Analisi e Progettazione del Software

spitfire

A.A. 2023-2024

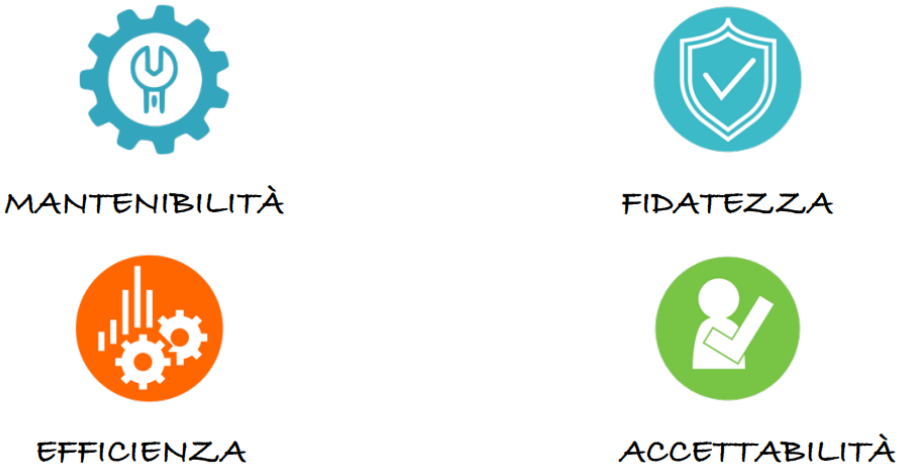
Contents

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:

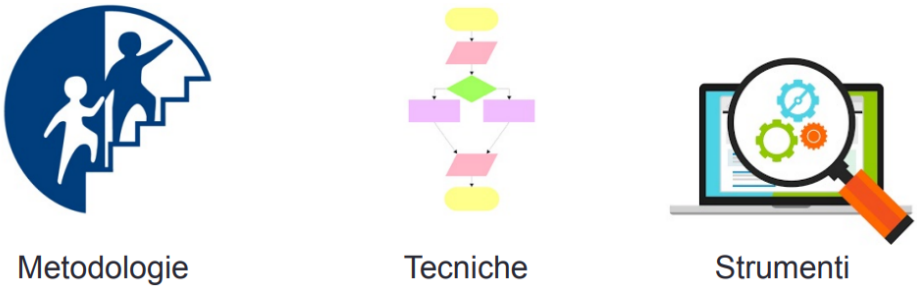


1.1 Introduzione all'ingegneria del software

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**":

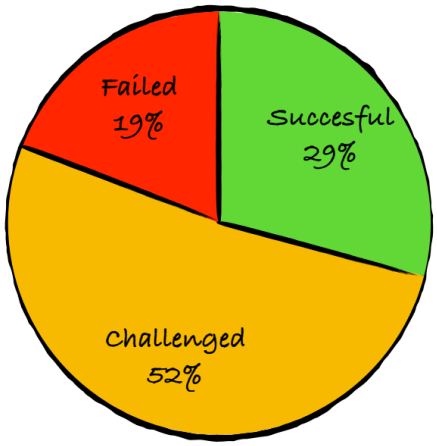
- **Disciplina ingegneristica**: Utilizzare metodi e teorie **appropriati** per risolvere i problemi tenendo conto dei vincoli **organizzativi e finanziari**
- **Tutti gli aspetti della produzione del software**: Non solo il **processo tecnico di sviluppo**. Anche la **gestione del progetto** e lo sviluppo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

La disciplina dell'ingegneria del software si occupa di:

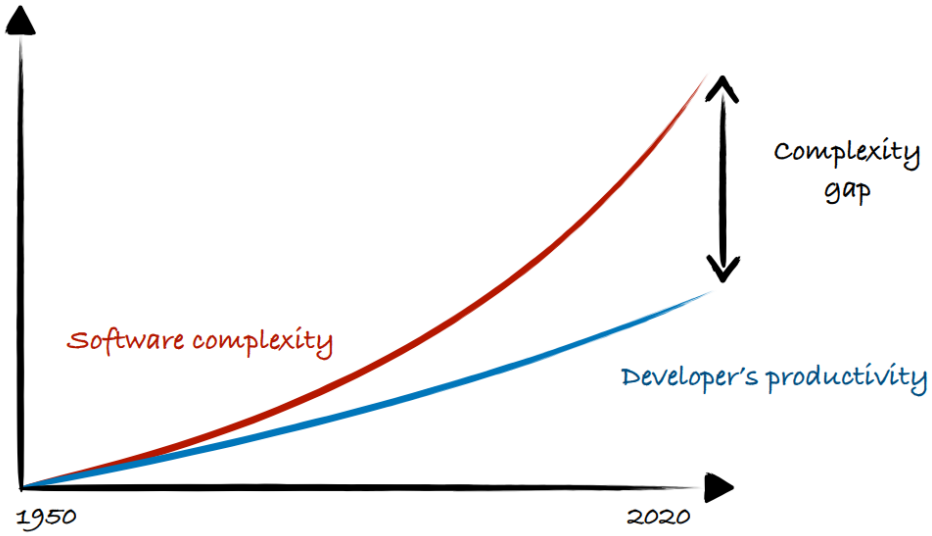


1.2 La crisi del software

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:

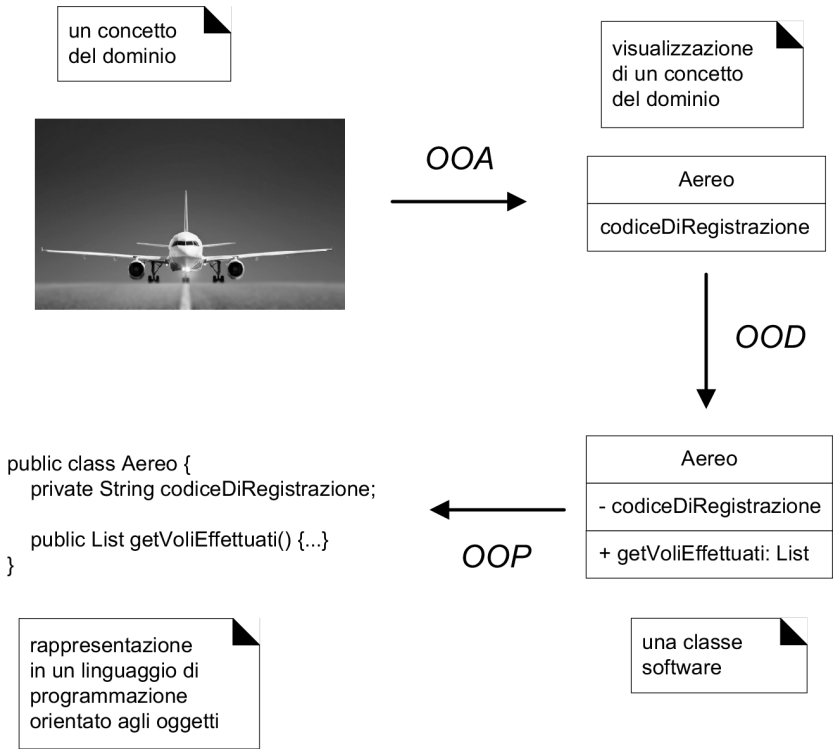
- **Progettazione orientata agli oggetti**
- **Progettazione di basi di dati**

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

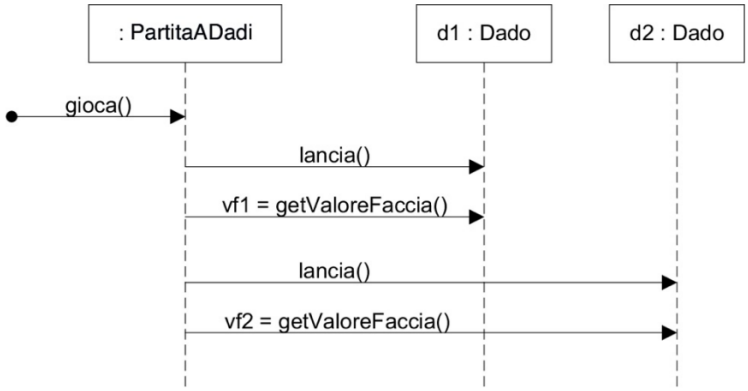
L'analisi dei requisiti può comprendere **storie o scenari** relativi al modo in cui l'applicazione può essere utilizzata dagli utenti; queste storie possono essere scritte come **casi d'uso**. I casi d'uso **non sono un elaborato ad oggetti** ma semplicemente delle storie scritte. Sono tuttavia uno 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:

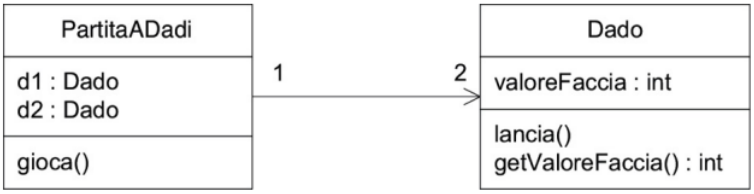


È interessante notare come **la progettazione degli oggetti software e dei programmi si può ispirare a un dominio del mondo reale**, tuttavia essa non è **nè un modello diretto nè una simulazione di questo dominio**. Quindi, per esempio, seppur nel mondo reale è il giocatore a lanciare il dado, nel progetto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

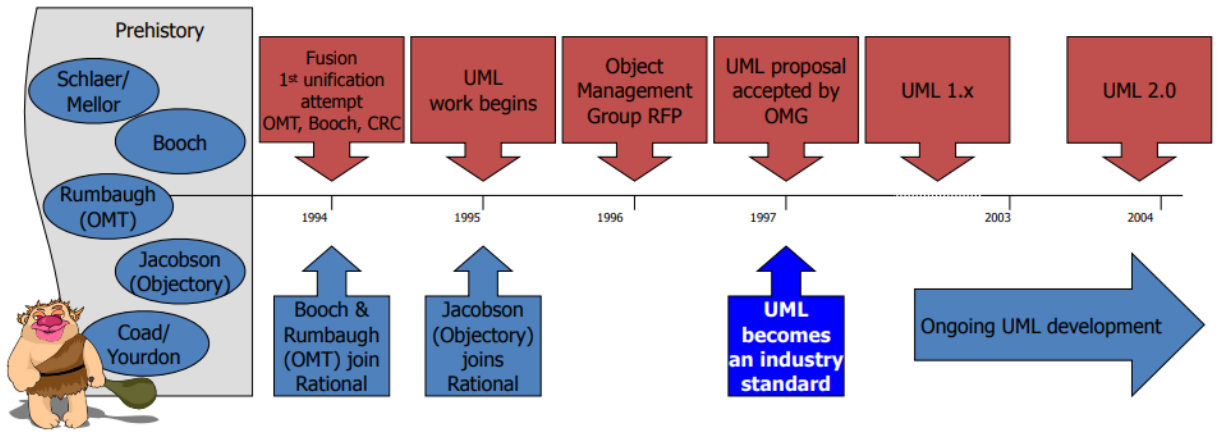
i loro attributi e metodi. Il diagramma delle classi di progetto è rappresentato nella seguente maniera:



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).

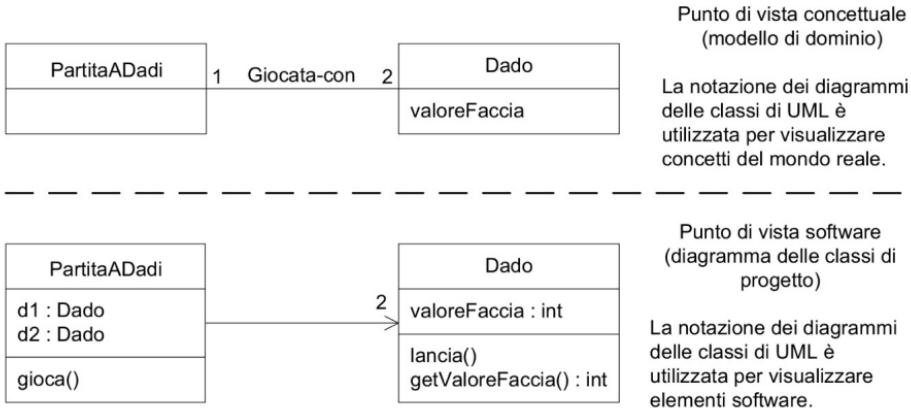
1.5.3 Due punti di vista per applicare UML

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**.

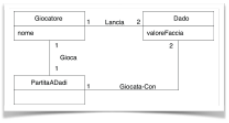
2 Processi per lo sviluppo del software

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

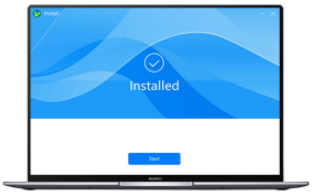


Analisi

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:

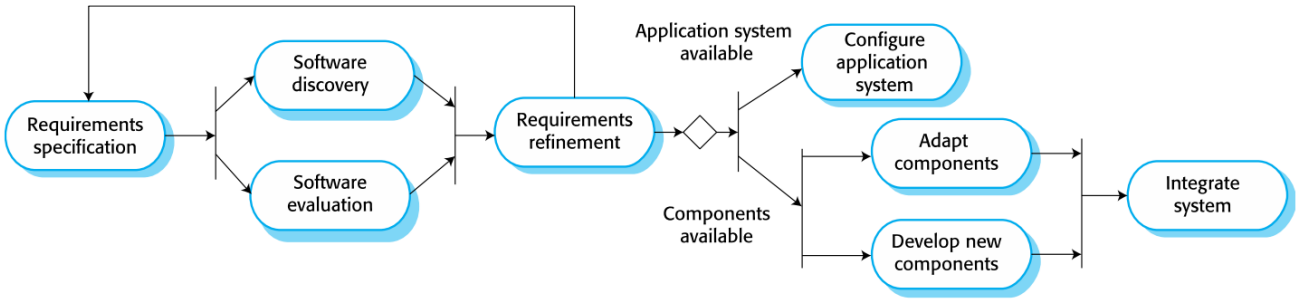
- **Complessità del progetto**: I progetti software possono variare **in termini di complessità**, da semplici applicazioni a sistemi complessi e "mission-critical". La complessità di un progetto **influenza il livello di formalismo e struttura** necessari nel processo di sviluppo
- **Dimensione del 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

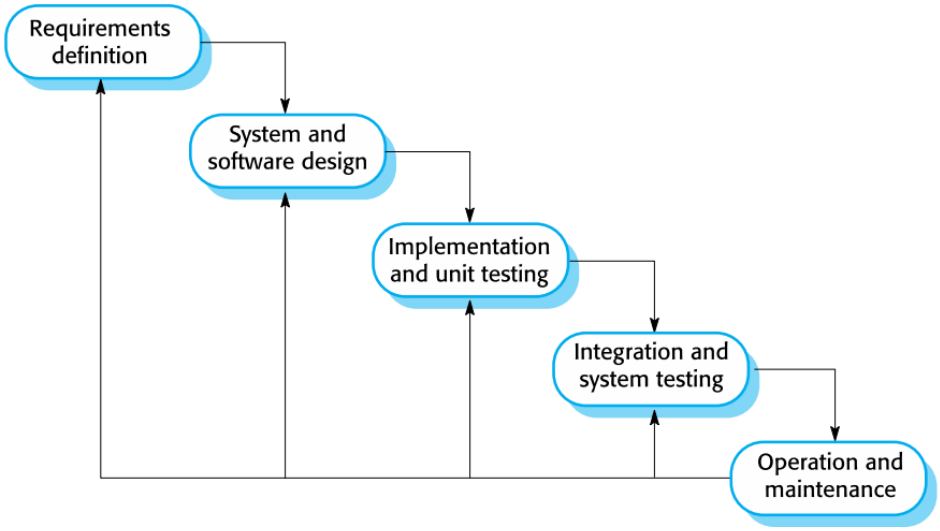
L'intero processo quindi **definisce un'ingegneria del software orientata al riuso**:



Vediamo ora i vantaggi e gli svantaggi di questo approccio:

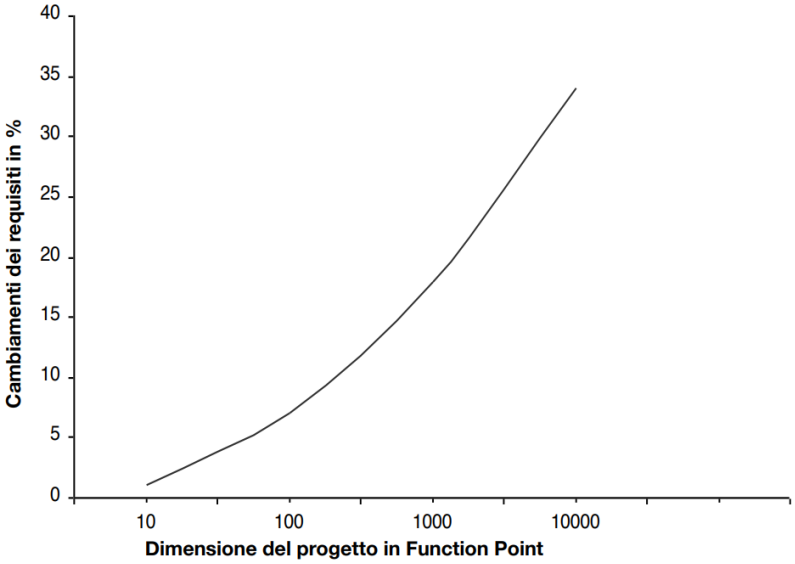
- **Vantaggio**: Riduzione dei costi e dei rischi, poiché **viene sviluppato meno software da zero**
- **Vantaggio**: Consegna più rapida del sistema al cliente
- **Svantaggio**: Si dovranno attuare dei **compromessi sui requisiti**, quindi il sistema potrebbe non soddisfare appieno le esigenze dell'utente
- **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?

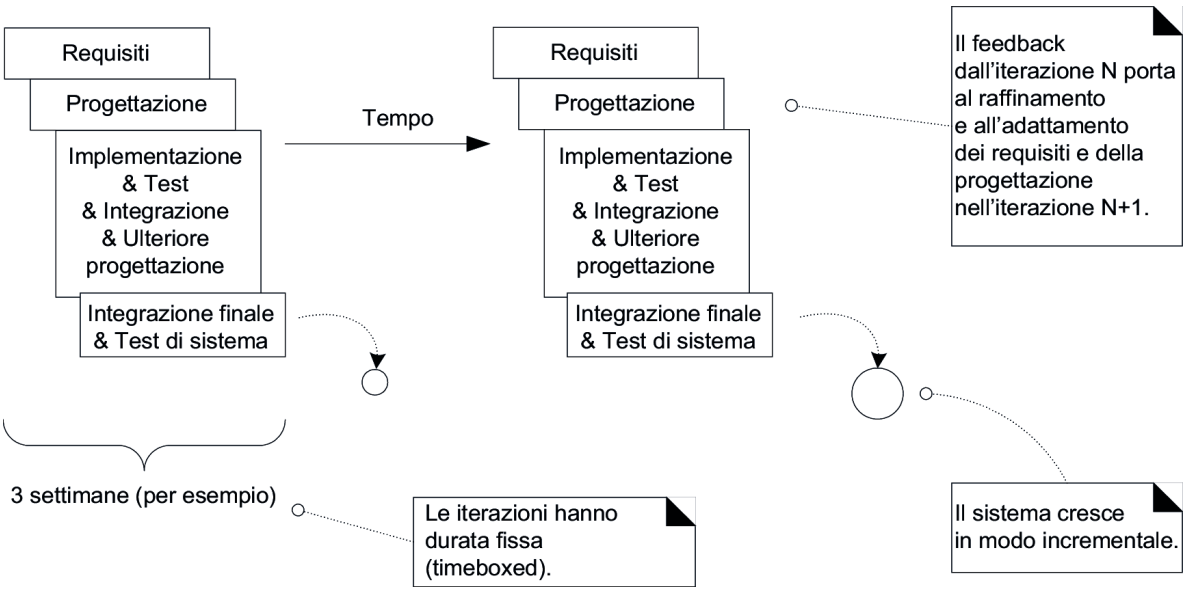
- La suddivisione **inflessibile** del progetti in **fasi distinte** rende difficile rispondere alle mutevoli esigenze di un cliente
  - Pertanto, questo modello è appropriato **solo quando i requisiti sono ben compresi e le modifiche saranno piuttosto limitate durante il processi di sviluppo**. Questo 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:

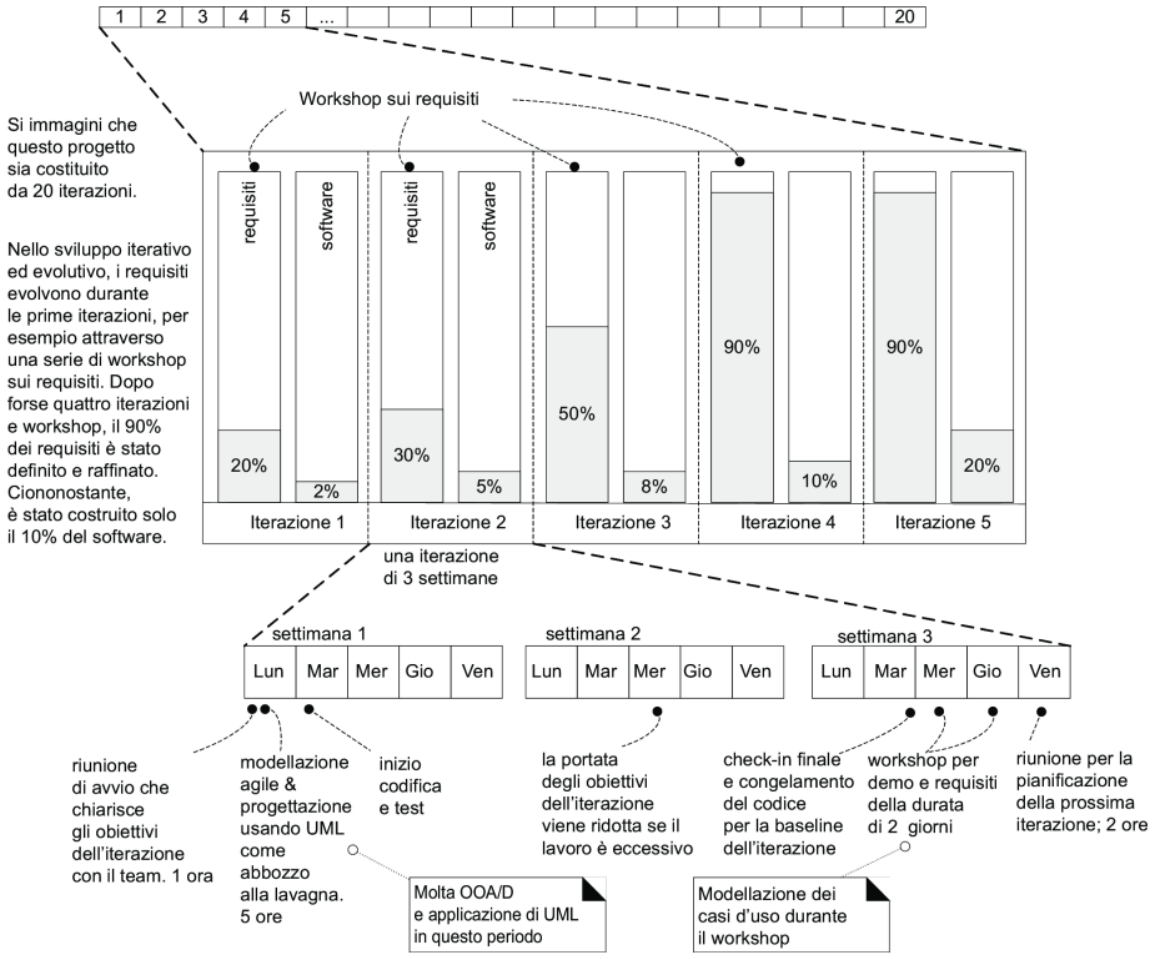
- Feedback proveniente dalle **attività iniziali di sviluppo**, dai **programmatori** che cercano di leggere le specifiche e da **dimostrazioni ai clienti** per raffinare i requisiti
- Feedback proveniente dai **test** e dagli **sviluppatori** che raffinano il progetto e i modelli
- Feedback circa **l'avanzamento 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)

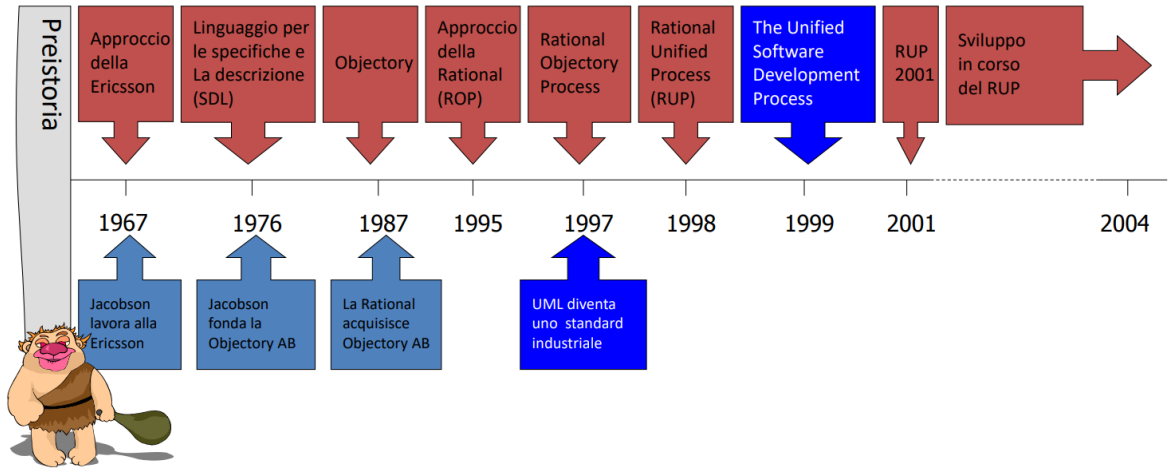
Il **processo unificato** (unified process) o **UP** è un processo iterativo diffuso per lo sviluppo software orientato agli oggetti. UP è molto **flessibile e aperto**: incoraggia infatti l'uso di **altre pratiche** prese da **altri processi iterativi**, come SCRUM o Extreme Programming. UP è:

- **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:

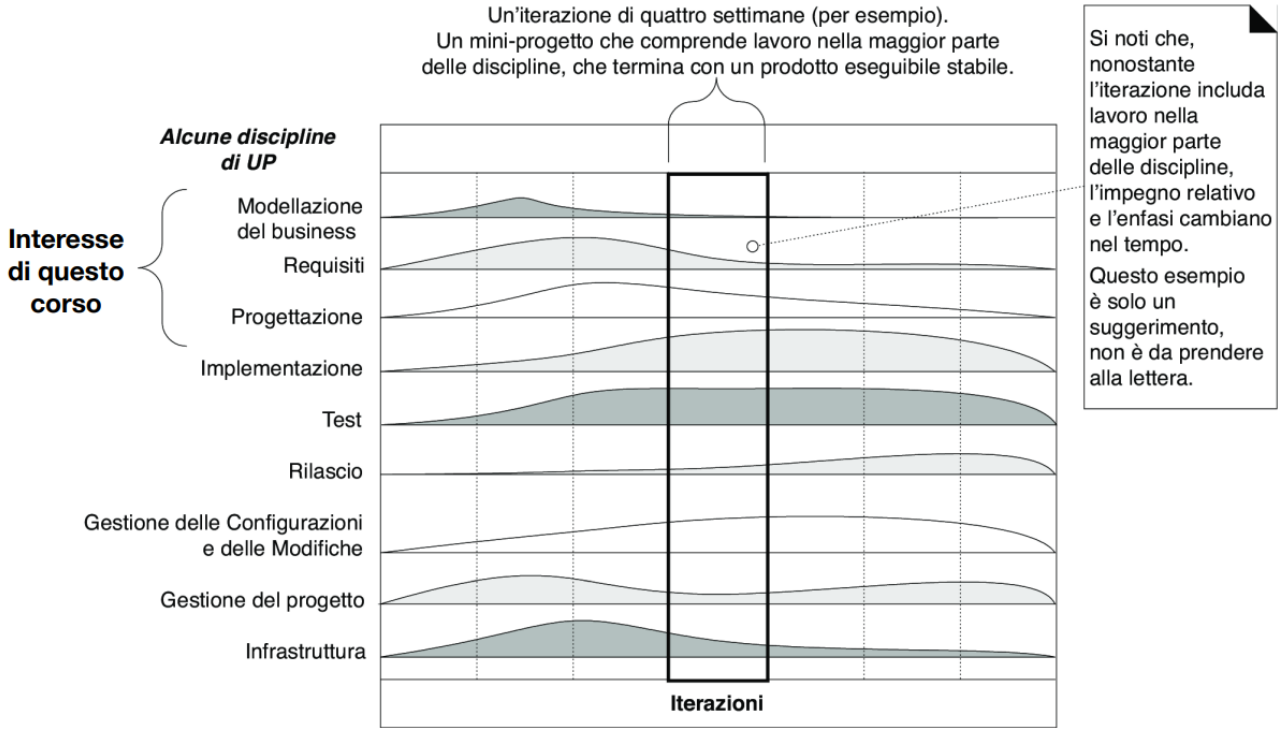
- **Pianificazione**
- **Analisi e progettazione**
- **Costruzione**
- **Integrazione e test**
- **Un rilascio**

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

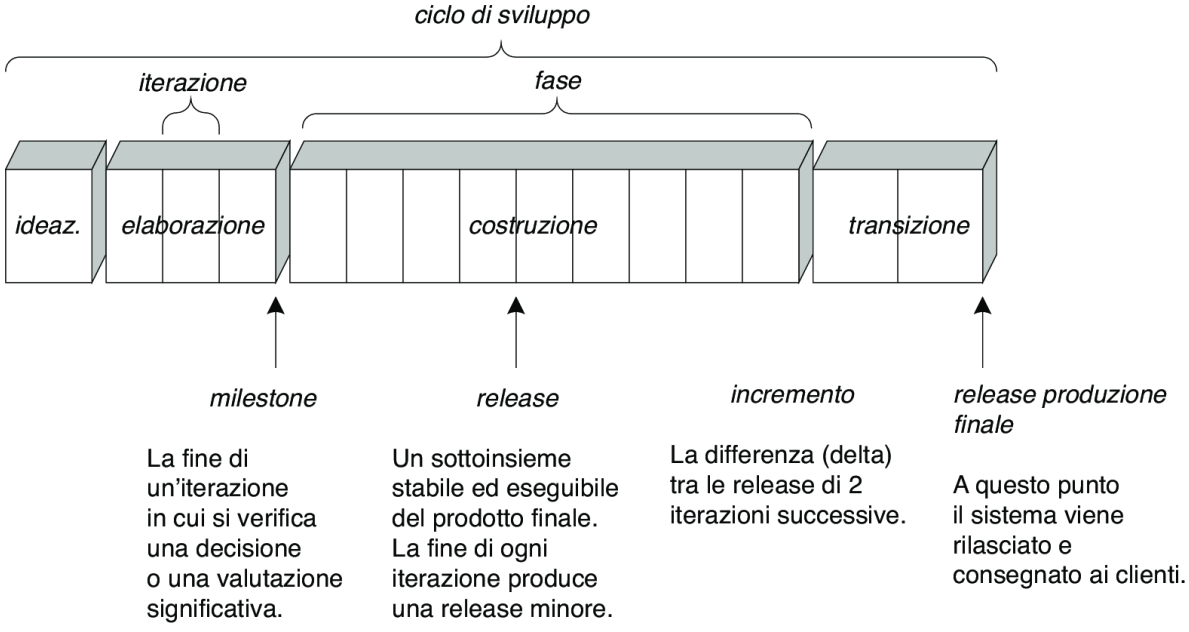
Ogni iterazione **genera una release**: una release è un **insieme di manufatti**, previsti e approvati. Essa fornisce una **base approvata per le successive attività di analisi e sviluppo**. Un **incremento** è la **differenza** tra una release e la successiva. Costituisce quindi un passo in avanti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

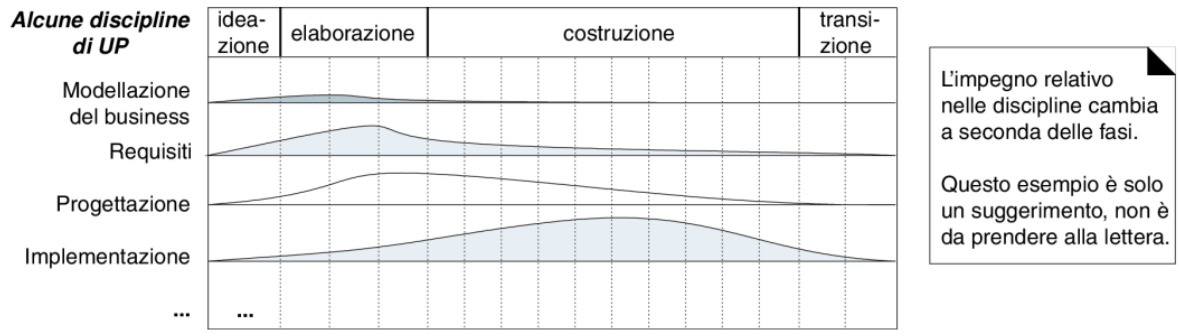
3.2 Fasi di UP

Un progetto UP organizza il lavoro in quattro **fasi temporali principali e successive**:

1. **Ideazione**: Visione **approssimativa**, studio economico, porta, stime **approssimative** dei costi e dei tempi
2. **Elaborazione**: Visione **raffinata**, implementazione **iterativa** del **nucleo dell'architettura**, risoluzione dei rischi maggiori, identificazione della **maggior parte** dei requisiti, stime più realistiche.
3. **Costruzione**: Implementazione **iterativa** degli elementi rimanenti, più facili e a rischio minore; preparazione al rilascio
4. **Transizione**: beta test, rilascio finale

Si noti che questo modello **non è a cascata**: l'ideazione **non è una fase di requisiti**; piuttosto è una breve fase di **fattibilità** in cui viene eseguita un'indagine **sufficiente** a sostenere la decisione di continuare o interrompere il progetto. Allo stesso modo, la fase di **elaborazione** non è una fase dei requisiti o dell'implementazione; piuttosto è una fase in cui **viene implementato il nucleo dell'architettura** e si risolvono i rischi maggiori. Vediamo un'esempio visuale:





3.3 Scenari di sviluppo di UP

Tra gli elaborati e le pratiche di UP, **quasi tutto è opzionale**

- Alcune pratiche e principi di UP **sono fissi**, come lo **sviluppo iterativo e guidato dal rischio** e **il controllo continuo della qualità**
- Tutte le attività e gli elaborati sono **opzionali**, con l'ovvia esclusione del codice

La scelta delle pratiche degli elaborati UP per un progetto può essere scritta in un breve documento chiamato **scenario di sviluppo**

Disciplina	Pratica	Elaborato	Ideazione	Elaboraz	Costr	Transiz
		Iterazione	I1	E1..En	C1..Cn	T1..T2
Modellazione del business	modellazione agile	Modello di Dominio		i		
	workshop requisiti					
Requisiti	workshop requisiti	Modello dei Casi d'Uso	i	r		
	esercizio sulla visione	Visione	i	r		
	votazione a punti	Specifica Supplementare	i	r		
		Glossario	i	r		
Progettazione	modellazione agile	Modello di Progetto		i	r	
	sviluppo guidato dai test	Documento dell'Architettura Software		i		
		Modello dei Dati		i	r	

i=Inizio, r=raffinamento

4 Metodologie e processi agili

Lo sviluppo agile è una forma di **sviluppo iterativo** che incoraggia *l'agilità*, ovvero una risposta **rapida e flessibile** ai cambiamenti. Le pratiche agili, come **Agile Modelling**, sono fondamentali per applicare UML correttamente. I metodi di sviluppo agile di solito applicano lo sviluppo **iterativo ed evolutivo**, con iterazioni brevi e timeboxed, fanno uso della **pianificazione iterativa**, promuovono le **consegne incrementali** e comprendono altri valori che incoraggiano l'agilità. Non è possibile dare una definizione di "**metodo agile**", poiché le pratiche adottate variano da metodo a metodo. Tuttavia una pratica di base, adottata da tutti i metodi, è quella che prevede **iterazioni brevi**, con un **raffinamento evolutivo dei piani, dei requisiti e del progetto**. Inoltre, essi promuovono pratiche e principi che riflettono una **sensibilità agile** per **la semplicità,**

la leggerezza, la comunicazione, i gruppi di lavoro auto-organizzanti e altro. Qualsiasi metodo iterativo **può essere applicato in maniera agile**, quindi anche UP, il quale **incoraggia l'inclusione di pratiche da metodi agili**.

4.1 Agile Modelling

L'atto puro della modellazione (creare diagrammi UML ecc...) deve essere un modo per **comprendere meglio il problema o lo spazio delle soluzioni**. Da questo punto di vista, lo scopo di "fare UML" non è quello di **creare e tradurre in codice i diagrammi prodotti**, ma piuttosto quello di **esplorare rapidamente le alternative e il percorso verso un buon progetto OO**. Questo modo di vedere, coerente con i metodi agili, è stato chiamato **modellazione agile** nel libro *Agile Modelling* [Ambler02]; esso è basato sulle seguenti pratiche e valori:

- Adottare un metodo agile **non significa evitare del tutto la modellazione**; infatti molti metodi agili hanno estensive parti di modellazione
- Lo scopo della modellazione e dei modelli è principalmente quello di **agevolare la comprensione e la comunicazione**, non di documentare.
- Non si deve modellare e applicare UML per **eseguire per intero o per la maggior parte la progettazione del software**. Si applichi UML sono alle parti del progetto che sono **difficili on insidiose**
- Si utilizzi lo strumento di modellazione **più semplice possibile**
- La modellazione **non è da fare da soli** ma in coppia (o a tre)
- Tenere presente che **ogni diagramma sarà incompleto e impreciso**
- Nell'abbozzo alla lavagna va usata una **notazione semplice e "abbastanza buona"**
- La modellazione per la progettazione OO dovrebbe essere fatta **dagli stessi programmatori che si occuperanno della programmazione**

4.2 UP Agile

UP non è stato pensato per essere **pesante o non agile**; anzi UP è stato concepito per essere adottato in uno **spirito di adattabilità e leggerezza**, come un **UP agile**. Ecco alcuni esempi di come ciò è possibile:

- Si preferisca un **insieme piccolo di attività** ed elaborati UP; bisogna quindi tenere presente quasi tutti gli elaborati di UP sono **opzionali**
- Dato che UP è **iterativo ed evolutivo**, i requisiti e la progettazione **non vengono completati prima dell'implementazione** ma emergono in modo **adattivo** durante una serie di iterazioni, anche sulla base dei feedback
- Si applichi UML **con le pratiche della modellazione agile**

- Non esiste un **piano dettagliato per l'intero progetto**. Esiste un piano di alto livello (chiamato il **piano delle fasi**) che stima la data della fine del progetto e di altre milestone principali, ma non descrive nel dettaglio i **passi a grana fine per raggiungere queste milestone**. Un piano dettagliato (chiamato il **piano dell'iterazione**) pianifica in maggior dettaglio un'unica iterazione: **la successiva**. La pianificazione dettagliata viene eseguita in modo adattivo da un'interazione all'altra

4.3 Scrum

Scrum[SB01] è un metodo agile che consiste di sviluppare e rilasciare prodotti software **con il più alto valore possibile per i clienti nel più breve tempo possibile**. Scrum si occupa principalmente dell'**organizzazione del lavoro e della gestione dei progetti** e meno agli aspetti tecnici dello sviluppo software; quindi lascia agli sviluppatori **libertà sulle tecnologie, sulle tecniche e sulle metodologie specifiche da utilizzare**. Di conseguenza può essere facilmente combinato con altri metodi.
Scrum è un approccio **incrementale e iterativo** allo sviluppo software; ciascuna iterazione, chiamata uno **Sprint**, ha una durata fissata, per esempio due settimane. Le iterazioni sono quindi **timeboxed**, e dunque non **non vengono mai estese**. In Scrum ci sono solo tre ruoli:

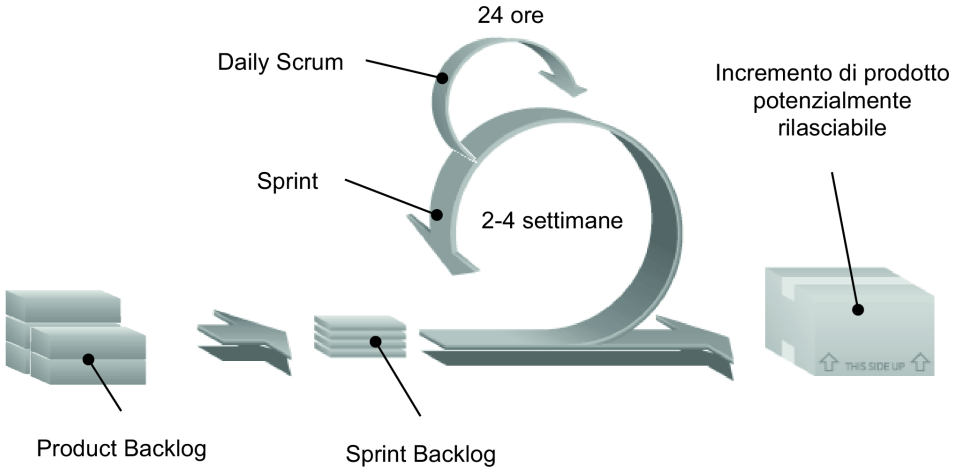
- **Product Owner**: Definisce le **caratteristiche** del prodotto software da realizzare e specifica le **le priorità** tra queste caratteristiche. Il suo obbiettivo è **massimizzare il valore del prodotto**.

- **Development Team**: è composto di solito da una **manciata di persone**, che possiedono le competenze necessarie per **sviluppare il software**. Il team è **auto-organizzato e auto-gestito** e opera con un alto grado di autonomia.

- **Scrum master**: aiuta l'intero gruppo ad **apprendere e applicare Scrum** al fine di ottenere il valore desiderato. Lo Scrum master **non è il manager del team**, ma piuttosto un istruttore e una guida, che serve, aiuta e protegge il Team.

Complessivamente tutti e tre formano un **Team Scrum**.
Il product owner definisce le **caratteristiche** del prodotto da realizzare nel **Product Backlog**, che è un insieme di **voci** (funzionalità e altri requisiti) ordinato per priorità. All'inizio del progetto, il Product Backlog descrive **tutte le caratteristiche del prodotto**; di iterazione in iterazione questo elaborato **viene aggiornato** e descrive le cose che devono essere ancora fatte per completare il prodotto. All'inizio di ciascuno Sprint, il team seleziona dal Product Backlog un insieme di voci da sviluppare durante quell'iterazione; questa scelta prende il nome di **Sprint Goal**, ovvero l'obbiettivo di sviluppo del team durante lo sprint corrente. Inoltre, il team compila lo **Sprint backlog** che specifica l'insieme dei compiti dettagliati per raggiungere lo Sprint Goal. Ogni giorno, all'inizio della giornata, il Team si riunisce **brevemente** in un **Daily Scrum** per verificare i propri progressi e per decidere i passi successivi necessari per completare il lavoro rimanente. Dopodiché, il team si mette al lavoro e **compie tutte le attività necessarie per l'analisi, la progettazione, la costruzione, l'integrazione e la verifica** del software. Il risultato di ciascuno Sprint deve essere un prodotto software

funzionante chiamato "**incremento di prodotto potenzialmente rilasciabile**". In linea con i principi di sviluppo **iterativo e incrementale**, il prodotto software parziale deve essere **funzionante e pienamente documentato per l'utente finale**. Le voci dello Sprint Backlog che sono state sviluppate in questo modo sono considerate **fatte**. Alla fine dello Sprint, nella **Sprint Review** il **Product Owner** e il **Team** presentano alle diverse parti interessate **l'incremento di prodotto software** che è stato sviluppato e ne fanno una **dimostrazione**. Lo scopo della Sprint Review è **ottenere un feedback su quanto è stato fatto**, anche per poter decidere che **cosa è utile fare nel prossimo Sprint**. Si procede così di Sprint in Sprint, in modo iterativo, fino a quando l'intero prodotto non è stato completato.



La caratteristica distintiva di Scrum tra i metodi agili è **l'enfasi sull'adozione di team auto-organizzanti e auto-organizzati**. Inoltre, Scrum è basato su un insieme di **elaborati ed eventi che hanno lo scopo di rendere visibili gli obbiettivi e il progresso delle iterazioni e di favorire un adattamento evolutivo del processo di sviluppo**. Facciamo un riassunto di alcune delle terminologie viste:

Termine Scrum	Definizione
Scrum	Un incontro giornaliero del team Scrum che esamina i progressi e definisce le priorità del lavoro da svolgere in quel giorno. Idealmente, questo dovrebbe essere un breve incontro faccia a faccia che includa l'intera squadra.
ScrumMaster	Lo ScrumMaster è responsabile di assicurare che il processo di Scrum sia seguito e guida il team nell'uso efficace di Scrum. È responsabile dell'interfacciamento con il resto dell'azienda e di assicurare che il team Scrum non venga deviato da interferenze esterne. Gli sviluppatori di Scrum sono convinti che lo ScrumMaster non debba essere considerato un project manager. Altri, tuttavia, potrebbero non sempre trovare facile vedere la differenza.
Sprint	Un'iterazione di sviluppo. Le sprint sono solitamente di 2-4 settimane.
Velocity	Una stima di quanto lavoro rimanente un team può fare in un singolo sprint. Capire la velocità di una squadra li aiuta a stimare ciò che può essere coperto in uno sprint e fornisce una base per misurare il miglioramento delle prestazioni.

Termine Scrum	Definizione
Team di sviluppo	Un gruppo auto-organizzatore di sviluppatori di software, che non dovrebbe superare le 7 persone. Sono responsabili dello sviluppo del software e di altri documenti essenziali del progetto.
Incremento potenzialmente rilasciabile	L'incremento software fornito da uno sprint. L'idea è che ciò dovrebbe essere "potenzialmente trasportabile", il che significa che si trova in uno stato finito e non è necessario alcun ulteriore lavoro, come il test, per incorporarlo nel prodotto finale. In pratica, ciò non è sempre realizzabile.
Product backlog	Questo è un elenco di elementi 'da fare'che il team Scrum deve affrontare. Possono essere definizioni di caratteristiche per il software, requisiti software, storie utente o descrizioni di compiti supplementari necessari, come la definizione dell'architettura o la documentazione utente.
Product owner	Un individuo (o eventualmente un piccolo gruppo) il cui compito è quello di identificare le caratteristiche o i requisiti del prodotto, dare la priorità a questi per lo sviluppo e rivedere continuamente la product backlog per garantire che il progetto continui a soddisfare le esigenze di business critiche. Il Product Owner può essere un cliente, ma potrebbe anche essere un product manager in una società di software o un rappresentante di altri stakeholder.

5 Iterazione 0: Analisi dei requisiti

L'iterazione 0 per gli **studi di caso** enfatizza i concetti fondamentali dell'**analisi dei requisiti**. Questa capacità costituisce un **prerequisito fondamentale** per l'analisi e la progettazione ad oggetti che verranno presentate nelle prossime iterazioni. Nello sviluppo iterativo è utile avere una "**iterazione 0**" iniziale, che ha lo scopo di **definire la visione del software da realizzare** e fornire una **stima approssimativa** dei tempi e dei costi. Questo passo iniziale corrisponde alla **fase di ideazione di UP**. Questa iterazione non ha quindi lo scopo di creare un sistema software eseguibile, ma si concentra soprattutto sull'avvio di altre attività ed elaborati, tra cui, appunto, **l'analisi dei requisiti**.

5.1 Ideazione

La maggior parte dei progetti richiede un breve passo iniziale dove si affrontano i seguenti tipi di domande:

- Qual'è la **visione** e qual'è lo **studio economico** per questo progetto?
- Il progetto è **fattibile**?
- **Comprare** e/o **costruire**?
- Stima **approssimativa e non affidabile** dei cosi (ordine di grandezza della spesa)
- Dovremmo procedere o fermarci?

In UP, l'ideazione è appunto il **breve** passo iniziale che permette di definire la visione del progetto e di ottenere una **stima**, approssimativa e non affidabile, dei **costi** e dei **tempi** di sviluppo. A tal fine, l'ideazione richiede *un po' di analisi dei requisiti*, come, per esempio, il 10% dei **requisiti funzionali** (casi d'uso) nonché l'analisi dei **requisiti non funzionali più critici**. È necessario tenere a mente che **lo scopo di questa fase non è quello di definire tutti i requisiti**, né quello di generare una stima o

un piano di progetto affidabili; se si sta facendo questo allora si sta sovrapponendo il **pensiero a cascata a UP**. L'idea è quella di **effettuare un'indagine sufficiente** per formarsi un'**idea razionale e giustificabile** sull'obbiettivo generale e sulla fattibilità del sistema software e decidere se **vale la pena di investire in un'indagine più approfondita** (scopo della fase di **elaborazione**). La maggior parte dell'analisi dei requisiti **avviene durante la fase di elaborazione, in parallelo alle prime attività di programmazione di qualità-produzione e di test**. Per la maggior parte dei progetti, la fase di ideazione dovrebbe essere **relativamente breve** (es. una settimana). Quindi:

- **Ideazione in una frase**: immaginarsi la portata del prodotto, la visione e lo studio economico
- **Il problema che risolve in una frase**: Le parti hanno un **accordo di base sulla visione del progetto**, e vale la pena investire su un'analisi più seria?

Lo scopo dell'ideazione è quindi quello di stabilire una visione iniziale comune per gli obbiettivi del progetto, stabilire se questo è fattibile e decidere se vale la pena di effettuare alcune indagini serie nell'elaborazione. Se è stato **deciso a priori che il progetto è fattibile** (per esempio perché il team ha già fatto progetti simili), allora la fase di elaborazione può essere anche **particolarmente breve**. Essa può comprendere il **primo workshop sui requisiti e la pianificazione per la prima iterazione**, per poi passare direttamente all'elaborazione.

5.1.1 Elaborati iniziati durante l'ideazione

Elaborato	Commento
Visione e Studio economico	Descrive gli obiettivi e i vincoli di alto livello, lo studio economico, e fornisce un sommario del progetto.
Modello dei Casi d'Uso	Descrive i requisiti funzionali. Durante l'ideazione vengono identificati i nomi della maggior parte dei casi d'uso, e circa il 10% dei casi d'uso viene analizzato in modo dettagliato.
Specifiche supplementari	Descrivono altri requisiti, per lo più non funzionali. Durante l'ideazione è utile avere un'idea dei requisiti non funzionali fondamentali che avranno un impatto significativo sull'architettura.
Glossario	Terminologia chiave del dominio e dizionario dei dati.
Lista dei Rischi e Piano di Gestione dei Rischi	Descrive i rischi (aziendali, tecnici, di risorse, di calendario) e le idee per attenuarli o rispondervi.
Prototipi e proof of concept	Per chiarire la visione e validare tecniche.
Piano dell'Iterazione	Descrive che cosa fare nella prima iterazione dell'elaborazione.
Piano delle Fasi e Piano di Sviluppo del Software	Ipotesi (poco precise) riguardo alla durata e allo sforzo della fase di elaborazione. Strumenti, persone, formazione e altre risorse.
Scenario di Sviluppo	Una descrizione della personalizzazione dei passi e degli elaborati di UP per questo progetto; UP viene sempre personalizzato per il progetto.

Ricordiamo che nello sviluppo iterativo, gli elaborati creati in questa fase **saranno parziali e verranno raffinati durante le prossime iterazione**; inoltre essi **non devono essere creati se non si ritiene che aggiungano un valore pratico effettivo**. Si noti che l'ideazione **può comprendere la programmazione di prototipi** per chiarire alcuni requisiti attraverso (di solito) prototipi **orientati all'interfaccia utente**. La tabella sopra mostra un ampio numero di elaborati per questa fase, tuttavia è necessario tenere a mente che **gli elaborati di UP vanno considerati opzionali** e bisogna scegliere di creare solo quelli necessari.

5.1.2 Ideazione e UML

Dato lo scopo dell'ideazione, è probabile che **non si faccia molto uso di UML in questa fase** se non per i **semplici diagrammi dei casi d'uso**. NGli elaborati prodotti in questa fase sono: ell'ideazione c'è una maggiore enfasi sulla **comprensione della portata del progetto e del 10% dei requisiti**; espressi soprattutto in **forma testuale**. L'applicazione e la creazione della maggior parte dei diagrammi UML avverrà nella fase di **elaborazione**.

5.2 Requisiti evolutivi

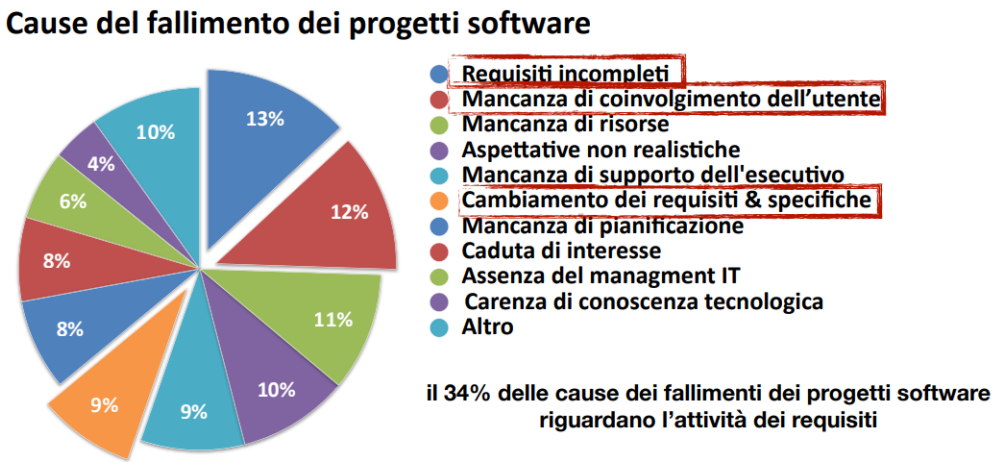
Ogni sistema software ha lo scopo di **risolvere** un determinato problema di interesse **per un insieme di utenti**. A tal fine, il sistema deve di solito fornire un **insieme di funzionalità**, relative alla **gestione di certe tipologie di informazioni**. Inoltre, il sistema deve possedere alcune caratteristiche di **qualità**, per esempio, in termini di sicurezza, affidabilità, ecc... Una possibile definizione di **requisito** è quindi la seguente:

Definizione 5.2.1 *Un **requisito** è una capacità o una condizione a cui il sistema, e più in generale il progetto, deve essere conforme*

I requisiti **derivano da richieste degli utenti del sistema** (e di altri parti interessate). Inoltre i requisiti vengono spesso **formalizzati** in un documento, una specifica o su un altro documento formale. Ci sono **due tipi** principali di requisiti:

- **Requisiti funzionali** (comportamentali): Descrivono il comportamento del sistema, in termini di **funzionalità fornite** ai suoi utenti. Questi requisiti sono di solito orientati all'uso del sistema e possono essere espressi, per esempio, sotto forma di **casi d'uso**. I requisiti funzionali comprendono anche gli aspetti **relativi alle informazioni che il sistema deve gestire**
- I **Requisiti non funzionali** (tutti gli altri requisiti): Non riguardano le specifiche funzioni del sistema ma sono relative a **proprietà del sistema nel suo complesso** (es. sicurezza, prestazioni, usabilità)

Una sfida primaria nell'analisi dei requisiti **è trovare, comunicare e ricordare** (il che di solito significa scrivere) ciò che è realmente necessario, in una forma che parli **chiaramente** al cliente e ai membri del team di sviluppo. Più generale, UP promuove un insieme di **best practices**, una delle quali è *gestire i requisiti*; cioè, nel contesto dei desiderata delle parti interessate, che sono **poco chiari e inevitabilmente cambieranno**, indica un **approccio sistematico per trovare, documentare, organizzare e tracciare i requisiti che cambiano di un sistema**.



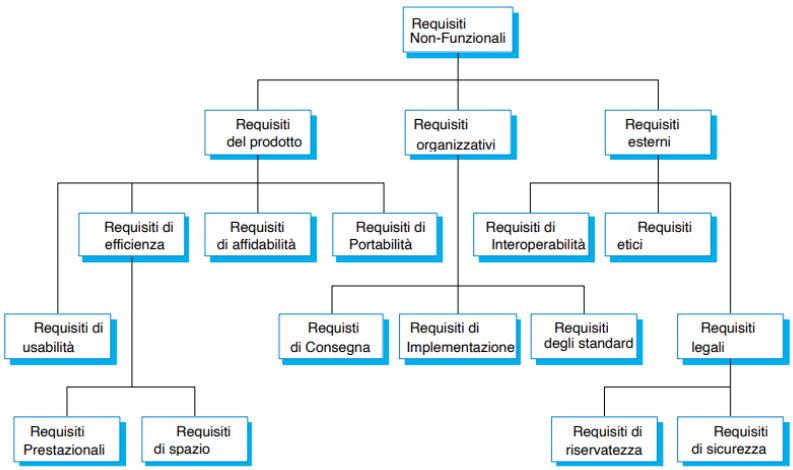
5.2.1 Proprietà dei requisiti funzionali

Altri problemi possono insorgere quando vi è **ambiguità nell'interpretazione dei requisiti funzionali**. Per esempio, la parola *verificare* potrebbe essere interpretata in **modi diversi** dall'utente e dal programmatore. In linea di principio, i **requisiti funzionali** dovrebbero essere:

- **Completi**: Dovrebbero includere la **definizione di tutti i servizi richiesti**
- **Coerenti**: Non devono **contenere informazioni contraddittorie**

In pratica, tuttavia, molto spesso, a causa della **complessità del sistema e dell'ambiente**, è impossibile produrre un documento completo e coerente sui requisiti.

5.2.2 Proprietà dei requisiti non funzionali



I **Requisiti non funzionali** definiscono le **proprietà e i vincoli del sistema**, come ad esempio affidabilità e tempo di risposta; oppure vincoli come la **capacità dei sistemi di I/O**, le rappresentazioni dei dati nelle interfacce di sistema ecc... Essi possono **vincolare il processo di sviluppo** del software in diversi ambiti, come ad esempio quali **standard di qualità usare** oppure **quali strumenti di sviluppo usare**. I requisiti non funzionali potrebbero **risultare più critici dei requisiti funzionali**.

In caso essi non siano soddisfatti, il sistema potrebbe **risultare inutilizzabile**. Lo schema sopra rappresenta i **tipi di requisiti non funzionali**. I requisiti non funzionali possono influire anche sull'architettura **complessiva** del sistema invece che solo sui singoli componenti. Un unico requisito non funzionale inoltre può generare **una serie di requisiti funzionali correlati** che definiscono i **servizi di sistema necessari** o può generare requisiti che **limitano quelli già esistenti**. I requisiti non funzionali possono essere tuttavia **molto difficili da individuare con precisione** e requisiti imprecisi possono essere **difficili da verificare**. Quindi, possiamo usare questa distinzione:

- **Obbiettivo**: Un'**intenzione generale dell'utente**, come, per esempio, la facilità d'uso
- **Requisito funzionale verificabile**: Una **dichiarazione** che utilizza alcune misure **oggettivamente verificabili**

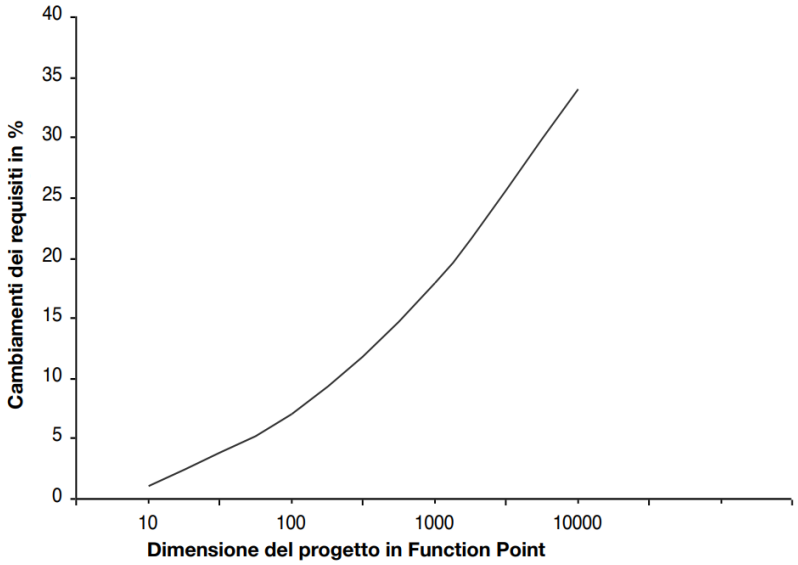
Gli obbiettivi sono utili agli sviluppatori poiché trasmettono le **intenzioni degli utenti del sistema**. Le seguenti sono **metriche per specificare i requisiti non funzionali**:

Proprietà	Misura
Velocità	Transazioni elaborate al secondo Tempi di risposta a utenti/eventi Tempo di refresh dello schermo
Dimensione	Mbytes Numero di chip ROM
Facilità d'uso	Tempo di addestramento Numero di maschere di aiuto
Affidabilità	Tempo medio di malfunzionamento Probabilità di indisponibilità Tasso di malfunzionamento Disponibilità
Robustezza	Tempo per il riavvio dopo malfunzionamento Percentuali di eventi causanti malfunzionamento Probabilità di corruzione dei dati dopo malfunzionamento
Portabilità	Percentuali di dichiarazioni dipendenti dall'architettura di destinazione Numero di architetture di destinazione

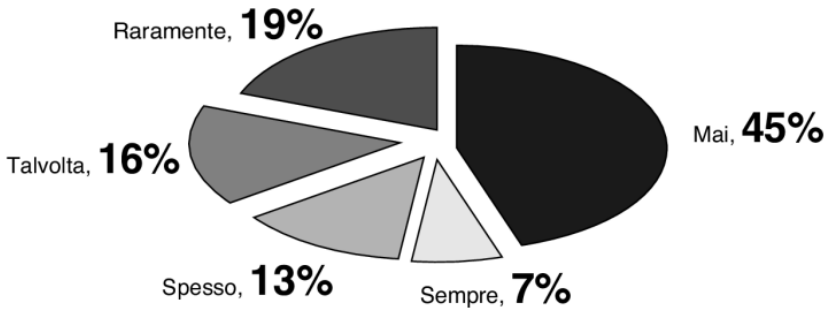
5.2.3 Requisiti evolutivi e a cascata a confronto

Un aspetto fondamentale dell'analisi dei requisti è la **gestione dei requisiti che cambiano**. Una gestione non realistica dei requisiti (e del loro cambiamento) è infatti alla base del **fallimento di molti progetti software**. Lo sviluppo iterativo **abbraccia il cambiamento nei requisiti** come una **guida** fondamentale per i progetti. Questo aspetto è estremamente importante ed è al centro delle differenze tra **sviluppo iterativo e sviluppo a cascata**. In UP e in altri metodi **evolutivi**, si inizia la programmazione di qualità e il test **molto prima che tutti i requisiti siano stati definiti** (forse all'inizio della programmazione se ne è definito il 10%/20%). A partire dagli anni ottanta, emersero prove che le convenzioni **seguite dal modello a cascata** sull'analisi dei requisiti, cioè **avere un fase di analisi e scrittura dei requisiti** che dovesse descrivere ed individuare **tutti i requisiti di un sistema**, erano **basate su principi errati**. In particolare, la vecchia convenzione considerava il processo di sviluppo del software come un **tradizionale processo di produzione di massa prevedibile**,

dove i requisiti di un prodotto **hanno un tasso di cambiamento basso**. Tuttavia, il software rientra nel dominio dello sviluppo di **nuovi prodotti** con **tassi di cambiamento alti e con grado di novità e scoperta elevati**. Infatti, in media, nei progetti software il **25% dei requisiti cambia**. Pertanto, qualsiasi metodo che cerchi di **congelare i requisiti di un sistema** o che si opponga al loro inevitabile cambiamento è **fondamentalmente difettoso e basato su un falso presupposto**. Secondo lo studio [Thomas01], effettuato su oltre mille progetti software, il fattore di fallimento più comune è stato il **tentare di applicare pratiche a cascata**: è stato menzionato come problema principale nell'82% dei progetti.



Un altro importante studio ([Johnson02]), svolto su migliaia di progetti, risponde alla seguente domanda: *se si segue un'analisi dei requisiti a cascata; quante delle caratteristiche specificate inizialmente sono effettivamente utili nel prodotto software finale*? I risultati sono abbastanza sorprendenti:



Dunque, quasi il **65%** delle caratteristiche specificate secondo un approccio a cascata si è rilevato inutile o poco utile. Questi risultati non implicano che **il modo giusto di procedere sia quello di scrivere codice sin dal primo giorno, dimenticandosi dell'analisi dei requisiti**. Invece, essi indicano che esiste una **via intermedia**: **l'analisi dei requisiti iterativa ed evolutiva** combinata con uno sviluppo **anticipato, iterativo e timeboxed**, nonché **partecipazione, valutazioni e feedback dei risultati parziali frequenti** da parte delle parti interessate.

5.2.4 Modi validi per trovare i requisiti

UP incoraggia un'acquisizione dei requisiti abile, attraverso, diverse tecniche:

- **Scrivere i casi d'uso con gli utenti**
- **Workshop dei requisiti** dove partecipano programmatori e clienti
- **Gruppi di lavoro** con rappresentanti dei clienti
- **Dimostrazione del risultato di ogni iterazione**, per ottenere un feedback istantaneo dal cliente

UP accoglie **qualsiasi metodo di acquisizione dei requisiti** fino a quando esso può aggiungere valore e **aumentare la partecipazione degli utenti**. Anche il metodo delle "storie" di XP può essere integrato in UP, tuttavia è una pratica molto difficile da realizzare poiché richiede **la presenza a tempo pieno del cliente o di un esperto**.

5.3 Tipi e categorie di requisiti

Nella gestione dei requisiti è spesso utile utilizzare un qualche sistema di classificazione come **checklist** per la copertura dei requisiti e per ridurre il rischio di non star tralasciando un qualche importante aspetto del sistema. Ci sono **diversi sistemi di qualificazione dei requisiti** ed essi sono pubblicati da **enti che si occupando di standard**, come ISO, SEI (Software Engineering Institute) ecc...
Ecco alcune categorie principali di requisiti:

- **Funzionale**: Requisito **funzionale**, caratteristiche e capacità funzionali
- **Usabilità**: Riguarda aspetti legati alla facilità d'uso del sistema; documentazione e aiuto per l'utente
- **Affidabilità**: Riguarda caratteristiche come la **disponibilità** (la quantità di tempo in cui il sistema è attivo e offre i suoi servizi agli utenti), la capacità del sistema di **tollerare i guasti** (fault tolerance) o di **poter essere ripristinato in seguito a fallimenti**
- **Prestazioni**: Riguarda caratteristiche come **tempi di risposta, throughput, capacità e uso delle risorse**
- **Sicurezza**: Riguarda la capacità del sistema di resistere ad **usi non autorizzati**, ma al tempo stesso di poter essere utilizzato dai suoi utenti legittimi
- **Sostenibilità**: È l'abilità del sistema software di essere **facilmente modificabile per consentire miglioramenti e riparazioni**. Riguarda aspetti come **adattabilità, manutenibilità, verificabilità, localizzazione, configurabilità, compatibilità**

Esistono anche altre categorie di requisiti, complementari e secondari, quali per esempio:

- **Vincoli di progetto**: Limitazioni su **risorse, linguaggi e strumenti da utilizzare, hardware,**...

- **Interoperabilità**: Vincoli imposti dalla necessità di **interagire e interfacciarsi con sistemi esterni**.
- **Operazionali**: Gestione del sistema nel suo contesto operativo
- **Fisici**: Per esempio, vincoli sulle dimensioni per l'hardware
- **Legali**: licenze e cose simili

Alcuni dei requisiti non funzionali sono chiamati **attributi di qualità del sistema**. Fra questi ci sono:

- **Usabilità**
- **Affidabilità**
- **Prestazioni**
- **Sostenibilità**

Gli attributi di qualità hanno una forte influenza sull'architettura del sistema.

5.4 Requisiti ed elaborati di UP

UP offre diversi elaborati dei requisiti che, come molti altri elaborati di UP, sono **opzionali**. I principali sono:

- **Modello dei casi d'uso**: Un insieme di **scenari tipici dell'utilizzo del sistema**. Usato principalmente per i requisiti funzionali
- **Specifiche supplementari**: Essenzialmente tutto ciò che **non rientra nei casi d'uso**. È usato principalmente per i requisiti non funzionali. È anche il posto per registrare delle **caratteristiche funzionali non espresse o non esprimibili come casi d'uso** (es. la generazione di un report)
- **Glossario**: Nella sua forma più semplice, il glossario definisce i **termini significativi**. Esso ha anche il ruolo di **dizionario dei dati** che registra i requisiti relativi ai dati, come **regole di validazione, valori accettabili** e così via. Il Glossario può definire nel dettaglio **qualsiasi elemento**, dall'attributo di un oggetto fino al parametro di un'operazione.
- **Visione**: Riassume i **requisiti ad alto livello** che sono dettagliati nel **modello dei casi d'uso** e nelle **specifiche supplementari**; contiene inoltre lo **studio economico** del progetto. Esso quindi è un **documento sintetico** usato per apprendere rapidamente le idee principali del progetto.
- **Regole di business**: Le **regole di business** (o regole di dominio) descrivono di solito requisiti che **trascendono il dominio del progetto software**. Esse sono richieste nel **dominio o nel business** ed è possibile che molte applicazioni vi si debbano **conformare** (es. le leggi fiscali di un certo stato). I dettagli delle regole di dominio **possono** essere registrati nelle **specifiche supplementari**, ma poiché di solito sono **più durature** e **sono applicabili a più progetti**, inserirle in un elaborato centrale, condiviso tra tutti gli analisti, permette un **miglior riuso dello sforzo di analisi**.

I requisiti sono scritti come **frasi in linguaggio naturale** integrate da **diagrammi e tabelle**. Il linguaggio naturale è usato perché è **espressivo, intuitivo e "universale"**, quindi può essere compreso sia dai clienti che dagli utenti.

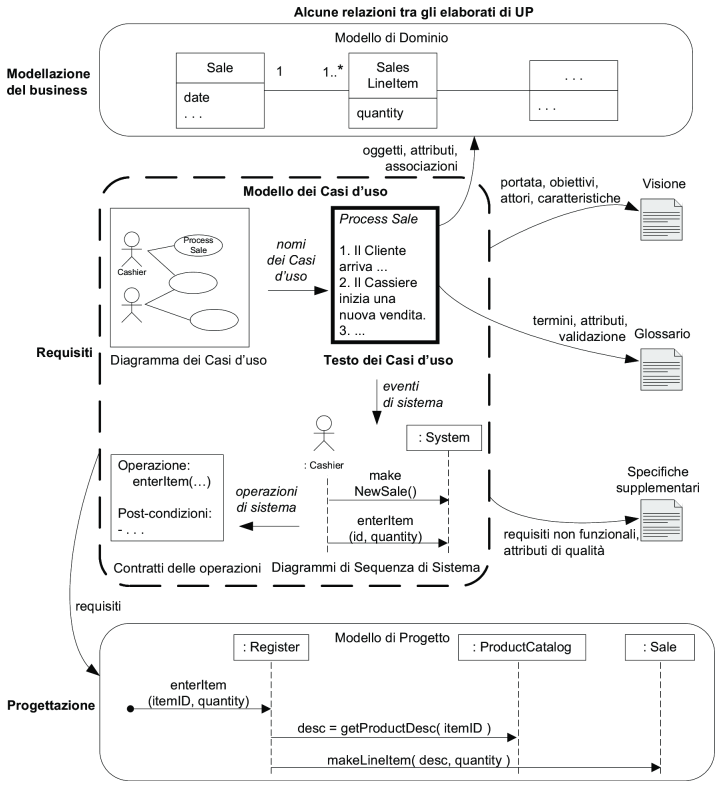
5.5 Linee guida per la scrittura dei requisiti

Possiamo seguire le seguenti linee guida quando scriviamo i requisiti:

- Ideare un **formato standard** ed utilizzarlo per tutti i requisiti
- L'utilizzo del linguaggio in modo **consistente**: utilizzare **DEVE** per i requisiti obbligatori, **DOVREBBE** per i requisiti desiderabili
- Utilizzare **l'evidenziazione del testo** per identificare le parti più importanti dei requisiti
- **Evitare il gergo informatico**
- Includere una **spiegazione razionale** del motivo per cui è necessaria una certa disposizione.

5.6 Casi d'uso

I **casi d'uso** sono **storie scritte** di un qualche attore che **usa un sistema per raggiungere degli obbiettivi**; essi sono ampiamente utilizzati per **scoprire e registrare i requisiti**. Essi influenzano molti aspetti di un progetto, compresa **l'analisi e la progettazione orientata agli oggetti**, tuttavia essi **NON SONO ELABORATI ORIENTATI AGLI OGGETTI**.



Innanzitutto, diamo delle definizioni preliminari:

Definizione 5.6.1 *Un **attore** è un qualcosa o un qualcuno dotato di **comportamento**, come una persona (caratterizzata da un **ruolo**), un **sistema informatico** oppure un'**organizzazione**.*

Definizione 5.6.2 *Uno **scenario** (o **istanza di caso d'uso**) è una **sequenza specifica di azioni e interazioni tra il sistema e alcuni attori**. Uno scenario descrive una **particolare storia nell'uso del sistema**, ovvero un **percorso attraverso il caso d'uso**.*

Definizione 5.6.3 *Un **caso d'uso** è una **collezione di scenari correlati**, sia di **successo** che di **fallimento**, che descrivono un attore che **usa un sistema per raggiungere un obbiettivo specifico**.*

Oppure

Definizione 5.6.4 *Un **caso d'uso** è un insieme di **scenari**, in cui ogni scenario è una **sequenza di azioni** che un sistema esegue per **produrre un risultato osservabile e di valore per uno specifico attore***

5.6.1 Modello dei casi d'uso

UP definisce il **modello dei casi d'uso** nell'ambito della disciplina dei **requisiti**. Si tratta dell'insieme di **tutti i casi d'uso descritti**; è un modello delle funzionalità del sistema e del suo ambiente. I casi d'uso **sono documenti di testo, non diagrammi**, quindi l'attività di scriverli è più un'attività di scrittura di testi, non di disegno di diagrammi. Il **modello dei casi d'uso** può includere, opzionalmente, **un diagramma UML dei casi d'uso** che mostra i nomi dei casi d'uso e degli **attori** e le **relazioni** tra essi. Esso costituisce un buon **diagramma di contesto** di un sistema e del suo ambiente, oltre a fornire un indice di facile consultazione dei nomi dei casi d'uso. Seppur i casi d'uso non sono **orientati agli oggetti**; ciò non costituisce un problema ma anzi, ne ampli **l'applicabilità e l'utilità**, poiché permettono di rappresentare i **requisiti come input utile per l'OOA/D classica**

5.6.2 Perché i casi d'uso

Molti metodi di analisi sono **troppo complessi** e vengono compresi sono dagli analisti, ma mettono in confusione l'uomo d'affari medio. Il mancato coinvolgimento dell'utente è tra le prime ragioni di **fallimento** dei progetti software [Larman03], e quindi è decisamente opportuno utilizzare tutto ciò che può contribuire al loro coinvolgimento. I casi d'uso sono un buon **metodo per mantenere la semplicità e consentire agli esperti di dominio di scrivere essi stessi i casi d'uso** o almeno di **partecipare alla loro scrittura**. Un altro valore dei casi d'uso è che mettono in risalto **gli obbiettivi degli utenti e il loro punto di vista**; i casi d'uso costituiscono la risposta alle domande:

- **Chi** utilizza il sistema?
- **Quali sono** i loro scenari di uso tipici?

- **Quali sono** i loro obbiettivi?

Questo pone un'enfasi **sull'utente**, invece di chiedere semplicemente quali devono essere le **caratteristiche funzionali del sistema**. Un'ulteriore punto di forza dei casi d'uso è dato dalla possibilità di **aumentare o diminuire il loro livello di dettaglio e formalità**.

5.6.3 I casi d'uso sono requisiti funzionali

I casi d'uso **sono** requisiti, soprattutto **requisiti funzionali o comportamentali**, che indicano cosa il sistema deve fare. Oltre agli aspetti funzionali, i casi d'uso possono essere **utilizzati anche per altri tipi di requisiti**, soprattutto se essi sono fortemente correlati ad altri casi d'uso. In molti metodi moderni, come UP, la scrittura dei casi d'uso è il **metodo consigliato** per la scoperta e definizione dei **requisiti**. Un punto di vista correlato è quello che definisce un caso d'uso come un **contratto relativo al comportamento di un sistema** [Cockburn01]

5.6.4 Tipi di attori

Un attore è **qualcosa o qualcuno dotato di comportamento**. Anche il **sistema in discussione** (SuD: System under Discussion) stesso è considerato un attore, quando **ricorre ad altri sistemi esterni**. Ci sono diversi tipi di attori:

- **Attori primari**: Utilizza **direttamente** i servizi del SuD, affinché vengano raggiunti degli obbiettivi utente. È utile identificare gli attori primari per **trovare gli obbiettivi degli utenti**, poiché essi guidano l'identificazione dei casi d'uso.

- **Attore finale**: Vuole che il SuD sia utilizzato **affinché si raggiungano i suoi obbiettivi**. Spesso, attore primario e finale **coincidono**, perché l'attore primario vuole raggiungere i propri obbiettivi usando il SuD. Tuttavia, vi può essere il caso in cui l'attore primario è **un intermediario che usa il sistema** invece che l'attore finale. È utile identificare gli attori finali per **trovare gli obbiettivi di questi attori**; che possono essere utili per guidare l'identificazione di altri importanti casi d'uso

- **Attore di supporto**: **Offre un servizio** (per esempio, informazioni) al SuD. Spesso è un sistema informatico, ma potrebbe essere una persona o un'organizzazione. È utile identificare gli attori di supporto per **chiarire le interfacce dei sistemi esterni usati dal SuD e i loro protocolli**.

- **Attore fuori scena**: Ha interesse nel **comportamento del caso d'uso**, ma non è un attore primario, finale o di supporto. È utile identificarli per garantire che **vengano individuati e soddisfatti tutti gli interessi necessari di tutte le parti interessate**. Gli interessi degli attori fuori scena sono spesso **sottili e facili da dimenticare** a meno che non vengano **nominati esplicitamente**.

5.6.5 Formati comuni per i casi d'uso

I casi d'uso possono essere scritti **usando diversi formati e livelli di formalità**:

- **Formato breve**: Riepilogo conciso di un solo paragrafo, normalmente relativo al solo scenario principale di successo.
  - **Quando va usato?** Durante **l'analisi iniziale dei requisiti**, per capire rapidamente l'argomento e la portata
- **Formato informale**: Più paragrafi, scritti in maneire **informale**, relativi a **vari scenari**
  - **Quando va usato?** Come nel caso del formato breve, solo con un livello di dettaglio maggiore
- **Formato dettagliato**: **Tutti i passi e tutte le variazioni** sono scritti nel dettaglio; ci sono anche delle **sezioni di supporto**, come le **pre-condizioni, le post-condizioni e le garanzie di successo**.
  - **Quando va usato?** Dopo che **molti casi d'uso sono stati definiti e scritti in formato breve**, alcuni casi d'uso (circa il 10%) che hanno maggiore valore e che sono **più significativi dal punto di vista dell'architettura** vengono scritti in formato dettagliato. Nelle iterazioni successive, anche gli altri casi d'uso verranno via via scritti in formato dettagliato.

Un buon template (e quello più diffuso dagli anni 90) è stato definito da **Alistair Cockburn**:

SELEZIONE DEL CASO D'USO	DESCRIZIONE
Nome del Caso d'Uso	Inizia con un verbo
Portata	Il sistema che si sta progettando
Livello	"Obiettivo utente" o "sottofunzione"
Attore Primario	Nome dell'attore primario
Parti Interessate e Interessi	A chi interessa questo caso d'uso e che cosa desidera
Pre-condizioni	Che cosa deve essere vero all'inizio del caso d'uso (e vale la pena di dire al lettore)
Garanzia di successo	Che cosa deve essere vero se il caso d'uso viene completato con successo (e vale la pena di dire al lettore)
Scenario Principale di Successo	Uno scenario comune di attraversamento del caso d'uso, di successo e incondizionato
Estensioni	Scenari alternativi, di successo e di fallimento
Requisiti speciali	Requisiti non funzionali correlati
Elenco delle variabili tecnologiche e dei dati	Varianti nei metodi di I/O e nel formato dei dati
Frequenza di ripetizione	Frequenza prevista di esecuzione del caso d'uso
Varie	Altri aspetti, ad esempio i problemi aperti

Vediamo come interpretare le sue varie sezioni:

- **Elementi del preambolo**: Il preambolo è composto da tutto ciò che precede lo **scenario principale e le estensioni**. Contiene informazioni che è importante leggere prima degli scenari del caso d'uso

- **Portata**: La portata descrive i **confini del sistema** in via di progettazione. Normalmente un caso d'uso descrive l'utilizzo di un sistema software. In questo caso viene chiamato **caso d'uso di sistema**. Adottando una portata più ampia, i casi d'uso possono anche descrivere il **modo in cui un'azienda o un'organizzazione viene utilizzata dai suoi clienti o dai suoi soci**. Una simile descrizione prende il nome di**caso d'uso di business**.
- **Livello**: I casi d'uso vengono classificati con un **livello**; possibili livelli, tra gli altri, sono il **livello obbiettivo utente** e il **livello di sottofunzione**.
  * **Obbiettivo utente**: un caso d'uso a livello di obbiettivo utente è il tipo più comune di caso d'uso; esso descrive gli **scenari con cui un attore primario può portare a termine il suo lavoro, raggiungendo i suoi obbiettivi**. Ciò corrisponde approssimativamente a un **processo di business elementare** (EBP) nell'ingegneria dei processi di business.
  * **Sottofunzione**: Un caso d'uso a livello di sottofunzione descrive invece dei **sotto-passi**, richiesti come **supporto al raggiungimento di un obbiettivo utente**.
- **Attore finale e attore primario**: L'attore **finale** è l'attore che vuole raggiungere un obbiettivo, e questo richiede l'esecuzione dei servizi del sistema. L'attore **primario** è l'attore che **usa direttamente il sistema**. Di solito i due coincidono, ma è possibile che l'attore finale sia un **intermediario**.
- **Elenco delle parti interessate degli interessi**: Le parti interessate e i loro interessi **suggeriscono e limitano ciò che il sistema deve fare**. Per dirlo con le parole di Cockburn: *Il sistema definisce un contratto tra le parti interessate, dove i casi d'uso descrivono nel dettaglio gli aspetti comportamentali di questo contratto...*
  Il caso d'uso come contratto per il comportamento raccoglie tutti e soli i comportamenti relativi alla soddisfazione degli interessi delle parti interessate. Questo consente di rispondere alla domanda: "**Che cosa va scritto nel caso d'uso?**"; la risposta è: "**Ciò che serve a soddisfare tutti gli interessi delle parti interessate**". Inoltre, iniziando a scrivere un caso d'uso partendo dai relativi interessi delle parti interessate, si ha a disposizione un metodo per ricordare quali dovrebbero essere le **responsabilità dettagliate del sistema**.

- **Pre-condizioni e garanzie di successo (post-condizioni)**: Le **pre-condizioni** descrivono **cosa deve essere sempre vero all'inizio del caso d'uso**. Le pre-condizioni **non vengono verificate all'interno del caso d'uso** ma si presuppongono vere. Normalmente una pre-condizione implica che **lo scenario di un'altro caso d'uso sia stato completato con successo**. È necessario notare che vale la pena di scrivere **solamente le pre-condizioni non banali o che vale la pena descrivere al lettore**. Le **garanzie di successo** o **post-condizioni** affermano **cosa deve essere vero quando è stato completato con successo il caso d'uso**, ovvero quando si completa **lo scenario principale di successo** o un percoso alternativo. La garanzia **deve soddisfare tutte le esigenze delle parti interessate**. Anche in questo caso, vale la pena di scrivere **solo le post-condizioni non ovvie o significative**.

- **Scenario principale di successo e i suoi passi (Flusso di base)**: Lo scenario principale di successo viene anche chiamato **happy path** oppure **flusso di base** o **flusso tipico**. Esso descrive **un percorso di successo comune** che soddisfa gli interessi di tutte le parti interessate. Lo scenario principale è costituito da una **sequenza di passi numerati**, che può contenere **passi da ripetere più volte**, ma che di solito non comprende alcuna condizione o diramazione (rimandate alla sezione delle estensioni). I passi che formano lo scenario possono essere di tre tipi:

  1. **Un'interazione tra attori**. Anche il sistema deve essere considerato un attore. I casi più comuni sono:
     (a) **Un attore** (utente) **interagisce con il sistema**, inserendo dati o effettuando una richiesta
     (b) **Il sistema interagisce con un attore** (utente), comunicandogli dei dati o fornendogli una risposta
     (c) **Il sistema interagisce con altri sistemi**
  2. Un **cambiamento di stato** da parte del sistema
  3. **Una validazione**

  Il primo di un caso d'uso **non rientra sempre in questa classificazione**, ma talvolta può indicare **l'evento trigger** che scatena l'esecuzione dello scenario. Similmente, nemmeno l'ultimo passo rientra sempre in questa classificazione, ma può descrivere il fatto che **l'attore ha effettivamente conseguito i suoi obbiettivi**. I nomi degli attori di solito vengono scritti **con l'iniziale maiuscola** per distinguerli meglio. Per indicare una ripetizione di passi, si **indica testualmente che i passi da n a k vanno ripetuti**

- **Estensioni (o Flussi alternativi)**: