Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 3° anno - 6 CFU

Ingegneria del Software

Professore:
Prof. Jacopo Soldani

Autore:
Filippo Ghirardini

Anno Accademico 2024/2025

Contents

Ingegneria del software

Realizzato da: Ghirardini Filippo

A.A. 2024-2025

---

1 Introduzione

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

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

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

1.0.1 Scopo

Lo scopo è quello di produrre software che sia:

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

1.1 Casi di studio

Richiamiamo alcune definizioni in ambito software:

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

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

1.1.1 Gemini V

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

1.1.2 Denver Airport

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

1.1.3 THERAC-25

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

1.1.4 Sistema anti-missile Patriot

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

1.1.5 London ambulance service

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

1.1.6 Ariane V

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

1.1.7 Toyota

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

1.1.8 METEOR

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

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

1.2 Storia

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

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

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

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

1.3 Aspetti importanti

Vediamo alcuni aspetti importanti dello sviluppo del software.

1.3.1 Specificità

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

1.3.2 Economia

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

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

1.3.3 Teamwork

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

2 Processo software

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

Ogni modello di processo software include:

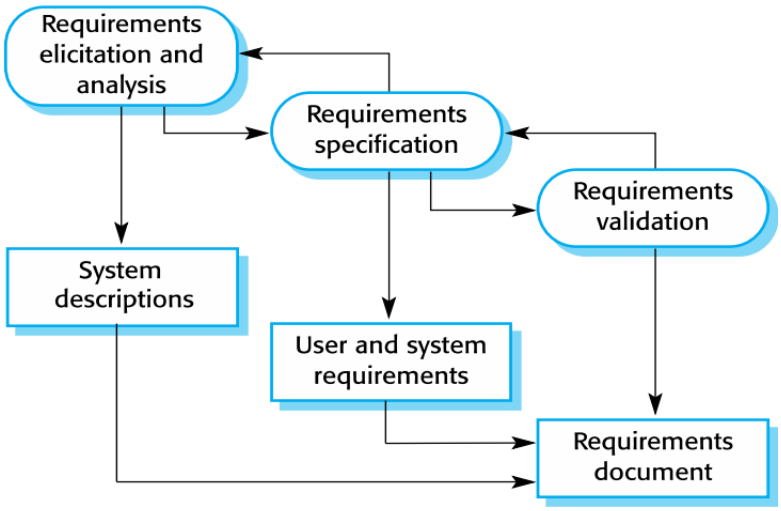
- **Specifica**: definizione di cosa deve essere fatto
- **Progettazione** e **implementazione**
- **Validazione**: verifica che il sistema rispetti le specifiche
- **Evoluzione**: modifica o aggiornamento del sistema

2.1 Fasi di sviluppo

2.1.1 Specifica

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

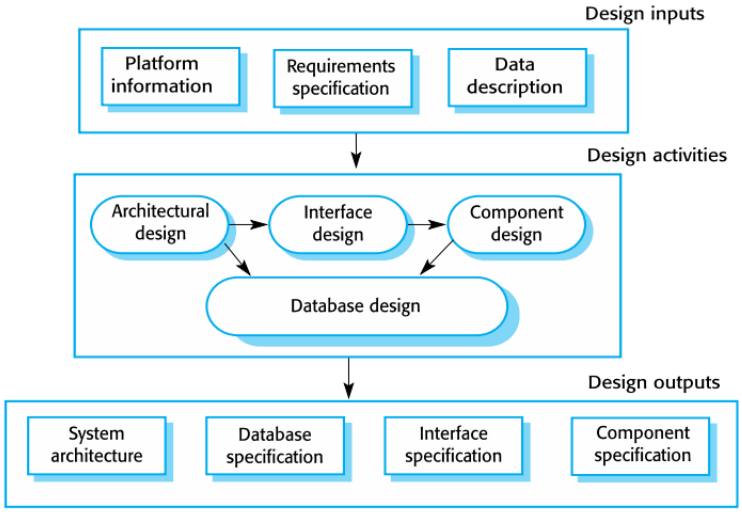
- **Estrazione** e **analisi** dei requisiti: cosa richiedono o si aspettano gli stakeholder
- **Specifica** dei requisiti: definirli in dettaglio
- **Convalida** dei requisiti: verificare che siano validi



2.1.2 Progettazione

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

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



2.1.3 Sviluppo

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

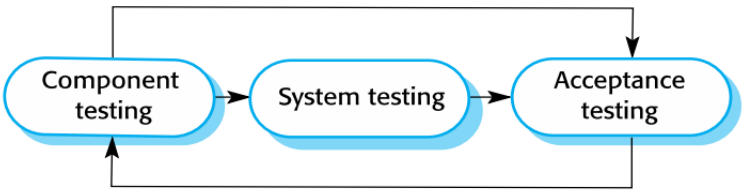
- **Programmazione** attività senza un processo standard
- **Debugging**: attività per identificare e correggere errori o bug

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

2.1.4 Validazione

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

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



2.1.5 Evoluzione

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

2.2 Modelli

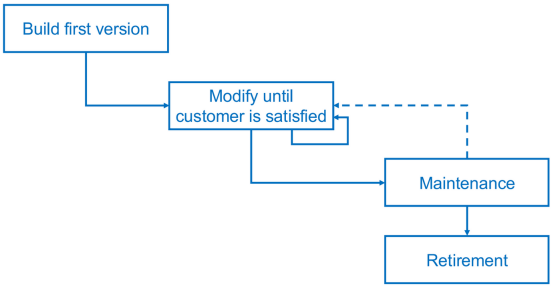
Definizione 2.2.1 (Modello)**.** *Il modello di un processo software fornisce una rappresentazione astratta del processo stesso:*

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

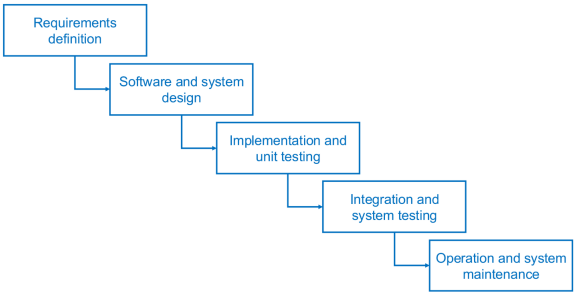
E.g. ISO 12207

2.2.1 Sequenziale

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

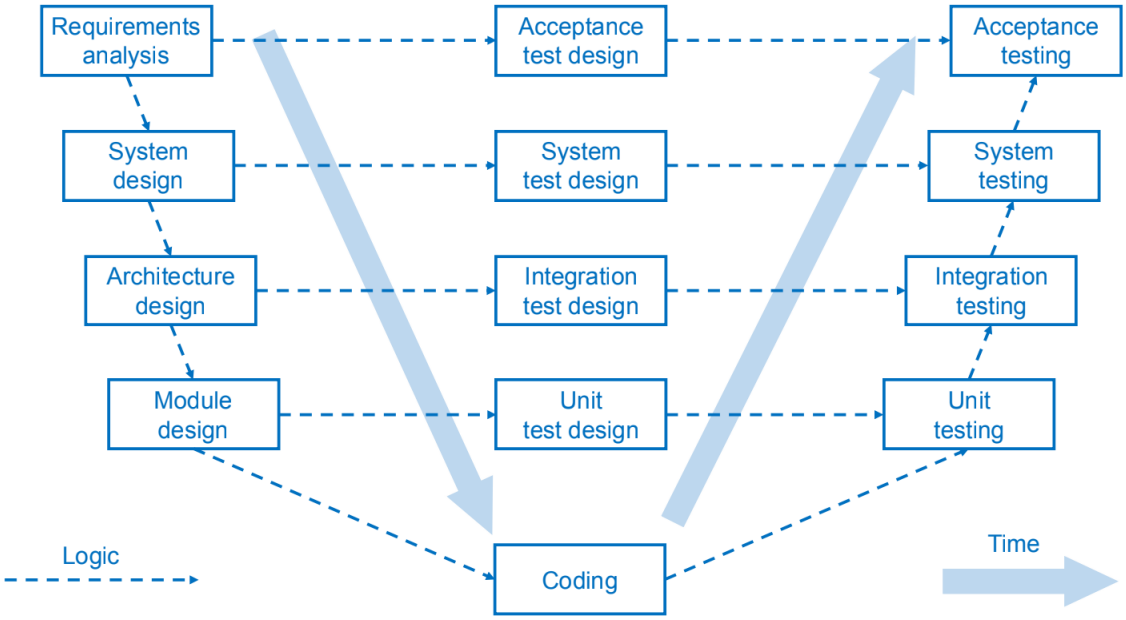


Modello a cascata Modello ideato da Royce nel 1970, fu il primo a distinguere e definire le fasi di un processo software, dando finalmente importanza all'analisi e alla progettazione prima della codifica. Ogni fase prima di procedere deve produrre un documento che deve essere approvato da chi di dovere.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



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

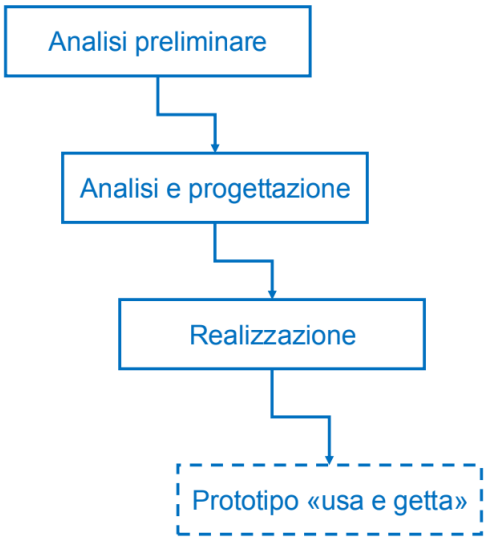
Modello a V In questo modello le attività di **sinistra** sono di analisi che scompongono i requisiti degli utenti in sezioni piccole; quelle di **destra** aggregano e testano il prodotto delle precedenti per verificare che le esigenze siano effettivamente rispettate.
Al centro troviamo la progettazione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



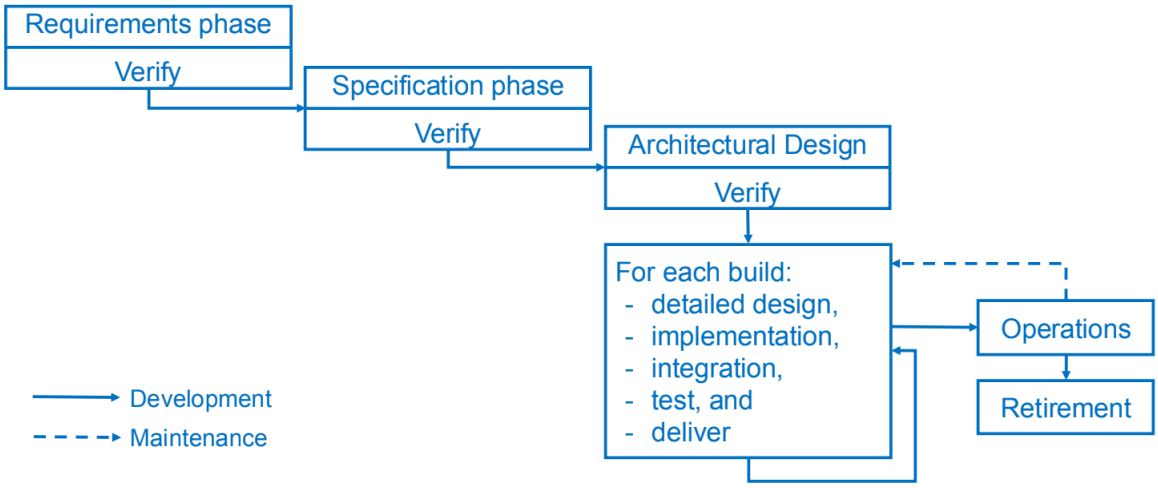
Note 2.2.2. Questo modello è uno standard SQA (qualità del software).

2.2.2 Iterativo

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



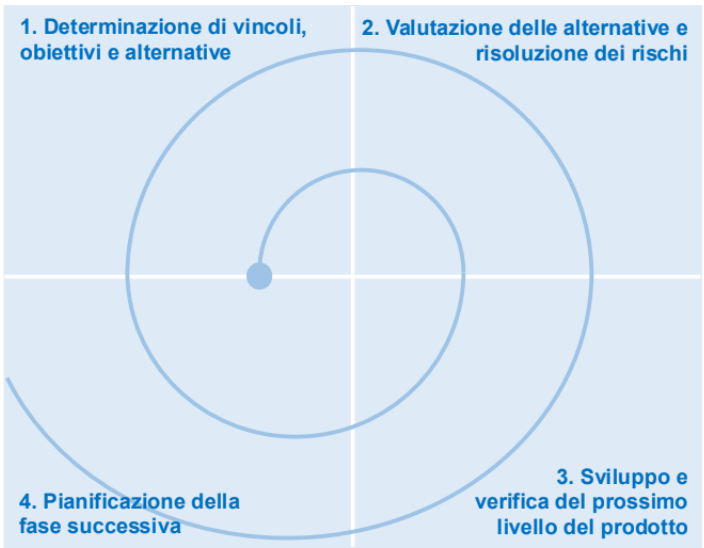
Modello incrementale In questo modello il sistema è costruito **iterativamente** aggiungendo nuove funzionalità a partire da requisiti definiti inizialmente.
Questo permette di **ritardare** fasi che per motivi esterni non possono proseguire e fa uscire versioni iniziali ed utilizzabili molto rapidamente,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



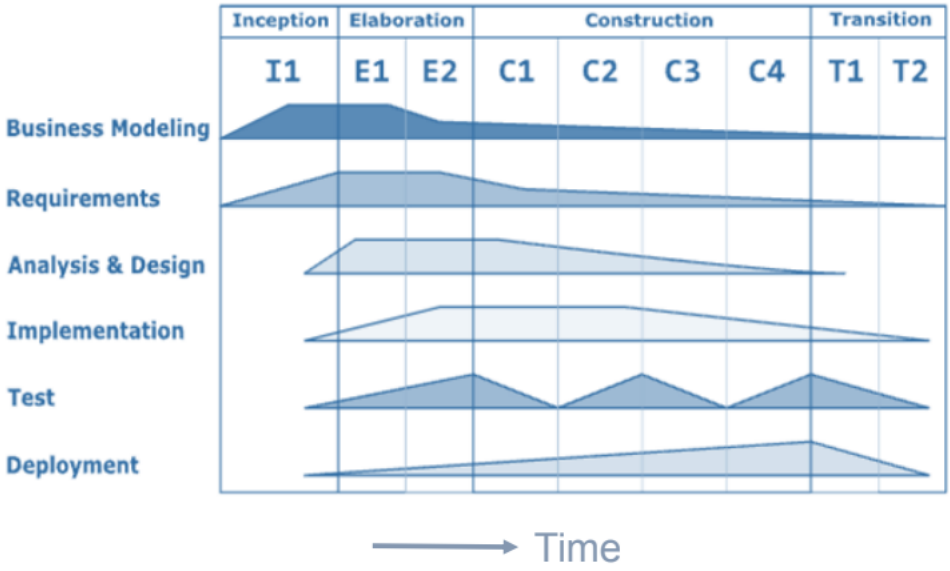
Modello a spirale Ideato da Bohem nel 1998, ispirato dal *plan-do-check-ackt* di Deming, divide ogni **iterazione** in quattro fasi:

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

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



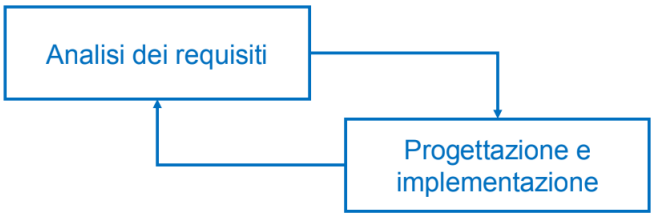
Unified process Ideato da Booch Et Al nel 1999, è guidato da **casi d'uso** e **analisi dei rischi** già a partire dalla raccolta e analisi dei requisiti. È un modello iterativo **incrementale** incentrato sull'**architettura**: nelle prime fasi c'è una definizione generale e i dettagli vengono lasciati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



2.2.3 Agile

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

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



Il modello agile segue i seguenti principi:

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

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

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

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

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

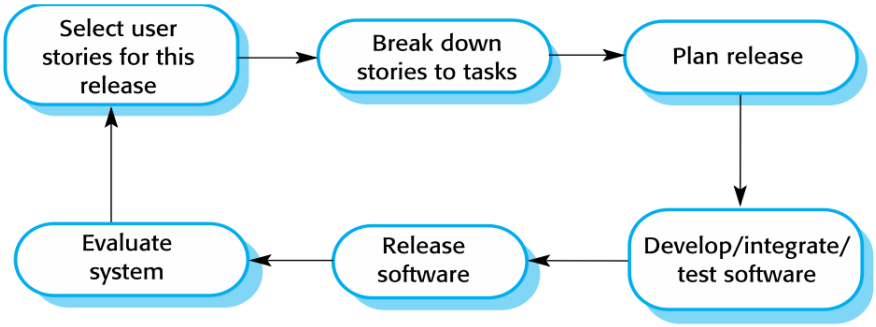
Il modello agile è **conveniente** per lo sviluppo di prodotti di piccola o media dimensione (fino a 50 sviluppatori) oppure in contesti di sviluppo di sistemi custom (meno regolamenti e restrizioni). Nella pratica comunque viene usato nella maggior parte dei team.

Il modello agile ha portato alla nascita di:

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

2.2.4 Extreme Programming

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



Alcune pratiche comuni nell'XP:

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

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

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

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

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

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

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

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

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

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

2.2.5 Scrum

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

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

Ci sono tre figure coinvolte:

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

Le fasi del metodo scrum sono:

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

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

2.3 Kanban

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

3 Analisi dei requisiti

Definizione 3.0.1. *Processo di studio e analisi delle esigenze del committente e dell'utente per giungere alla definizione del dominio del problema e dei requisiti del sistema.*

L'analisi serve a capire **cosa** fare e non come farlo, identificando i confini del sistema SW, il modo in cui interagisce con l'ambiente e la qualità con cui lo fa.
È un processo **fondamentale** in quanto permette di individuare e risolvere difetti in maniera meno costosa rispetto alle altre fasi di sviluppo software.
Il prodotto dell'analisi dovrà essere un documento e/o modello che descrive il **dominio** del sistema, i suoi **requisiti** ed opzionalmente il *manuale utente* e i *casi di test*.

3.1 Studio di fattibilità

È la fase preliminare per stabilire l'opportunità di realizzare o meno un sistema software. Si basa su:

- **Descrizione** del sistema e delle necessità degli utenti
- Analisi di **mercato**:
  - Mercato *attuale* e *futuro*
  - *Costo* di produzione
  - *Redditività* attesa
- Analisi tecnica di **realizzabilità**:
  - Strumenti necessari
  - Soluzioni algoritmiche e architetturali
  - Hardware
  - Processo

3.2 Dominio

Per la costruzione del dominio sono necessari:

- Un **glossario**: collezione di termini rilevanti al caso specifico. Costruito strada facendo dagli analisti e può essere riutilizzato.
- Modello **statico** e **dinamico**

Ci si deve concentrare su: **entità**, **relazioni**, **processi** e **comportamenti**.

Esempio 3.2.1 (House of cars)**.** House of Cars è un parcheggio verticale multipiano, formato da 10 colonne e 24 piani per colonna, 12 sotto al livello strada e 12 sopra.

Glossario

- *Colonna*: colonna di posti auto dotata di un sollevatore centrale che raggiunge tutti i piani del parcheggio; ha un proprio locale di ricezione auto; comprende un carrello
- *Carrello*: carrello per movimentare le vetture; è dotato di "forchette" in corrispondenza delle ruote
- *Cella*: formata da due box affiancati: può quindi contenere due auto
- *Gruppo di spostamento elettromeccanico*: controlla il carrello che trasla la vettura nelle celle posizionate davanti e dietro al sollevatore
- ...

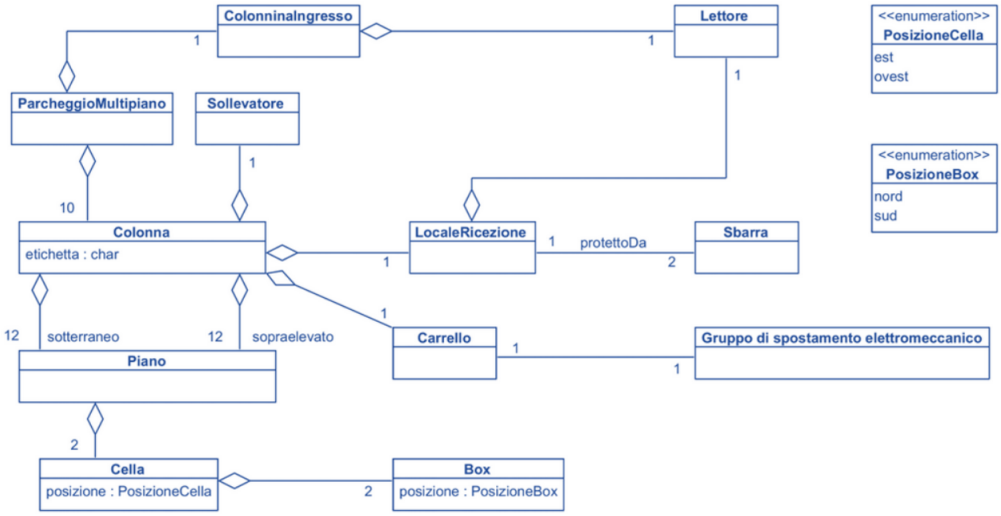


Figure 1: Modello statico

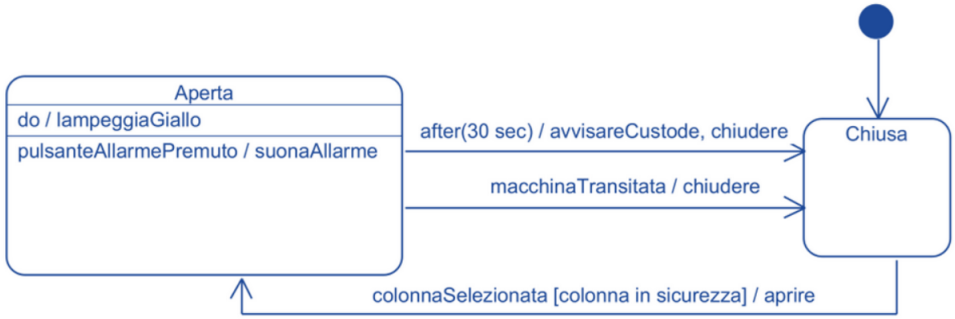


Figure 2: Modello dinamico

3.3 Requisiti

Un requisito è una proprietà che deve essere garantita dal sistema per soddisfare una necessità dell'utente.

Definizione 3.3.1 (IEEE definition)**.** *Un requisito è:*

- *Una condizione (o capacità) necessaria a un utente per risolvere un problema o raggiungere un obiettivo*
- *Una condizione (capacità) che deve essere soddisfatta/posseduta da un sistema per soddisfare un contratto, uno standard, una specifica, o altri documenti formali*

I requisiti possono essere di due tipi (che devono rimanere separati):

- **Funzionali**: descrivono le **funzionalità** che il sistema deve realizzare in termini di *azioni*, *risposte agli input* e *comportamenti in casi particolari.*
- **Non funzionali**: descrivono le **proprietà** del SW:
  - **qualità**, e.g. efficienza, affidabilità, sicurezza
  - **processo**, e.g. standard di processo, linguaggi usati, metodo di sviluppo
  - **esterne**, e.g. interoperabilità e vincoli legislativi
  - **fisici**, e.g. hardware, rete

Osservazione 3.3.1. Un requisito deve essere **ben posto**, idealmente nella forma **assertiva**

$$\text{Il } < \text{sistema} > \text{ deve } < \text{funzionalità/proprietà} >$$

Definizione 3.3.2 (Documento dei requisiti)**.** *È un documento che specifica **cosa** deve fare il sistema e con quali **vincoli**. È un contratto tra lo sviluppatore e l'utente che in genere specifica anche la scadenza.*

3.3.1 Acquisizione

L'acquisizione può avvenire in diversi modi:

- **Interviste**
  - *Strutturate*
  - *Non strutturate*
- **Questionari** a risposta multipla
- Costruzione di **prototipi** (anche su carta)
- **Osservazione** di futuri utenti al lavoro: i **casi d'uso** devono includere la sequenza di eventi corretta ed eventuali comportamenti inattesi
- **User stories**: utilizzato nei processi *Agile*, i requisiti sono descritti con un template predefinito

  $$\text{As a } < \text{user role} > \text{ I want } < \text{goal} > \text{ so that } < \text{benefit} >$$

  che viene messo su un foglio in formato $A6$ (facile, visibile e rende possibile appenderli e collegarli). Questa tecnica però non è **scalabile**, è **vaga** e **informale** e spesso non include dettagli sui requisiti **non funzionali**.
- **Studio** di documenti

3.3.2 Elaborazione

Durante questa fase i requisiti vengono **espansi** e **raffinati** e viene scritta la prima bozza del documento. Questo deve essere strutturato in:

1. **Introduzione**: perché il sistema è desiderabile e come si inquadra negli obiettivi più generali del cliente
2. **Glossario**
3. **Requisiti funzionali**
4. **Requisiti non funzionali**
5. **Architettura**: strutturazione in sottoinsiemi
6. Specifica dei requisiti del software: specifica dettagliata dei requisiti **funzionali**
7. **Modelli astratti** del sistema, formali o semi-formali
8. **Evoluzione**: modifiche previste
9. **Appendici**: individuazione ed eventuale descrizione della piattaforma hardware, requisiti di database, manuale utente, piani di test
10. **Indici**

3.3.3 Convalida

In questa fase si controlla il documento dei requisiti per evitare i seguenti errori:

- **Omissioni** o incompletezza: mancata presenza di un requisito
- **Inconsistenza**: contraddizione tra i requisiti o tra un requisito e l'ambiente di lavoro
- **Ambiguità**: significati multipli. In particolare si deve prestare attenzione a:
  - **Quantificatori**: e.g. tutti, sempre, ogni, niente, ogni, qualsiasi
  - **Disgiunzioni**: e.g. AND, OR, XOR
  - **Coordinazione**: e.g. *Viaggerò in treno o in traghetto e in macchina*
  - **Referenziale e anafore**: in base ai pronomi e a come vengono utilizzati
  - **Vaghezze**: quando vengono usati *aggettivi qualificativi* o *avverbi non misurabili* (e.g. *appropriato*)
  - **Verbi deboli** e **forme passive**: e.g. *potere* oppure forme passive senza complemento di agente
- **Sinonimi ed omonimi**
- Non devono esserci **dettagli tecnici**
- **Ridondanze** in una stessa sezione

Note 3.3.1. Non usare doppie negazioni.

Per validare un documento già strutturato esistono varie tecniche.

Walkthrough : lettura sequenziale dei documenti.

Lemmario : si utilizzano i termini del glossario con puntatori ai requisiti che li nominano, aiutando a trovare *inconsistenze*, *omonimi*, *sinonimi* e *ridondanze*.

Natural Language Processing : software che fanno un'analisi del documento alla ricerca di errori. Alcuni esempi sono: QuARs, TIGER-PRO e RQA

Prototipi : presentazione di prototipi al committente

Note 3.3.2. Eventuali errori di quelli sopra descritti devono sempre essere discussi con il cliente.

3.3.4 Negoziazione

In questa fase si assegnano delle **priorità** ai requisiti basandosi sulle **esigenze** del committente e sui **costi** e **tempi** di produzione. Durante questa fase alcuni requisiti possono essere cancellati o posticipati.

MoSCoW Questa tecnica assegna una priorità ai requisiti dividendoli nelle seguenti classi:

- **Must have**: obbligatori e irrinunciabili per il cliente
- **Should have**: non necessari ma utili
- **Could have**: opzionali ma relativamente utili
- **Want to have**: contrattabili per successive versioni

3.3.5 Gestione

Per gestire i requisiti è fondamentale assegnargli un **identificatore univoco** oltre che diversi attributi:

- **Stato**: proposto, approvato, rifiutato o incorporato
- **Priorità**
- **Effort** espresso in giorni e personale
- **Rischio** da un punto di vista tecnico
- **Stabilità**
- **Versione** di destinazione

È importante che i requisiti siano **tracciabili** per poter risalire ai *componenti del sistema*, al loro *codice* e ai *test*.

Note 3.3.3. È importante allegare il documento dei requisiti al contratto o, nel caso in cui non sia ancora pronto, prevedere una rinegoziazione.

4 UML

Unified Modelling Language è un linguaggio di modellazione che supporta la descrizione e la progettazione di progetti software (in particolare OO). Descrive da un punto di vista **strutturale** e **comportamentale**, del dominio e del codice, della progettazione e della fase finale.
Utilizza **famiglie grafiche** che comprendono diversi punti di vista, sono correlate e facilmente interpretabili.

4.1 Modello

Un modello è un'**astrazione** a partire dai dettagli, del sistema o dominio usato per specificarne il *comportamento* e la *struttura*. È espresso con un formalismo ed è descritto da un insieme di **viste**.
È uno strumento di documentazione, comunicazione e discussione fondamentale per un progetto di sviluppo software.

4.1.1 Modellazione

Un modello può essere:

- **Statico**: vengono usate **entità** e **relazioni** per descrivere tutti gli aspetti indipendenti dal tempo:
  - Concetti del *dominio*
  - Componenti dell'*architettura*
  - Classi di *realizzazione*
- **Dinamico**: modella il comportamento delle entità descritte in quello statico

In questa fase si decide anche il livello di astrazione.

4.1.2 Rappresentazione

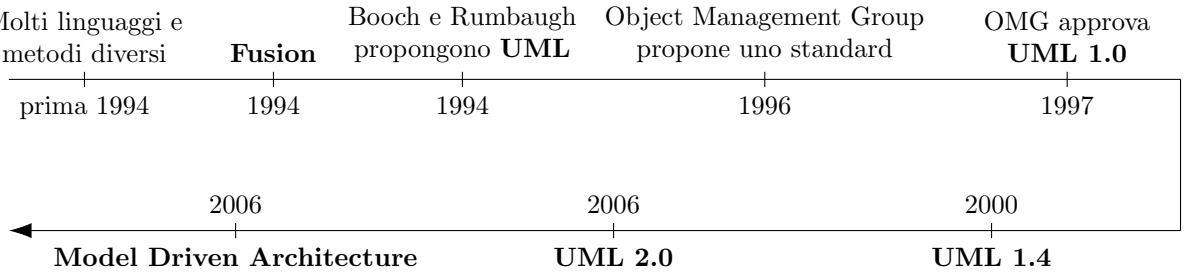
Si rappresenta con un linguaggio formale o semi formale.

4.1.3 Utilizzo

A seconda del livello del modello, ha scopi diversi:

- **Sketch**: non completo, usato per le descrizioni iniziali
- **Blueprint**: sufficiente a creare un sistema capace di funzionare
- **Eseguibile**: talmente completo e preciso da generare automaticamente il codice

4.2 Storia



4.3 Diagrammi

4.3.1 Casi d'uso

Questo diagramma modella i requisiti del sistema: raccoglie quelli *funzionali*, li elabora e li documenta. Il modello statico è il diagramma dei casi mentre quello dinamico contiene le narrative a loro associate.

Definizione 4.3.1 (Attore)**.** *Un attore è un'**entità estranea** al sistema che **interagisce** direttamente con esso in un determinato **ruolo**. Può essere di tre tipi:*

- *Utente **umano** in un determinato **ruolo***
- *Un **sistema** differente*
- ***Tempo***

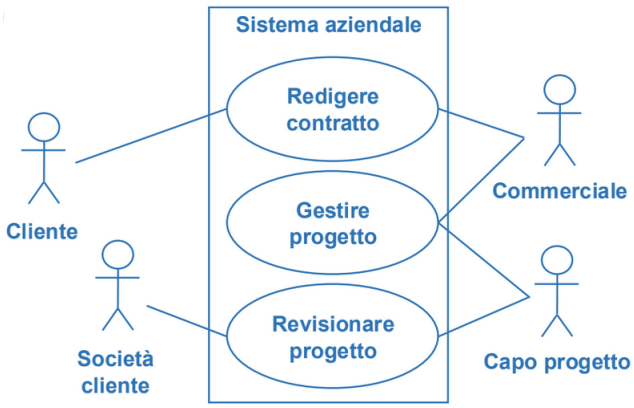
Definizione 4.3.2 (Caso d'uso)**.** *Un caso d'uso è un **compito** che un **attore** può svolgere con l'aiuto del sistema. Viene espresso come insieme di scenari.*

Definizione 4.3.3 (Scenario)**.** *Uno scenario è una sequenza di **interazioni** tra sistema e attori. Viene espresso come scambio di messaggi.*

La modellazione di questo tipo di diagramma si divide nelle seguenti fasi:

1. Individuare il **confine** del sistema
2. Individuare gli **attori**
3. Individuare i **casi d'uso**
4. Individuare le **relazioni** tra attori e casi d'uso
5. Specificare il caso d'uso con una **narrativa**

Notazione In questo tipo di diagramma abbiamo *attori* e *casi d'uso* indicati con la lettera maiuscola. Le *relazioni* che rappresentano lo scambio di messaggi sono linee e il *confine del sistema* è un rettangolo intorno ai casi d'uso.



Note 4.3.1. L'associazione tra attori e casi d'uso è di tipo **molti a molti** in quanto un attore può essere associato a più casi e un caso a più attori.

Note 4.3.2. Un caso può essere **iniziato** solamente da uno ed un solo attore, detto **principale** (che può essere il Tempo). Ci sono casi specifici in cui non ci sono attori.

Narrativa La narrativa descrive il modello dinamico, ovvero gli **scenari** rilevanti per un caso d'uso dal punto di vista degli attori, compreso il principale. La **struttura** è la seguente:

Nome	Nome del caso d'uso
ID	Identificatore univoco del caso
Breve descrizione	...
Attore primario	Colui che avvia il caso
Attori secondari	Tutti gli attori che interagiscono
Precondizioni	Ciò che deve valere prima dell'esecuzione
Sequenza degli eventi principale	Sequenza di passi
Postcondizioni	Ciò che deve valere dopo l'esecuzione
Sequenze alternative degli eventi	Errori, ramificazioni e interruzioni

Scenario Uno scenario è un'**istanza** di un caso d'uso, quindi una sequenza di passi che produce un risultato osservabile da uno o più attori. Portano dalla *precondizione* alla *postcondizione*.

Definizione 4.3.4 (Pre e post condizioni)**.** *Precondizioni e postcondizioni sono* ***asserzioni*** *che devono necessariamente essere vere in uno stato. Si esprimono con* ***predicati*** *e* ***formule logiche****. Non sono MAI azioni.*

Definizione 4.3.5 (Flusso di narrativa)**.** *Per ogni stato σ che soddisfa la* ***precondizione****, l'esecuzione del caso d'uso a partire da σ produce uno stato σ' che soddisfa la* ***postcondizione****. Questo a meno di imprevisti elencati nella sequenza alternativa.*

Sequenza principale Indica la sequenza di passi che compongono il caso d'uso. È **numerata** e ogni passo ha la struttura

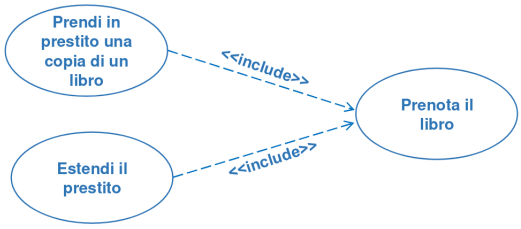
$$< \text{numero} > . < \text{soggetto} >< \text{azione} >< \text{complementi} >$$

Il primo passo è detto **attivazione** ed è compiuto sempre dall'attore principale.

Generalizzazione È possibile generalizzare gli **attori**, ad esempio quando c'è bisogno di un unico attore principale (e.g. *professore* e *assistente* sono *ricercatori*), o i **casi d'uso** (e.g. *card* e *cash* sono *pagamenti*). Bisogna prestare attenzione perché il classificatore specializzato **eredita** tutte le relazioni di quello padre a meno che non sia esplicitato il contrario.

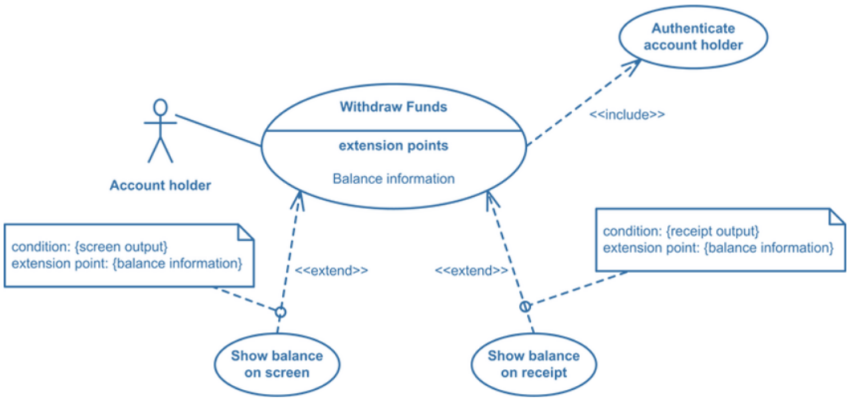
Definizione 4.3.6 (Stereotipo)**.** *Gli stereotipi sono parole chiave racchiuse tra <<>> che annotano gli elementi di un diagramma, precisandone il significato.*

Inclusione Un caso d'uso può incorporarne un altro tramite l'**inclusione**, ovvero la chiamata del primo ha come conseguenza la chiamata del secondo.



Nella narrativa l'inclusione può avvenire in maniera *istanziabile* quando viene avviato da un attore primario o *non istanziabile* quando viene eseguito solo dopo l'inclusione da parte di un altro caso.

Estensione Quando il primo caso d'uso *può* prevederne un altro, si dice che lo incorpora. Di conseguenza le estensioni sono opzionali e per specificare quando si presentano è possibile usare un **extension point** con all'interno la condizione che si deve verificare.



4.3.2 Classi e oggetti

Il diagramma delle classi descrive il tipo degli oggetti che fanno parte di un sistema e le relazioni statiche tra di esse. Mostra inoltre le **proprietà** e le **operazioni** delle classi. Può essere utilizzato per descrivere il *dominio* o per fare una *progettazione* di dettaglio.

Definizione 4.3.7 (Oggetto)**.** *Un oggetto è un'identità caratterizzata da un'**identità**, uno **stato** (valori e rispettivi attributi) e un **comportamento** (operazioni che lo definiscono).*

Definizione 4.3.8 (Classe)**.** *Una classe descrive un insieme di oggetti con caratteristiche simili, ovvero dello stesso tipo. Cattura un **concetto** nel dominio del problema o della realizzazione.*

In UML la classe contiene:

- **Nome**: maiuscolo singolare
- **Attributi** tipizzati con modificatori di visibilità:
  - *Pubblic* + : accessibile ad ogni elemento che può vedere e usare la classe
  - *Protected* # : accessibile ad ogni discendente
  - *Private* - : solo le operazioni della classe vi hanno accesso
  - *Package* ~ : solo gli elementi dello stesso *package* vi hanno accesso
- **Operazioni** tipizzate

Attributi La sintassi degli attributi è la seguente:

$$\text{visibilità } \textit{nome} : \text{tipo}[\text{molteplicità}] = \text{ valoreIniziale } \{\text{proprietà}\}$$

Note 4.3.3. La molteplicità [1] può essere omessa.

Degli esempi di **proprietà** sono *ordered*, *unique* o vincoli in generale (e.g. $\{> 0, < 10\}$).

Note 4.3.4. Quando viene usato per la definizione del dominio, si omettono generalmente *operazioni* e *modificatori di visibilità* e si inseriscono solo gli *attributi* utili ad esso.

Operazioni La sintassi delle operazioni è la seguente:

$$\text{visibilità } \textit{nome}(\text{lista parametri}) : \text{tipoRitorno}$$

Dove la lista dei parametri può essere l'**insieme vuoto** o una **dichiarazione** di parametro:

$$\text{lista parametri} ::= \emptyset \quad | \quad \text{dichiarazione}$$
$$\text{dichiarazione} ::= \text{nome} : \text{tipo} = \text{default}$$

Note 4.3.5. L'unica parte obbligatoria della sintassi è il **nome**, sia nelle operazioni che in *eventuali* parametri.

Note 4.3.6. Attributi e operazioni **statici**, quindi con ambito di classe, sono sottolineati.

Enumerazioni Le enumerazioni sono usate per specificare un insieme di valori **prefissati** ovvero tutti i valori che un attributo può assumere.
In UML hanno sono *classi* con un nome (il tipo) e l'elenco dei valori. Sono etichettate dallo *stereotipo*

<< enumeration >>

4.3.3 Relazioni

Una relazione rappresenta un legame tra due o più oggetti, di solito istanze di classi diverse.

Tra classificatori	**Tra oggetti**
Associazione	Collegamento
Generalizzazione	*(non definita)*
Realizzazione	*(non definita)*
Dipendenza (d'uso, d'istanza, etc..)	

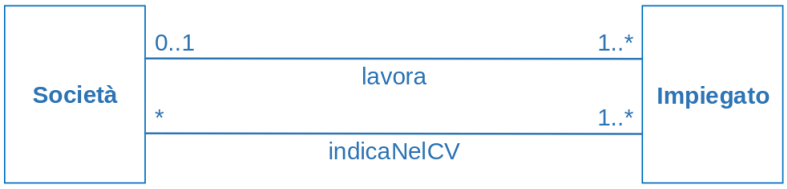
Associazione Un'associazione è una linea retta con:

- **Nome**, di solito un verbo
- **Ruoli**, di solito sostantivi
- **Verso** di lettura, opzionale

Di solito si usa o il *nome* o i *ruoli*, raramente entrambi.
È importante anche specificare i **vincoli di molteplicità**, ovvero il numero di oggetti coinvolti nell'associazione in un dato istante. Si possono definire:

- Con un **numero positivo**
- Con il **simbolo indefinito** (*), ovvero per un qualunque numero ≥ 0
- Indicando gli **estremi dell'intervallo**, dove quello *inferiore* può essere ≥ 0 e quello superiore un numero positivo o il simbolo indefinito

Esempio 4.3.1. Esempio di molteplicità:



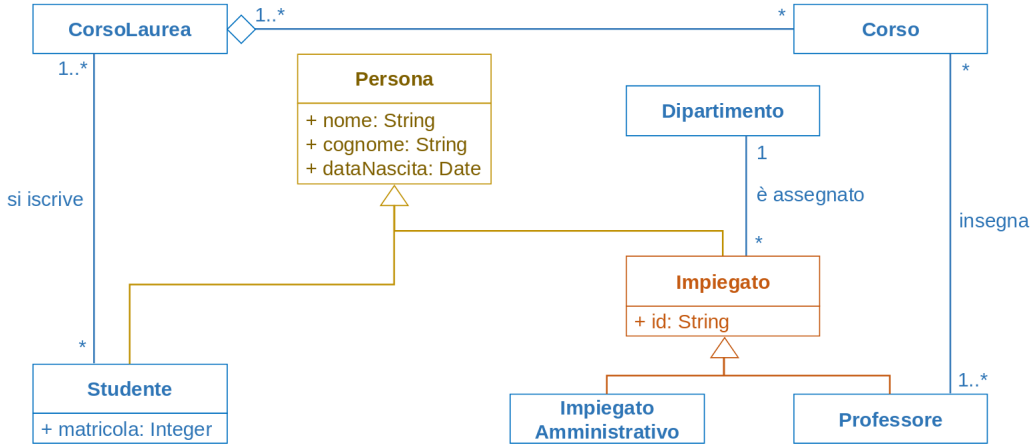
Un associazione può mettere in relazione un'entità con se stessa, in questo caso è detta **riflessiva**.
Esistono due tipi specifici di associazioni che vengono utilizzati per specificare se un oggetto fa parte o meno di un'altra classe:

- **Aggregazione** (◊), poco forte, usata quando la classe "parte" esiste anche senza quella "tutto"
- **Composizione** (♦), molto forte, usata quando la classe "parte" non esisterebbe senza quella "tutto". Non ha un nome.

Esempio 4.3.2. Esempio di aggregazione e composizione:

Generalizzazione È una relazione tra un elemento generico G e uno specializzato S, che è consistente con il primo ma contiene più informazione. Per il *principio di sostituzione di Liskov* S può essere sostituito con G.

Una *superclasse* generalizza le sottoclassi e queste **ereditano** *attributi*, *operazioni*, *relazioni* e *vincoli*. La sottoclasse può aggiungere caratteristiche e ridefinire determinate operazioni.



Definizione 4.3.9 (Classe astratta)**.** *Nell'ambito della generalizzazione, quando una classe esiste puramente per essere poi estesa e specializzata, viene definita* ***astratta****. In questo caso o si usa il nome in corsivo o si indica*

$$\{abstract\}$$

Definizione 4.3.10 (Interfaccia)**.** *Un'interfaccia è un'entità che contiene solamente il comportamento ma nessuno stato. Viene indicata dallo stereotipo*

$$<< interface >>$$

Dipendenza Una dipendenza è una relazione tra una classe **cliente** e una **fornitore**, dove il primo dipende dal secondo e una modifica nel secondo influenza il primo. Viene indicata da una linea tratteggiata con uno dei seguenti stereotipi a seconda del caso

$$<< \text{use} >> \qquad << \text{create} >>$$

4.3.4 Classi di analisi

Le classi di analisi corrispondo a concetti concreti del *dominio*, e.g. il contenuto del glossario. Viene spesso raffinata in una o più classi di *progettazione*. Il loro compito è quello di **astrarre** un elemento del dominio. Generalmente devono seguire le seguenti caratteristiche:

- Numero ridotto di funzionalità
- Evitare classi *onnipotenti*
- Evitare funzioni travestite da classi
- Evitare gerarchie di ereditarietà ≥ 3

Esistono principalmente due approcci per identificare le classi di analisi:

- **Data-driven**: si identificano i dati del sistema e si dividono in classi, tipico per la fase di analisi
- **Responsibility-driven**: si identificano le responsabilità e si dividono in classi, tipico per la fase di progettazione

Analisi nome-verbo Una tecnica per eseguire l'identificazione dove ogni **sostantivo** corrisponde ad una *classe* o *attributo* mentre ogni **verbo** ad un'operazione.
Consiste in:

1. Individuazione delle classi
2. Assegnazione di attributi e responsabilità
3. Individuazione delle relazioni

Le problematiche principali sono i sinonimi che portano a classi **inutili** e le classi **nascoste** perché implicite nel dominio.

4.3.5 Diagramma degli oggetti

Un oggetto si rappresenta con il **nome**, la **classe** che istanzia, sottolineati e la lista degli **attributi** (nome, tipo e valore).

Collegamento Un collegamento istanzia un'associazione nel diagramma delle classi collegando due o più oggetti. Non ha nome, ha sempre molteplicità $1:1$ e può includere i ruoli

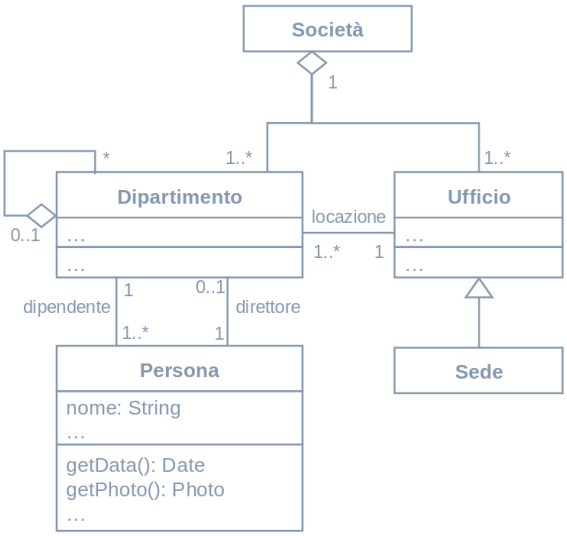


Figure 3: Diagramma delle classi

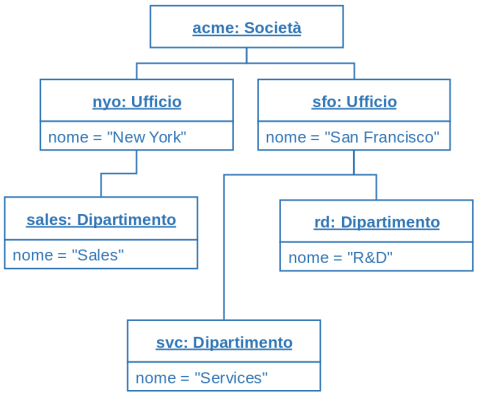
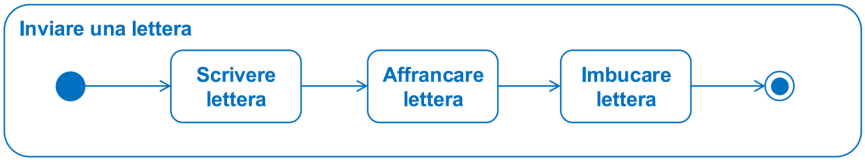


Figure 4: Diagramma degli oggetti

4.3.6 Diagramma delle attività

Il diagramma delle attività modella un **workflow** e descrive come **coordinare** un insieme di **azioni**, quali *sequenze*, *scelte*, *iterazioni* e *concorrenza*. Nello specifico modella un'attività in cui sono coinvolte una o più entità.
In UML è contenuta in un rettangolo dagli angoli smussati e ha un nome.



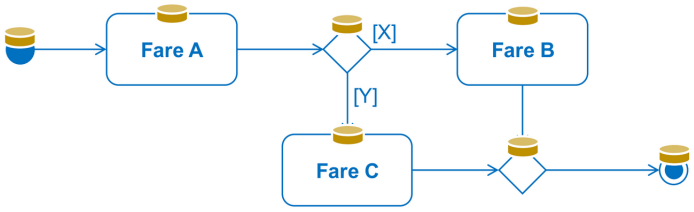
Il contenuto dell'attività è un **grafo diretto**, dove i **nodi** rappresentano le componenti dell'attività (azioni e nodi di controllo quali inizio e fine) mentre gli **archi** rappresentano il **control flow**, ovvero i possibili path di esecuzione.

Azioni Le azioni sono rettangoli con gli angoli smussati che contengono il nome che è sempre un **verbo**. Sono **atomiche** e non interrompibili.
Ogni azione ha solo un arco entrante ed uno uscente e quest'ultimo viene attraversato ad azione terminata, simulando il passaggio di un **token** che permette l'esecuzione.

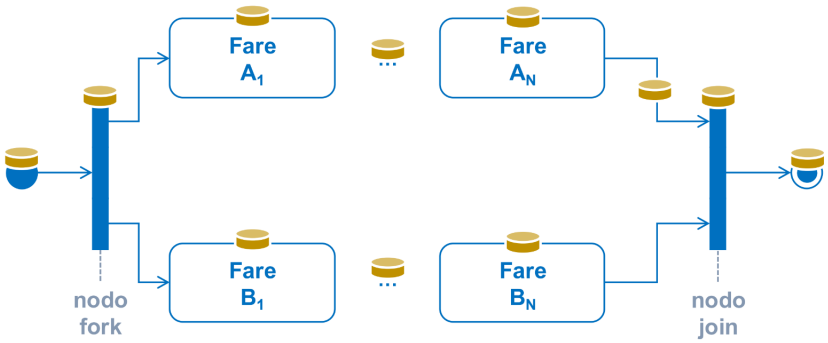
Note 4.3.7. I nomi delle azioni devono indicarne l'esecuzione e quindi dovrebbero essere verbi all'indicativo, imperativo o infinito oppure sostantivi che indichino un'azione.

Nodi di controllo Sono nodi che alterano il control flow. Possono essere:

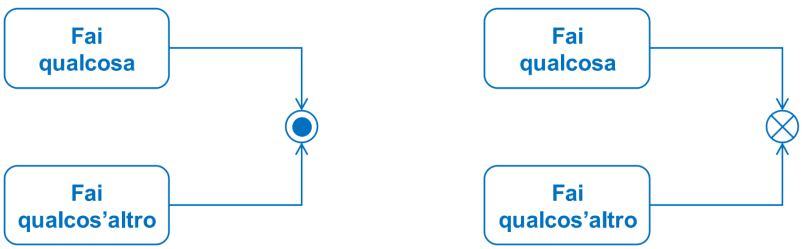
- **Decisione** e **fusione**: sono rappresentati da rombi, nel primo caso con condizioni che devono coprire tutti i possibili casi ($X \vee Y \vee \ldots \vee Z = \text{true}$) e che portano a rami differenti. È possibile utilizzare la clausola [*else*]. Non è obbligatorio che ci sia un nodo di fusione corrispondente.



- **Fork** e **join**: la fork moltiplica i token e ne restituisce uno per ogni ramo uscente, al contrario la join (non ne serve per forza una per ogni fork), attende un token da ogni ramo, li consuma tutti e ne restituisce uno.



- **Fine attività** e **fine flusso**: gli unici nodi che possono avere più archi entranti. Il primo, appena riceve un token, termina l'intera attività mentre il secondo termina solo il flusso corrispondente.



- **Segnali** ed **eventi**: possono inviare un **segnale asincrono**, riceverlo o accettare un **evento temporale** (assoluto o relativo) I nodi di accettazione non hanno bisogno di archi entranti: se

Invio Accettazione Evento temporale

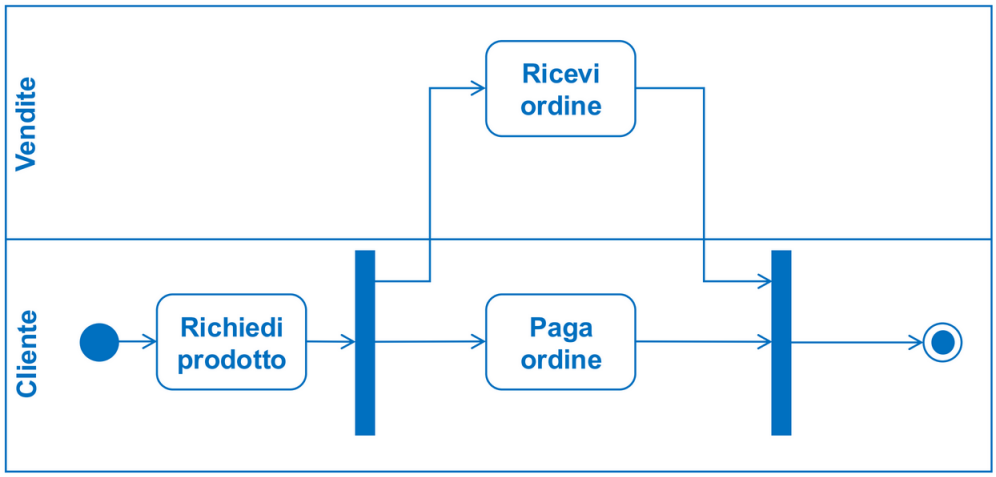
non c'è si genera un token quando si verifica l'evento, altrimenti si attende prima di passarlo al nodo successivo.

Note 4.3.8 (Fork e merge). È possibile eseguire una *fork* e poi una *merge* invece della join, ma questo vorrà dire che le operazioni dopo saranno eseguite più volte.

Note 4.3.9 (Azioni vs eventi). Un'**azione** si usa quando è effettuata da una o più entità che stiamo descrivendo mentre un **evento** riguarda quelle esterne.

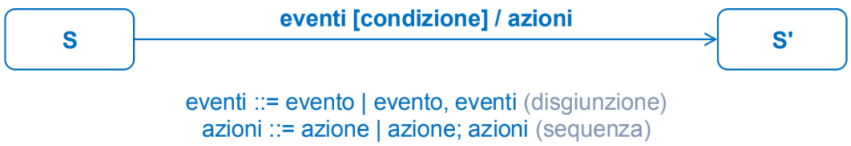
Sotto attività Un diagramma può contenere un riferimento ad un'attività secondaria e viene indicato con ⫛.

Partizionamento Sono divisioni dell'attività che permettono di assegnare le responsabilità, spesso usate ad esempio in combinazione con la divisione delle unità operative di un modello business.

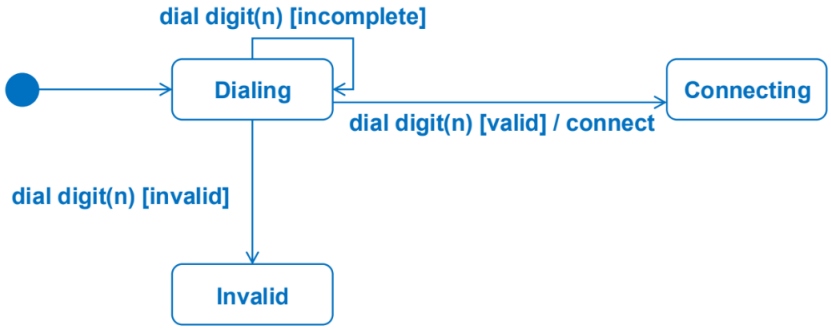


4.3.7 Diagramma degli stati

Il comportamento di una classe viene descritto da un **grafo stati-transizioni**. Questo contiene gli **stati** significativi di un oggetto durante la sua vita e come questo può **transire** (in seguito ad eventi come messaggi) tra di essi.
Gli elementi principali sono lo stato **iniziale**, quello **finale**, gli stati **intermedi** e le **transizioni**. Queste ultime definiscono la risposta di un oggetto all'occorrenza di un evento e possono essere prese solo quando si verifica l'evento e l'eventuale **condizione** è vera. Comportano inoltre l'esecuzione di eventuali **azioni**.



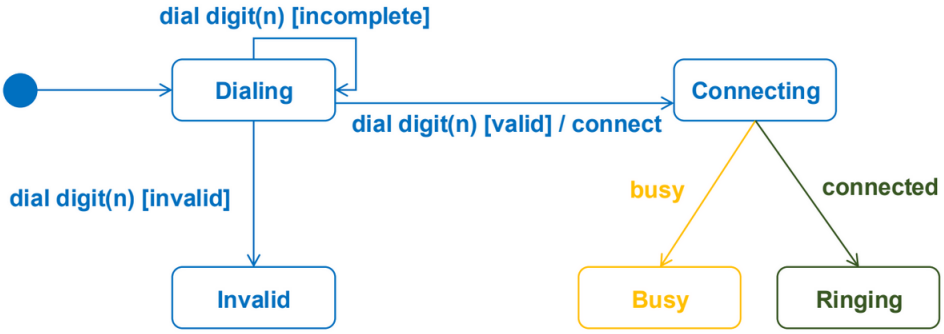
Esempio 4.3.3. Un esempio di diagramma a stati con una classe *Telefono* e un'operazione *dial digit(n)*



Eventi Un evento occorre **istantaneamente** e va introdotto solo se ha degli effetti. Se lo stato **non ha transizioni** per quell'evento, allora lo si ignora, altrimenti se ne ha **più di una**, si fa una scelta non-deterministica. Un evento può essere:

- Un'**operazione** o un **segnale**: ricezione do una chiamata di metodo o un segnale con parametri e tipi compatibili
$$op(a : T)$$
- **Variazione**, quando *exp* si verifica. Non avviene istantaneamente ma lo diventa appena si verifica l'espressione.
$$when(exp)$$
- **Temporale**, dopo che l'oggetto è stato fermo per un certo periodo di tempo
$$after(t)$$

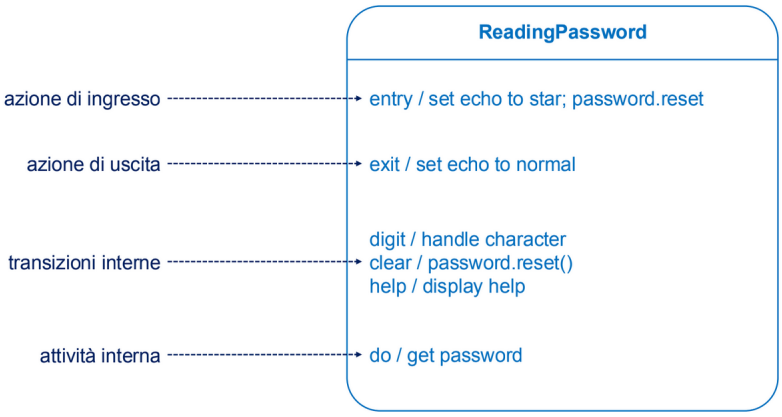
Esempio 4.3.4. Continuiamo l'esempio di prima aggiungendo due eventi di *variazione*.



Stato Di seguito i comportamenti di un oggetto in determinate situazioni mentre esso si trova in un certo stato:

- **entry**: azione di entrata eseguita all'ingresso in uno stato
- **do**: azione interna eseguita in modo continuativo mentre l'oggetto è in quello stato. Non necessita di eventi scatenanti, consuma tempo e può essere interrotta
- **exit**: azione di uscita eseguita all'ingresso in uno stato
- Eventuali **transizioni interne** che vengono eseguite in risposta ad un evento

Esempio 4.3.5. Prendiamo ad esempio lo stato di lettura di una password:



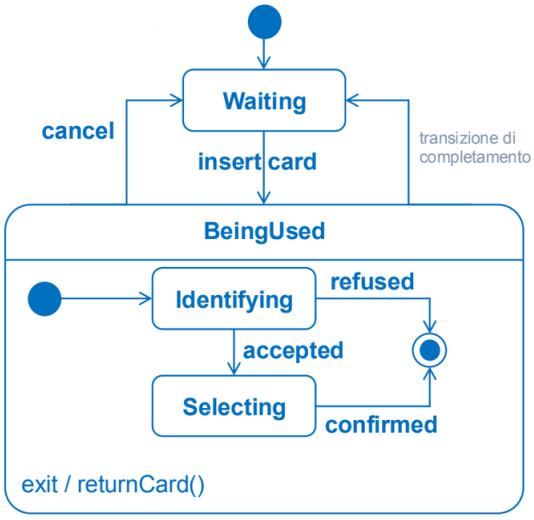
Note 4.3.10. I nomi degli stati devono indicarne la *permanenza* e quindi dovrebbero essere aggettivi, participi passati o gerundi.

Stato composito In questo caso lo stato include un'ulteriore macchina a stati.
Il punto d'**ingresso** è determinato dalla freccia:

- se arriva al bordo si inizia dallo stato iniziale della macchina interna
- altrimenti dallo stato indicato dalla freccia

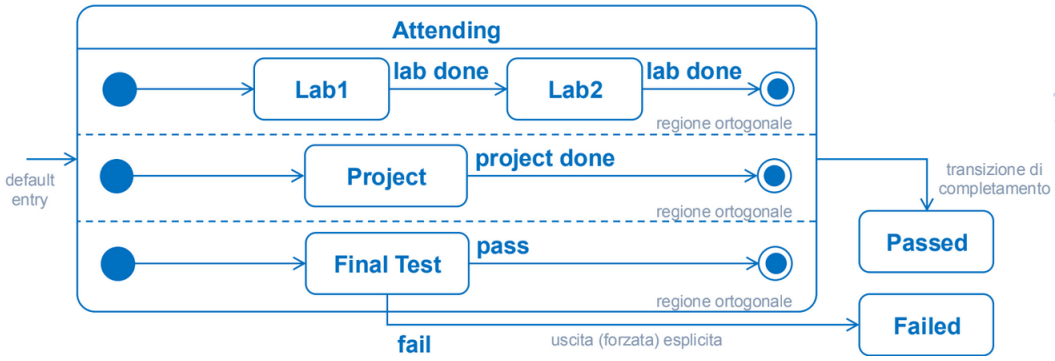
All'**uscita** ci possono essere tre casi:

- Transizioni etichettate che partono dal bordo e possono essere attivate da qualsiasi stato interno
- Dallo stato finale si procede alla **transizione di completamento**
- Transizioni da stati interni che attraversano il bordo



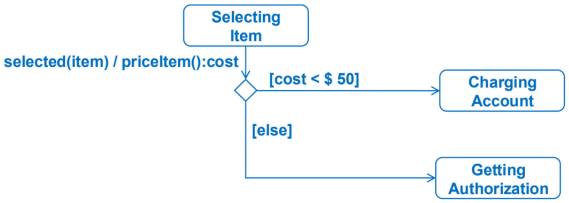
Stati compositi paralleli È possibile avere stati compositi **paralleli**: possono esserci più sottostati attivi contemporaneamente, uno per regione. L'ingresso (**default entry**) arriva sul bordo e prosegue a tutti i sottostati iniziali. La transizione di uscita si può presentare in tre casi:

- Transizione di completamento attivata al raggiungimento di tutti i sottostati finali
- Transizione che esce da un solo sottostato e li termina tutti
- Trasmissione che parte dal bordo, sempre attivabile e termina tutti i sottostati

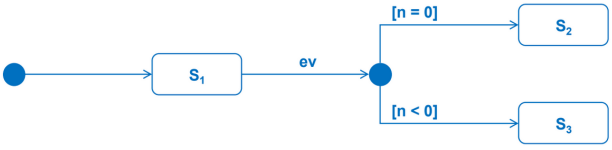


Pseudo-stati Esistono tre tipi di pseudo-stati:

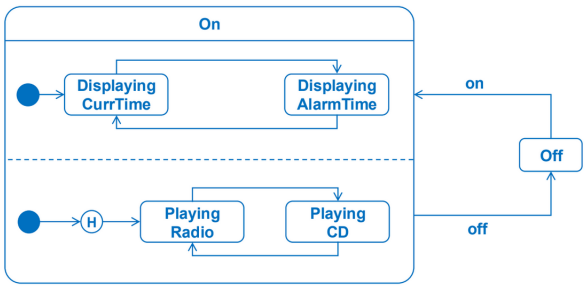
- **Scelta**: scelta basata su condizioni determinate **dinamicamente**. La disgiunzione (OR) di tutte le guardie deve essere sempre vera ed è ammesso il non-determinismo



- **Giunzione**: pseudo-stato da cui escono e/o entrano due o più transizioni. Le condizioni sono valutate **staticamente** e **prima degli eventi** che attivano le transizioni in ingresso



- **History**: permette di ripristinare lo stato della precedente attivazione dello stato composito

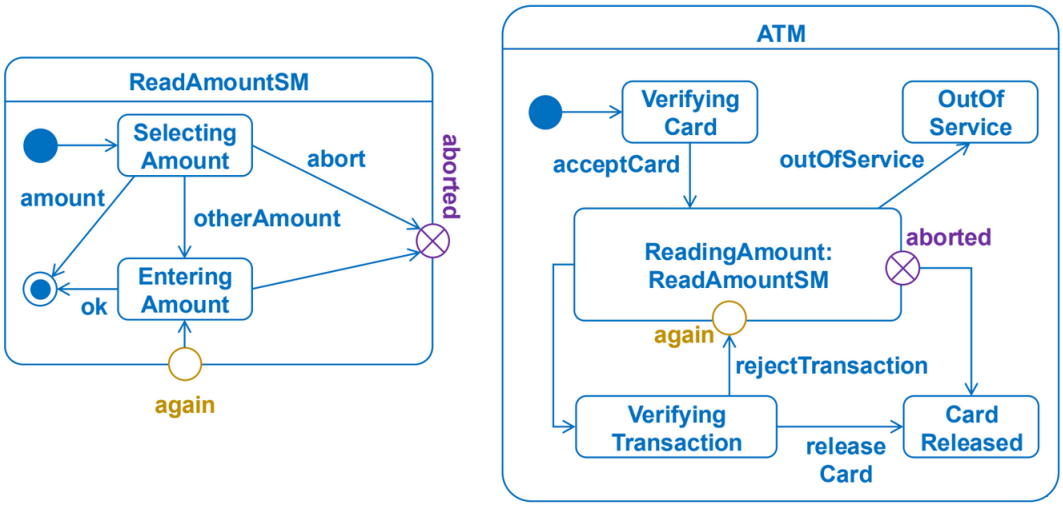


Sottomacchina Una sottomacchina permette di descrivere uno stato composito in una macchina a parte. Ha un **nome** o **tipo** e le sue istanze sono nella forma

nomeIstanza : Tipo

Bisogna definire **entry** ed **exit** point per collegare la sottomacchina alle transizioni di quella principale. Le **transizioni di completamento** scattano quando:

- Si raggiunge la **terminazione**:
  - Stato finale per lo *stato composito sequenziale*
  - Stati finali di tutte le regioni ortogonali per lo *stato composito parallelo*
  - Exit point
- Alla terminazione di **entry** e/o **do**
- Al raggiungimento di uno pseudo-stato **giunzione**



Note 4.3.11. L'attività **exit** viene eseguita quando scatta la transizione di completamento.

Osservazione 4.3.1 (Attività vs stati)**.** Il diagramma degli stati serve a mostrare l'evoluzione di un oggetto in risposta a degli eventi e descrive quindi l'evoluzione delle istanze. Quello delle attività, invece, permette di mettere in ordine le attività da fare descrivendo il flusso delle azioni da svolgere.

5 Progettazione

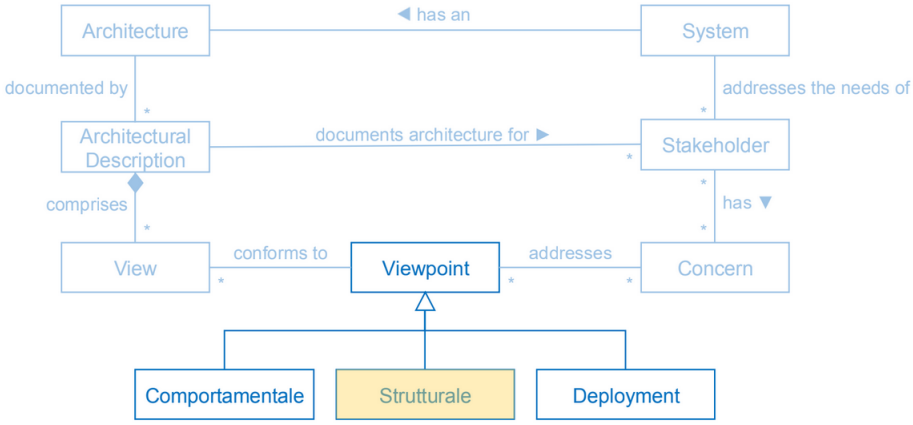
La fase di progettazione è tra quella della specifica (cosa fare) e quella della programmazione. Come risultato produce un'**architettura** che descrive **come** fare.
Possono esserci due livelli di astrazione:

- **Architetturale**: si scompone un sistema in più sottosistemi, ne si identificano e specificano le parti e le interconnessioni
- Di **dettaglio**: indica come la specifica di ogni parte sarà realizzata

Definizione 5.0.1 (Architettura software)**.** *L'architettura di un sistema software è la* ***struttura*** *del sistema, costituita dalle parti del sistema, dalle* ***relazioni*** *tra le parti e dalle loro proprietà visibili.*

Definizione 5.0.2 (Stile architetturale)**.** *Uno stile architetturale caratterizza una famiglia di architetture con caratteristiche simili (e.g. client-server, microservizi). Le funzionalità e le interazioni tra i componenti spesso seguono degli standard.*

Un'architettura si sviluppa in diverse **viste** e **stili**.



5.1 Vista comportamentale

Anche detta **component-and-connector**, descrive un sistema software come **composizione di componenti**, compreso di:

- **Interfacce** dei componenti
- Caratteristiche dei **connettori**
- Struttura del sistema in esecuzione (flusso dei dati, parallelismo, replicazioni, etc..)

Questa vista permette l'analisi delle **caratteristiche di qualità** a tempo di esecuzione (prestazioni, affidabilità, etc..) e di documentare lo **stile** dell'architettura.

5.1.1 Componente

Una componente software è un'unità concettuale di decomposizione del sistema a tempo di esecuzione. Incapsula un insieme di **funzionalità** e/o **dati**, restringendone l'accesso tramite delle **interfacce**.

Definizione 5.1.1 (Componente)**.** *Una componente software è un'unità di software* ***indipendente*** *e* ***riutilizzabile****.*

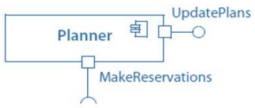
Definizione 5.1.2 (Sistema software)**.** *Un sistema software è una composizione di componenti software basata sulla connessione di più componenti e realizzata con interfacce dei componenti e connettori,*

In UML un componente è un **classificatore** e si compone di:

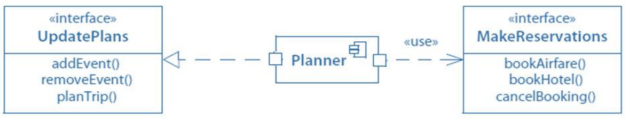
Porti I porti ne identificano i punti di interazione. Può essercene più di uno, possono fornire o richiedere una o più *interfacce* e possono avere *nomi* e *molteplicità*.

I porti possono avere la specifica delle interfacce in due modi:

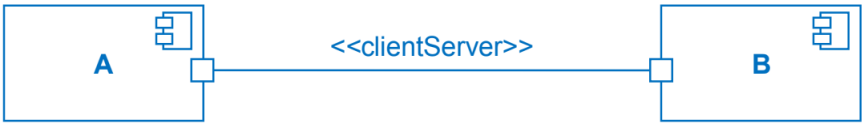
- **Sintetica**: si indica solo quando è *richiesta* (forchetta) o offerta (*lollipop*)



- **Estesa**: si specifica per esteso le operazioni richieste ed offerte



Connettori I connettori sono **canali di interazione** tra i porti di componenti. Non hanno un descrittore specifico e vengono rappresentati come associazioni. Per dare più informazioni si indica lo **stile** della connessione o con uno **stereotipo** o indicando i **ruoli**.



5.1.2 Stili

In questo tipo di vista uno stile architetturale è caratterizzato dalle **caratteristiche generali** delle componenti e dalle loro **interazioni** (quindi *porti* e *connettori*).

Pipes & filters Questo stile consiste in un flusso di elaborazione di dati che viaggiano lungo le **pipe** e sono processati dai **filter**. In particolare i *filter* sono i **componenti** che trasformano i flussi di dati mentre le *pipe* sono i **connettori** che fungono da canale di comunicazione unidirezionale bufferizzato che preserva l'ordine di ingresso.

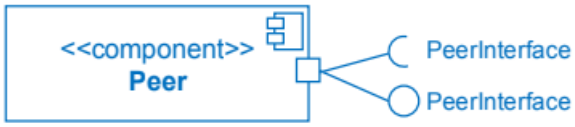
Client-server Questo stile prevede due componenti che possono essere su macchine diverse. Il **server** offre il servizio, aspetta le richieste dati ad un porto e può servirne su di esso più alla volta. Il **client** invia richieste al server e attende una risposta.

Il server in particolare prevede un **RequestListener** in attesa di richieste e un **RequestHandler** per ognuna di esse. Quest'ultimo elabora le richieste e:

- Se **stateless** le gestisce ognuna in maniera indipendente
- Se **stateful** consente richieste *composite* che consistono in più richieste *atomiche*, mantenendo un record di esse **sessione**

Master-slave Questo stile è una variazione di *client-server* in cui c'è solamente un servente (**slave**) e un cliente (**master**)

Peer to Peer Questo stile è una variazione di *client-server* dove tutti i componenti sono sia client che server e avviene uno scambio di servizi alla pari.

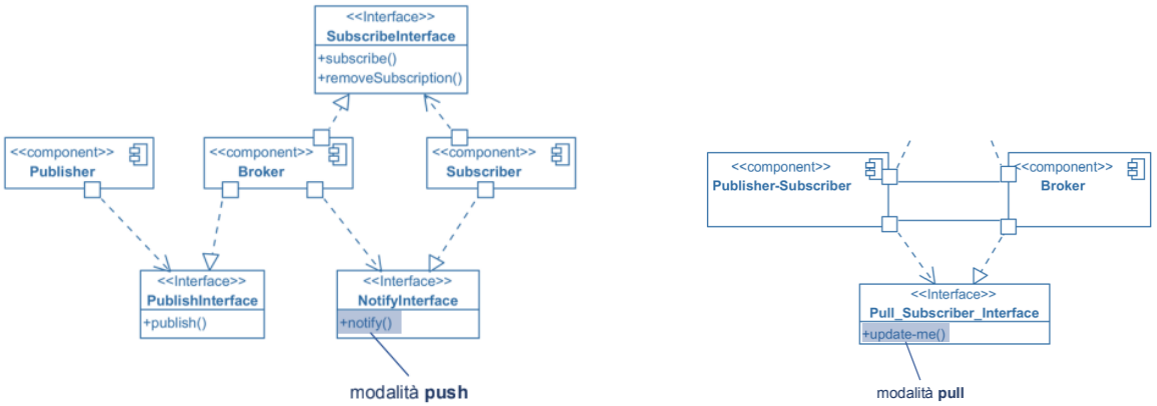


Publish-Subscribe In questo stile i componenti interagiscono in modo **event-based**. Abbiamo tre tipi di componenti:

- **Publisher**: produce classi di eventi
- **Subscriber**: si iscrive alle classi rilevanti
- **Broker**: smista gli eventi pubblicati

Un componente può essere sia *publisher* che *subscriber*. Tra di loro comunicano tramite il *broker*.I publisher non sanno quanti/quali subscriber ci siano e viceversa, garantendo **scalabilità**. Questo stile può funzionare in due modi:

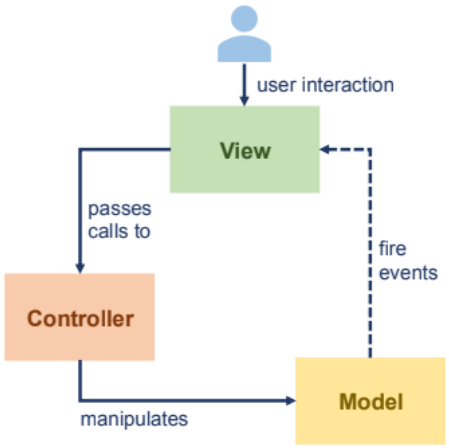
- **Push**: il broker invia attivamente i messaggi ai subscriber, controllandone la frequenza
- **Pull**: è il subscriber che manualmente recupera i messaggi dal broker. Migliora la scalabilità e la flessibilità dato che servono meno broker.



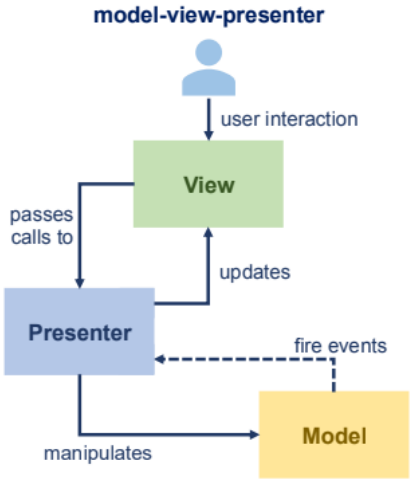
Process coordinator Un componente funge da **process coordinator** mentre gli altri sono passivi e non conoscono il loro ruolo nel processo ma si limitano a contribuire. Il *coordinator* conosce la sequenza di passi necessaria, invoca le funzionalità richieste e fornisce una risposta.

Model-View-Controller/Publisher Questo stile si basa su tre elementi:

- **Model**: nucleo funzionale che implementa la business logic dell'applicazione e ne rappresenta i dati su cui essa lavora
- **View**: presentazione del model all'utente, possono essercene diverse
- **Controller**: controlla l'input dell'utente e lo traduce in operazioni da eseguire su *model* oppure **presenter** in alternativa



Controller



Presenter

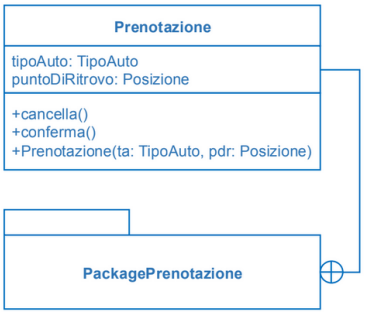
5.2 Vista strutturale

Descrive la struttura di un sistema software come insieme di **unità di realizzazione**, ovvero di codice. Permette di analizzare le **dipendenze**, progettare i **test** e valutare la **portabilità**.
La vista strutturale in UML include:

- **Classi** con specifica delle *operazioni* più dettagliata rispetto alla descrizione del dominio
- **Package**

5.2.1 Decomposizione

Una classe può far parte (essere contenuta in) un package, che a sua volta può far parte di uno più grande. La decomposizione può essere fatta per **apprendimento** del sistema o per l'**allocazione del lavoro** e può seguire tre criteri:



- Incapsulamento per **modificabilità**
- Supporto alle scelte **costruisci/compra**
- Moduli comuni in **linee di prodotto**

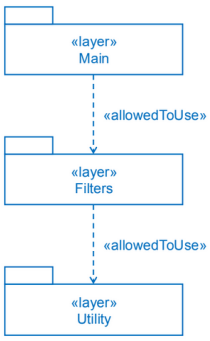
Note 5.2.1. La relazione "parte di" si può rappresentare anche con l'inclusione grafica in un package.

5.2.2 Uso

Un modulo A può usarne un altro B per soddisfare i suoi requisiti. Questo permette di pianificare uno sviluppo incrementale e creare test di unità ed integrazione. Sono permessi i cicli anche se pericolosi.

5.2.3 Strati

Nella vista strutturale a strati ogni elemento (**strato**) è un insieme coeso di moduli a volte raggruppati in segmenti, il quale offre un'interfaccia pubblica per i suoi servizi. La relazione *allowedToUse* è un caso particolare di quella di uso ed è **antisimmetrica** e **non implicitamente transitiva**.
Questa vista favorisce modificabilità, portabilità e controllo della complessità.



5.2.4 Generalizzazione

In questa vista gli elementi sono i **moduli** (classi o packages) che hanno una relazione di generalizzazione tra di loro. Permette di rappresentare relazioni di sotto tipo tra classi e la relazione tra un framework[1] e una sua specializzazione nel caso dei package.

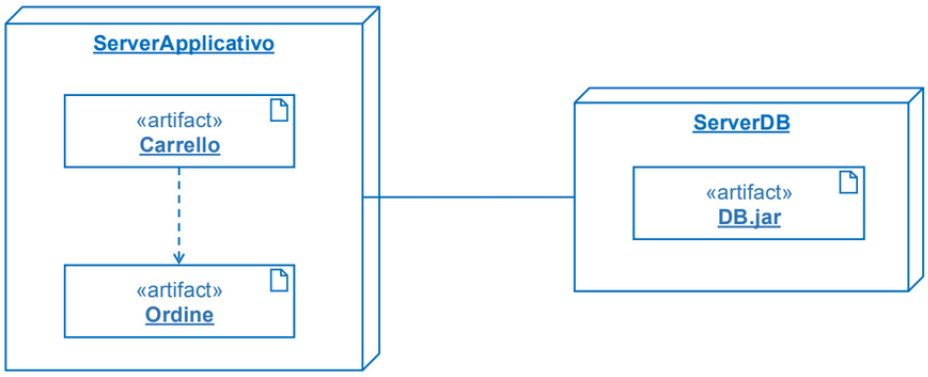
5.3 Vista di deployment

Descrive l'allocazione del software su ambienti di esecuzione e permette di valutarne **prestazioni** e **affidabilità**. Contiene:

- **Artefatti** prodotti da un processo di sviluppo software o dal funzionamento di un sistema, rappresentati dallo stereotipo <<*artifact*>>
- **Ambienti di esecuzione** o nodi hardware, rappresentati come *parallelepipedi*

Questi elementi possono avere due tipi di relazioni:

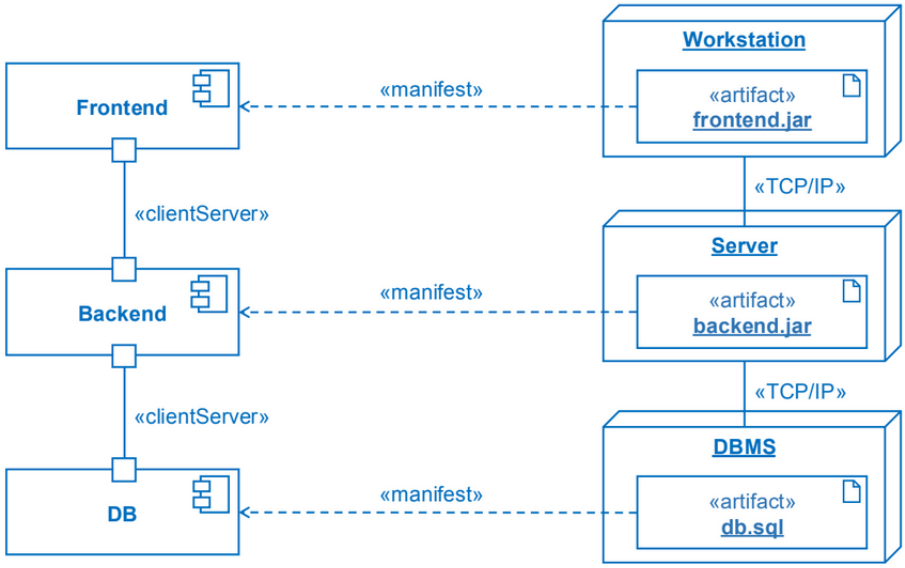
- **Dislocazione** di artefatti negli ambienti, rappresentata con il *contenimento*
- **Interconnessione** tra gli ambienti di esecuzione, rappresentate con relazioni stereotipate



[1] Collezione di classi, anche astratte, con relazioni d'uso tra loro.

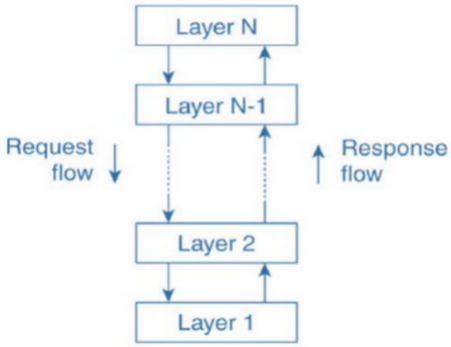
5.4 Vista ibrida

La vista ibrida si compone di quella comportamentale e di quella di deployment su di un'applicazione 3-tier.



5.5 Altri esempi

5.5.1 Architettura a livelli

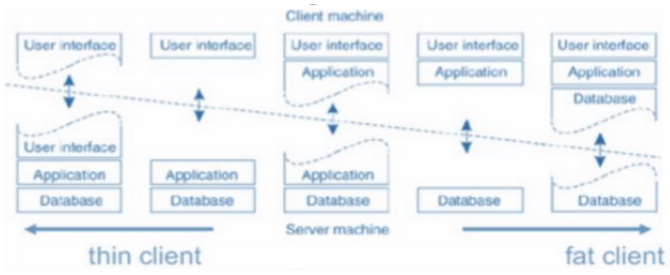


I componenti sono organizzati in livelli (layer) dove uno a livello i può invocarne uno del livello sottostante $i-1$. Le richieste scendono lungo la gerarchia mentre le risposte salgono.

5.5.2 Architettura multi-livello

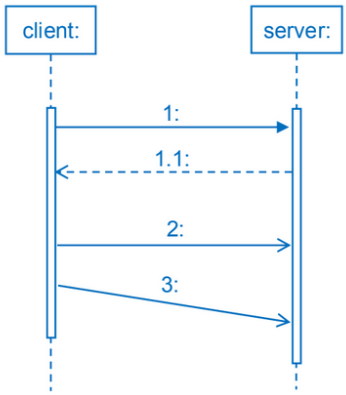
In questa architettura avviene un mapping tra livelli logici (layer) e fisici (tier). Può avere da 1 ad N livelli, dove all'aumentare di questi si guadagna in **flessibilità**, **funzionalità** e possibilità di **distribuzione** ma si introducono problemi di **prestazione**.
Dal punto di vista fisico, l'architettura può essere:

- **1-tier**: mainframe e terminale
- **2-tier**: macchina client e singolo server
- **3-tier**: ciascun livello su una macchina separate



6 Diagrammi di sequenza

I diagrammi di sequenza descrivono le **interazioni** tra oggetti riorganizzandole in una sequenza temporale. Nella fase di *analisi* dei casi d'uso formalizzano la sequenza principale degli eventi mentre in fase di *progettazione* illustrano come l'architettura realizza i requisiti.



In UML per ogni partecipante viene disegnata una **linea di vita** che può essere tratteggiata quando l'oggetto è inattivo e doppia-continua quando è attivo. Per rappresentare le **interazioni** vengono usate frecce, che possono essere:

- **Sincrone**
- **Ritorno**
- **Asincrone**
- Asincrone con **consumo di tempo**

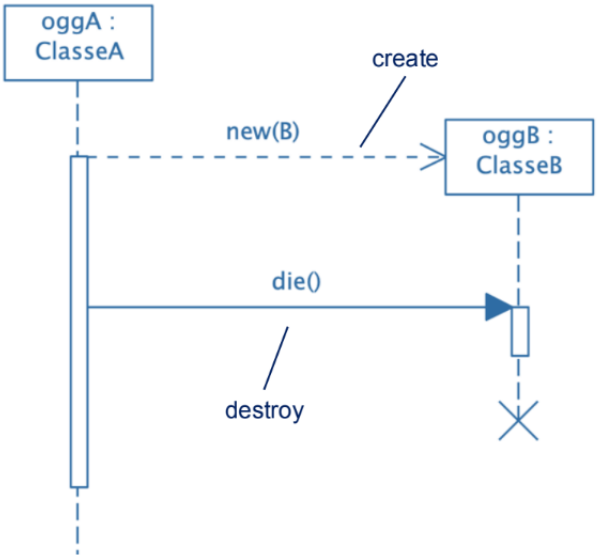
6.1 Etichette

È possibile mettere etichette ai messaggi:

- n è il numero del messaggio nella sequenza
- **attr** è l'attributo a cui assegnare il valore restituito
- **name** identifica e descrive il messaggio
- **arg** sono i parametri
- **value** rappresenta il valore restituito

6.2 Creazione e distruzione

Un oggetto può crearne o eliminarne un altro attraverso lo scambio di messaggi.

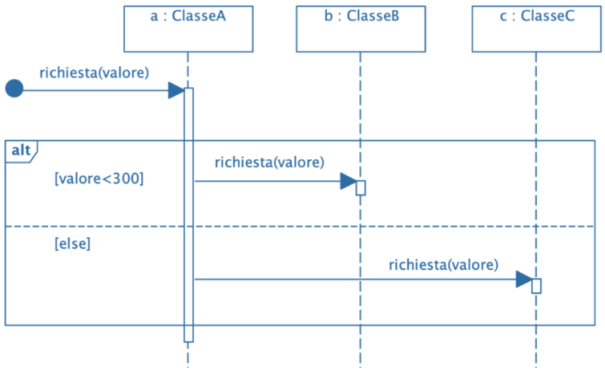


6.3 Frame

6.3.1 Condizionale

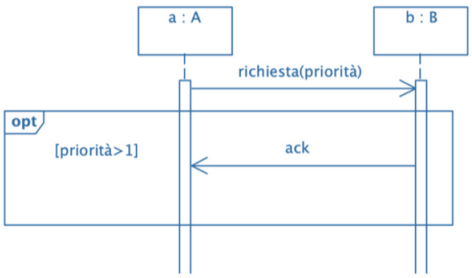
È identificato dalla parola **alt**. I subframe possono essere etichettati con guardie:

- Se non c'è la guardia, è *true*
- Se ci sono più guardie vere è una situazione di non determinismo
- Se ci sono tutte le guardie false, si salta il frame



6.3.2 Opzionale

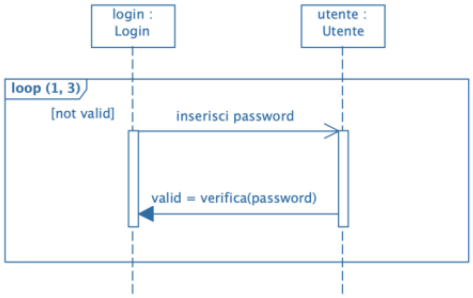
Il frame opzionale è identificato dalla parola chiave **opt** e le interazioni sono eseguite solo se la guardia è vera, altrimenti il frame viene saltato.



6.3.3 Iterativo

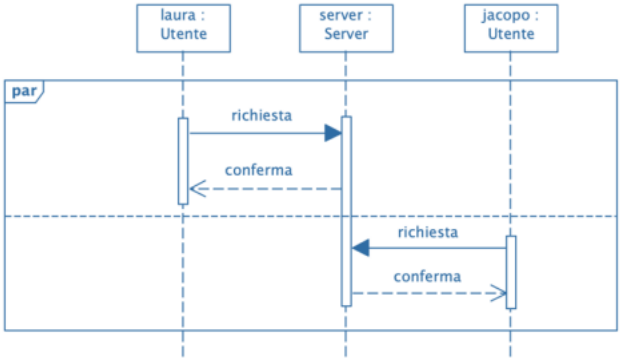
Questo frame ripete il suo contenuto da *min* a *max*, **loop(min, max)** e finché la **condizione** è vera. Quindi le implementazioni dei classici cicli sono:

- **while**: loop(0,*)[guardia] e loop[guardia]
- **do-while**: loop(1,*)[guardia]
- **for**: loop(n,n) e loop(n)



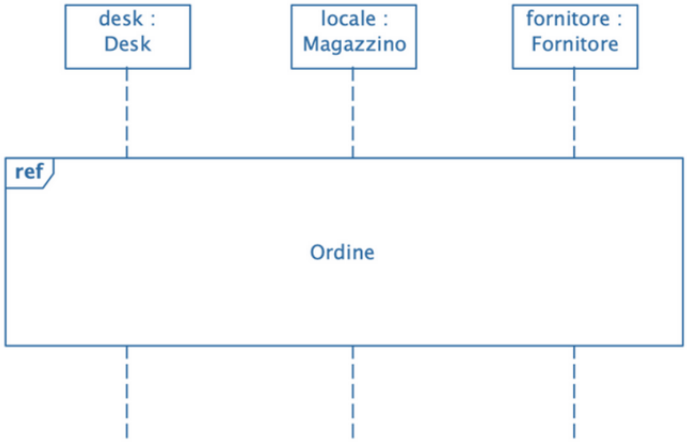
6.3.4 Parallelo

Identificato da **par**, indica che le interazioni nei sotto-frammenti sono eseguite in parallelo con una semantica ad interleaving.



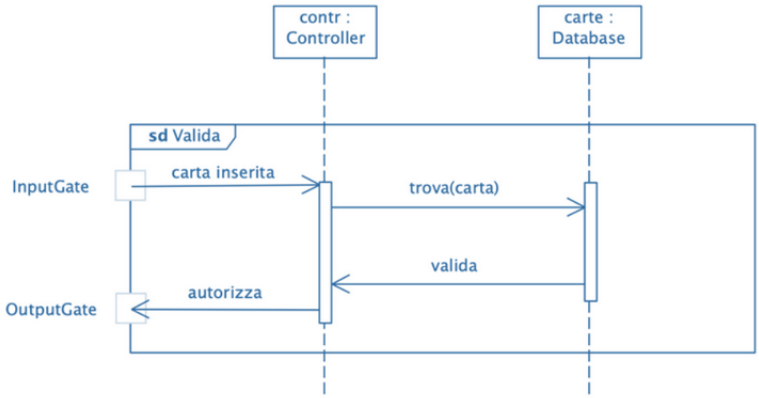
6.4 Inclusione

È possibile includere un'interazione definita altrove tramite **ref**.



6.5 Gate

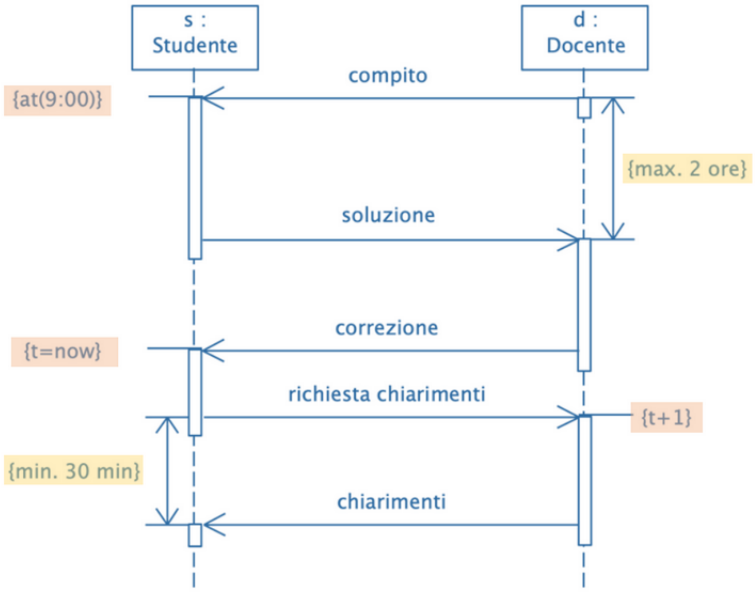
Un gate è un punto di ingresso o di uscita sul bordo di un diagramma o di un frame. Consente la spedizione e la ricezione di messaggi ed è identificato da un nome.



6.6 Vincoli di durata

Sono espressi tra parentesi graffe e consentono di specificare:

- **quando** avviene un evento, tramite **at**(orario) o **now**
- **quanto** tempo tra due eventi, che può essere un numero effettivo o un range (min, max)



7 Principi di progettazione

Le tecniche di progettazione e le good practice mirano a produrre un sistema che realizzi **requisiti funzionali** e di **qualità**, mantenendolo facilmente **manutenibile** e **riusabile**.

7.1 Principi generali

7.1.1 Information hiding

Consiste nella separazione tra l'**interfaccia** visibile e il **corpo**, ovvero l'implementazione che è privata e/o nascosta. I componenti o i moduli sono come **black box** che forniscono e richiedono funzionalità. Rendono nota all'esterno solo la loro **interfaccia** ma nascondono algoritmi e strutture dati usati internamente.
I vantaggi principali sono:

- **Comprensibilità**: non servono i dettagli implementativi per usare una feature
- **Manutenibilità**: si può cambiare il corpo di un'unità senza cambiarne altre
- **Team work**: corpi di unità diverse possono essere sviluppate da team indipendenti
- **Sicurezza**: i dati di un'unità sono modificabili solo internamente

Osservazione 7.1.1 (Incapsulamento)**.** L'incapsulamento è diverso dall'information hiding: il primo rappresenta la capacità degli oggetti di mantenere al loro interno attributi e metodi ed è una proprietà dei linguaggi di programmazione. Permette, *senza garantire*, l'information hiding tramite i costrutti *public* e *private*

Lo standard de-facto per l'accesso agli attributi privati, che quindi ne nasconde la rappresentazione, è tramite un'interfaccia di getter e setter:

- **getter**: restituisce un attributo come valore e senza cambiare lo stato dell'oggetto
- **setter**: modifica lo stato dell'oggetto cambiando il valore dell'attributo

7.1.2 Astrazione

Il principio di astrazione si applica a:

- **Controllo**: una procedura è vista come modulo di astrazione del flusso di controllo, nascondendo l'algoritmo utilizzato. Sono organizzate in classi di moduli e costituiscono la libreria.
- **Dati**: la struttura dati, che regolamenta accesso e modifica, è astratta, garantendo un'interfaccia stabile indipendentemente dal cambio di implementazione. Non sono puramente funzionali in quanto le operazioni offerte possono cambiare lo stato.

7.1.3 Coesione

La coesione è quanto strettamente correlate sono le funzionalità offerte da un modulo o da una componente. Funzionalità vicine devono stare nella stessa unità e un sistema è coeso se tutti gli elementi di ogni unità sono strettamente collegati. Esistono diversi tipi di coesione:

- **Funzionale**: gli elementi collaborano per realizzare una funzionalità
- **Temporale**: gli elementi sono azioni che devono essere eseguite in uno stesso arco di tempo (difficilmente coese e riutilizzabili)
- **Logica**: gli elementi sono correlati logicamente ma non funzionalmente (e.g. raccolta dati, analisi e generazione dei report). Possono essere operazioni correlate ma funzioni significativamente diverse o operazioni debolmente connesse e difficilmente riutilizzabili
- **Accidentale**: gli elementi non sono correlati ma sono piazzati assieme

7.1.4 Disaccoppiamento

Indica quanto sono slegate le unità di progettazione in termini di dipendenze funzionali o messaggi. In un sistema molto accoppiato (**high coupling**) le modifiche alle varie unità impattano anche le altre a distanza, rendendo difficile la manutenzione. In quelli poco accoppiati (**low coupling**) l'impatto delle modifiche è limitato.

Osservazione 7.1.2. Un alto grado di coesione contribuisce a ridurre il grado di accoppiamento.

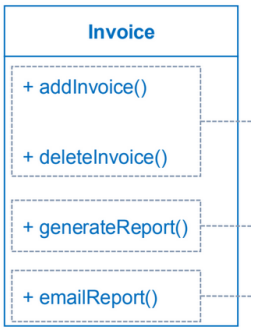
7.2 Collezione di principi

7.2.1 SOLID

SOLID include cinque principi di base per progettazione e sviluppo object oriented.

Single Responsibility Una classe o metodo deve avere un solo motivo per cambiare: un cambiamento deve impattare solo la classe che realizza quella funzionalità, se ci sono più motivi vanno create più classi. Deve quindi essere **funzionalmente coesa**.
Le eccezioni a questa regola sono:

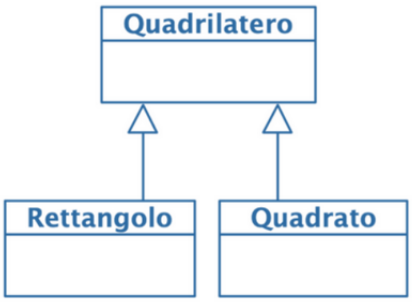
- La separazione di una classe introdurrebbe complessità non necessaria
- Un motivo di cambiamento è tale se è una reale possibilità di cambiamento nel sistema



Open Closed Le entità software devono essere aperte per estensione ma chiuse per modifiche. Se i requisiti cambiano si deve estendere il comportamento aggiungendo nuovo codice, senza cambiare quello esistente e funzionante. Si può implementare tramite classi **astratte** e **concrete**, **delega** e **plugin**.

Liskov Substitution Deriva dal principio di sostituzione di Barbara Liskov.

Definizione 7.2.1 (Principio di sostituzione)**.** *Dato S* ***sottotipo*** *di T, o_s e o_t oggetti di tipo rispettivamente S e T e P un programma definito in termini di T. Se o_s è usato al posto di o_t, il comportamento di P è immutato.*



Interface Segregation Le interfacce devono essere a **grana fine** e **specifiche** per ogni cliente: i client non devono dipendere da interfacce che non usano e si devono mettere solo i metodi utilizzati.

Esempio 7.2.1. Ad esempio un lavoratore lavora e mangia. Può essere umano o robot ma quest'ultimo non può mangiare. Conviene quindi separare l'interfaccia del lavoratore da quello del mangiatore e il robot implementerà solo la prima.

Dependency Inversion Si deve programmare per l'interfaccia e non per l'implementazione: i moduli di alto livello non devono dipendere da quelli di basso livello ma devono entrambi dipendere da **astrazioni** che a loro volta non devono dipendere dai dettagli.

Esempio 7.2.2. Dati dei moduli per leggere, scrivere e copiare, è necessario che questi non dipendano, ad esempio, dal mezzo su cui eseguire le operazioni. Potrebbero essere implementate per la lettura da tastiera e scrittura su schermo oppure lettura da tastiera e scrittura su disco.

Dependency Injection È una forma di inversione delle dipendenze dove il cliente non sa come costruire i servizi che vuole chiamare ma delega ad un **iniettore**, che poi a sua volta passa al client i servizi. È quindi sufficiente sapere solo le interfacce dei servizi che definiscono come il client può usarli e in questo modo le responsabilità di costruzione e di utilizzo sono separate.

7.2.2 GRASP

Questa famiglia di principi di progettazione si compone di: **General Responsibility Assignment Software Patterns**. La progettazione è guidata dai **casi d'uso** che vengono realizzati in termini di oggetti collaborativi utilizzando diagrammi e pattern e assegnando responsabilità alle classi.

In particolare le **responsabilità** sono obblighi che un oggetto ha, definiti in termini di comportamento, e legate al dominio del problema. Possono essere di due tipi:

- **Fare**: fare qualcosa (e.g. creare un oggetto o fare un calcolo) o iniziare, controllare e coordinare le azioni di altri oggetti
- **Conoscere**: conoscere i dati privati, gli oggetti correlati, i dati che possono derivare o calcolare. Generalmente si deducono dal modello di dominio.

Osservazione 7.2.1. Una responsabilità non è un metodo: questi ultimi sono implementati per soddisfare le prime e la traduzione da responsabilità a metodi dipende dalla **granularità** della prima.

7.3 Delega

L'ereditarietà presenta alcuni problemi: obbliga ad ereditare tutti i metodi anche se non rilevanti o non adatti e rende la sottoclasse strettamente legata all'implementazione della superclasse. La delega, al contrario, rende esplicito l'utilizzo parziale e consente di **controllare quanti e quali** metodi la classe delegata usa. Il costo è rappresentato dai metodi di delega aggiuntivi.

La delega riduce leggermente le **prestazioni** per l'invocazione di un altro oggetto rispetto all'uso di un metodo ereditato. Inoltre non può essere usata con classi astratte e non impone una struttura al progetto.

7.4 Qualità del software

7.4.1 Functional suitability

Il grado con cui un prodotto o un sistema fornisce funzionalità che soddisfino le richieste esplicite ed implicite quanto usato sotto certe condizioni. Si divide in:

- **Functional completeness**: quando le funzionalità offerte coprono i task specificati e gli obiettivi degli utenti considerati
- **Functional correctness**: quanto siano accurati i risultati forniti agli utenti considerati
- **Functional appropriatness**: quanto le funzionalità offerte facilitano il raggiungimento del task e gli obiettivi considerati

7.4.2 Performance efficiency

Il grado con cui un sistema esegue le sue funzioni nella quantità di tempo specificata e con determinati parametri di output assieme alla sua efficienza nell'uso delle risorse date determinate condizioni. Si divide in:

- **Time behaviour**: quanto vengono rispettati i requisiti in termine di *response time* e *throughput*
- **Resource utilization**: quanto vengono rispettati i requisiti in termini di *quantità* e *tipologia* di risorse computazionali utilizzate
- **Capacity**: quanto vengono rispettati i requisiti in termini di capacità (limiti massimi)

7.4.3 Compatibility

Il grado con cui un sistema scambia informazioni con altri ed esegue funzioni mentre condivide lo stesso ambiente comune e le stesse risorse. Si divide in:

- **Co-existance**: quanto il sistema rimane efficiente mentre condivide ambiente e risorse
- **Interoperability**: quanto il sistema riesce a scambiare informazioni con altri e sfruttare quelle ricevute

7.4.4 Interaction capability

Il grado con cui un sistema può interagire con utenti specifici per scambiare informazioni nell'interfaccia utente per completare task in diversi contesti. Si divide in:

- **Appropriatness recognizability**: quanto gli utenti riconoscono se il sistema è appropriato
- **Learnability**: quanto le funzionalità sono apprendibili da un utente in un certo lasso di tempo
- **Operability**: quanto sia facile operare e controllare il sistema
- **User error protection**: quanto il sistema prevenga errori operativi
- **User engagement**: quanto *engaging* sia il sistema
- **Inclusivity**: quanto il sistema può essere utilizzato da utenti con background diversi
- **User assistance**: quanto il sistema può essere usato da utenti diversi per gli stessi task
- **Self-descriptiveness**: quanto bene sono presentate le informazioni e le funzionalità per rendere ovvio l'utilizzo del sistema

7.4.5 Reliability

Il grado con cui un sistema esegua determinate funzioni in determinate condizioni per un certo periodo di tempo. Si divide in:

- **Faultlessness**: quanto il sistema funziona senza fallimenti
- **Availability**: quanto il sistema sia operativo ed accessibile quando richiesto
- **Fault tolerance**: quanto il sistema riesca a ripristinare lo stato desiderato in caso di fallimento

7.4.6 Security

Il grado con cui un sistema riesce a difendersi da attacchi e proteggere le informazioni e i dati. Si divide in:

- **Confidentiality**: quanto il sistema assicura che i dati siano accessibili solo a persone autorizzate
- **Integrity**: quanto il sistema assicura che lo stato del sistema e i dati siano protetti da modifiche non autorizzate
- **Non-repudiation**: quanto sia dimostrabile che azioni ed eventi abbiano avuto luogo
- **Accountability**: quanto sono tracciabili le azioni di una entità
- **Authenticity**: quanto è dimostrabile che l'identità di un soggetto o risorsa sia effettivamente quella
- **Resistance**: quanto il sistema è resistente da attacchi di malintenzionati

7.4.7 Maintainability

Il grado di efficacia ed efficienza con cui un sistema può essere modificato per migliorarlo, correggerlo o adattarlo a cambi dell'ambiente e dei requisiti. Si divide in:

- **Modularity**: quanto poco i cambiamenti di un modulo di un sistema impattano sugli altri
- **Reusability**: quanto è riusabile il sistema e le sue parti
- **Analyzabilty**: quanto è facile analizzare l'impatto di un eventuale cambiamento, diagnosticare le cause di eventuali problemi o identificare le parti da modificare
- **Modifiability**: quanto sia modificabile il sistema senza degradarne le qualità
- **Testability**: quanto sia facile determinare i criteri di test e testare effettivamente il sistema

7.4.8 Flexibility

Il grado con cui un prodotto può essere adattato ai cambiamenti dei requisiti, del contesto o dell'ambiente.

- **Adaptability**: quanto il sistema è adattabile a cambiamenti HW e SW degli ambienti di esecuzione
- **Scalability**: quanto bene il sistema gestisce i cambiamenti di workload e variability
- **Installability**: quanto efficientemente il sistema può essere installato o disinstallato
- **Replaceability**: quanto il sistema può sostituirne un altro sviluppato per gli stessi scopi

7.4.9 Safety

Il grado con cui un prodotto evita stati in cui la vita umana, la salute o gli oggetti siano messi in pericolo.

- **Operational constraint**: quanto il sistema è vincolato a rimanere nei parametri di sicurezza
- **Risk identification**: quanto eventuali rischi di sicurezza sono identificati
- **Fail safe**: quanto in caso di fallimenti o pericoli il sistema riesca ad agire in safe mode
- **Hazard warning**: quanto il sistema fornisca avvisi su rischi inaccettabili per gli operatori in modo che essi possano reagire spontaneamente
- **Safe integration**: quanto il sistema riesca a mantenere la *safety* se integrato con altri sistemi

7.5 Qualità delle implementazioni

7.5.1 Client-Server 2-N tier

Disponibilità	I server di ogni tier possono essere replicati quindi in caso di fallimento il problema sarebbe minimo.
Fault tolerance	Se un client sta comunicando con un server che fallisce, la richiesta viene reindirizzata ad un server replicato in maniera trasparente.
Modificabilità	Il disaccoppiamento e la coesione di questa architettura favoriscono la modificabilità.
Performance	Buone performance ma bisogna tenere d'occhio il numero di threads, la velocità delle comunicazioni e il volume dei dati scambiato.
Scalabilità	Basta replicare i server di ogni tier. Unico bottleneck potrebbe essere la base di dati in comune.

7.5.2 Pipes and filters

Disponibilità	Avendo componenti e possibilità di connetterli sufficienti a formare una catena.
Fault tolerance	Occorre riparare una catena interrotta usando componenti replica.
Modificabilità	Presente se le modifiche interessano una o poche componenti.
Performance	Dipende dalla capacità del canale di comunicazione e dalla performance del filtro più lento.
Scalabilità	Ok.

7.5.3 Publish-Subscribe

Disponibilità	Si possono creare cluster di dispatcher.
Fault tolerance	Si crea un dispatcher replica.
Modificabilità	Si possono aggiungere publisher e subscriber ponendo attenzione al formato dei messaggi.
Performance	Ok ma con un compromesso tra velocità e requisiti tipo affidabilità e/o sicurezza.
Scalabilità	Con un cluster si può gestire un volume molto elevato di messaggi.

7.5.4 P2P

Disponibilità	Dipende dal numero di nodi, si assume di si.
Fault tolerance	Gratis.
Modificabilità	Si se si è interessati solo alla parte di comunicazione.
Performance	Dipende dal numero di nodi, dalla rete e dagli algoritmi.
Scalabilità	Gratis.