Analisi e Progettazione del Software

spitfire

A.A. 2023-2024

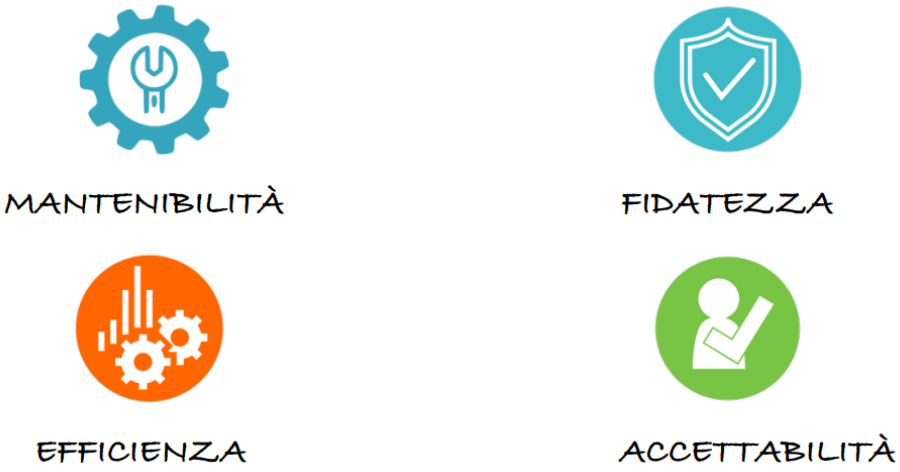
Contents

1 Introduzione

Che cos'è il **software**? Esso è **un programma per computer** unito alla **documentazione ad esso associata**, la quale specifica e comprende **requisiti, modelli di progetto, manuale utente,...**
I prodotti software possono essere:

- **Generici**: sviluppati per un ampio insieme di clienti (elaboratori di testo, database,...)
- **Personalizzati** (custom): sviluppati per un singolo cliente in base alla sue esigenze specifiche

Un nuovo prodotto software può essere **creato da zero, personalizzando software già esistenti o riusando parti o software già esistente**. Le caratteristiche essenziali di un buon software sono:

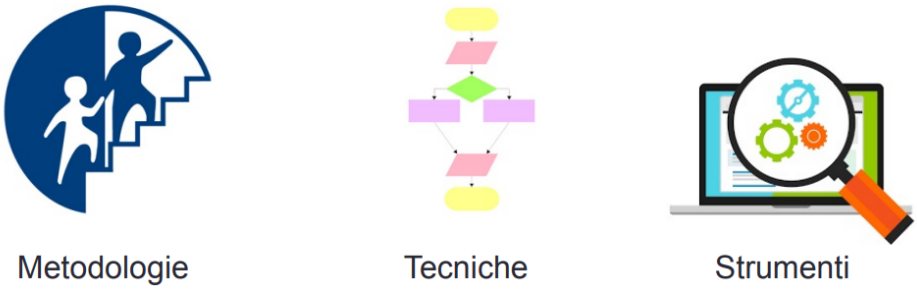


1.1 Introduzione all'ingegneria del software

Che cos'è **l'ingegneria del software**? **L'ingegneria del software** è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione del software di buona qualità, dalle **prime fasi della specifica del sistema fino alla manutenzione del sistema** dopo la messa in uso. Vediamo cosa si intende per **disciplina ingegneristica** e "**Tutti gli aspetti della produzione del software**":

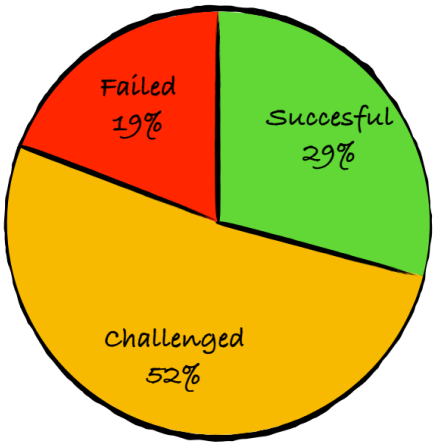
- **Disciplina ingegneristica**: Utilizzare metodi e teorie **appropriati** per risolvere i problemi tenendo conto dei vincoli **organizzativi e finanziari**
- **Tutti gli aspetti della produzione del software**: Non solo il **processo tecnico di sviluppo**. Anche la **gestione del progetto** e lo sviluppo di **strumenti** ,metodi ecc... per supportare la produzione del software

La disciplina dell'ingegneria del software si occupa di:

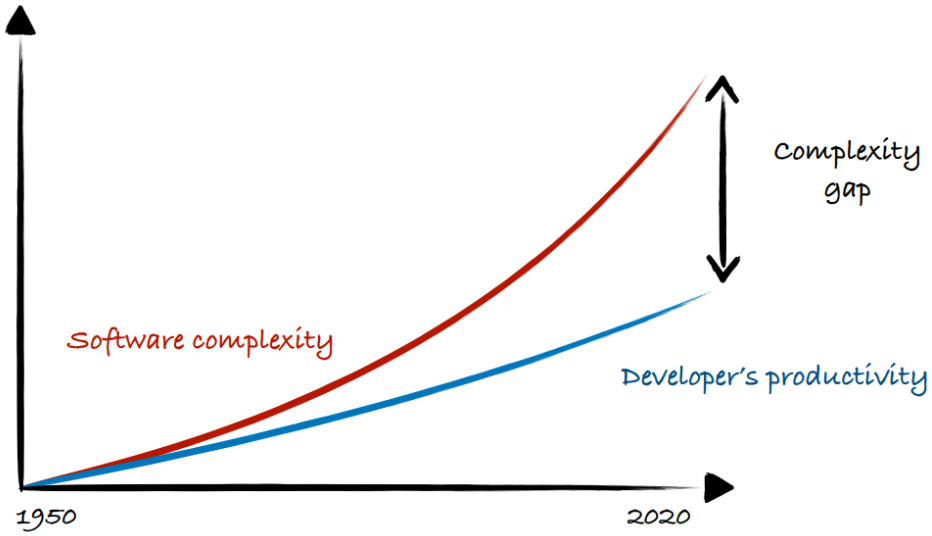


1.2 La crisi del software

Il termine **crisi del software** (o software crisis) è usato nell'ambito dell'ingegneria del software per descrivere l'impatto della **rapida crescita** della potenza degli elaboratori e la **complessità** dei problemi che dovevano esseri affrontati. Le parole chiavi della software crisis erano **complessità, attese e cambiamento**. Il concetto di software crisis emerse negli anni '60.



Standish Group 2015 Chaos Report



Le cause della crisi del software erano legate alla **complessità dei processi software** e alla **relativa immaturità dell'ingegneria del software** Per superare la crisi infatti si dovettero introdurre:

- **Management**
- **Organizzazione**, attraverso **analisi e progettazione**
- **Teorie e tecniche** come la **programmazione strutturata e ad oggetti**
- **Strumenti**, come gli IDE
- **Metodologie**, tra cui il **modello a cascata e il modello agile**

1.3 Analisi e progettazione

Che cosa sono **analisi e progettazione**?
L'analisi enfatizza un'**investigazione del problema e dei requisiti** invece che una soluzione: per esempio, se si vuole realizzare un nuovo sistema di trading online, bisognerà capire **come questo sistema verrà utilizzato** e **quali sono le sue funzioni**.
"Analisi" è un termine ampio con più accezioni, tra cui:

- **Analisi dei requisiti**, cioè un'investigazione dei requisiti del sistema
- **Analisi orientata agli oggetti**, cioè un'investigazione degli oggetti di dominio

La **progettazione** enfatizza una soluzione **concettuale** (software e hardware) che **soddisfa i requisiti**, anziché la relativa implementazione. Per esempio, la descrizione di uno schema di base di dati e di oggetti software. Nella progettazione vengono spesso **esclusi dettagli di basso livello o "ovvi"** (o almeno "ovvi" per coloro a cui è destinato il software).
Infine i progetti possono essere **implementati** e la loro implementazione (ovvero il codice) esprime il progetto realizzato vero e completo. Come nel caso dell'analisi, anche "progettazione" è un termine con più accezioni, tra cui:

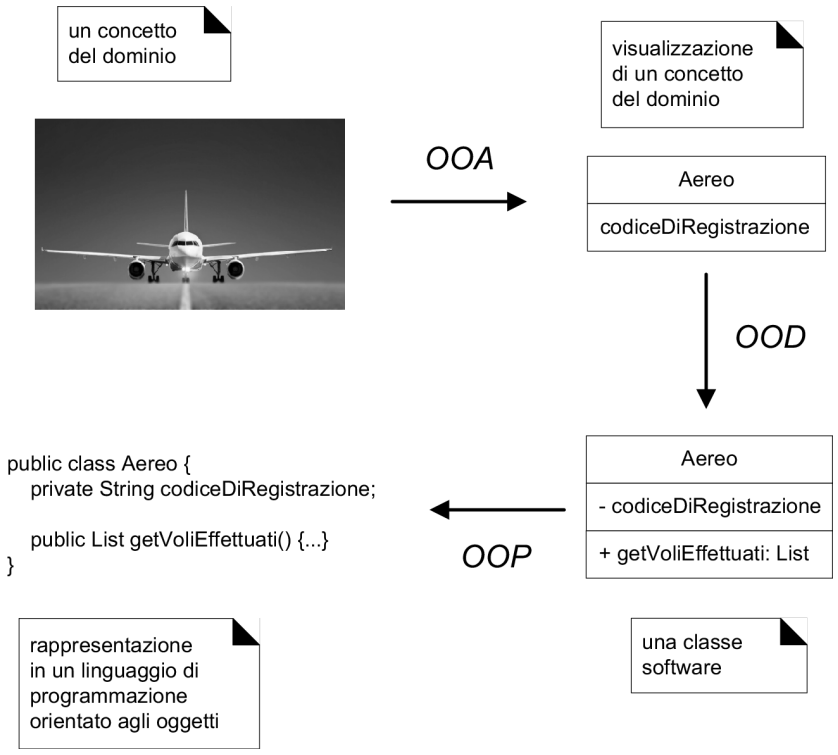
- **Progettazione orientata agli oggetti**
- **Progettazione di basi di dati**

L'analisi e la progettazione possono essere riassunti con la seguente frase:

Fare la cosa giusta(analisi) **e fare la cosa bene**(progettazione)

1.3.1 Analisi e progettazione orientata agli oggetti

Durante **l'analisi orientata agli oggetti** c'è un enfasi sull'**identificazione** e la **descrizione degli oggetti**, o dei **concetti**, nel **dominio del problema**. Per esempio, nel caso di un sistema informatico per voli aerei, alcuni dei concetti possono essere *Aereo, Volo* e *Pilota*.
Durante **la progettazione orientata agli oggetti** (o più semplicemente **progettazione a oggetti**) l'enfasi è sulla **definizione di oggetti software** e sul **modo in cui questi collaborano per soddisfare i requisiti**. Per esempio un oggetto software *Aereo* può avere un attributo *codiceDiRegistrazione* e un metodo *getVoliEffettuati*.



Infine durante **l'implementazione** o la **programmazione orientata agli oggetti**, gli oggetti progettati vengono implementati, per esempio implementando la classe *Aereo* in un linguaggio ad oggetti. Dunque, analisi e progettazione **hanno obbiettivi diversi che vengono perseguiti in maniera diversa**. Tuttavia, come mostrato dall'esempio sopra, esse sono **attività fortemente sinergiche** che sono **correlate fra loro** e con le **altre attività dello sviluppo del software**.

1.4 Introduzione ai diagrammi e ai passi fondamentali dello sviluppo software

Vediamo una breve introduzione dei **vai diagrammi e dei passi fondamentali** legati allo sviluppo software.

1.4.1 Definizione dei casi d'uso

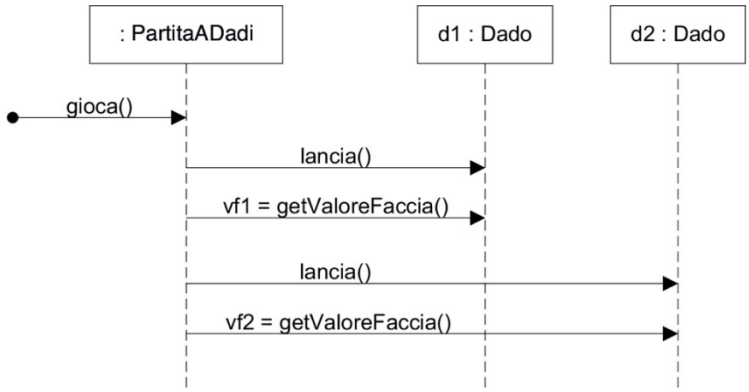
L'analisi dei requisiti può comprendere **storie o scenari** relativi al modo in cui l'applicazione può essere utilizzata dagli utenti; queste storie possono essere scritte come **casi d'uso**. I casi d'uso **non sono un elaborato ad oggetti** ma semplicemente delle storie scritte. Sono tuttavia uno strumento **diffuso nell'analisi dei requisiti**. Facciamo un'esempio:
Gioca una partita a Dadi: Il giocatore chiede di lanciare i dadi. Il Sistema presenta il risultato: se il valore totale delle facce dei dadi è sette, il giocatore ha vinto; altrimenti ha perso.

1.4.2 Definizione di un modello di dominio

L'analisi orientata agli oggetti è interessata alla **creazione di una descrizione del dominio da un punto di vista ad oggetti**. Vengono identificati i **concetti, gli attributi e le associazioni considerati significativi**. Il risultato può essere espresso come un **modello di dominio** che mostra i concetti o gli oggetti **significativi** del dominio. Esso è rappresentato nel seguente modo:

1.4.3 Definizione dei diagrammi di interazione

La **progettazione ad oggetti** è interessata alla **definizione di oggetti software, delle loro responsabilità e collaborazioni**. Una notazione comune per illustrare queste collaborazione è un **diagramma di sequenza** (un tipo di diagramma UML). Esso mostra lo scambio di messaggi **tra oggetti software**, dunque l'invocazione di **metodi**. Esso è rappresentato nel seguente modo:

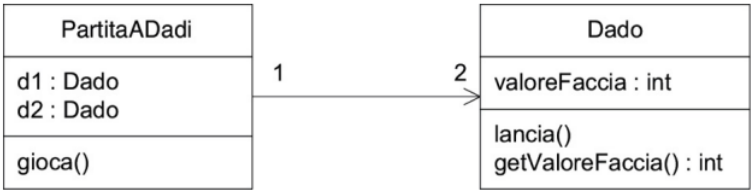


È interessante notare come **la progettazione degli oggetti software e dei programmi si può ispirare a un dominio del mondo reale**, tuttavia essa non è **nè un modello diretto nè una simulazione di questo dominio**. Quindi, per esempio, seppur nel mondo reale è il giocatore a lanciare il dado, nel progetto software è l'oggetto *PartitaADadi* che "lancia" i dadi.

1.4.4 Definizione dei diagrammi di classe di progetto

Accanto a una visione dinamica delle **collaborazioni tra oggetti**, mostrata dai diagrammi di interazione, è utile mostrare una **vista statica** delle definizioni di classi mediante un

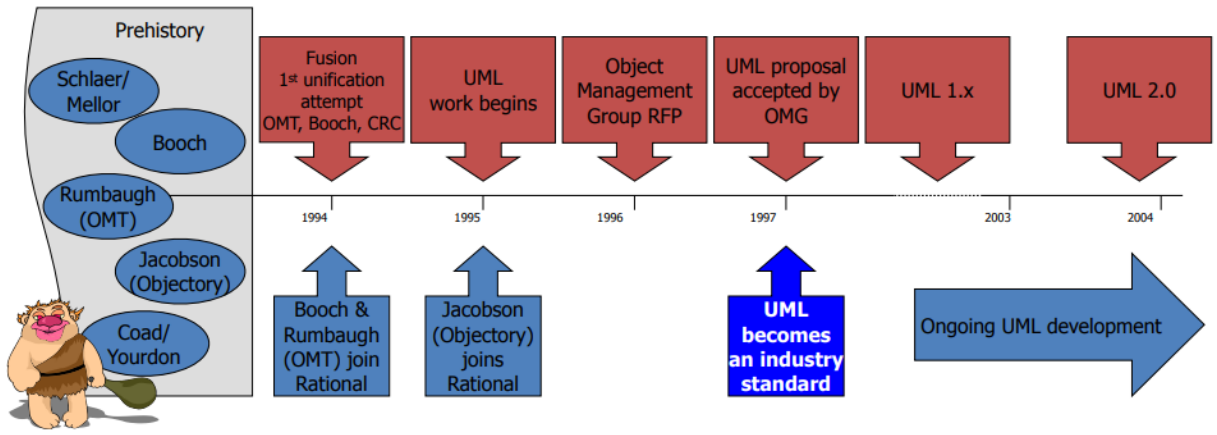
diagramma delle classi di progetto, che mostra le classi software con i loro attributi e metodi. Il diagramma delle classi di progetto è rappresentato nella seguente maniera:



Diversamente dal modello di dominio, che illustra **classi del mondo reale**, questo diagramma mostra **classi software**. Si noti che, benché questo diagramma delle classi di progetto **non sia uguale al modello di dominio**, i nomi e il contenuto delle classi sono **simili**. In tal modo i **progetti e i linguaggi Object Oriented** (OO) sono in grado di **favorire un salto rappresentazionale basso** tra i componenti software e il nostro modello mentale di un dominio, **migliorando la comprensione**.

1.5 UML

Unified Modelling Language, abbreviato **UML**, è un **linguaggio visuale** per la **specifica, la costruzione e la documentazione degli elaborati** di un sistema software. UML rappresenta una **collezione di best practices di ingegneria**, dimostratesi vincenti nella modellazione di sistemi vasti e complessi; inoltre esso **favorisce la divulgazione delle informazioni nella comunità dell'ingegneria del software** in quanto è *standard de facto*. Bisogna però tenere a mente che **UML non è una metodologia ma un linguaggio!**
Il termine *visuale* della definizione è un punto fondamentale. UML è uno standard de facto per la **notazione di diagrammi per disegnare o rappresentare figure** (con del testo) **relative al software**, e in particolare, al software OO. A un livello più profondo, di particolare interesse per i produttori di strumenti per **MDA** (Model Driven Architecture) alla base della notazione UML c'è il **meta-modello di UML** che descrive la **semantica** degli elementi di modellazione, tuttavia non è necessario che lo sviluppatore lo conosca. Presentiamo ora una breve storia di UML:



Il più significativo aggiornamento di UML è avvenuto nel **2003**:

- Maggiore **consistenza**
- Semantica definita in maniera **più chiara e dettagliata**
- **Nuovi diagrammi**
- Compatibilità con le precedenti versioni (1.x)

Altra parola importante è *unified*: UML vuole essere un **linguaggio unificante** sotto diversi aspetti:

- **Storico** (OMT, Booch, CRC, Objectory)
- **Ciclo di sviluppo** (sintassi visuali per tutte le fasi)
- **Domini applicativi** (dai sistemi embedded ai sistemi gestionali)
- **Linguaggi e piattaforme di sviluppo** (.Net, Java, C#,...)
- **Processi di sviluppo** (UP, BPM, ...)

1.5.1 UML e gli oggetti

UML modella i sistemi come **una serie di oggetti che collaborano fra loro**. Si hanno quindi due strutture:

- **Struttura statica**:
  - **Quali** tipi di oggetti sono necessari
  - **Come** sono correlati
- **Struttura dinamica**:
  - **Ciclo di vita** di questi oggetti
  - **Come collaborano** per fornire le funzionalità richieste

1.5.2 Tre modi di applicare UML

Fowler [Fowler03] descrive tre modi per applicare UML:

- **UML come abbozzo**: Diagrammi **informali e incompleti** (spesso abbozzati a mano su una lavagna bianca), che vengono creati per **esplorare parti difficili dello spazio del problema o della soluzione**, sfruttando l'espressività dei linguaggi visuali.
- **UML come progetto**: Diagrammi di progetto abbastanza dettagliati che vengono utilizzati per:
  1. **Il reverse engineering**, ovvero per visualizzare e comprendere meglio **del codice già esistente** mediante dei diagrammi UML. In questo caso, uno strumento UML legge il codice sorgente o binario per **generare** (di solito) **dei diagrammi UML dei package, delle classi e di sequenza**. Questi "progetti" possono aiutare il lettore a capire i principali elementi, le strutture e le collaborazioni

2. **Il forward engineering**, ovvero per la **generazione di codice**. In questo caso, alcuni diagrammi dettagliati possono fornire una **guida alla generazione di codice** da fare manualmente o automaticamente con un strumento. Solitamente, i diagrammi sono utilizzati per **specificare una parte di codice**, mentre il resto del codice viene scritto da uno sviluppatore durante la codifica, magari applicando UML come abbozzo.

- **UML come linguaggio di programmazione**: La specifica **completamente eseguibile** di un sistema software con UML. Il codice viene generato **automaticamente** e non viene normalmente normalmente né visto né modificato dagli sviluppatori; quindi UML viene usato come vero e proprio **linguaggio di programmazione**. Questo utilizzo di UML richiede un modo **pratico** per rappresentare sotto forma di di diagrammi **tutto il comportamento o la logica** (probabilmente tramite diagrammi di interazione e di stato). Si tratta di un approccio **ancora in corso di sviluppo** sia in termini di teoria sia in termini di usabilità e robustezza degli strumenti.

La **modellazione agile** enfatizza l'uso di UML come **abbozzo**; si tratta di un metodo comune per applicare UML, spesso con un elevato ritorno in termini di **investimento di tempo** (che è normalmente breve).

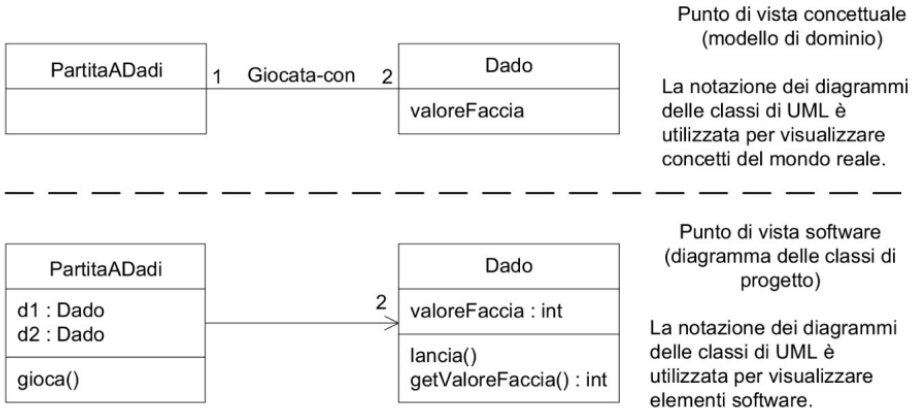
1.5.3 Due punti di vista per applicare UML

UML descrive dei tipi **grezzi** di diagrammi, come i diagrammi delle classi e i diagrammi di sequenza; tuttavia UML **non impone un particolare punto di vista di modellazione per l'uso di questi diagrammi**; quindi la stessa notazione può essere usata secondo **due punti di vista** (o prospettive) e **tipi di modelli**:

- **Punto di vista concettuale**: I diagrammi sono scritti e interpretati **come descrizioni di oggetti del mondo reale** o nel dominio di interesse

- **Punto di vista software**: I diagrammi, che utilizzano **la stessa notazione del punto di vista concettuale**, descrivono astrazioni o componenti software. In particolare, i diagrammi possono descrivere:

  1. **Implementazioni software** con riferimento a una particolare tecnologia
  2. **Specifiche e interfacce** di componenti software, ma **indipendentemente** da ogni possible implementazione



Quindi, in pratica, UMl viene usato:

1. **Nell'analisi**, principalmente secondo il **punto di vista concettuale**
2. **Nella progettazione**, principalmente secondo il **punto di vista software**

1.5.4 Significato di classe

Nell'UML grezzo, abbiamo chiamato "classi" un insieme di oggetti; ma questo termine racchiude una **varietà di casi**: oggetti fisici, concetti astratti, elementi software, eventi e così via. In particolare, una classe UML è un caso particolare di un modello UML generale chiamato **classificatore**, che è qualcosa che ha delle caratteristiche strutturali e/o comportamentali e comprende **classi, attori, interfacce e casi d'uso**. Un metodo **impone una terminologia alternativa sovrapposta all'UML grezzo**; in particolare, ci adegueremo a quella di **UP** (Unified Process), che chiama:

- **Classe concettuale**: Oggetto o concetto **del mondo reale** da un punto di vista **concettuale**. Il modello di dominio di UP contiene **classi concettuali**
- **Classi software**: Una classe che rappresenta un **componente software**, da un punto di vista **software**, indipendentemente dal processo, metodo o linguaggio di programmazione. Il modello di progetto di UP contiene **classi software**.

1.5.5 Vantaggi della modellazione visuale

Disegnare e leggere UML implica che si sta lavorando in **modo visuale**. La modellazione visuale ci permette di sfruttare le capacità del nostro cervello di **comprendere rapidamente simboli, unità e relazioni nelle notazioni** (prevalentemente bidimensionali) a "rettangoli e linee". I diagrammi ci aiutano a vedere o esaminare meglio il **quadro generale** e le relazione tra elementi dell'analisi del software e allo stesso tempo ci permettono di **ignorare o nascondere i dettagli poco interessanti**.

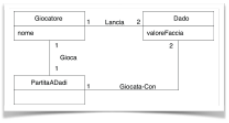
2 Processi per lo sviluppo del software

Un **processo per lo sviluppo del software** (o **processo software**) definisce un approccio disciplinato per la **costruzione**, il **rilascio** e la **manutenzione del software**. Definisce quindi **chi fa che cosa, quando e come** per raggiungere un certo obbiettivo. In particolare:

- **Cosa** sono le **attività**
- **Chi** sono i **ruoli**
- **Come** sono le **metodologie**
- **Quando** riguarda **l'organizzazione temporale** delle attività

Le attività fondamentali di un processo di sviluppo sono:

Requisiti

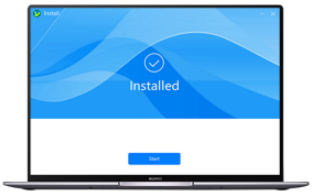


Analisi

Progettazione

Implementazione

Validazione



Rilascio e installazione

Manutenzione
Ed Evoluzione

Gestione
del Progetto

Ciò che distingue i processi software gli uni dagli altri è tuttavia **sono le scelte che riguardano l'organizzazione temporale delle attività** (quando), ovvero il modo in cui essi rispondono alle domande:

- Per quanto tempo continueremo a svolgere questa attività
- Cosa faremo dopo?

2.1 Processi agili e basati sul piano

I processi **orientati al piano** sono processi in cui tutte le attività sono **pianificate in anticipo** e i progressi del progetto sono **misurati rispetto a questo piano**. Invece, nei **processi agili**, la pianificazione è **incrementale**, quindi risulta più facile modificare il processo per riflettere le mutevoli esigenze del cliente. In pratica, **la maggior parte dei processi software include elementi di entrambi gli approcci**. È necessario notare che **non esistono processi software totalmente sbagliati o corretti**.

2.2 Modelli di processo software

In pratica, la maggior parte dei sistemi di grandi dimensioni vengono sviluppati usando processi che incorporano elementi dei seguenti modelli:

- **Sviluppo a cascata**: Modello **basato sul piano**. Fasi separate di **specifica** e di **sviluppo**
- **Sviluppo incrementale**: **Specifica, sviluppo e validazione** si alternano. Può essere guidato dal piano o agile
- **Integrazione e configurazione**: Il sistema viene assemblato a partire da **componenti esistenti e configurabili**. Può essere basato sul piano o agile

Tuttavia "**quale scegliere?**" è una domanda assolutamente non banale. Diamo quindi dei motivi per il quale esistono diversi processi software:

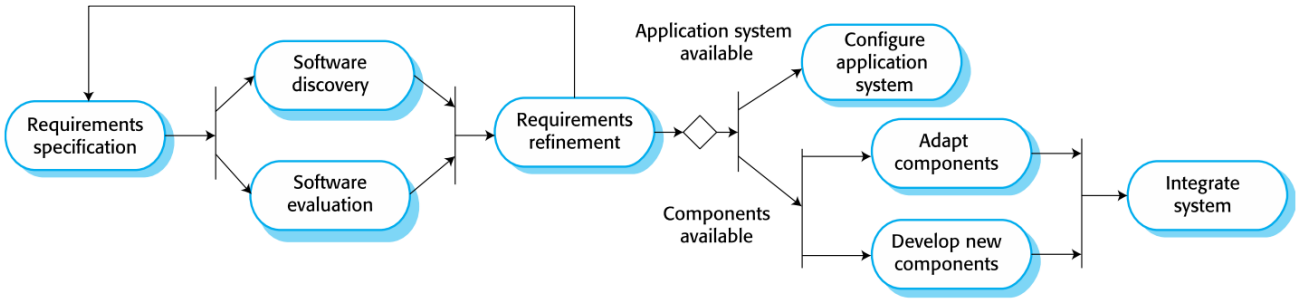
- **Complessità del progetto**: I progetti software possono variare **in termini di complessità**, da semplici applicazioni a sistemi complessi e "mission-critical". La complessità di un progetto **influenza il livello di formalismo e struttura** necessari nel processo di sviluppo
- **Dimensione del team**: Il numero di persone che lavorano ad un progetto software **influenza come il lavoro viene organizzato e coordinato**. Team di grandi dimensioni avranno quindi bisogno di **un processo più formale** rispetto a team di piccole dimensioni
- **Budget e tempistiche**: Il budget e le tempistiche di un progetto software **influenzano il modo in cui il progetto viene organizzato e gestito**. Progetti software con **budget e tempistiche limitate** avranno bisogno di un processo software **più snello** rispetto a progetti con budget e tempistiche flessibili
- **Rischio**: Il **rischio** associato ad un progetto software influenza il **livello di rigore e controllo** necessari nel processo di sviluppo. Maggiore è il fattore di rischio, maggiore sarà il **rigore** che il processo software dovrà avere.

2.2.1 Integrazione e configurazione

Questo modello è basato sul **riutilizzo del software**, in cui i sistemi sono **integrati da componenti o sistemi applicativi esistenti** (talvolta chiamati **sistemi COTS**: *commercial-off-the-shelf*). Gli elementi riutilizzati possono essere **configurati** in modo da adattarli alle esigenze del cliente. Il riutilizzo è oggi il sistema standard per la costruzione di molti tipi di software aziendali.
Quali sono però i tipi di "software riutilizzabile"?

- Sistemi applicativi **stand-alone** (COTS) configurati per l'uso in un particolare ambiente
- **Collezioni di oggetti** sviluppate come **pacchetti** da integrare con un **framework** di componenti (es. .NET o J2EE)
- **Servizi web** sviluppati secondo lo **standard di servizio** e invocabili in maniera remota

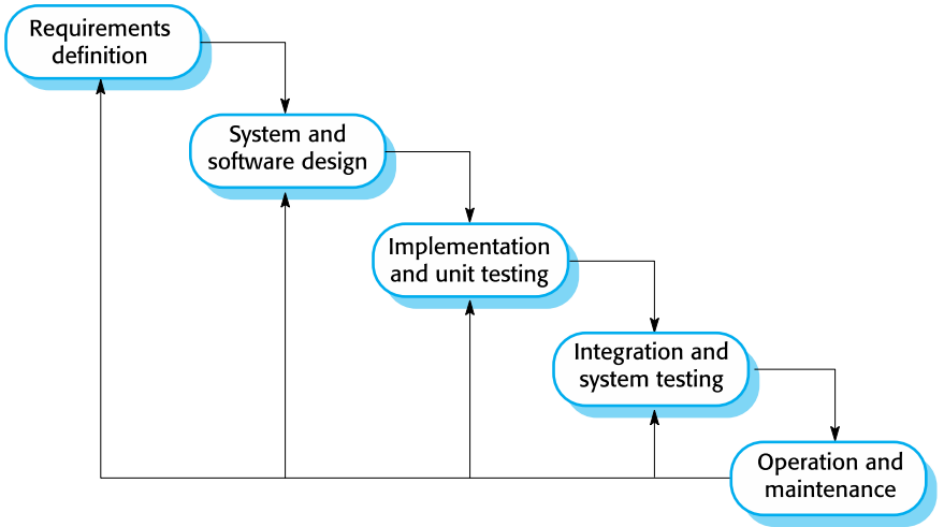
L'intero processo quindi **definisce un'ingegneria del software orientata al riuso**:



Vediamo ora i vantaggi e gli svantaggi di questo approccio:

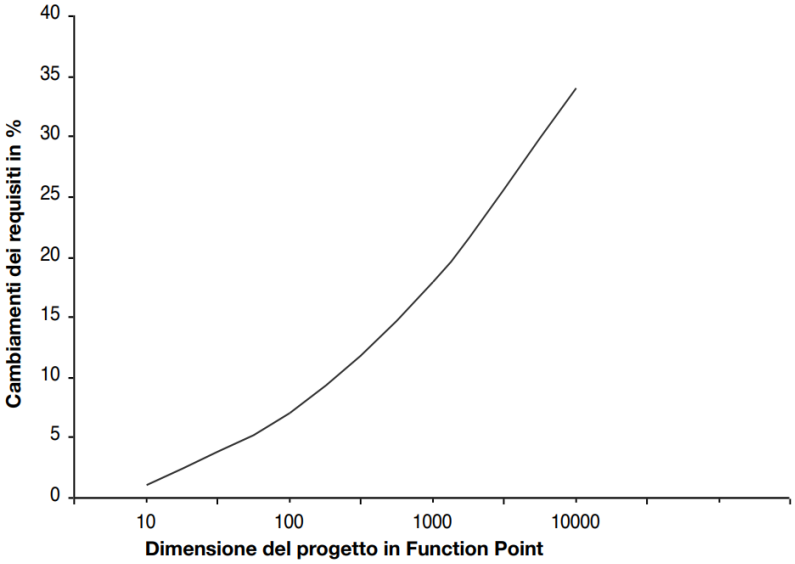
- **Vantaggio**: Riduzione dei costi e dei rischi, poiché **viene sviluppato meno software da zero**
- **Vantaggio**: Consegna più rapida del sistema al cliente
- **Svantaggio**: Si dovranno attuare dei **compromessi sui requisiti**, quindi il sistema potrebbe non soddisfare appieno le esigenze dell'utente
- **Svantaggio**: Perdita di controllo sull'**evoluzione** delle varie componenti che formano il sistema

2.2.2 Processo a cascata



Il processo a cascata è il processo software **più vecchio** tra quelli utilizzati ancora oggi. Il processo software con ciclo di vita a cascata (o **sequenziale**) è, in prima approssimazione, basato su uno svolgimento **sequenziale delle diverse attività di sviluppo del software**. All'inizio di un progetto, vengono definiti in dettaglio **tutti i requisiti** (o almeno la maggior parte di essi); allo stesso modo, più o meno all'inizio del progetto, si cerca di stabilire un **piano temporale dettagliato e "affidabile"** delle attività da svolgere (non è detto che lo sia). Poi si prosegue con la **modellazione** (analisi e progettazione) e viene creato un **progetto completo del software**. Solo a questo punto inizia la **programmazione del sistema software**, a cui seguiranno **verifica, rilascio e manutenzione**. Si noti come **ogni fase descritta inizia solo quando la precedente finisce**. Ad essere precisi, il processo a cascata **permette la possibilità di feedback e cicli tra le attività**, ma la maggior parte delle organizzazioni che applica questo processo considera di solito una **sequenzialità stretta** fra le varie fasi. Il principale svantaggio del processo a cascata è quindi quello di **avere difficoltà ad accogliere i cambiamenti a processo avviato**. Il processo a cascata, tuttavia, **risulta una pratica mediocre per la maggior parte dei progetti software** ([Larman03] e [LB03]): infatti il processo a cascata è associato ad una **percentuale elevata di fallimenti**. Perché quindi questo processo è così soggetto a frequenti fallimenti?

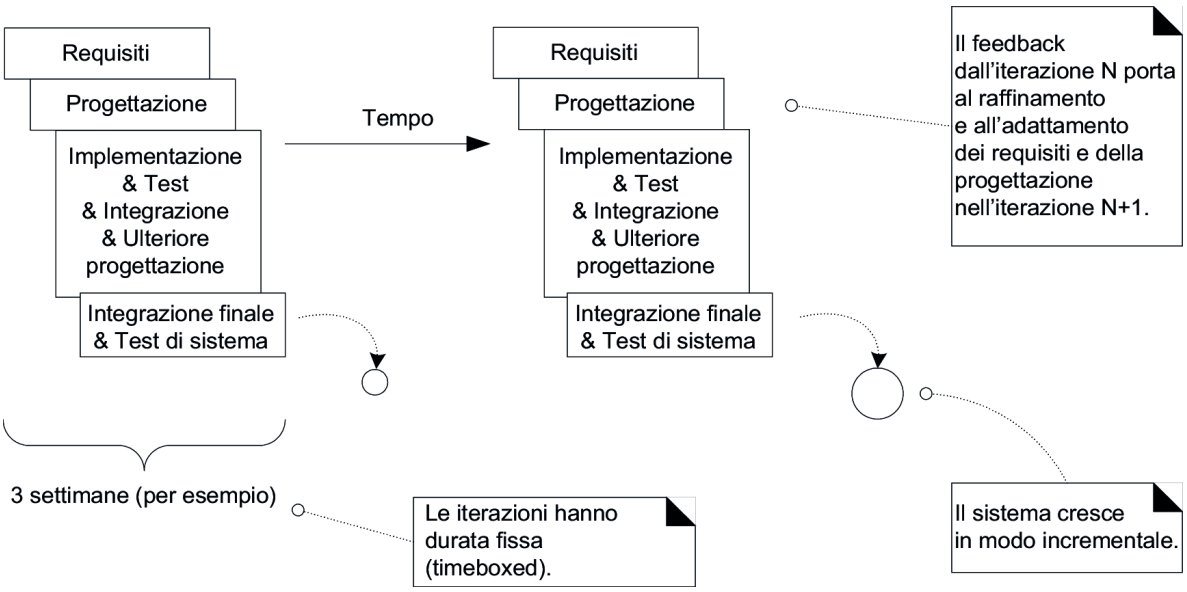
- La suddivisione **inflessibile** del progetti in **fasi distinte** rende difficile rispondere alle mutevoli esigenze di un cliente
  - Pertanto, questo modello è appropriato **solo quando i requisiti sono ben compresi e le modifiche saranno piuttosto limitate durante il processi di sviluppo**. Questo però non accade, come mostrato dal seguente grafico dello studio [Jones97]:



- Il modello a cascata viene utilizzato principalmente per progetti di **ingegneria dei sistemi di grandi dimensioni**, in cui un sistema viene sviluppato in diversi siti.
  - In questo caso, la natura pianificata del processo a cascata **aiuta a coordinare il lavoro**

2.2.3 Sviluppo iterativo, incrementale ed evolutivo

Una pratica fondamentale di molti processi software moderni (come UP e SCRUM) è lo **sviluppo iterativo**. In questo approccio al ciclo di vita, lo sviluppo è suddiviso in **una serie di mini progetti** dalla durata temporale fissa (es. 3 settimane, si dicono quindi **timeboxed**) chiamate **iterazioni**; il risultato di ciascuna iterazione è **un sistema eseguibile, testato e integrato, ma parziale**. Ciascuna iterazione prevede le proprie fasi di **analisi dei requisiti, progettazione, implementazione e test**. Il ciclo di vita iterativo si basa sul **susseguirsi di ampliamenti e raffinamenti** di un sistema nel corso di **molteplici iterazioni**, con **feedback e adattamenti ciclici** come guide essenziali per **convergere verso un sistema appropriato**. Il sistema quindi **cresce in modo incrementale** nel tempo. Poiché il feedback e l'adattamento fanno **evolvere il sistema nel tempo**, questo processo si dice anche **evolutivo**.



Quindi lo sviluppo iterativo, incrementale ed evolutivo si basa su un atteggiamento di **accettazione del cambiamento** e sull'**adattamento** come guide **inevitabili e di fatto essenziali**. Tuttavia questo non significa che questo processo supporti uno sviluppo **caotico** e che gli sviluppatori continuino a cambiare direzione in base alle richieste estemporanee del cliente ("feature creep"). Una via di mezzo **è possibile**:

- Ciascuna iterazione comporta la scelta di un **di un piccolo sottoinsieme di requisiti, una rapida progettazione, implementazione e test**. Seppur nelle iterazioni iniziali ciò che si produce **sarà lontano da ciò che si vuole ottenere**, ciò permette al cliente di dare feedback in maniera **rapida e precoce**; i quali potranno essere **analizzati dal team di lavoro** per ottenere delle **indicazioni pratiche e significative**; ciò permette al team anche di avere un'opportunità di **modificare o adattare la comprensione dei requisiti e il progetto**. Oltre al chiarimento dei requisiti, attività quali i **test di carico** dimostreranno se il progetto e l'implementazione parziale sono nella direzione giusta o se è necessaria una modifica dell'architettura.
- Il lavoro procede mediante una serie di **cicli strutturati di costruzione-feedback-adattamento**
- Nel tempo, attraverso il **feedback iterativo** e **l'adattamento**, il sistema sviluppato **evolve** e **converge** verso i requisiti corretti e il progetto più appropriato
  - Non bisogna stupirsi se nelle prime iterazioni lo **scostamento** dal sistema desiderato è maggiore che in quelle successive
  - L'instabilità dei requisiti e del progetto **tende a diminuire nel tempo**, tuttavia nelle iterazioni finali è **difficile ma non impossibile** che si verifichi un **cambiamento significativo dei requisiti**

Quali sono quindi i **vantaggi** dello sviluppo iterativo, incrementale ed evolutivo?

- **Minore probabilità di fallimento del progetto**, **migliore produttività**, **percentuali più basse di difetti**

- Riduzione precoce, anziché tardiva, dei **rischi maggiori** (tecnici, requisiti obbiettivi ecc...)
- Progresso visibile **sin dall'inizio**
- **Feedback precoce**, coinvolgimento dell'utente e adattamento, che portano a un sistema che soddisfa al meglio le esigenze reali delle parti interessate
- **Gestione della complessità**, cioè il team non viene sopraffatto dalla "**paralisi da analisi**" o da **passi molto lunghi e complessi**
- Ciò che si apprende nel corso di un iterazione **può essere usato per migliorare le successive**

Tuttavia questo processo **non è privo di svantaggi**:

- Il processo **non è visibile**: i manager hanno bisogno di documenti **costanti** per tenere traccia del processo di sviluppo; tuttavia se il sistema continua a cambiare non è conveniente continuare a produrre documenti che riflettono ogni versione del sistema
- La struttura del sistema **tende a degradarsi** con l'aggiunta di nuovi incrementi: a meno che non si dedichi tempo e denaro al **refactoring** per migliorare il software, le aggiunte tendono a **corrompere la struttura del sistema**; quindi incorporare sempre più modifiche software diventa sempre più **difficile e costoso**

Lo sviluppo iterativo è basato sul fatto che nei sistemi complessi e mutevoli, il feedback e l'adattamento sono **incrementi chiave** per il successo:

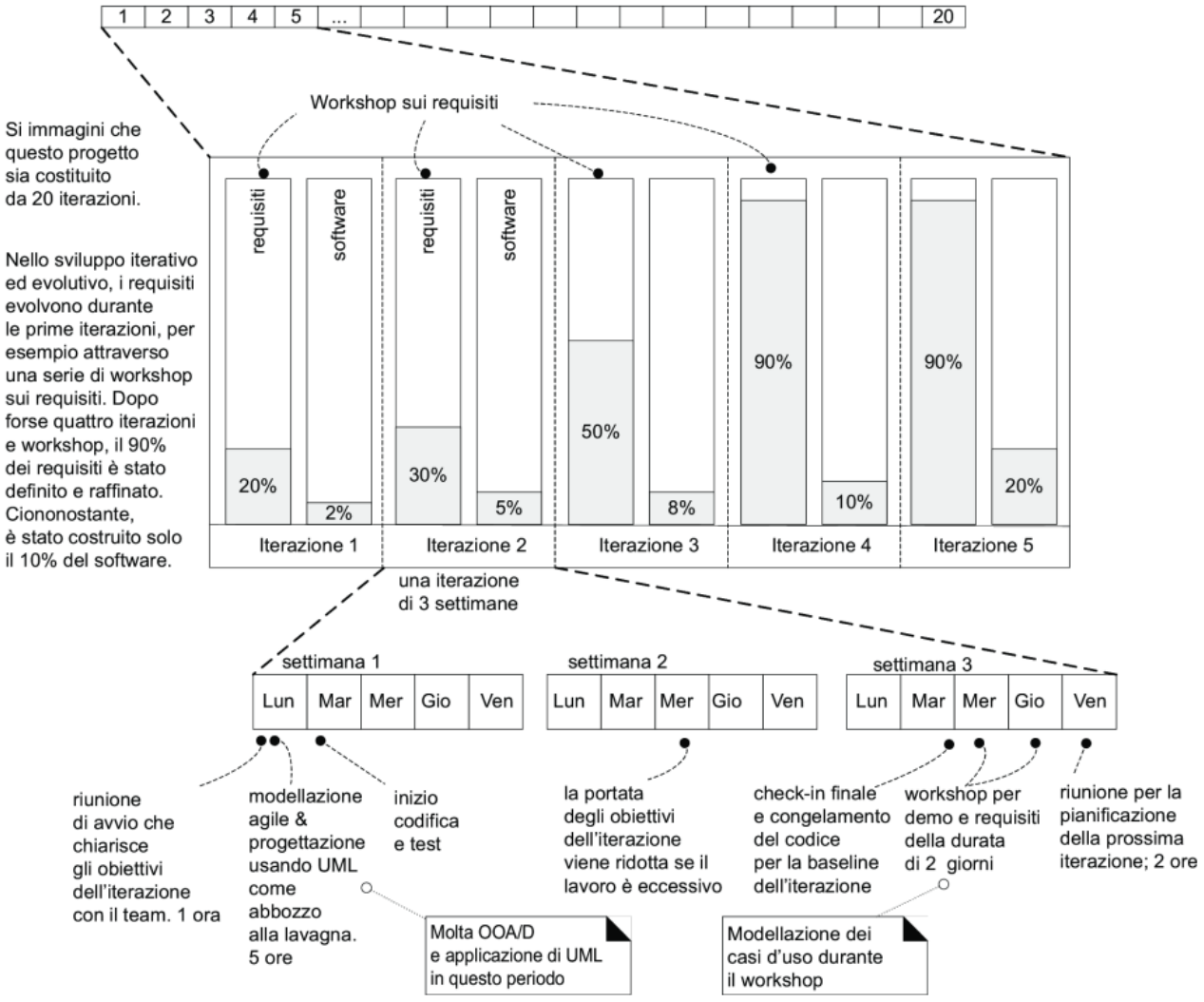
- Feedback proveniente dalle **attività iniziali di sviluppo**, dai **programmatori** che cercano di leggere le specifiche e da **dimostrazioni ai clienti** per raffinare i requisiti
- Feedback proveniente dai **test** e dagli **sviluppatori** che raffinano il progetto e i modelli
- Feedback circa **l'avanzamento del team** nell'affrontare le prime caratteristiche, per raffinare le **stime di tempo e di costi**
- Feedback proveniente dal **cliente e dal mercato** per assegnare/modificare le **priorità** alle caratteristiche da affrontare nell'iterazione successiva

Una pratica fondamentale dello sviluppo iterativo è quella di avere **iterazioni di lunghezza fissata**, cioè **timeboxed**; il periodo consigliato è dalle **due alle sei settimane**. Iterazioni più lunghe sono invece contrarie allo spirito dello sviluppo iterativo, poiché esso richiede **un feedback costante da parte del cliente**; con un periodo più lungo di sei settimane la complessità **cresce** e il feedback viene **ritardato**. Iterazioni più corte di due settimane invece **difficilmente permettono di sviluppare abbastanza software** per avere un feedback significativo. In questo processo di sviluppo **non è consentito ritardare la fine di un'iterazione**: se risulta difficile portare a termine tutti i requisiti che si erano previsti per una particolare iterazione, è meglio **spostarli**

all'iterazione successiva piuttosto che modificare la durata dell'iterazione. Un'iterazione di durata fissa è detta **timeboxed**.
Bisogna anche fare attenzione che il **pensiero a cascata non si infiltri nello sviluppo iterativo**; segni di questa infiltrazione sono:

- Si è scritto **la maggior parte dei requisiti o dei casi d'uso** prima dello sviluppo
- Si è creato in modo **dettagliato e completo delle specifiche, dei modelli o il progetto** prima di iniziare l'implementazione

L'adozione dello sviluppo iterativo richiede che il software venga realizzato in modo **flessibile**, affinche l'impatto dei cambiamenti sia **il più basso possibile**. A tal fine il codice (e il progetto del software) devono essere **facilmente modificabili**. Per facilitare questo aspetto il codice deve essere quindi **leggibile e facilmente comprensibile**, la mancanza di questa qualità infatti rende difficile implementare i cambiamenti in modo incrementale; inoltre è necessario usare degli **strumenti metodologici opportuni**; per esempio **tecnologie ad oggetti, sviluppo guidato dai test e refactoring**. Vediamo un esempio visuale sul come avviene uno sviluppo iterativo, incrementale ed evolutivo se assumiamo 20 iterazioni (assumiamo di star seguendo UP):



Un'attività critica dello sviluppo iterativo è la **pianificazione delle iterazioni**, cioè la definizione delle **attività da svolgere in ogni iterazione**. Se si sta seguendo un processo iterativo, è necessario evitare di **tentare di pianificare l'intero progetto in modo dettagliato sin dalla prima iterazione**. Piuttosto, i processi iterativi promuovono una **pianificazione iterativa** (o adattiva), in cui in ciascuna iterazione viene stabilito il **piano di lavoro dettagliato di una singola iterazione**. In UP, la pianificazione viene effettuata alla **fine dell'iterazione corrente** per decidere le attività della **seguente iterazione**. In SCRUM, la pianificazione viene effettuata **all'inizio dell'iterazione** per stabilire il piano dell'iterazione **corrente**. Lo sviluppo iterativo promuove la pianificazione **guidata dal rischio e guidata dall'utente**. Ciò significa che gli obbiettivi delle iterazioni iniziali vengono scelti

1. Per **identificare e attenuare i rischi maggiori**
2. Per **costruire e rendere visibili** le caratteristiche a cui il cliente tiene di più

In particolare, la progettazione guidata dal rischio contiene in sè la pratica dello **sviluppo centrato sull'architettura**: le prime iterazioni si concentreranno sulla **costruzione, test e la stabilizzazione del nucleo dell'architettura**. Infatti, è un rischio molto alto **non avere un'architettura di base solida**.
Importante per il processo iterativo è il **non cambiare gli obbiettivi dell'iterazione**: durante ciascuna iterazione, i requisiti su cui operare **vengono prima fissati** (pianificazione iterativa) e poi **bloccati**, cioè non sono più modificabili. Durante ciascuna iterazione, quindi, il team **può lavorare al suo meglio**, poiché:

- I requisiti sono **bloccati**, quindi il team non può essere **nè interrotto ne disturbato durante l'iterazione**
- I committenti possono interagire con il team di sviluppo **solo alla fine dell'iterazione**

Durante un'iterazione è tuttavia possibile che il team di sviluppo decida di **cambiare il piano dell'iterazione**, per esempio quando valuta, a metà dell'iterazione, la possibilità di raggiungere gli obbiettivi prefissati nella durata prevista.

3 Unified Process (UP)

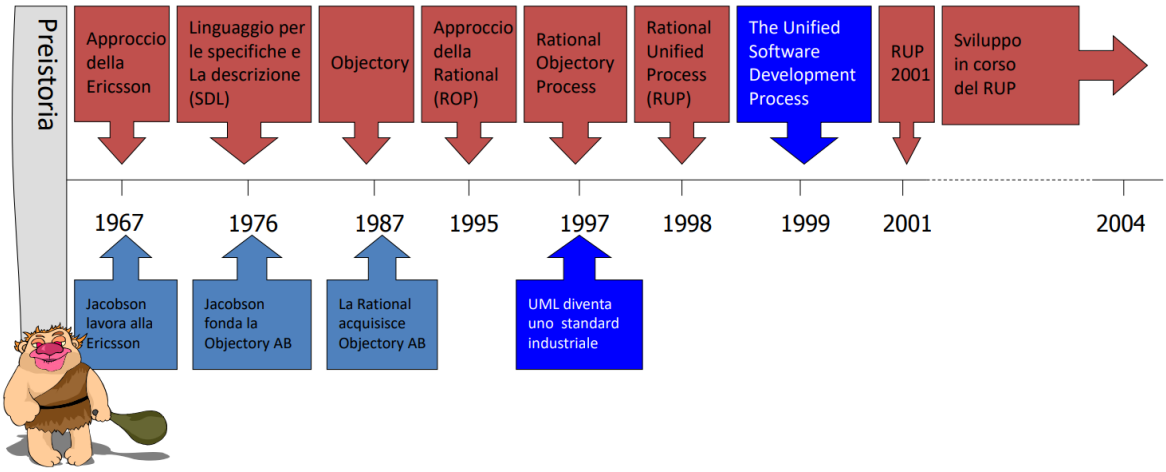
Il **processo unificato** (unified process) o **UP** è un processo iterativo diffuso per lo sviluppo software orientato agli oggetti. UP è molto **flessibile e aperto**: incoraggia infatti l'uso di **altre pratiche** prese da **altri processi iterativi**, come SCRUM o Extreme Programming. UP è:

- **Pilotato dai casi d'uso** (requisiti) e dai **fattori di rischio**
- Incentrato **sull'architettura**
- **Iterativo, incrementale ed evolutivo**

L'idea principale da apprezzare e praticare in UP è lo **sviluppo iterativo, evolutivo e incrementale** con **timeboxing breve**. Ulteriori best practices e concetti chiavi di UP sono:

- Affrontare le problematiche di **rischio maggiore** e valore elevato nelle **iterazioni iniziali**
- Impegnare gli utenti **continuamente** sulla valutazione, il feedback e i requisiti
- Creare un'architettura **coesa** nelle iterazioni iniziali
- Verificare continuamente le **qualità**: testare **spesso, presto e in modo realistico**
- Applicare i **casi d'uso**, se appropriato
- Fare della **modellazione visuale** (con UML)
- Gestire attentamente i requisiti
- Gestire le richieste di cambiamento e le configurazione

Vediamo una sua breve storia:



3.1 Iterazioni e discipline

Le iterazioni sono **concetti chiave** in UP. Esse sono come un mini-progetto che include:

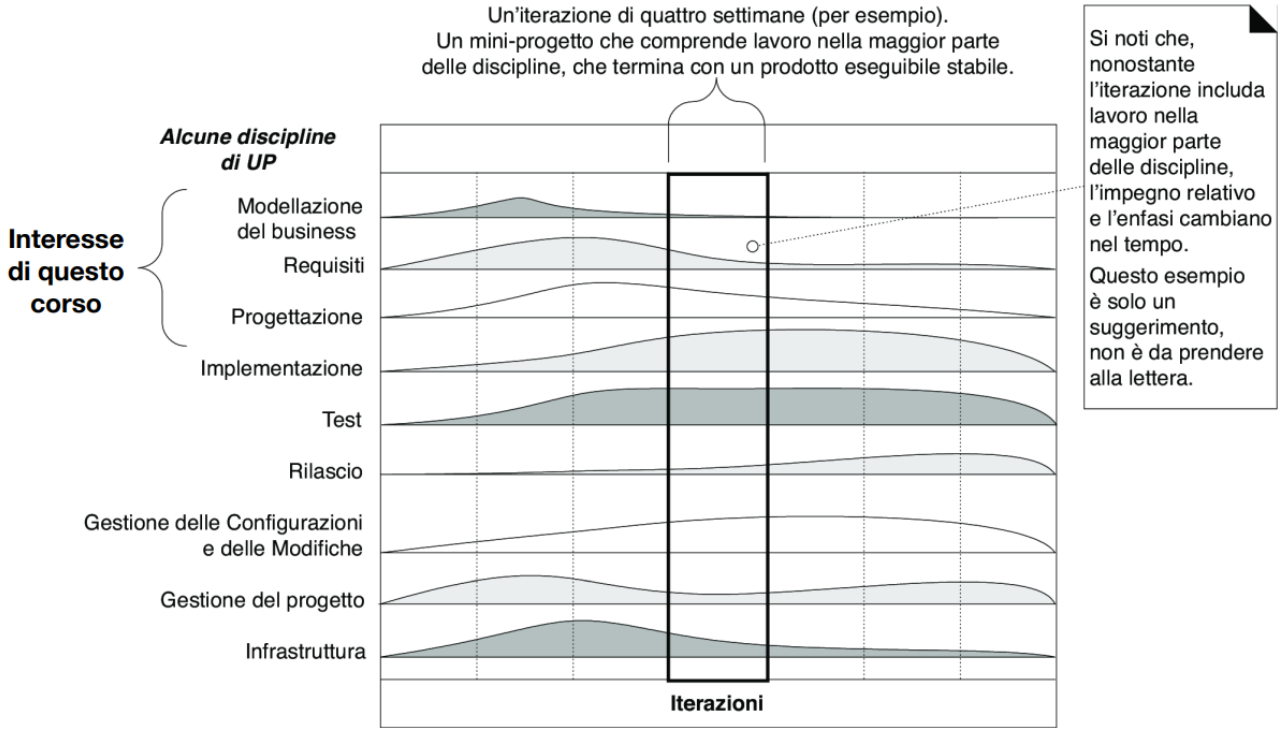
- **Pianificazione**
- **Analisi e progettazione**
- **Costruzione**
- **Integrazione e test**
- **Un rilascio**

Poiché UP è un processo iterativo, si arriva al **rilascio finale** dopo una **serie di iterazioni**. Le iterazioni **possono sovrapporsi**; ciò permette lo **sviluppo parallelo e il lavoro flessibile in grandi squadre**. Tuttavia richiede un'attenta pianificazione. UP colloca le attività lavorative in **discipline**; una disciplina è **un insieme di attività e dei relativi elaborati in una determinata area**, come, per esempio, l'area

dell'analisi dei requisiti. In UP, un **elaborato** è il termine generico con cui si fa riferimento ad un qualsiasi **prodotto di lavoro** (codice, schema di basi di dati, ecc...). UP definisce diverse discipline ed elaborati, ma noi ci concentreremo su:

- **Modellazione di business**: L'elaborato **Modello di dominio**, per visualizzare i concetti significativi nel dominio di applicazione
- **Requisiti**: Gli elaborati **Modello dei casi d'uso e Specifica supplementare**, per descrivere i **requisiti funzionali e non funzionali**
- **Progettazione**: L'elaborato **Modello di progetto**, per il progetto degli oggetti software

Un elenco più ampio è il seguente:



Come si può vedere sopra, anche se ogni iterazione può prevedere **tutti i flussi di lavoro**, la collocazione dell'iterazione all'interno del ciclo di vita del progetto **determina una maggiore enfasi** su uno dei flussi di lavoro.

3.1.1 Release

Ogni iterazione **genera una release**: una release è un **insieme di manufatti**, previsti e approvati. Essa fornisce una **base approvata per le successive attività di analisi e sviluppo**. Un **incremento** è la **differenza** tra una release e la successiva. Costituisce quindi un passo in avanti verso il rilascio finale del sistema.

3.2 Fasi di UP