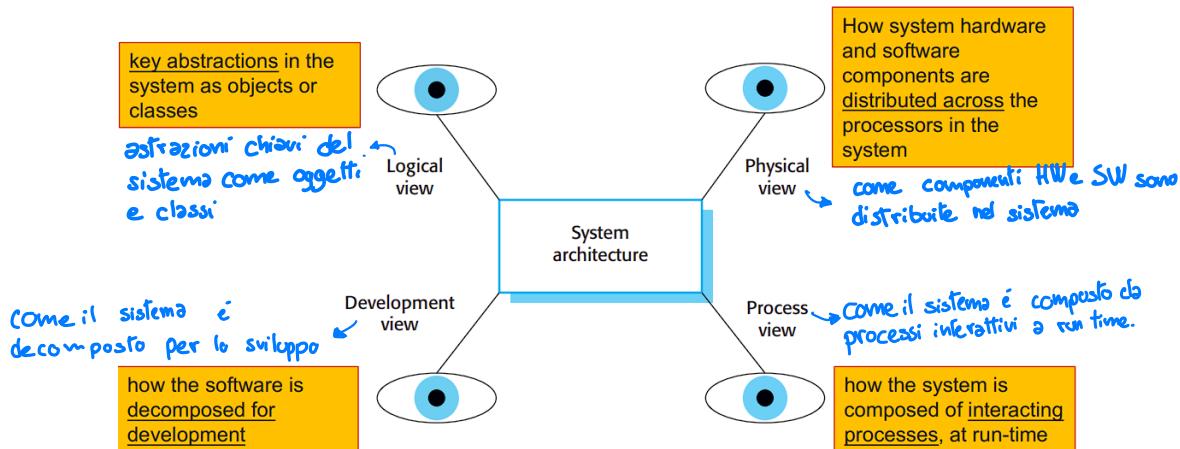
La rappresentazione è informale con diagrammi a blocchi che mostrano relazioni tra entità (spesso criticati perché mancano di significato anche per la loro marcata astrazione).

Tra gli utilizzi invece ritroviamo una semplificazione riguardo le discussioni sulla progettazione del sistema e una documentazione dell'architettura (mostra componenti le loro connessioni e interfacce).



## Viste Architetturali

Ci sono varie viste che possiamo dare di una stessa architettura, che forniscono prospettive diverse di quest'ultima:

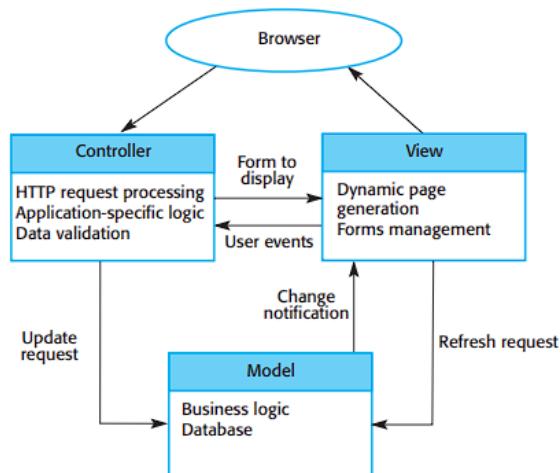


## Pattern Architetturali

### Model-View-Controller(MVC)

- **Model**: responsabile di gestire lo stato del sistema
- **View**: responsabile del render UI
- **Controller**: responsabile della gestione degli eventi dalla UI

### Esempio di MVC



### Quando uso MC?

Usato quando ci sono molti modi di vedere e interagire con gli stessi dati o quando il modo in cui le viste interagiscono con i dati è soggetto a evoluzione

## Vantaggi

- permette ai dati di cambiare indipendentemente dalla loro rappresentazione
- supporta presentazioni degli stessi dati in maniere diverse
- i cambiamenti nella logica dei dati sono immediatamente disponibili a tutte le viste

## Svantaggi

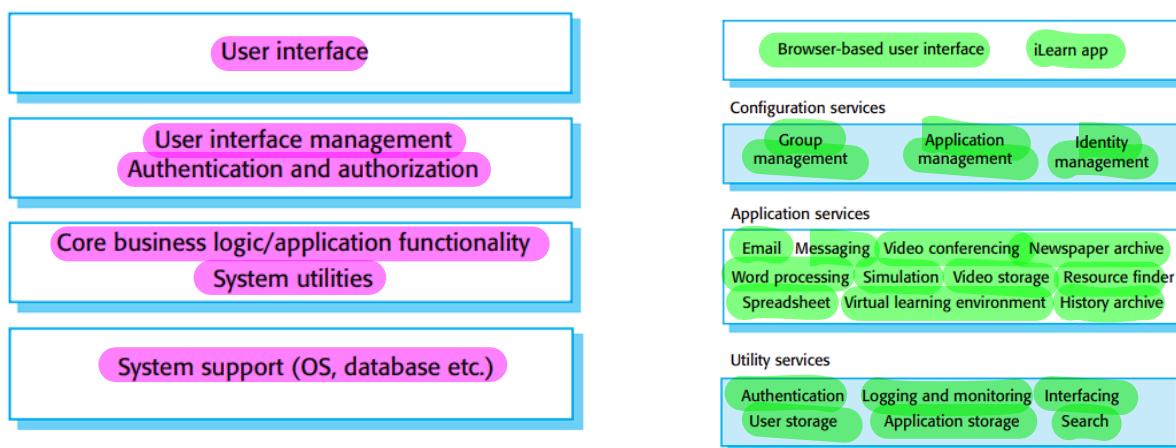
Tra gli svantaggi abbiamo che c'è una complessità del codice aggiuntiva nel momento in cui il modello dei dati e le interazioni con questi sono semplici.

## Architettura Stratificata

Ogni funzionalità è in uno strato diverso, ogni livello si appoggia al livello inferiore.

Combacia bene con lo sviluppo incrementale, è portabile.

## Esempio



## Quando la uso?

Quando sviluppiamo software sopra sistemi già esistenti, quando lo sviluppo è diviso tra vari team (uno strato a testa) o se ci sono requisiti di sicurezza multilivello.

## Vantaggi

- Permette di rimpiazzare un livello fintanto che l'interfaccia di comunicazione tra livelli adiacenti è rispettata.

- Si possono introdurre ridondanze tra i vari layer in maniera da incrementare l'affidabilità del sistema.

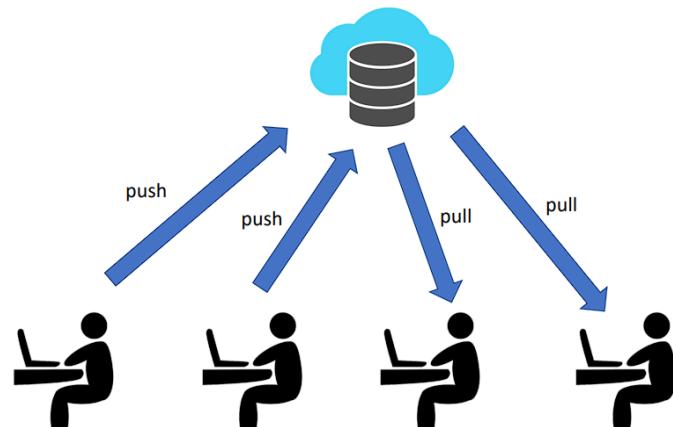
### Svantaggi

- Nella pratica una separazione così netta e pulita dei livelli non è banale da raggiungere
- Possono esserci problemi di performance perché si può arrivare a processare molti livelli (da alto livello che richiede magari passo per tanti altri livelli sotto).

## Architettura Repository

I dati condivisi sono mantenuti in un database centrale(repository) e possono essere acceduti da vari sottosistemi, tra questi però non c'è comunicazione diretta.

### Esempio



### Quando lo uso?

Quando ho grandi volumi di dati generati che devono essere mantenuti per molto tempo o sistemi data driven dove l'inclusione di dati nella repository triggerà azioni.

### Vantaggi

I componenti sono indipendenti, inoltre i cambiamenti fatti su una risorsa da un componente sono immediatamente disponibili a tutti gli altri perché propagati tramite la repository e infine i dati possono essere gestiti consistentemente dato che sono tutti nello stesso posto.

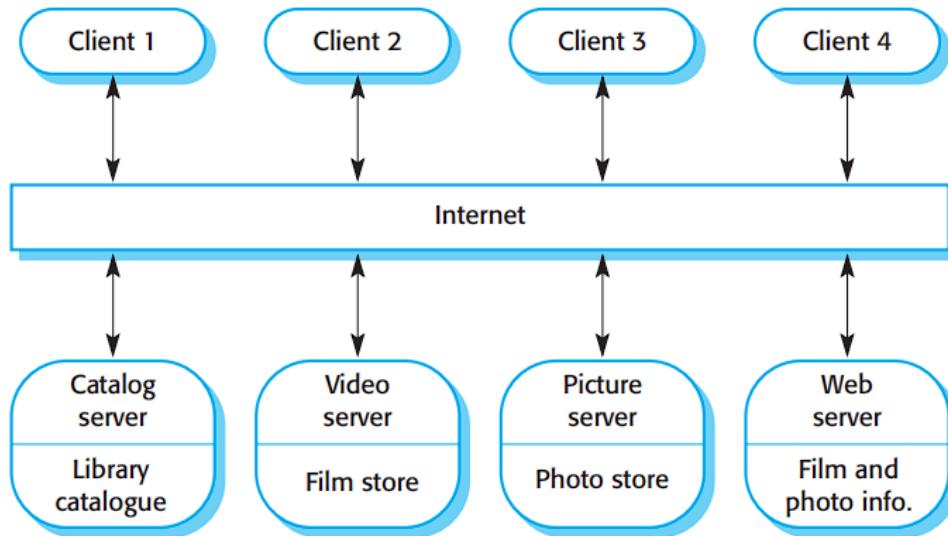
### Svantaggi

- C'è un single point of failure, ossia se salta il componente centrale, salta tutto il sistema.  
↳ più facile subire attacchi DDoS
- possibile inefficienza: tutte le comunicazioni mediate dalla repository
- distribuire la repository tra vari computer può risultare difficile.

## Architettura Client-Server

Insieme di server stand-alone che forniscono servizi specifici e insieme di client che richiedono questi ultimi ai server, inoltre una rete che collega client con server.

### Esempio di Client-Server



### Quando lo uso?

Quando i dati condivisi in un database devono essere acceduti da più posizioni e dato che i server sono replicabili per fare load-balancing.

### Vantaggi

- I server possono essere distribuiti sulla rete
- Funzionalità generali sono disponibili a tutti i client senza che siano implementate da tutti i server

### Svantaggi

- ogni servizio è un single point of failure

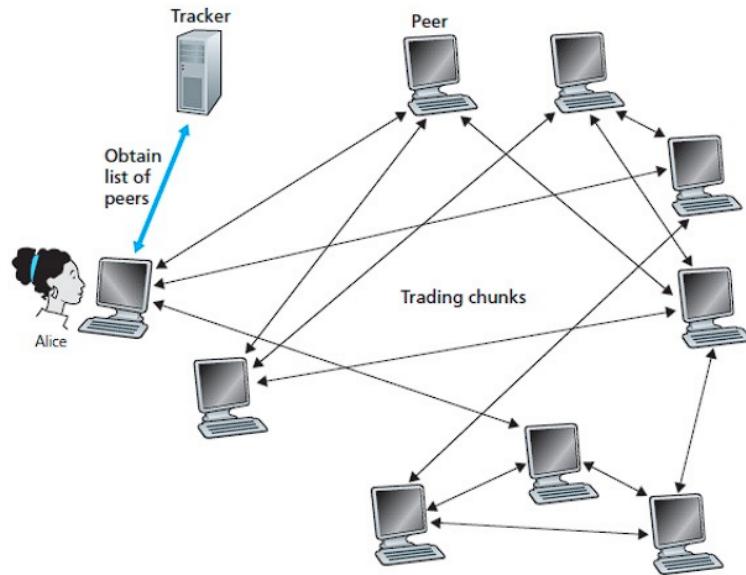
- la performance dipende dalla rete
- problemi di management quando i server sono posseduti da differenti organizzazioni

## Peer to Peer

Ogni componente fa contemporaneamente sia da client che da server, difatti fornisce un'interfaccia che specifica servizi offerti e servizi richiesti, se un peer dovesse aver bisogno di dati comunica con gli altri peer per trovare chi glieli fornisce.

P2P scala molto bene ed è fault tollerant, i dati difatti possono essere replicati tra vari peers.

### Esempio di Peer to Peer

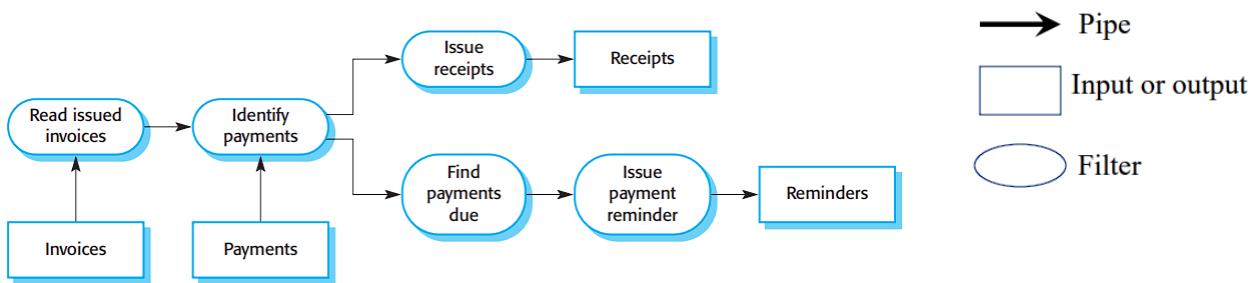


## Pipe & Filter

Qui i dati vengono elaborati per step successivi, ad esempio prima generiamo i dati, poi li filtriemo, in seguito li analizziamo e infine li rappresentiamo.

È molto appropriata quando le elaborazioni dei dati sono fatte in batch, difatti è molto usato in sistemi di data processing, mentre non è assolutamente appropriato per sistemi interattivi.

### Esempio di Pipe & Filter



## Quando la usiamo

La usiamo in applicazioni di processamento dei dati e quando gli input sono processati in fasi diverse per fornire output in relazione (output precedente è input del successivo).

### Vantaggi

- È facile da capire e supporta il riuso delle varie trasformazioni
- Il workflow è simile alla struttura di tanti processi business
- L'evoluzione aggiungendo trasformazioni è diretta
- Può essere implementato come un sistema sequenziale o concorrente

### Svantaggi

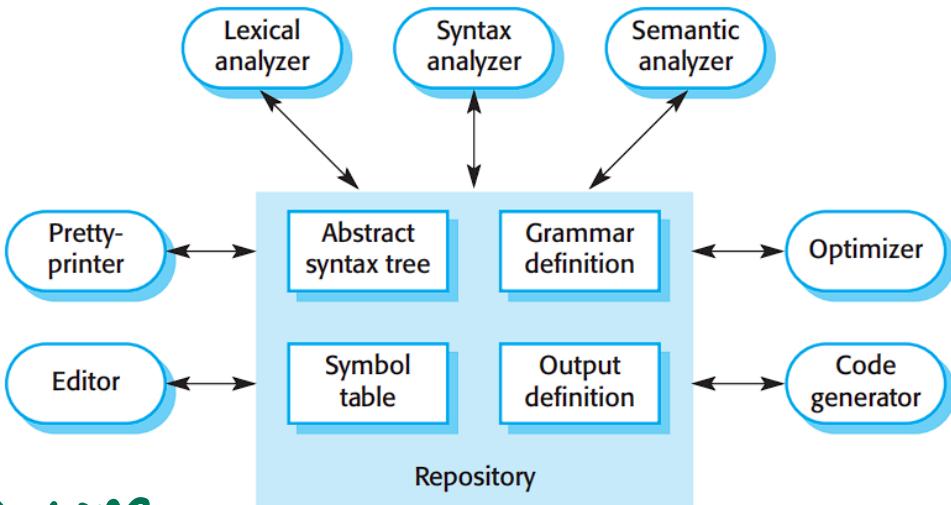
- Il formato dei dati deve essere comune a tutti gli step di trasformazione
- Ogni trasformazione deve fare parsing dei dati come concordato, ciò incrementa l'overhead.

## Architetture Eterogenee

Ovviamente si possono combinare tra di loro i vari pattern architetturali se vantaggioso.

Li si può organizzare in una composizione gerarchica, ossia ad esempio ho un sistema pipe & filter che ha un sottosistema organizzato a livelli.

Nelle architetture eterogenee inoltre un componente può avere due ruoli diversi in due pattern distinti, ad esempio un sottosistema accede ai dati tramite una repository ma comunica con gli altri componenti tramite una pipe.



## Validazione

### Program Testing

→ *Garantire che le nuove implementazioni funzionino*  
*Garantire che non interferiscono con le vecchie*

I test mi permettono di non testare a mano

Gli obiettivi del **testing** sono dimostrare che il programma fa quello che dovrebbe fare (ossia ciò che ha richiesto il customer) e trovarne eventuali difetti.

Per dimostrare che il sistema rispetta i requisiti facciamo:

- almeno un test per ogni requisito.
- almeno un test per ogni feature principale del sistema e combinazioni di queste.

Quando parliamo di testing abbiamo il **sut(system under test)**, ossia il sistema che stiamo verificando, al quale passiamo input e ci da output, che viene poi confrontato con quello atteso e da un esito del test (pass/fail), ricordandoci che il testing può solo mostrare la presenza di errori non la loro assenza.

## Confidenza

Il livello di confidenza che ho rispetto al fatto che il programma faccia ciò che deve dipende da:

- **software purpose**: quanto critico per un'organizzazione è il sistema
- **user expectation**: nel senso che le aspettazioni possono essere anche basse e quindi errori più tollerati.
- **marketing environment**: arrivare prima sul mercato potrebbe risultare più importante di trovare difetti.

## Inspection/review vs Testing

Le ispezioni o review, a differenza del testing sono statiche, ossia il codice non viene eseguito, tra i vantaggi che troviamo:

- gli errori non sono mascherati
- può essere eseguita su codice non completo
- può rinforzare altri attributi quali: inefficienze, qualità della programmazione.

## Fasi del testing

Ci sono 3 fasi principali:

1. **Development testing:** il sistema è testato dagli sviluppatori
2. **Release testing:** un team separato esegue un test della versione completa del sistema prima della release.
3. **User testing:** gli user del sistema o membri interni non esperti testano il sistema.

## Development Testing

Tipicamente fatto dal team che sviluppa il sistema, l'obiettivo è trovare bug nel software, operazione che quindi va a braccetto con il debugging ossia il processo di trovare problemi nel codice e sistemerli.

Durante la fase di development testing troviamo 3 stage:

- **unit testing:** i componenti vengono testati singolarmente.
- **component testing:** si uniscono vari componenti e si testa l'unione.
- **system testing:** si testa il sistema completo

→ Test di funzionalità  
permette agli sviluppatori di  
coprire se c'è stato raggiunto  
il Definition of Done

### Unit testing

Nello **unit testing** individuiamo le unità come **funzioni, classi, componenti con interfacce** per accedere alle loro funzionalità.

Se volessimo

**coprire tutti i casi di test di una classe** dovremmo testare tutte le operazioni associate a tale oggetto, settare e interrogare tutti gli attributi e provare l'oggetto in tutti i possibili stati, è spesso difficile coprire tutti questi casi.

Lo unit test è strutturato in questa maniera:

- **parte di setup:** il sistema viene inizializzato e messo in condizioni da essere testabile.
- **parte di chiamata:** le funzionalità sotto test vengono stressate.
- **parte assertiva:** risultati confrontati con risultati attesi

Se possibile lo unit test deve essere **automatizzato** così che tutti i test siano eseguiti senza bisogno di interventi manuali.

A volte capita che ci siano delle dipendenze che non sono ancora state implementate, come ad esempio un database non ancora disponibile.

Il **mock** è un'unità con la stessa interfaccia dell'unità mancante che può essere utilizzata per simulare l'effettiva funzionalità che poi ci sarà in release, nell'esempio del database potrebbe essere un database hardcoded con dati costanti.

Ci aspettiamo di testare scenari che riflettono operazioni normali del sistema o scenari che siamo a conoscenza sollevare problemi comuni(input anomali o output sbagliati/crash).

Dato che non possiamo testare un programma con tutti i possibili input facciamo **input partitioning**, ossia creiamo gruppi di input che hanno **caratteristiche simili** e che quindi dovrebbero essere processati nella stessa maniera, a questo punto per ogni gruppo prendiamo un **rappresentante** della classe e con questo facciamo il test.

Come rappresentanti possiamo considerare **valori centrali** della partizione(tipico) o **valori ai limiti** della partizione(atipici).

Tra le linee guida del testing troviamo che: bisogna scegliere input che generino tutti i messaggi d'errore, buffer overflow, ripetere lo stesso input più di una volta, forzare output non validi, forzare risultati molto grandi o piccoli.

Se invece dovessimo operare con delle sequenze come array, proviamo varie sequenze, prendiamo il primo, il centrale e l'ultimo elemento e proviamo anche sequenze di 0 o 1 elementi.

## Component Testing

Si uniscono più componenti assieme per testarne le interfacce. L'obiettivo è scopare problemi dovuti a errori nelle interfacce.

Ci sono vari tipi di interfacce: **da ascoltare**.

- **Parameter interface:** dati passati da un metodo a un altro
- **Shared Memory interface:** blocchi di memoria condivisi tra procedure/funzioni
- **Procedural interface:** sottosistemi encapsulano set di procedure per chiamarle da altri.
- **Message passing interface:** sottosistemi richiedono servizi ad altri sottosistemi

Tra gli errori comuni delle interfacce troviamo:

- **uso improprio dell'interfaccia:** parametri di tipo sbagliato o in ordine sbagliato.
- **faintendimeno dell'interfaccia:** un componente chiamante fa presupposti sul comportamento del componente chiamato che sono errati. **Ad esempio**, passare una lista da cercare che dovrebbe essere ordinata, ma non lo è.
- **errori di timing:** il chiamato e chiamante operano con tempi diversi (leggere un messaggio prima che sia pronto).

## Linee guida per il testing delle interfacce

Chiamare procedure con valori di parametri agli estremi assumibili, provare con puntatori nulli, test che fanno fallire i componenti, stress testing in sistemi di passaggio di messaggi e in sistemi di memoria condivisa variare l'ordine in cui i componenti sono attivati.

## System Testing

Lo scopo è testare le interazioni tra componenti in maniera da controllare che i componenti siano compatibili, interagiscono correttamente e trasferiscono i giusti dati con giusti tempi tra le loro interfacce.

### System testing basato su use cases

I casi d'uso identificano interazioni e possono essere utilizzati come base per il test, ogni use case coinvolge vari componenti, quindi testare i casi d'uso forza le interazioni tra questi.

Il diagramma di sequenza associato ai casi d'uso documenta i componenti e le interazioni che stanno venendo testati, i test dovrebbero anche considerare le eccezioni e verificare che siano correttamente gestite.

### Testing policies

Testare tutte le possibili esecuzioni di un sistema è impossibile, si possono però utilizzare delle **policies** per considerare il testing adeguato, esempi di policies sono:

- tutte le istruzioni del programma devono essere eseguite da almeno un test
- tutte le funzioni accessibili tramite un menu dovrebbero essere testate così come dovrebbero esserlo tutte le combinazioni di funzioni accessibili dal menu.
- se sono forniti input dall'utente, tutte le funzioni devono essere testate con input corretti o errati.

### Test Driven Development

Introdotto con lo sviluppo agile, quello che succede:

- si identifica la nuova feature incrementale da implementare
- si scrivono i casi di test per la feature
- si eseguono i test che falliscono
- si implementa la feature
- quando tutti i test passano allora la feature viene rilasciata.

L'obiettivo di fare ciò è capire cosa fa e cosa è la nuova feature ancora prima di implementarla.

### Vantaggi

- **code coverage:** c'è almeno un caso di test per ogni segmento di codice
- **regression test:** è sempre possibile (automaticamente) eseguire tutti i test cases, così tutte le feature esistenti vengono testate ogni volta che una nuova feature è aggiunta
- **simplified debugging:** un test che fallisce si riferisce a una sola feature
- **system documentation:** i test possono essere utilizzati per capire le feature del sistema.

so da dove  
può venire  
l'errore

## Release Testing

È la seconda fase del program testing, solitamente è responsabilità di un altro team rispetto a quello di sviluppo(che invece cerca bug e difetti), controlla se sono stati mantenuti i requisiti, solitamente è un processo **black-box** ossia conosco input-output ma non di più

Requirements:	Tests:
If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a <u>warning</u> 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 <u>ignore</u> an allergy warning, they shall provide a <u>reason</u> why this has been ignored	2. Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the <u>warning</u> is issued by the system. 3. Set up a patient record in which <u>allergies to two or more</u> drugs are recorded. Prescribe both of these drugs <u>separately</u> and check that the <u>correct warning</u> for each drug is issued. 4. Prescribe <u>two drugs</u> 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.

## Scenario testing

Lo scenario è una storia che descrive un modo in cui il sistema potrebbe essere utilizzato, un testing degli scenari quindi può includere vari requisiti e in caso appunto scenari e user stories siano disponibili dalla fase di ingegneria dei requisiti possono essere direttamente utilizzati come scenari di test

## Performance Testing

L'obiettivo è quello di dimostrare che il sistema può processare il carico di lavoro per lui pensato. Ad esempio testiamo il mix di lavori che devono essere gestiti dai lavori come 90% transazioni di tipo A, 5% tipo B e il resto C,D,E.

Si procede con dei test in serie aumentando il carico di lavoro fino a quando il sistema non cede.

Se invece parliamo di stress testing, siamo in un contesto in cui il sistema è volutamente sovraccaricato per capire il suo comportamento quando fallisce.

## User Testing

Le influenze provenienti dall'environment dello user possono impattare su affidabilità, performance, usabilità e robustezza del sistema.

Ci sono diversi tipi di user testing tra cui troviamo:

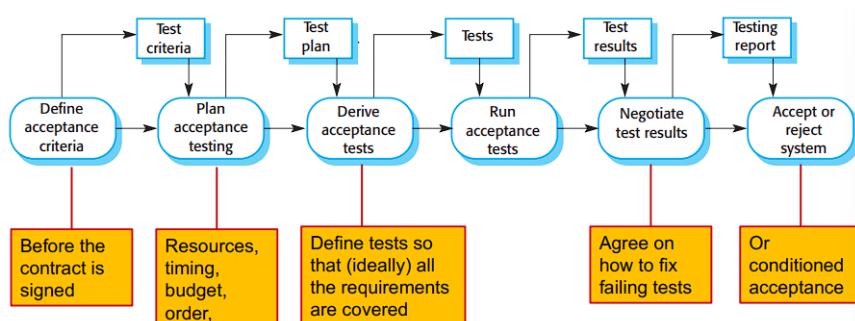
- **alpha testing:** pochi user provano il software lavorando con il team di sviluppo presso il centro del team di sviluppo
- **beta testing:** un gruppo largo di utenti prova il sistema
- **acceptance testing:** il customer prova il sistema per decidere se è pronto per essere accettato e messo in funzione nell'ambiente del customer

### Acceptance testing process

Accettare il sistema implica l'avvenire del pagamento finale per il software, il processo di accettazione è così strutturato:

Si definiscono i criteri, che se soddisfatti, comportano l'accettazione del sistema, si definisce poi il piano di accettazione quindi risorse, tempi, budget. Si prosegue quindi derivando i casi di test che verranno eseguiti e si eseguono questi test, i risultati poi vengono negoziati, se ci sono test falliti si capisce come sistemare questa condizione di incorrettezza, per finire il sistema viene o meno accettato e di conseguenza pagato o non pagato.

- 1 Definisci i criteri  
2 Pianifichi i test  
3 Derivi i test  
4 Lanci i test  
5 Negozio (Non obbl.)  
6 Accetti o meno



### Metodi agili nell'acceptance testing

Nei metodi agili il customer è parte del team di sviluppo ed è responsabile di prendere decisioni per l'accettabilità del sistema, i test sono definiti dal customer e integrati con altri test e vengono runnati automaticamente quando sono fatti cambiamenti. Non c'è una fase di acceptance testing separata, il problema principale **finire**

# Evoluzione

## Refactoring



Una modifica della struttura interna del software per renderlo più facile da capire e più economico da modificare senza cambiarne il suo comportamento osservabile. - Martin Fowler.

*del software dal punto di*

*qualitativo e della manutenzione*

Quindi quando parliamo di refactoring parliamo del processo in cui facciamo miglioramenti ad un programma per rallentare il degrado pian piano che ci facciamo dei cambiamenti. Possiamo vederlo anche come una manutenzione preventiva, quando facciamo refactoring miglioriamo la struttura per ridurne la complessità o renderla più facile da capire, ovviamente ci si concentra sul migliorare ciò che esiste e non introduce nuove cose.

Facciamo il refactor quindi per:

- limitare il decadimento della qualità del codice
- migliorare leggibilità
- semplificare il codice
- clean-up del codice esistente
- semplificare testing
- semplificare manutenzione

Bisogna prestare attenzione alla differenza che c'è tra **reengineering** e **refactoring**:

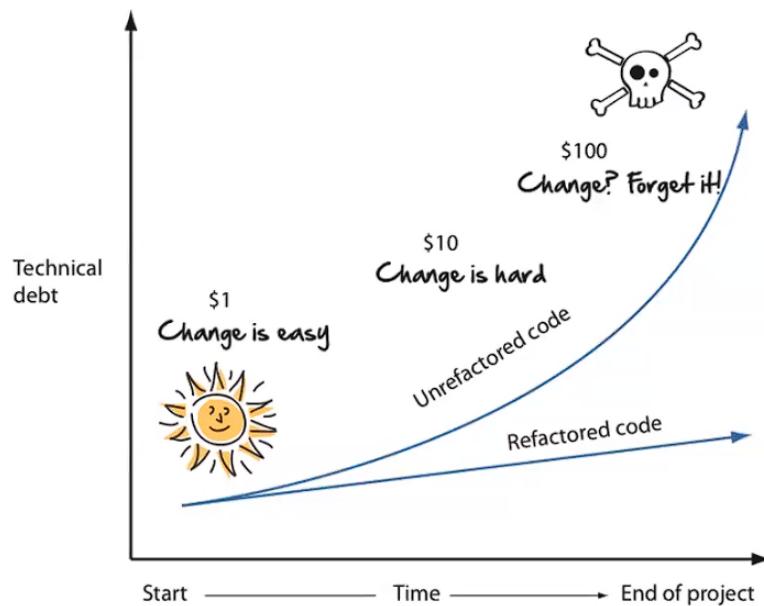
*viene modificato il sistema*

- **Reengineering:** prende luogo dopo che un sistema è stato manutenuto per qualche tempo e con i costi di manutenzione in aumento.
  - si usano tool automatici e un sistema di re-engineer per creare un nuovo sistema più manutenibile
- **Refactoring:** processo continuo di miglioramento durante lo sviluppo e evoluzione del sistema

Il **debito tecnico**(technical debt) sono tutti quei problemi di comprensibilità del codice che sono disposti ad avere che però probabilmente causeranno problemi nel futuro(collega deve mettere mano sul mio codice e non capisce), se quindi ad esempio

uso identificatori non efficaci sto causando un problema nel futuro e accumulando debito che pagherò nel futuro.

Se non uso il refactor quindi accumulo molto debito tecnico arrivando ad un punto in cui sistemare questi problemi costerà troppo, mentre con il refactoring riesco ad attenuare questo debito.



Per quanto riguarda quindi il **quando** applicare il refactoring, durante lo sviluppo in generale più spesso possibile, in casi più specifici però lo applichiamo:

- prima di aggiungere una nuova feature(più facile su un codice refactored)
- quando è stato fixato un bug
- quando si identifica **code smell**

## Code smells

Indicatore del fatto che c'è qualcosa che non va nel codice, è un inizio non è certo, potrebbe essere banalmente uno stile di programmazione o qualcosa che riduce la leggibilità/capibilità del codice, solitamente gli strumenti di software metric identificano che il codice è affetto da code smells, esempi ne sono:

- **codice duplicato:** stesso codice o molto simile ripetuto in più punti del programma, può essere rimosso e implementato come metodo o funzione chiamata quando

opportuno.

- **metodi lunghi:** meglio riprogettarli come metodi corti
- **switch-case statements:** *→ sostituito con polimorfismo di*
- **data clumping:** succede quando lo stesso gruppo *di* dati rioccorre in vari posti, possono essere rimpiazzati con oggetti che encapsulano tutti i dati
- **generalità speculativa:** accade quando gli sviluppatori includono generalità in un programma in caso fosse necessaria in futuro, risolvibile semplicemente eliminandola.

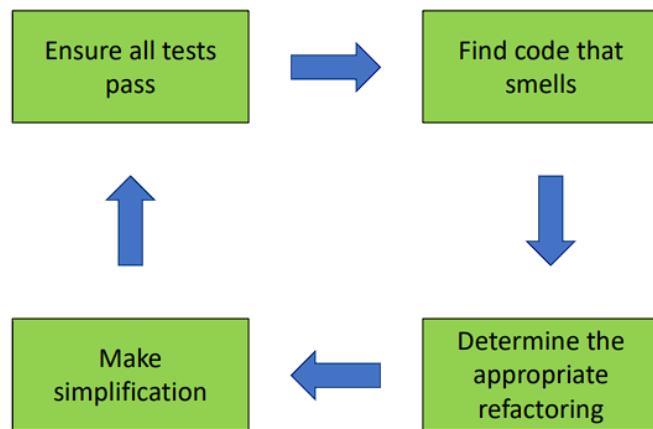
Altri esempi di code smell di cui troviamo 3 classi sono:

- **tropo codice:** metodi lunghi, classi grandi, codice morto,...
- **tropo poco codice:** data class(solo getter/setter), catch vuoto, poco codice nelle classi
- **outside code:** commenti eccessivi

## Software clone

Codice duplicato con o senza cambiamenti, è fonte di bug propagation spesso a causa del copy paste del bug stesso, è un problema per la manutenzione in quanto dobbiamo trovare tutti i cloni e cambiare la stessa cosa in tutti i cloni, tra il 5 e 20% dei sistemi software contiene codice duplicato.

## Processo di refactoring



Per quanto riguarda il ritmo di refactoring, una volta trovato del code smell è sempre preferibile procedere per piccoli step, selezionando piccole parti di codice e seguendo procedure definite (refactoring catalogue), buildare, fare i test ed eventualmente poi continuare con altri passettini.

## Cataloghi di refactoring

Alcuni casi di refactoring sono semplici e soprattutto piccoli, ad esempio la rideconomia di un metodo piuttosto che l'estrazione di un metodo, capita che ci siano casi più complessi come ad esempio il rimpiazzare condizionali con polimorfismo.

## Come applicare refactoring

Diciamo che ci sono due strade principali:

- applicare il refactor **manualmente**
- utilizzare tool **automatizzati**

Ovviamente l'utilizzo di tool automatici è a prescindere più conveniente, ad esempio se un nome di una classe viene cambiato, tutte le referenze dovrebbero essere cambiate consistentemente (costruttore, filename, oggetti), bisogna considerare che i tool automatici però supportano solo il simple refactoring, in entrambi i casi dopo aver fatto refactor bisogna eseguire i test.

## Casi di refactor interessanti

Tra le cose che possiamo fare c'è l'**estrazione di un'interfaccia**, ossia la separazione dell'implementazione di una classe dalla sua interfaccia, esempio ne è l'interfaccia List e le effettive classi che ereditano ArrayList, LinkedList, ..., mi permette di dare le dipendenze all'interfaccia e avere libertà sull'implementazione che può risultare più efficiente o meno.

Altra cosa che si può fare è il **pull-up** refactoring ossia spostare campi di una sottoclasse in una superclasse

C'è anche **extract method** e **move method** dove con quest'ultimo sposto un metodo da una classe ad un'altra, esempio devo capire se una persona partecipa a un progetto, il metodo può essere implementato dalla classe persona o dalla classe progetto.

Altro refactor è quello in cui facciamo il **replace di temp con query** ossia, al posto di utilizzare una variabile temporanea in un metodo, possiamo creare un metodo ad hoc che ritorna lo stesso valore di quella variabile e chiamare il metodo

Nel caso avessi una classe con troppe responsabilità, detta anche **god class**, è bene spezzarla e crearne delle classi più piccole

Invece per quanto riguarda i refactor più complessi troviamo refactor come **replace inheritance with delegation** piuttosto che **replace conditional with polymorphism** o ancora **separate domain from presentation**:

Lo al posto di fare if o switch  
cosa uso il polimorfismo

- **replace inheritance with delegation**: in questo caso una sottoclasse usa solo una porzione della superclasse e non è interessata nel resto di essa,
- **replace conditional with polymorphism**
- **separate domain from presentation**

divide le responsabilità  
delle classi

IoT inga hanno un proprio strumento che permette di visualizzare lo stato effettivo delle delivery effettuando anche rollback se necessario

Riutilizzo dei componenti mediante la loro repository dove sono definiti template che possono essere combinati per costituire i nuovi progetti

Wurst Phoenix → Non fa test → segue sviluppo  
driven development

**Continuos Integration**: i membri del team integrano frequentemente il loro lavoro su una repository condivisa. Ogni integrazione viene verificata da un processo automatico che compila il codice ed esegue test automatici. L'obiettivo è individuare tempestivamente problemi di integrazione e garantire che il SW risulti sempre funzionale e pronto per il rilascio.

**Continuos Deployment**: le modifiche vengono automaticamente rilasciate nell'ambiente di produzione dopo il superamento di test automatizzati. Ciò consente una consegna rapida del SW, garantendo aggiornamenti frequenti e correzione dei bug

Struttura work flow

