Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 3° anno - 6 CFU

Ingegneria del Software

Professore:
Prof. Jacopo Soldani

Autore:
Filippo Ghirardini

Anno Accademico 2024/2025

Contents

Ingegneria del software

Realizzato da: Ghirardini Filippo

A.A. 2024-2025

---

1 Introduzione

L'ingegneria del software è modo in cui produciamo il software, dall'esplorazione del problema al ritiro del prodotto dal mercato. Riguarda tutti gli **strumenti**, le **tecniche** e i **professionisti** coinvolti nelle seguenti fasi:

1. Analisi dei requisiti
2. Specifica
3. Progettazione
4. Implementazione
5. Integrazione
6. Mantenimento
7. Ritiro

Definizione 1.0.1 (Processo software)**.** *È un approccio sistematico per* ***sviluppo****,* ***operatività****,* ***manutenzione*** *e* ***ritiro*** *del software.*

1.0.1 Scopo

Lo scopo è quello di produrre software che sia:

- Fault free
- Consegnato in tempo
- Rispetti il budget
- Soddisfi le necessità
- Sia facile da modificare

1.1 Casi di studio

Richiamiamo alcune definizioni in ambito software:

Definizione 1.1.1 (Robustezza)**.** *Capacità di un software di mantenere il suo corretto funzionamento anche quando sottoposto a condizioni anomale (errori, eccezioni o input non validi). Contribuisce a garantire che il sistema sia affidabile e che possa continuare ad operare anche in situazioni critiche non previste.*

Definizione 1.1.2 (Fault tolerance)**.** *Capacità di un software di rilevare, gestire e riprendersi da errori o guasti senza causare interruzioni significative nei servizi o la perdita di dati.*

1.1.1 Gemini V

Missione nello spazio con equipaggio. Al suo rientro la navicella atterrò ad $80km$ dal punto previsto a causa di un **errore nel modello** (uno sviluppatore inserì la rotazione terrestre sbagliata).

1.1.2 Denver Airport

Il progetto prevedeva lo smistamento automatico dei bagagli con un sistema troppo complesso: i tempi di costruzione furono notevolmente allungati, i costi furono più del previsto e non c'era **fault tolerance** (il guasto di un singolo PC bloccava l'intero sistema). Alla fine venne abbandonato.

1.1.3 THERAC-25

Una macchina da radioterapia con un software progettato male e poco robusto: in caso di errore le emissioni di raggi non venivano sempre terminate ed era possibile da parte dell'operatore ignorarlo. Causò 3 decessi su 6 pazienti.

1.1.4 Sistema anti-missile Patriot

Un sistema poco robusto che non riuscì ad evitare un attacco in Arabia Saudita con conseguenti 28 morti. La causa fu l'uso oltre il tempo di progettazione: 100 ore contro le 14 previste.

1.1.5 London ambulance service

Un sistema per l'ottimizzazione delle ambulanze a Londra, migliorando i percorsi e istruendo vocalmente gli autisti. In questo caso era l'analisi del problema ad essere errata. Inoltre la UI era inadeguata, gli utenti poco addestrati e non era previsto un backup.

1.1.6 Ariane V

Un lanciatore per razzi che si è autodistrutto dopo 40 secondi a causa di un errore di sviluppo e test inefficienti: veniva usato una variabile a $16bit$ per un valore a $64bit$, causando un overflow.

1.1.7 Toyota

Il software per le macchine Toyota tra il 2000 e il 2013 fu mal progettato e causava acceleramenti involontari del veicolo.

1.1.8 METEOR

Prima metro al mondo ad essere automatizzata, locata a Parigi. Fu un successo grazie alla sua ottima progettazione.

Questo ci porta a concludere che data la forte presenza del software nel mondo di oggi è importante utilizzare tecniche di ingegneria del software per renderlo affidabile, rapido da produrre e sostenibile.

1.2 Storia

Tra il 1963 e il 1964, durante lo sviluppo dei sistemi di guida e navigazione per la missione Apollo, viene coniato il termine software engineering da Margaret Hamilton.
Nel 1968 la NATO organizza una conferenza al riguardo in quanto la qualità del software era bassa ed era necessario decidere tecniche e paradigmi per la produzione di software.
Nel 1994 viene fatta un'analisi media del software prodotto che fa vedere come:

- Il 16.2% del software sia stato prodotto in tempo
- Il 52.7% sia stato in ritardo, a causa di difficoltà iniziali, cambiamento della piattaforma e difetti finali
- Il 31.1% non sia stato completato per obsolescenza prematura, incapacità e mancanza di fondi

In particolare le cause di abbandono evidenziate furono relativi a tre aspetti:

- Aspettative incomplete, che cambiano nel tempo o che non sono chiare
- Mancanza di risorse e aspettative su funzionalità e tempi irrealistiche

1.3 Aspetti importanti

Vediamo alcuni aspetti importanti dello sviluppo del software.

1.3.1 Specificità

Per natura, un software è diverso da altri prodotti ingegnerizzabili in quanto non è necessariamente vincolato da materiali e leggi fisiche. Inoltre non si consuma e non ha costi aggiuntivi per ogni unità prodotta.
Ad esempio la **fault tolerance** è una specificità, in quanto quando un software crasha lo si fa ripartire cercando di minimizzare gli effetti del problema.

1.3.2 Economia

L'ingegneria del software si occupa anche di cercare soluzioni vantaggiose dal punto di vista economico. In particolare è importante notare come il costo maggiore derivi sempre dalla **manutenzione** successiva alla produzione del software. Questa può essere di due tipi:

- *Correttiva*: rimuove gli errori lasciando invariata la specifica
- *Migliorativa*: cambia le specifiche
  - *Perfettiva*: migliora la qualità del software, fornisce nuove funzionalità o ne migliora di esistenti
  - *Adattativa*: modifiche a seguito del cambiamento del contesto

1.3.3 Teamwork

La maggior parte del software è prodotto in team. Questo porta a diversi problemi, come l'interfaccia tra le varie componenti del codice e la comunicazione tra i membri del team. L'ingegneria del software deve quindi gestire anche i rapporti umani e l'organizzazione di un team.

2 Processo software

Definizione 2.0.1 (Processo software)**.** *Sequenza di attività necessarie a sviluppare un sistema software.*

Ogni modello di processo software include:

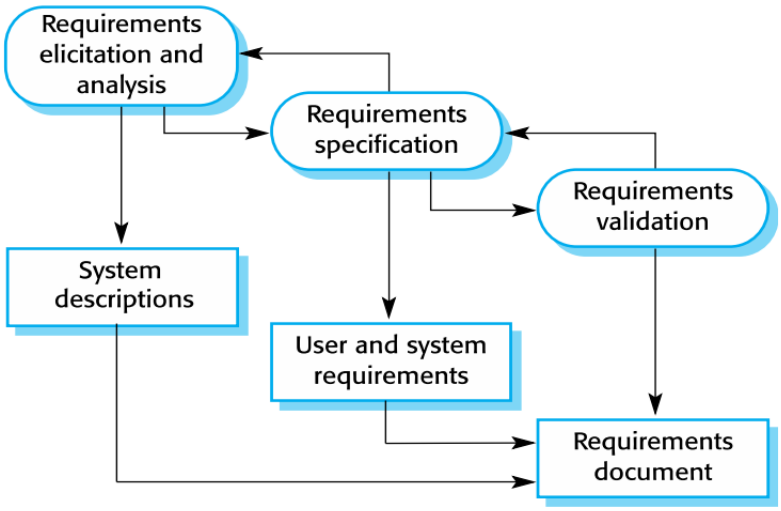
- **Specifica**: definizione di cosa deve essere fatto
- **Progettazione** e **implementazione**
- **Validazione**: verifica che il sistema rispetti le specifiche
- **Evoluzione**: modifica o aggiornamento del sistema

2.1 Fasi di sviluppo

2.1.1 Specifica

Questa fase stabilisce quali **servizi** sono richiesti e quali **vincoli** ci sono. È quindi un processo di *ingegneria dei requisiti*:

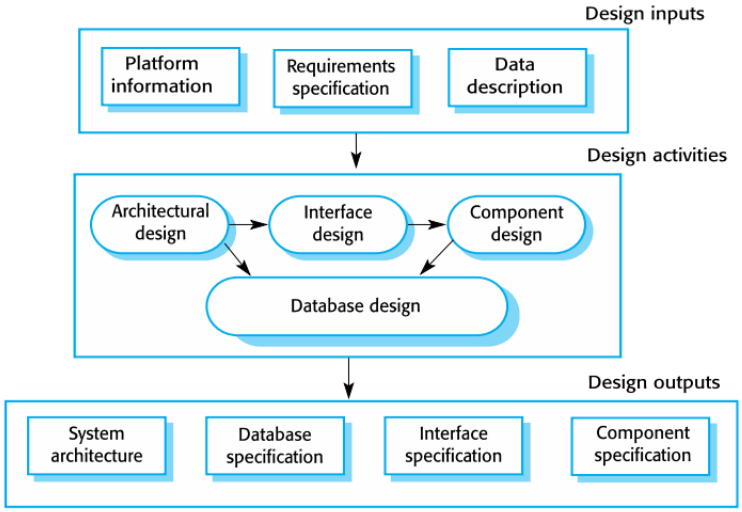
- **Estrazione** e **analisi** dei requisiti: cosa richiedono o si aspettano gli stakeholder
- **Specifica** dei requisiti: definirli in dettaglio
- **Convalida** dei requisiti: verificare che siano validi



2.1.2 Progettazione

In questa fase si definisce una struttura software che realizzi la specifica, analizzando quattro aspetti:

- **Architectural design**: identificazione della struttura in termini di componenti e di come si relazionano tra di loro
- **Database design**: definizione delle strutture dati necessarie e della loro rappresentazione in database
- **Interface design**: definizione delle interfacce tra le diverse componenti del sistema
- **Component design**: definizione in dettaglio delle varie componenti, identificando quelle realizzabili con elementi già esistenti



2.1.3 Sviluppo

La struttura progettata nella fase precedente viene ora realizzata tramite uno o più applicativi da implementare o da configurare. Include:

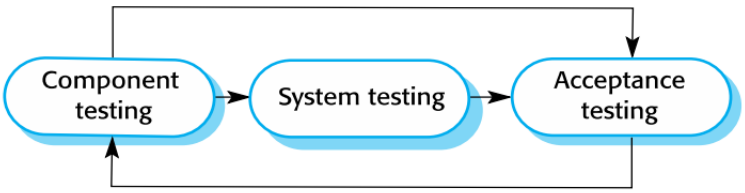
- **Programmazione** attività senza un processo standard
- **Debugging**: attività per identificare e correggere errori o bug

Spesso la progettazione e lo sviluppo sono svolte in **interleaving**.

2.1.4 Validazione

Verifica e **validazione** hanno lo scopo di dimostrare che un sistema è conforme alle specifiche e soddisfa i requisiti del cliente. Viene spesso fatta tramite **testing** con casi derivati dalla specifica dei dati che poi dovranno essere realmente utilizzati. Si suddivide in:

- **Component** testing: i componenti sviluppati sono testati indipendentemente
- **System** testing: il sistema è testato interamente prestando particolare attenzione alle *emergent properties*
- **Customer** testing: il sistema è testato con i dati del cliente



2.1.5 Evoluzione

I cambiamenti sono inevitabili in quanto le richieste del cliente possono cambiare nel tempo o possono uscire nuove tecnologie più aggiornate. Questi portano a dei **rework** costosi che richiedono la ripetizione parziale delle fasi precedentemente descritte.
Per ridurre i costi è importante anticipare i cambiamenti e garantire quindi **change tolerance**. Questo è più facile con i modelli incrementali.

2.2 Modelli

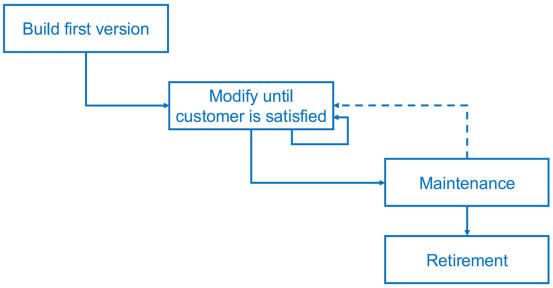
Definizione 2.2.1 (Modello)**.** *Il modello di un processo software fornisce una rappresentazione astratta del processo stesso:*

- ***Suddivisione*** *del processo in attività: cosa fare, quando farlo, come e cosa si ottiene*
- ***Organizzazione*** *delle attività: ordinamento, criteri di terminazione*

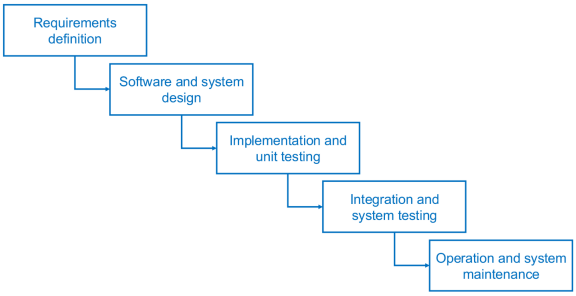
E.g. ISO 12207

2.2.1 Sequenziale

Build and fix Questo modello non prevede alcuna specifica o progettazione: lo sviluppatore scrive un programma e lo modifica ripetutamente finché non soddisfa il cliente. Adeguato solo per progetti molto piccoli.

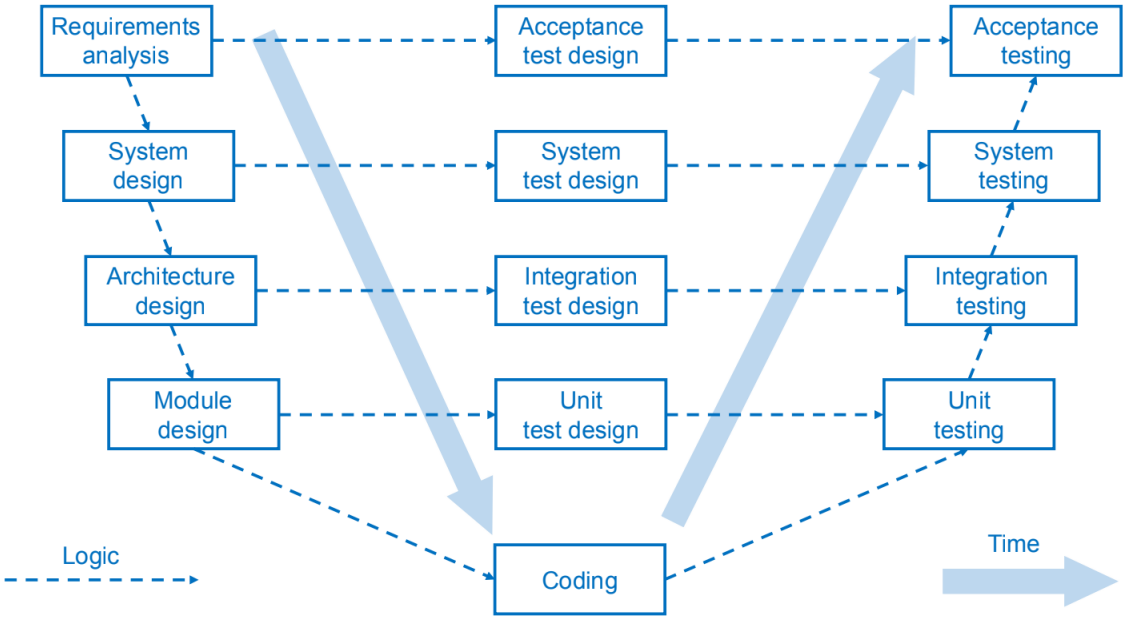


Modello a cascata Modello ideato da Royce nel 1970, fu il primo a distinguere e definire le fasi di un processo software, dando finalmente importanza all'analisi e alla progettazione prima della codifica. Ogni fase prima di procedere deve produrre un documento che deve essere approvato da chi di dovere. I **contro** principali sono l'enorme quantità di *documenti* prodotti e l'estrema *rigidità*: il cliente vede solo il prodotto finale che spesso non va bene e si deve rincominciare da capo.



Note 2.2.1. Royce stesso riconosce i problemi del suo modello e propone un alternativa con un **feedback loop** da una fase alla precedente.

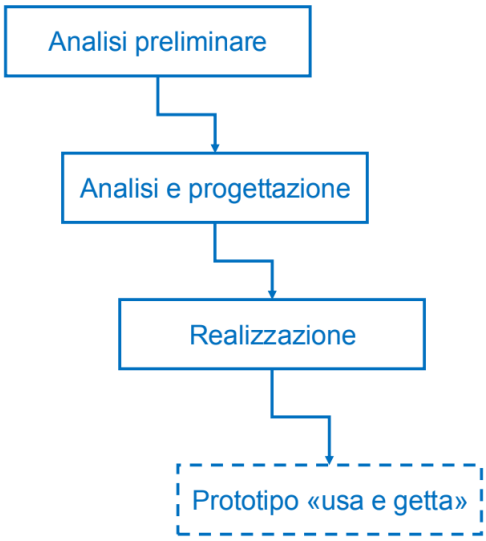
Modello a V In questo modello le attività di **sinistra** sono di analisi che scompongono i requisiti degli utenti in sezioni piccole; quelle di **destra** aggregano e testano il prodotto delle precedenti per verificare che le esigenze siano effettivamente rispettate.
Al centro troviamo la progettazione dei **test** da eseguire prima della codifica.



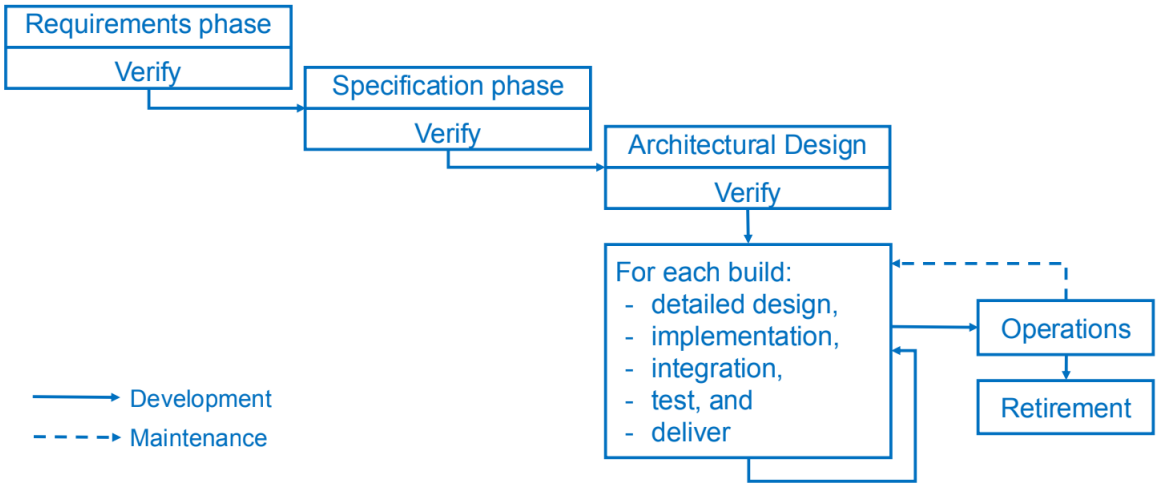
Note 2.2.2. Questo modello è uno standard SQA (qualità del software).

2.2.2 Iterativo

Rapid prototyping o evolutivo Consiste nel costruire velocemente un prototipo per permettere al committente di sperimentarlo. In questo modo il cliente può descrivere meglio i requisiti, sopratutto quando anche a lui non sono chiari.



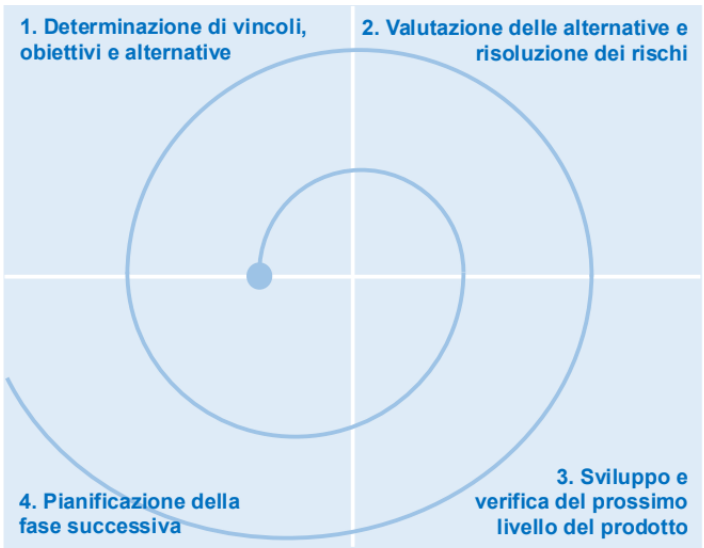
Modello incrementale In questo modello il sistema è costruito **iterativamente** aggiungendo nuove funzionalità a partire da requisiti definiti inizialmente.
Questo permette di **ritardare** fasi che per motivi esterni non possono proseguire e fa uscire versioni iniziali ed utilizzabili molto rapidamente, in modo da ricevere anche **feedback** che aiutino a correggere il prodotto a basso costo.
I contro principali sono che il processo di sviluppo non è molto visibile e c'è il rischio che diventi un *build and fix*: la struttura del sistema tende a degradarsi e diventa più costoso fare il refactoring.



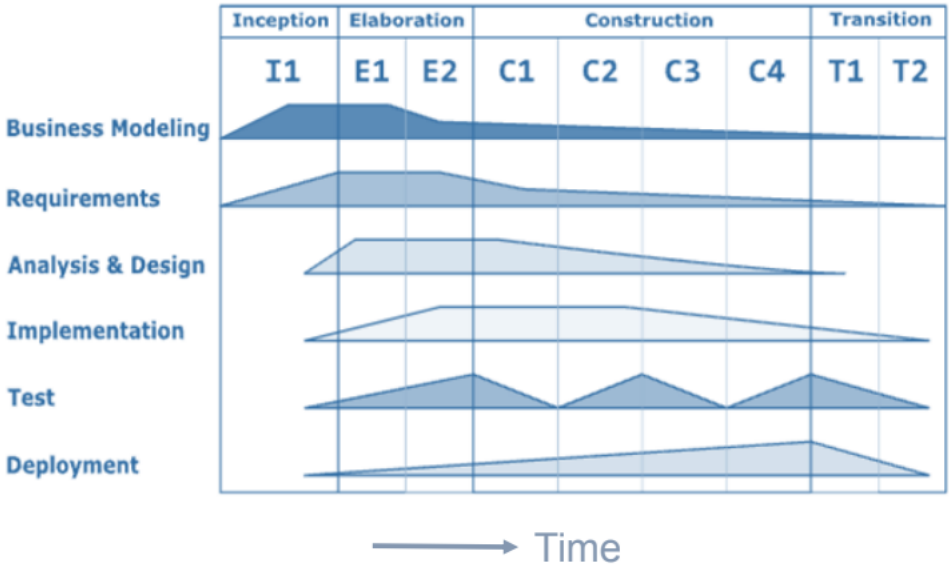
Modello a spirale Ideato da Bohem nel 1998, ispirato dal *plan-do-check-ackt* di Deming, divide ogni **iterazione** in quattro fasi:

1. Definizione degli obiettivi
2. Analisi dei rischi
3. Sviluppo e validazione
4. Pianificazione del nuovo ciclo

È un modello astratto da istanziare decidendo cosa fare in ogni iterazione e in ogni fase, applicandolo volendo ai cicli tradizionali. Si concentra molto sugli aspetti gestionali: pianificazione delle fasi, **risk-driven** (guidato dall'analisi dei rischi) e comunicazione con il cliente.



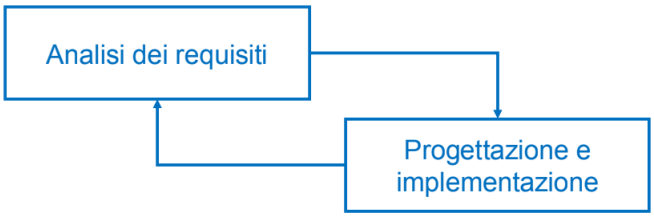
Unified process Ideato da Booch Et Al nel 1999, è guidato da **casi d'uso** e **analisi dei rischi** già a partire dalla raccolta e analisi dei requisiti. È un modello iterativo **incrementale** incentrato sull'**architettura**: nelle prime fasi c'è una definizione generale e i dettagli vengono lasciati alle fasi successive. Questo permette di avere subito una visione generale del sistema che diventa poi facilmente adattabile.



2.2.3 Agile

Oggi è sempre più importante la **rapidità** nello sviluppo e nel rilascio del software in quanto i requisiti delle aziende cambiano molto velocemente e con essi diventa impossibile avere un sistema stabile di requisiti.
Il modello agile viene introdotto negli anni '90 proprio per ridurre i tempi sopra descritti:

- Le fasi di specifica, progettazione e sviluppo sono eseguite in **interleaving**
- Il sistema è sviluppato con versioni incrementali valutate assieme agli stakeholder
- Rilascio frequente
- Supporti allo sviluppo, e.g. automated testing



Il modello agile segue i seguenti principi:

Customer involvement I clienti devono essere coinvolti durante il processo di sviluppo per fornire, prioritizzare e valutare le iterazioni del sistema.

People not process Le abilità del team devono essere riconosciute e gli sviluppatori devono essere liberi di sviluppare a modo loro, purché venga mantenuta comunicazione.

Mantain semplicity Cercare di ridurre sempre al minimo la complessità nello sviluppo e nel sistema. Bisogna mantenere il codice semplice ma avanzato tecnicamente a discapito di una documentazione mantenuta al minimo.

Incremental delivery Il sistema software è sviluppato in versioni incrementali, con il cliente che specifica i requisiti da soddisfare in ciascuna versione.

Embrace change Accettare che i requisiti cambieranno nel tempo e rendere facile la loro implementazione.

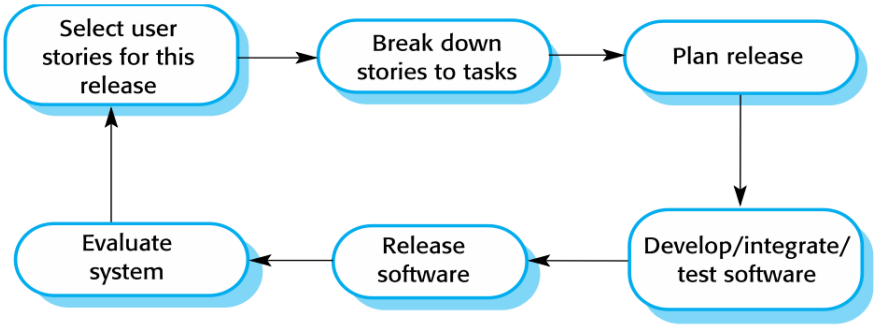
Il modello agile è **conveniente** per lo sviluppo di prodotti di piccola o media dimensione (fino a 50 sviluppatori) oppure in contesti di sviluppo di sistemi custom (meno regolamenti e restrizioni). Nella pratica comunque viene usato nella maggior parte dei team.

Il modello agile ha portato alla nascita di:

- **Continuous Integration**: integrazione continua di tutte le modifiche o aggiunte all'interno di un *main branch*, validata tramite automatic build e testing
- **Continuous deployment**: dispiegamento continuo e *automatico* delle nuove versioni ottenute tramite CI

2.2.4 Extreme Programming

L'extreme programming è un approccio estremo all'approccio iterativo e agile. Prevede che nuove versioni minori siano rilasciate più volte in un giorno, versioni incrementali rilasciate al cliente ogni due settimane e tutti i test sono eseguiti per ogni build.



Alcune pratiche comuni nell'XP:

Planning incrementale I requisiti sono raccolti sotto forma di user stories divise in **task**. Quelle da includere nella release sono determinate in base al tempo disponibile e alla loro priorità.

Release piccole Si parte con una piccola release iniziale che garantisca le funzionalità di base si procede con piccole e frequenti release che aggiungono funzionalità.

Design semplice La progettazione si concentra solo sui requisiti correnti e deve essere comprensibile a tutti.

Test-first developement Si sviluppano prima i test del codice stesso (a volte generati automaticamente).

Refactoring Il refactoring deve essere eseguito continuamente appena ci si rende conto del miglioramento necessario, mantenendo sempre il codice semplice, facilmente manutenibile e auto esplicativo.

Pair programming Gli sviluppatori lavorano in coppia in modo che ci sia sviluppo e supporto reciproco.

Collective ownership Le coppie lavorano su tutte le aree del sistema e la responsabilità è quindi condivisa.

Sustainable pace Ridurre al minimo il lavoro straordinario in quanto abbassa qualità e produttività.

On-site customer Un rappresentante del cliente deve essere disponibile al team per fornire o prioritizzare i requisiti.

Il modello di Extreme Programming si concentra principalmente su aspetti tecnici e per questo non è facilmente integrabile nelle organizzazioni. Di conseguenza il metodo non è molto diffuso ma alcuni degli aspetti che tratta sono stati trasportati in altri approcci.

2.2.5 Scrum

Scrum è un metodo agile per lo sviluppo iterativo e incrementale di un sistema software. L'idea è di ottenere un processo in cui un insieme di persone si muovono all'unisono per raggiungere un obiettivo che soddisfi la squadra.

Definizione 2.2.2 (Product backlog)**.** *Documento che contiene tutti i requisiti attualmente conosciuti.*

Ci sono tre figure coinvolte:

- **Product owner**: chi identifica le caratteristiche del prodotto, decide le priorità e revisiona il **product backlog** per assicurarsi che vengano rispettati i requisiti
- **Scrum master**: figura responsabile di assicurare che il processo avvenga efficacemente, garantendo le condizioni ambientali e motivazionali al meglio assicurandosi che non ci siano interferenze esterne (senza però avere autorità sul team)
- **Developement team**: gruppo autogestito di sviluppatori non più grande di 7 persone che si occupa dello sviluppo e della documentazione

Le fasi del metodo scrum sono:

1. **Pre-game phase**, ovvero la pianificazione, che a sua volta si compone di:
   - **Planning sub-phase**: definizione del sistema che deve essere sviluppato in termini di product backlog
   - **Architecture sub-phase**: design di alto livello del sistema, inclusa l'architettura, in base agli elementi del backlog
2. **Game phase**, ovvero lo sviluppo. Il sistema viene sviluppato attraverso una serie di **sprint**, ovvero un ciclo iterativo nel quale vengono sviluppate o migliorate delle funzionalità. Ogni sprint dura da una settimana ad un mese e include le classiche fasi di sviluppo. Si divide nelle seguenti fasi:
   (a) Si parte dal product backlog che contiene la lista dei **requisiti** da fare (TBD). Da questi vengono selezionati dal team e dal cliente quelli da sviluppare.
   (b) Si procede con la pianificazione, gestita dal product owner, e con la creazione dello **sprint backlog**
   (c) Inizia lo sviluppo da parte dei diversi team che rimangono isolati e protetti dallo scrum master; si occupa anche di organizzare brevi meeting giornalieri di aggiornamento.
   (d) Al termine dello sprint il prodotto viene revisionato in un incontro tra team, clienti ed eventuali utenti
   (e) Tra uno sprint ed il successivo viene organizzato un evento di **retrospettiva** dove il team riflette, impara e si adatta con l'obiettivo di migliorare
3. **Post-game phase**: conclude il processo di sviluppo e il prodotto viene preparato per il rilascio (test, integrazione, documentazione, formazione e marketing)

I vantaggi di questo approccio sono che il prodotto e **partizionato** in sotto problemi più facili da gestire, i requisiti non ancora pronti non ostacolano lo sviluppo, c'è molta comunicazione, i clienti ottengono increment periodici e si stabilisce un rapporto di fiducia.

2.3 Kanban

Il Kanban è un approccio all'organizzazione di un progetto che consiste nel dividere le attività tra **To Do**, **Work In Progress** e **Done**. Questi vengono poi visualizzati tramite una tabella. Viene inoltre imposto un limite alla categoria WIP che riduce il **task switching** e rende più facile trovare i colli di bottiglia. Ottimizza in generale l'**efficienza**.