Ingegneria del Software

Angelo Passarelli

November 30, 2023

Appunti basati sulle lezioni e dispense della professoressa Laura Semini [1]

[1] http://didawiki.cli.di.unipi.it/doku.php/informatica/is-a/start

Contenuti

0 Introduzione **5**

1 Modelli di Ciclo di Vita **6**
1.1 Modelli Sequenziali 6
1.1.1 Build-and-Fix 6
1.1.2 Modello a Cascata 7
1.1.3 Modello a V 7
1.2 Modelli Iterativi 8
1.2.1 Rapid Prototyping 8
1.2.2 Modello Incrementale 8
1.2.3 Modello a Spirale 10
1.3 Unified Process 11
1.4 Processi Agili 11
1.4.1 Il Manifesto di Snowbird 12
1.4.2 eXtreme Programming 12
1.4.3 SCRUM 13

2 Analisi dei Requisiti **15**
2.1 Studio di Fattibilità 15
2.2 Dominio 15
2.3 Requisiti 15
2.4 Documento dei Requisiti 16
2.5 Fasi dell'Analisi dei Requisiti 16
2.5.1 Acquisizione 16
2.5.2 Elaborazione 16
2.5.3 Convalida 17
2.5.4 Negoziazione 18
2.5.5 Gestione 18
2.6 Casi d'uso 19
2.7 User Stories 19

3 Linguaggio UML e Casi d'Uso **20**
3.1 UML 20
3.2 Diagramma dei Casi d'Uso 20
3.3 Narrativa dei Casi d'Uso 21
3.3.1 Inclusione 22
3.3.2 Estensione 23
3.4 Classi e Oggetti 24
3.4.1 Diagramma delle Classi 24
3.4.2 Relazioni 25
3.4.3 Aggregazione e Composizione 27
3.4.4 Generalizzazione 27
3.4.5 Classi Astratte 28
3.4.6 Interfacce 28

3.4.7 Dipendenze 28
3.4.8 Classi Associazione 28
3.4.9 Classi di Analisi 29
3.4.10 Diagramma degli Oggetti 30
3.5 Diagramma delle Attività 30
3.5.1 Segnali & Eventi 31
3.5.2 Sotto-Attività 32
3.5.3 Partizioni 32
3.6 Macchina a Stati 33
3.6.1 Eventi 33
3.6.2 Stato Composito Sequenziale 35
3.6.3 Stato Composito Parallelo 36
3.6.4 Sottomacchine 36
3.6.5 Pseudostati 37

4 Architetture Software 39
4.1 Vista Comportamentale 39
4.1.1 Stili Architetturali 41
4.2 Vista Strutturale 42
4.2.1 Vista Strutturale a Strati 42
4.2.2 Vista Strutturale di Generalizzazione 42
4.3 Vista Logistica 42

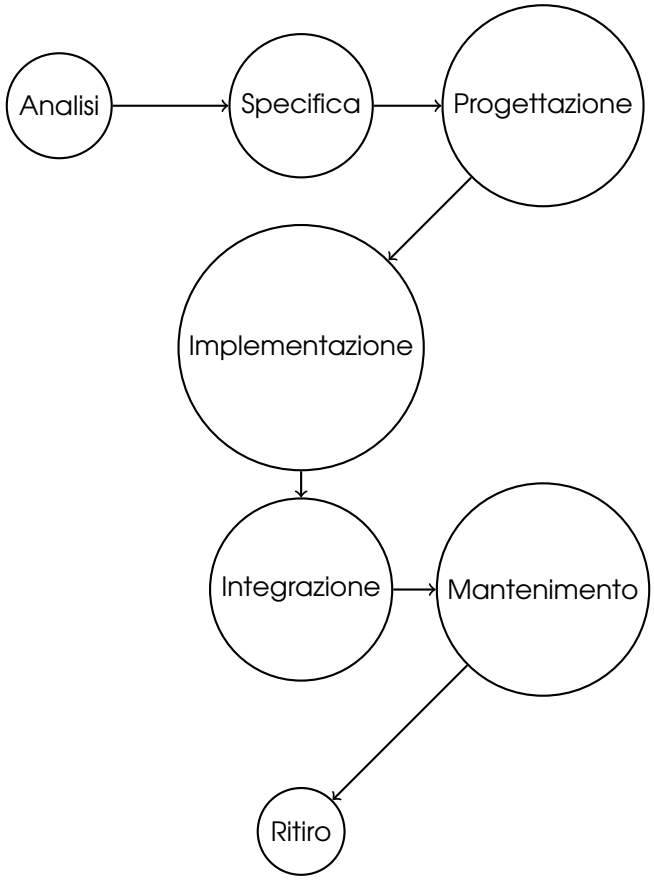
5 Diagrammi di Sequenza 44

6 Progettazione Software 47
6.1 Principi Generali 47
6.1.1 Information Hiding 47
6.1.2 Coesione 48
6.1.3 Disaccoppiamento 48
6.2 SOLID 48
6.3 GRASP 49
6.4 Qualità del Software 49
6.5 Stili dell'Architettura 50
6.6 Design Pattern 51
6.6.1 Strategy 52
6.6.2 State 52
6.6.3 Factories 53
6.6.4 Singleton 54
6.6.5 Adapter 55
6.6.6 Proxy 56
6.6.7 Decorator 57

7 Verifica e Validazione **58**
7.1 Verifica Statica 59
7.2 Verifica Dinamica 60
7.2.1 Progettazione Casi di Test 60
7.3 Testing Combinatorio 62
7.4 Pairwise Testing 62
7.5 Criteri White-Box 62
7.6 Fault Based Testing 63
7.7 Oracolo ed Output Attesi 64
7.8 Test di Sistema 65

0 Introduzione

Fasi del progetto



Specificità del Software

1. **Fault toulerance**: capacità del software di essere tollerante ai guasti.
2. **Difetto latente**: difetto nascosto che si trova difficilmente in fase di testing; e anche nel caso comparisse è quasi impossibile da ritrovare.
3. **Robustezza**: capacità di funzionare anche con input non previsti e/o non testati.
4. Il software non presenta costi materiali e nemmeno costi marginali, ovvero il costo di un'unità del prodotto.

5. Infine il software non si consuma nel tempo, ma potrebbe diventare obsoleto.

La Manutenzione

Costi La fase di manutenzione è quella che richiede costi più alti. Per evitare uno spreco durante questa fase è necessario studiare bene l'analisi dei requisiti, in quanto un errore in questa fase si propagherà in modo esponenziale, in termini di costi, nelle fasi successive.

La manutenzione si divide in:

- **Manutenzione Correttiva**: rimuove gli errori, lasciando invariata la specifica.
- **Manutenzione Migliorativa**: consiste nel cambiare quella che è la specifica, e a sua volta può dividersi in:
  - Perfettiva: modifiche per migliorare e/o introdurre nuove funzionalità.
  - Adattiva: modifiche indotta da cambiamenti esterni, come leggi o modifiche all'hardware o al sistema operativo.

Stakeholders

- Fornitore: colui che sviluppa il software.
- Committente: chi lo richiede e paga.
- Utente: chi lo usa.

1 Modelli di Ciclo di Vita

Definizione (Processo Software)**.** Con processo software si indica il percorso da seguire per sviluppare un prodotto o più nello specifico un software. Fanno parte del processo sia gli strumenti e le tecniche per lo sviluppo che i professionisti coinvolti.

1.1 Modelli Sequenziali

1.1.1 Build-and-Fix

Il prodotto è sviluppato senza alcuna fase di progettazione preliminare, lo sviluppatore scrive il software e poi lo modifica ogni volta che non soffisfa il committente.

Contro Diventa improponibile per progetti grandi e la manutenzione diventa difficile senza documentazione nè specifica.

1.1.2 Modello a Cascata

Questo modello è stato il primo a distingure il processo software in più fasi, evidenziando l'importanza della progettazione e dell'analisi.

Viene chiamato anche modello *document driven* dato che ogni fase produce un documento, e per passare alla successiva occorre aver approvato il documento della fase precedente.

Contro Troppo pesante da seguire, inoltre non si può tornare indietro, e mancando l'interazione con il cliente, se non è soddisfatto, và tutto ripetuto dall'inizio.

1.1.3 Modello a V

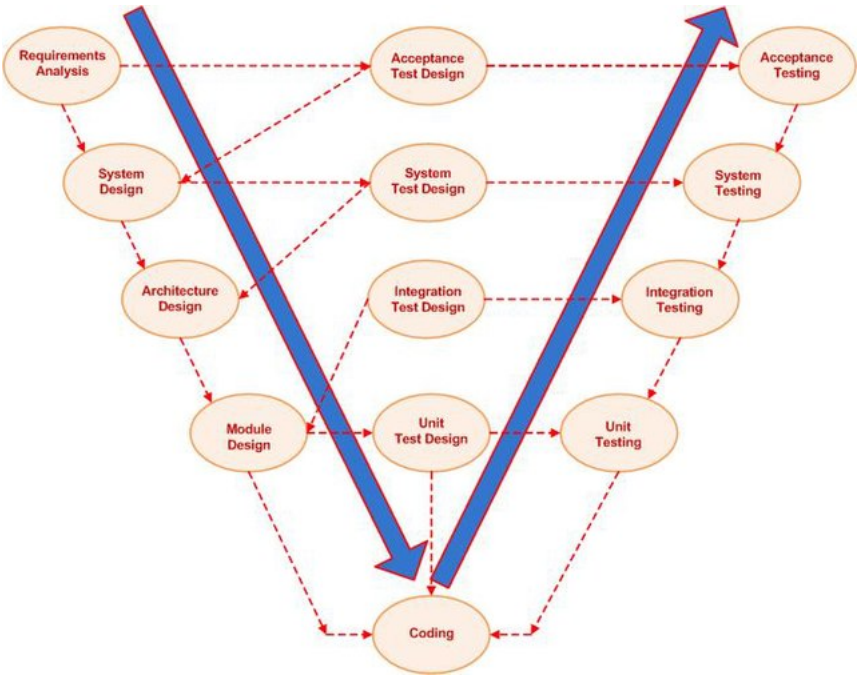


Figure 1: Le frecce blu rappresentano il *tempo*, mentre quelle tratteggiate le *dipendenze*

Questo modello evidenzia come sia possibile progettare i test durante le fasi di sviluppo (quelle a sinistra, prima della fase di *coding*). Mentre sulla destra sono presenti i test veri e propri che devono verificare e convalidare l'attività in corrispondenza sulla sinistra.

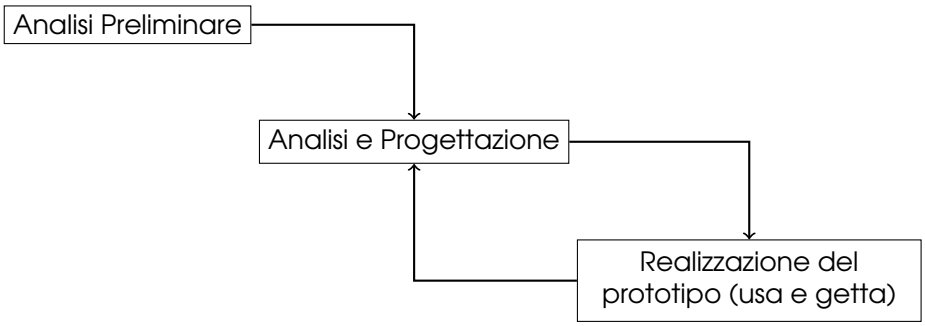
Standard SQA Questo modello è uno degli standard *SQA* (Software Quality Assurance), usato per descrivere le attività di test durante il processo di sviluppo.

1.2 Modelli Iterativi

1.2.1 Rapid Prototyping

L'obbiettivo è quello di costruire rapidamente un prototipo del software per permettere al committente di sperimentarlo.

Questo modello diventa utile quando i requisiti non sono chiari, quindi ogni prototipo aiuterà il cliente a descriverli meglio.

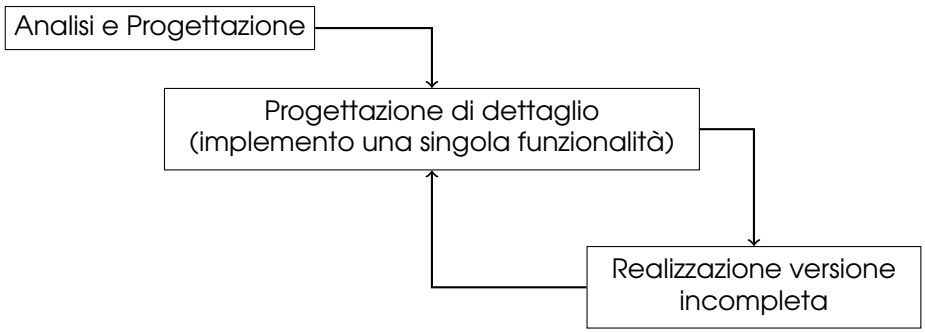


1.2.2 Modello Incrementale

Il software viene costruito in modo iterativo, aggiungendo di volta in volta nuove funzionalità.

I requisiti e la progettazione vengono definiti inizialmente, per questo è possibile applicarlo solo in caso di requisiti stabili.

Contro Se non viene realizzata una buona progettazione, questo modello sfocia in un *Build-and-Fix*.

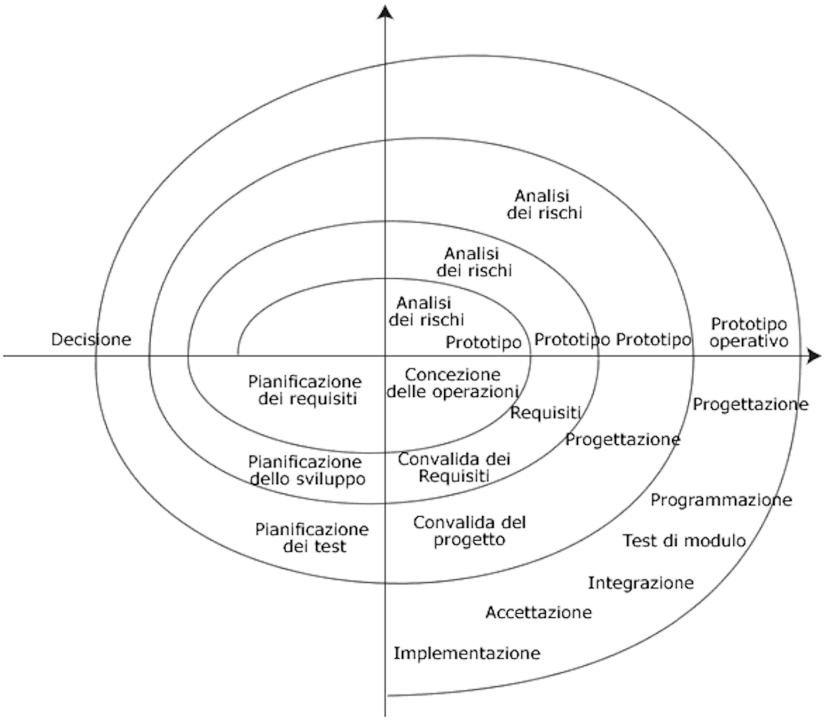

Analisi e Progettazione
Progettazione di dettaglio
(implemento una singola funzionalità)
Realizzazione versione
incompleta

1.2.3 Modello a Spirale

In questo caso ogni iterazione è formata da 4 fasi che corrispondono ai quadranti del piano:

1. *Quadrante in alto a sinistra*: definizione degli obiettivi e dei vincoli.
2. *Quadrante in alto a destra*: analisi e risoluzione dei rischi.
3. *Quadrante in basso a destra*: sviluppo e verifica del prossimo livello.
4. *Quadrante in basso a sinistra*: pianificazione della fase successiva.

Questo modello viene anche chiamato *risk driven* in quanto è incentrato principalmente sull'analisi dei rischi. Inoltre si ispira profondamente al metodo iterazivo *plan-do-check-act cycle* [2]



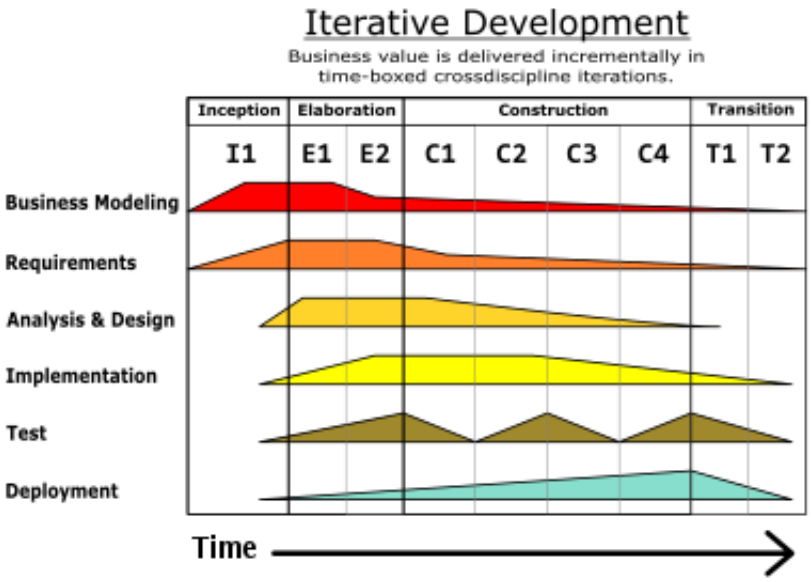
[2] https://it.wikipedia.org/wiki/Ciclo_di_Deming

1.3 Unified Process

In questo modello vengono distinte quattro fasi chiamate *Inception*, *Elaboration*, *Construction* e *Transition*. Ogni fase può presentare un numero variabile di iterazioni anche in base alla dimensione del progetto.

Questo modello viene definito iterativo incrementale, *incrementale* perchè alla fine di ogni iterazione si ottiene un rilascio del sistema con funzionalità in più o migliorate rispetto al rilascio precedente.

Inoltre viene data molta importanza all'architettura del sistema, infatti già dalle prime fasi ci si concentra soprattutto sull'architettura anche se a livello molto superficiale, lasciando i dettagli alle fasi successive. In questo modo è molto facile avere una visione generale del sistema che sarà facilmente modellabile sulla variazione dei requisiti. Per questo, piuttosto che dai requisiti, ci si fà guidare principalmente dai *casi d'uso* e dall'*analisi dei rischi*.



1.4 Processi Agili

Definizione (Metodo Agile)**.** Con metodo agile si intende un metodo per lo sviluppo del software che si basa principalmente sul coinvolgimento del committente. Questa metodologia si riferisce ai principi del *Manifesto di Snowbird* del 2001.

I concetti chiave di questi processi sono:

- Continuous Integration: rendere il più automatico possibile la consegna e l'integrazione dei singoli moduli.
- Continuous Delivery: rilascio frequente e supportato delle nuovi versioni del software.
- DevOps: *Development* e *Operations*, ovvero maggiore collaborazione tra sviluppatori e responsabili della manutenzione, della sicurezza e dell'infrastruttura dell'azienda.

1.4.1 Il Manifesto di Snowbird

Il *Manifesto di Snowbird* si fonda su quattro punti fondamentali:

1. Comunicazione: la comunicazione fra tutti gli attori del progetto è centrale, soprattutto le interazioni e la collaborazione con i clienti.
2. Semplicità: si mantiene il codice sorgente il più semplice possibile, ma comunque avanzato tecnicamente, in questo modo si riduce la documentazione al minimo indispensabile.
3. Feedback: sin dal primo giorno di sviluppo il codice viene testato, in modo da poter rilasciare versioni ad intervalli molto frequenti.
4. Coraggio: dare in uso il sistema il prima possibile ed implementare i cambiamenti richiesti man mano.

Di seguito sono riportati due modelli che si basano sui *processi agili*.

1.4.2 eXtreme Programming

Si basa su un insieme di consuetudini:

- *Pianificazione flessibile*: è basata su un insieme di scenari proposti dagli utenti e i programmatori vengono coinvolti direttamente.
- *Rilasci frequenti*: più o meno ogni 2-4 settimane, e alla fine si ricomincia con una nuova pianificazione.
- *Progetti semplici*: comprensibili a tutti.
- *Testing*: test basati sui singoli scenari e con supporto automatico.
- *Test Driven Development*: i casi di test vengono definiti prima della scrittura del codice.
- *Cliente sempre a disposizione*
- *Programmazione a coppie*: viene usato un solo terminale, una persona svolge il ruolo di *driver* che scrive il codice, mentre un'altra fà il *navigatore*, ovvero controlla il lavoro del *driver* attivamente.

- *No al lavoro straordinario*
- *Collettivizzazione del codice*: accesso libero e continua integrazione.
- *Code Refactoring*: modificare il codice senza cambiare il suo comportamento e commentarlo il più possibile.
- *Daily Stand Up Meeting*

1.4.3 SCRUM

Definizione (SCRUM)**.** Con *SCRUM* si intende un processo *iterativo* ed *incrementale*, dove alla fine di ogni iterazione vengono fornite un insieme di funzionalità potenzialmente rilasciabili.

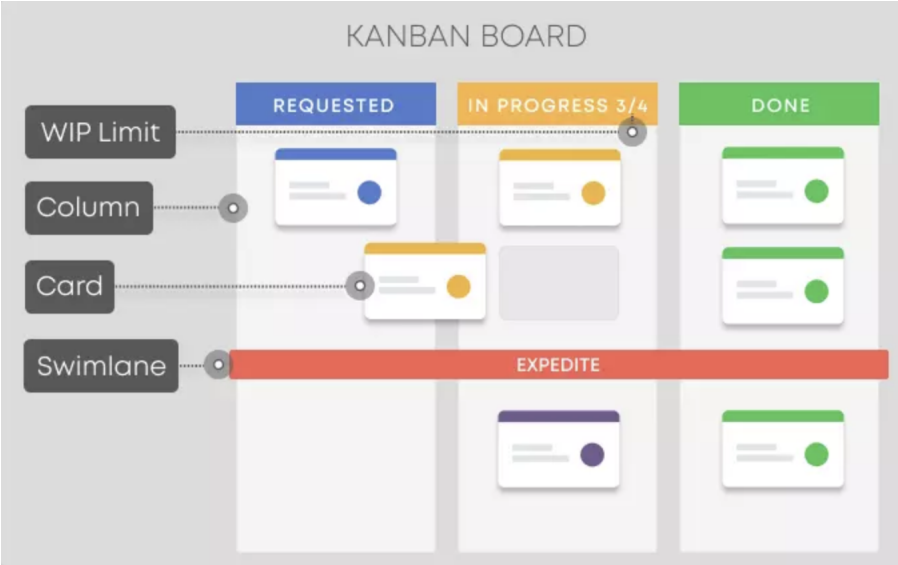
Il processo è diviso in tre fasi:

1. **Pre-game phase**:
   (a) Planning sub-phase: viene creata una *Product Backlog List* che contiene tutti i requisiti conosciuti.
   (b) Architecture sub-phase: viene già pianificato il design di alto livello e l'architettura del sistema.
2. **Development phase**: in questa fase il sistema viene sviluppato attraverso una serie di *Sprint*, ovvero cicli iterativi nei quali vengono sviluppate o migliorate una serie di funzionalità, e ogni sprint può durare circa 1-4 settimane. Lo *Sprint* ovviamente include le classiche fasi di sviluppo del software.
3. **Post-game phase**: il prodotto viene preparato per il rilascio, ovvero si prepara l'*integrazione*, i *test*, la *documentazione* per l'utente e la preparazione del materiale di *marketing*.

I ruoli principali durante l'esecuzione di un processo *SCRUM* sono tre:

- **Product Owner**: ci si riferisce a quella persona responsabile di accettare o rifiutare i risultati di un lavoro e di poter terminare uno *Sprint*, inoltre fà da raccordo fra tutti soggetti interessati nel progetto.
- **Membri del Team**: i membri decidono cosa fare in ogni *Sprint*, ogni team è indipendente e i membri non fanno capo ad alcun project manager. Ogni membro ha diverse specializzazioni (*cross-functional*), in modo tale da non avere persone con troppo carico di lavoro e ognuno si occupa di un singolo lavoro alla volta.
- **Scrum Master**: non ha alcuna autorità sul team, ma si occupa di supportarlo e motivarlo, garantendo anche le condizioni ambientali per lavorare al meglio.

Kanban Board Questa lavagna permette di gestire al meglio il flusso del lavoro. Come mostrato in figura è presente un *Work In Progress Limit* che definisce un limite alla quantità di post-it che possono essere presenti in ogni colonna. Questo limite permette di completare più velocemente i singoli lavori, in modo tale di dare qualcosa al cliente il prima possibile e di individuare facilmente i colli di bottiglia che possono rallentare gli altri lavori. Inoltre permette di ridurre il *task switching*, ovvero il lavoro su più task contemporaneamente.



Gli eventi che fanno parte di uno *Sprint* sono i seguenti:

1. Sprint planning: il *product owner* gestisce l'evento di pianificazione dello *Sprint*.
2. Daily meeting: i membri del team e gli SCRUM master si ritrovano davanti la *kanban* e discutono delle difficultà che hanno riscontrato.
3. Review: alla fine di di una modifica concreta al software, questo viene ispezionato in collaborazione con gli utenti per ottenere un feedback e per discutere su cambiamenti o nuove idee.
4. Retrospettiva: questa fase permette di riflettere, studiare e adattarsi per lo *Sprint* successivo.

2 Analisi dei Requisiti

Definizione (Analisi dei Requisiti)**.** Si intende il processo di studio e analisi delle esigenze del committente e dell'utente per giungere alla produzione di un documento che definisce il *dominio* del problema e i *requisiti* del software. In alcuni casi si definiscono anche i *casi di test* e il *manuale utente*.

Prima di passare alla fase vera e propria di *analisi dei requisiti* occorre seguire una fase preliminare per stabilire la realizzabilità del progetto software.

2.1 Studio di Fattibilità

Si basa principalmente sulla descrizione del software e delle necessità dell'utente. In seguito vengono svolte due analisi:

- Analisi di Mercato: si fà un confronto con il mercato attuale e si stimano i costi di produzione e quanto l'investimento può essere remunerativo.
- Analisi Tecnica: si studiano tutti gli strumenti per la realizzazione del progetto, come i software, le architetture, gli hardware e gli algoritmi. Inoltre si studia come deve essere fatta la prototipazione del software e la futura ricerca.

2.2 Dominio

Il **dominio** è il contesto in cui il software opera. Per definirlo occorre costruire un glossario, ovvero una collezione di definizioni di termini rilevanti in quel dominio specifico e che può essere riusato in progetti successivi nello stesso dominio. Inoltre occorre definire un modello statico, quindi come interagiscono fra loro gli elementi del dominio staticamente, e un modello dinamico, ovvero come si comporta il dominio in base all'avvenire di un determinato evento che può coinvolgere gli utenti. Questi due modelli possono essere descritti sia tramite l'uso del linguaggio UML [3], sia usando la semplice descrizione testuale.

2.3 Requisiti

Definizione (Requisito)**.** Il requisito è una proprietà che deve essere garantita dal sistema per soddisfare una qualsiasi necessità dell'utente.

I requisiti possono dividersi in due categorie:

[3] `https://it.wikipedia.org/wiki/Unified_Modeling_Language`

- **Requisiti funzionali**: quelli che descrivono le funzionalità e il comportamento del software.
- **Requisiti non funzionali**: descrivono le proprietà del software o del processo di sviluppo. Ad esemprio le caratteristiche di efficienza e affidabilità, l'interfaccia, il linguaggio di programmazione e l'ambiente di sviluppo scelti, i vincoli legislativi e i requisiti hardware o di rete.

I requisiti possono essere descritti mediante l'uso di diversi linguaggi, formali o meno. In questo caso si vede la descrizione dei requisiti mediante la produzione di un documento scritto in linguaggio naturale.

2.4 Documento dei Requisiti

Questo documento è un contratto tra lo sviluppatore e il committente, che elenca i requisti e i vincoli che il software deve soddisfare, e specifica anche una *deadline* per la consegna del progetto.

2.5 Fasi dell'Analisi dei Requisiti

L'*analisi dei requisiti* viene svolta in cinque passi:

1. **Acquisizione**
2. **Elaborazione**
3. **Convalida**
4. **Negozazione**
5. **Gestione**

2.5.1 Acquisizione

Il team di analisti incontra i membri dell'organizzazione del committente e si procede con la raccolta dei requisiti che può avvenire tramite: semplici interviste, questionari, costruzione di prototipi (anche su carta), studio di documenti o l'osservazione di possibili utenti mentre lavorano.

2.5.2 Elaborazione

Viene scritta la prima bozza del *documento dei requisti*, dove quest'ultimi vengono trattati in modo più approfondito. La struttura del documento deve essere la seguente:

Introduzione
Glossario
Definizione dei Requisiti Funzionali
Definizione dei Requisiti Non Funzionali
Architettura: la strutturazione del software in sottosistemi.
Specifica dettagliata dei Requisiti Funzionali
Modelli astratti: descrivere il sistema in base a ciascun punto di vista.
Evoluzione del sistema: successivi cambiamenti.
Appendici: descrizione della piattaforma hardware, database, manuale utente e i piani di test.
Indici: costruire un lemmario, quindi una lista di termini che puntano ai requisiti che li usano.

Nota Bene Nella descrizione dei requisiti occorre sempre usare la forma assertiva. Esempio:

Il <sistema> deve <funzionalità>/<proprietà>

2.5.3 Convalida

Nella fase di convalida occorre revisionare il documento per far sì che vengano evitati i seguenti difetti:

- Omissioni: requisiti mancanti.
- Inconsistenze: contraddizione tra i requisiti o tra un requisito e il contesto.
- Ambiguità: vaghezze o requisiti che possono avere più significati. Le ambiguità all'interno del linguaggio naturale possono essere portate da quantificatori, disgiunzioni oppure possono presentarsi ambiguità di coordinazione (nel caso si usano sia la *o* che la *e* nella stessa frase) oppure referenziale nell'uso non chiaro di pronomi. Inoltre occorre sempre evitare verbi deboli, forme passive, ovvero verbi senza un soggetto esplicito ed anche negazione e doppie negazioni.
- Sinonimi e Omonimi: termini diversi con lo stesso significato e termini uguali con significato diverso.
- Presenza di dettagli tecnici
- Ridondanza: può esserci, ma solo tra sezioni diverse del documento.

Le principali tecniche di convalida dei requisiti sono:

- Deskcheck
  - Walkthrough: ovvero il documento viene analizzato per intero sequenzialmente.
  - Ispezione: la lettura del documento è strutturata, utilizzando per esempio la tecnica del lemmario.
- Uso di strumenti di analisi del linguaggio naturale.
- Uso di prototipi.

Una volta trovati i difetti, è importante ricordarsi che vanno sempre risolti con il committente.

2.5.4 Negoziazione

In questa fase vengono assegnate delle priorità ai requisiti in base alle esigenze del committente e ai costi e tempi di produzione.

Questa fase è importante per decidere se alcuni requisiti possono essere cancellati oppure sviluppati successivamente.

MoSCoW Questa è una tecnica per dare priorità ai requisiti, i quali vengono divisi in quattro classi:

- Must have: requisiti irrinunciabili per il cliente.
- Should have: non necessari ma utili.
- Could have: non molto utili, da realizzare solo se c'è tempo.
- Want to have: da sviluppare in successive versioni.

2.5.5 Gestione

Questa fase si occupa di tre aspetti principali:

- Identificazione: ad ogni requisito viene assegnato un identificatore univoco.
- Attributi: ad ogni requisito vengono assegnati attributi relativi allo stato, priorità, sforzo in termini di giorni e/o personale da impiegare, rischio, e la versione destinazione per quanto riguarda lo sviluppo incrementale.
- Tracciabilità: ovvero la capacità di descrivere e seguire la vita di un requisito. Viene costruita una mappa tra i requisiti e le componenti del sistema, il codice ed i test.

Contratto Il *documento dei requisiti* precede la stipula del contratto.

2.6 Casi d'uso

I **casi d'uso** sono un altro modo per acquisire i requisiti, valutando le interazioni degli utenti col sistema e indicando al committente i risultati attesi. I casi d'uso oltre ad includere la sequenza corretta di eventi attesi devono anche presentare i comportamenti inattesi, ovvero le eccezioni.

2.7 User Stories

Sono un'altra tecnica di raccolta dei requisiti utilizzata principalmente nei *processi agile* e consiste nell'utilizzo di *user story cards* per scrivere i requisiti, che in linea generale hanno questa forma:

Nel mio ruolo di <ruolo utente>, *ho bisogno che il sistema* <funzione>, *al fine di* <beneficio>

Contro Il problema principale delle *user stories* è la scalabilità, ovvero sono difficili da trasporre su grandi progetti e problematiche se il team è distribuito geograficamente. Inoltre, essendo informali e brevi non sono adatte per raggiungere degli accordi legali e raramente includono i *requisiti non funzionali*.

3 Linguaggio UML e Casi d'Uso

Definizione (Modello)**.** Un *modello* è un'astrazione del dominio, usato per specificarne la natura e il comportamento.

I modelli possono classificarsi in:

- **Modelli Statici**: vengono rappresentate le *entità* e le *relazioni* fra esse per permettere di descrivere al meglio il dominio, le componenti architetturali e le classi da realizzare.
- **Modelli Dinamici**: vengono modellati i comportamenti delle entità descritte nel *modello statico*.

Un modello può essere:

- Una bozza o sketch, quindi un modello molto incompleto, usato principalmente per descrizioni iniziali.
- Un progetto dettagliato chiamato blueprint, che permette ai programmatori di realizzare direttamente il software senza prendere decisioni di progettazione.
- Un eseguibile, talmente preciso e completo da poter generare il codice in automatico partendo solo dal modello.

3.1 UML

Definizione (UML)**.** L'*Unified Modeling Language* è un linguaggio di modellazione unificato che ha il compito di supportare la descrizione e il progetto di software, nello specifico di applicazioni *object oriented*, ma permette anche di descrivere i modelli da più punti di vista in modo molto comprensibile sia dai clienti che dagli utenti.

3.2 Diagramma dei Casi d'Uso

Permette di descrivere i *requisiti funzionali* del sistema, catturando nello specifico le funzionalità viste dall'esterno (lato utente).

Definizione (Attore)**.** Un **attore** è un'entità esterna al sistema, che interagisce con esso. Gli attori possono essere classificati in:

- Un utente umano che possiede un determinato ruolo.
- Un altro sistema.
- Il tempo.

All'interno del diagramma gli attori sono delle classi e sono indicati con un nome in maiuscolo. Dato che sono classi è possibile fare delle generalizzazioni sugli *attori*, ovvero è possibile creare delle gerarchie.

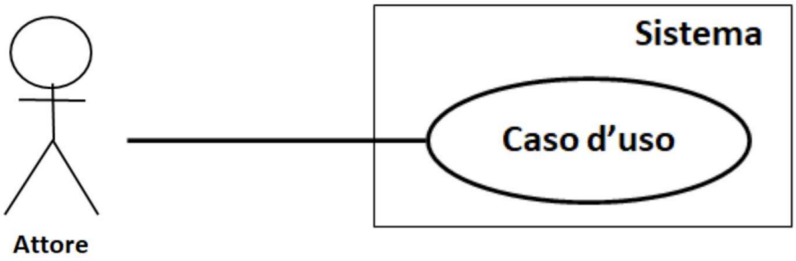
Definizione (Caso d'Uso)**.** Un **caso d'uso** è una funzionalità o un servizio offerto dal sistema a uno o più attori, e viene espresso tramite un insieme di *scenari*.

All'interno del diagramma, anche i casi d'uso sono scritti in maiuscolo, e per descriverli vengono usati dei *verbi* che ne indicano il compito.

Il *diagramma dei casi d'uso* oltre ad essere composto da *attori* e da *casi d'uso*, presenta anche:

- Relazioni: tra gli attori e i casi d'uso che rappresentano un'interazione.
- Il confine del sistema: un rettangolo disegnato intorno ai casi d'uso per indicare il confine del sistema.

È importante specificare che un caso d'uso è sempre iniziato da un solo attore, chiamato attore principale. Inoltre, possono essere presenti casi d'uso non collegati ad alcun attore.



3.3 Narrativa dei Casi d'Uso

Per poter descrivere il *modello dinamico*, viene redatto un documento che permette di rappresentare gli scenari di ogni caso d'uso dal punto di vista di ogni attore coinvolto.

La descrizione di un caso d'uso segue questa struttura:

Nome
Breve descrizione
Attore primario
Attori secondari
Precondizioni
Sequenza degli eventi principale
Postcondizioni
Sequenze alternative degli eventi

Definizione (Precondizioni e postcondizioni)**.** Le **precondizioni** e le **post-condizioni** sono dei predicati che devono sempre essere veri in uno stato: per le *precondizioni* prima di iniziare il caso d'uso, per le *post-condizioni* alla fine. La relazione tra *precondizioni*, *postcondizioni* e sequenza principale ed alternativa ha a che fare con la *logica di Hoare* [4], infatti è possibile costruire la seguente *tripla di Hoare*:

$$\{Precondizione\}\ Sequenza\ Principale\ \{Postcondizione\}$$

Ciò significa che per ogni stato σ che soddisfa la *precondizione*, se l'esecuzione della *sequenza principale* nello stato σ termina producendo uno stato σ', allora la *postcondizione* nello stato σ' deve essere vera.

Questo però significa che se l'esecuzione della *sequenza principale* non termina o termina in modo inaspettato come indicato nella *sequenza alternativa*, allora la *postcondizione* non è garantita.

Definizione (Scenario)**.** Uno **scenario** è un'istanza di un caso d'uso, ovvero una sequenza di interazioni tra il sistema e gli attori che produce un risultato osservabile.

Gli scenari descritti dalla *sequenza degli eventi principale* sono quelli che portano alle *postcondizioni*.

La *sequenza degli eventi principale* elenca i passi che compongono il caso d'uso ed ogni passo presenta la seguente sintassi:

```
<numero>. <soggetto><azione><complementi>
```

Il primo passo, inoltre, è sempre compiuto dall'*attore principale*. All'interno della sequenza possono anche presenti *condizioni* e *cicli*, scritti in pseudocodice.

3.3.1 Inclusione

L'*inclusione* permette di creare una relazione di dipendenza tra casi d'uso.

Il caso d'uso *incluso* può essere *istanziabile* (o *completo*), quando è avviato da un attore, oppure *non istanziabile*, quando viene eseguito solo quando è incluso.

[4] `https://it.wikipedia.org/wiki/Logica_di_Hoare`

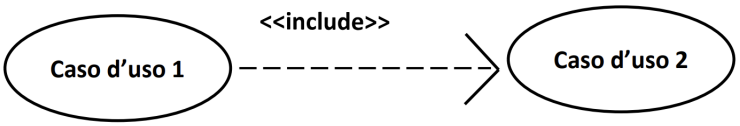


Figure 2: Il *Caso d'uso 1* include il *Caso d'uso 2*.

Nota Bene È importante non usare la relazione di inclusione per fare decomposizione di un caso d'uso.

3.3.2 Estensione

L'*estensione*, a differenza dell'*inclusione*, non è una dipendenza, ma permette a un caso d'uso di incorporarne opzionalmente un altro.

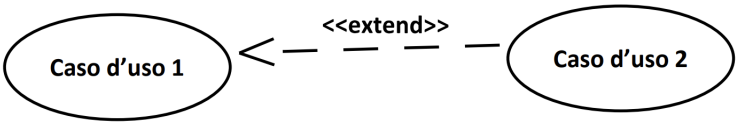
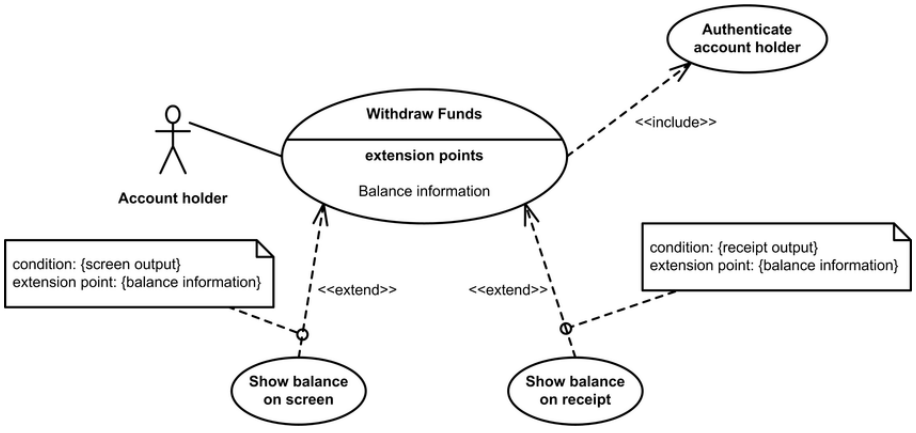


Figure 3: Il *Caso d'uso 1* può essere esteso dal *Caso d'uso 2*.

Extension Points Gli *extension points* sono una notazione che permette di identificare quando e dove inserire l'estensione. Si collega un vincolo alla freccia `<<extend>>` indicando la condizione che deve essere vera affichè l'estensione venga applicata.



3.4 Classi e Oggetti

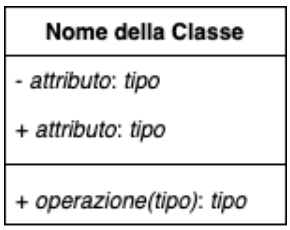
Definizione (Oggetto)**.** Un **oggetto** è un entità caratterizzata da un'*identità*, uno *stato*, ovvero i valori degli *attributi* dell'oggetto, e da un *comportamento*, quindi le operazioni che lo definiscono.

Definizione (Classe)**.** Una **classe** è un insieme di *oggetti* con caratteristiche simili, quindi che hanno lo stesso tipo.

Una classe permette di catturare gli elementi del dominio del sistema.

3.4.1 Diagramma delle Classi

Descrive le proprietà e le operazioni di un classe, il tipo degli oggetti, e le relazioni *statiche* fra essi.



Una classe viene rappresentata in questo modo, con il nome sempre in maiuscolo, la sezione che riguarda gli attributi e quella che riguarda le operazioni. Il simbolo $+$ indica un attributo/operazione privata, mentre il $-$ ne indica uno pubblico. Esistono anche altri due simboli per la *visibilità*: il $\#$, che indica che l'attributo o l'operazione è accessibile anche alle classi discendenti nella gerarchia, mentre la $\sim$ che indica l'accessibilità solo alle classi nello stesso *package*. Inoltre, per specificare che un attributo o un'operazione sono *statici*, si sottolineano.

Nonostante ciò, quando si usa il diagramma delle classi per descrivere il dominio, sia le operazioni che la visibilità e i dettagli implementativi degli attributi si omettono.

Attributi La sintassi degli attributi è la seguente:

```
Visibilità Nome: Tipo[Molteplicità] = ValoreDefault {Proprietà}
```

La *molteplicità* indica gli array di valori, quando è uguale a 1 può essere omessa. Invece, le *proprietà* possono essere sui valori dell'attributo, oppure nel caso la molteplicità è maggiore di 1 si può usare `{ordered}` per indicare che nell'array l'ordine degli elementi conta, e `{unique}` per dire che gli elementi dell'array non devono avere ripetizioni.

Operazioni La sintassi delle operazioni è la seguente:

```
Visibilità Nome: (ListaParametri) : TipoRitorno
```

Con `ListaParametri` definita dalla seguente grammatica:

$$ListaParametri ::= \varnothing \mid DichiarazioneParametro,\ ListaParametri$$

$$DichiarazioneParametro ::= Nome : Tipo = ValoreDefault$$

Enumerazioni Sono usate per specificare una lista prefissata di valori che un *attributo* può assumere. In UML sono etichettate con lo stereotipo `<<enumeration>>`.

Figure 4: Esempio di enumerazione con l'attributo *colore*.

3.4.2 Relazioni

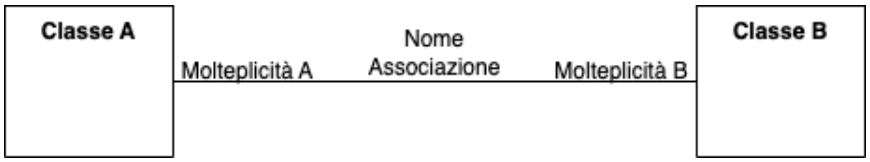
Definizione (Relazione)**.** Una **relazione** rappresenta un legame tra due o più oggetti.

In UML viene rappresentata da una linea retta con sopra scritte il nome dell'associazione, il ruolo di ogni classe, ed eventualmente una freccia che sta ad indicare il verso di lettura dell'associazione.

In generale si usa specificare o solo il *nome dell'associazione* o solo i *ruoli*.

Molteplicità La *molteplicità* indica il numero di oggetti coinvolti in un'associazione in un dato istante. Può essere definita in uno dei seguenti modi:

- Con un numero positivo.
- Con $*$ che indica un valore indefinito.
- Indicando gli estremi inferiore e superiore di un intervallo (es. 2..4, 0..5, 7..*).



Associazioni Riflessive Si riferiscono alla stessa classe e, in questo caso, è fondamentale il ruolo dei due oggetti.

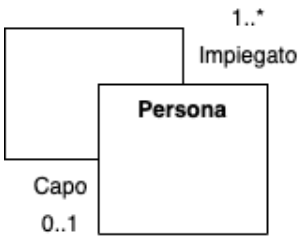


Figure 5: Esempio di associazione riflessiva.

3.4.3 Aggregazione e Composizione

L'**aggregazione** e la **composizione** sono tipi di relazioni in cui viene specificato che un oggetto di una classe *fà parte* di un oggetto di un'altra classe. L'*aggregazione* è meno forte come relazione, quindi si verifica quando le *classi parte* hanno un significato anche senza la presenza della *classe tutto*. Nella *composizione*, invece, la *classe parte* ha un senso solo se legata alla *classe tutto*.

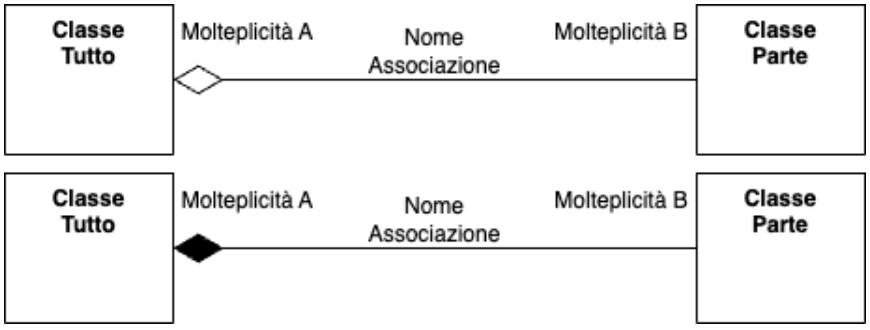


Figure 6: Sopra *aggregazione*, sotto *composizione*.

3.4.4 Generalizzazione

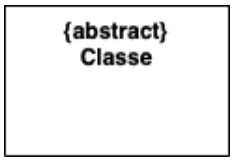
Consiste in una relazione tra una classe più generica e una più specializzata: la classe specializzata o *sottoclasse* eredita tutte le caratteristiche della *superclasse*, può aggiungerne delle altre, e può ridefinire delle *operazioni*.

In questa relazione vale il *Principio di Sostituzione di Liskov* [5], ovvero un oggetto della classe specializzata può essere usato al posto di un oggetto della superclasse.

[5] `https://it.wikipedia.org/wiki/Principio_di_sostituzione_di_Liskov`

3.4.5 Classi Astratte

Le **classi astratte** definiscono delle classi che non sono implementate completamente.



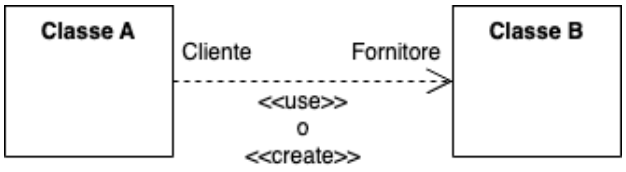
3.4.6 Interfacce

Le **interfacce** si usano in fase di *progettazione* per definire delle classi con solo operazioni e senza attributi.

Nota Bene Sia le *interfacce* che le *classi astratte* non possono essere istanziate, ma si utilizzano nelle *gerarchie* per definire la "*struttura*" di classi più complete.

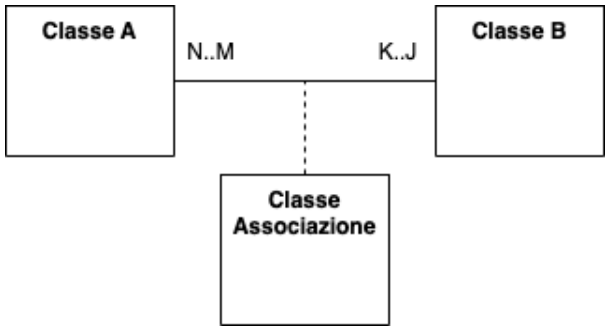
3.4.7 Dipendenze

La dipendenza è una relazione in cui le classi hanno un ruolo di *cliente* e di *fornitore*. Questo avviene quando un parametro di un'operazione della classe *cliente* è un oggetto della classe *fornitore*, o se un'operazione del *cliente* restituisce o crea dinamicamente un oggetto del tipo del *fornitore*.



3.4.8 Classi Associazione

Un'associazione può avere dei propri attributi, rappresentati con una **classe associazione**. Per ogni coppia di classi, può esistere solo una *classe associazione*.



3.4.9 Classi di Analisi

Le **classi di analisi** corrispondono ai concetti concreti del dominio, per esempio i termini del glossario. Queste classi hanno un numero ridotto di *funzionalità* e durante la loro definizione occorre evitare di:

- Definire classi "*onnipotenti*".
- Definire classi che in realtà sono delle *funzioni*.
- Definire delle *gerarchie* troppo profonde (più di 3 livelli).
- Specificare troppo i tipi e i valori degli *attributi*.

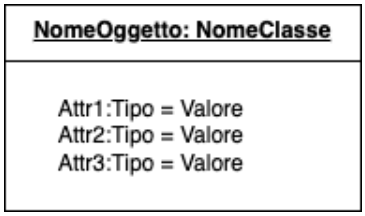
Inoltre le *operazioni* e gli *attributi* vanno indicati solo quando sono veramente utili.

Le principali tecniche di definizione delle *classi di analisi* sono:

- Data Driven: durante la *fase di analisi*, si identificano tutti i dati del sistema e si dividono in classi.
- Responsibility Driven: durante la *fase di progettazione* si identificano le operazioni e si dividono in classi.

L'analisi nome-verbo consiste nell'associare ai *sostantivi* le classi e gli attributi, mentre ai *verbi* le operazioni. Successivamente si individuano le relazioni fra le classi. Utilizzando questo tipo di approccio occorre prestare attenzione ai casi di sinonimia per evitare di definire classi inutili; e bisogna saper individuare le classi nascoste del dominio, cioè quelle che non vengono mai menzionate esplicitamente.

3.4.10 Diagramma degli Oggetti



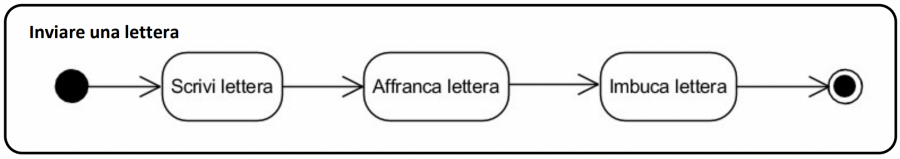
In questo caso, un collegamento tra *oggetti* è un'**istanza d'associazione**, non ha un nome ma si possono indicare i ruoli. Inoltre non esiste la molteplicità, in quanto è sempre 1 a 1.

3.5 Diagramma delle Attività

Sono utili per descrivere delle attività relative a un qualsiasi *oggetto* o *classe*, ovvero un insieme di azioni che possono essere *sequenziali*, *condizionali*, *concorrenti* e *iterative*.

Si usano sia nella fase di *analisi* per modellare un processo o un caso d'uso, ma anche per descrivere l'*operazione* di una classe o per modellare un algoritmo in fase di *testing*.

Il contenuto di un'attività è un *grafo diretto* in cui i nodi sono le *azioni* o i *nodi di controllo*, mentre gli archi sono i possibili percorsi percorribili per l'attività.



Azioni È importante sottolineare che le azioni devono essere **atomiche**, cioè non interrompibili. Inoltre per ogni azione ci può essere solo una freccia entrante e una uscente. Quando un'azione è giunta al termina avviene una transazione automatica chiamata **token** che porta all'azione successiva.

Nodi di Decisione Un nodo di decisione deve poter coprire tutti i casi possibili, in modo che il *token* non possa bloccarsi. Opzionalmente le *guardie* possono essere *mutualmente esclusive*. Inoltre, dato un *nodo decisione* non è obbligatoria la presenza di un *nodo fusione*.

Le *guardie* si scrivono sempre tra parentesi `[]`.

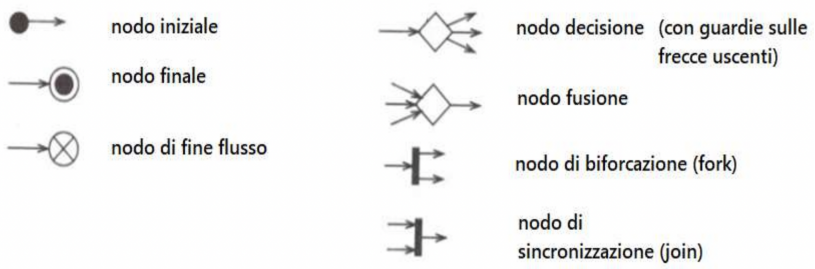


Figure 7: Nodi di Controllo

Fork & Join La *fork* moltiplica i token, producendone uno per ogni uscita. La *join* invece attende un token per ogni freccia entrante e, quando li consuma tutti, ne esce solo uno. Come per i nodo di decisione non è necessaria una *join* per ogni *fork*.

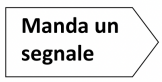
Nodo di Fine Attività Quando un qualsiasi token raggiunge questo nodo tutta l'attività termina. Solo su questi nodi, e su quelli di *fine flusso*, possono essere presenti più archi entranti; ciò sta a significare che il primo token che arriva termina l'attività.

Nodo di Fine Flusso Questo tipo di nodo non termina l'attività, ma consuma il token.

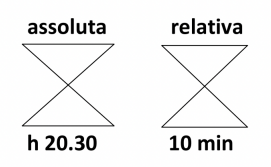
3.5.1 Segnali & Eventi

- Accettazione di un evento esterno.

- Invio di un segnale. L'invio di segnali è asincrono, cioè non blocca l'attività.



- Accettazione di un evento temporale.

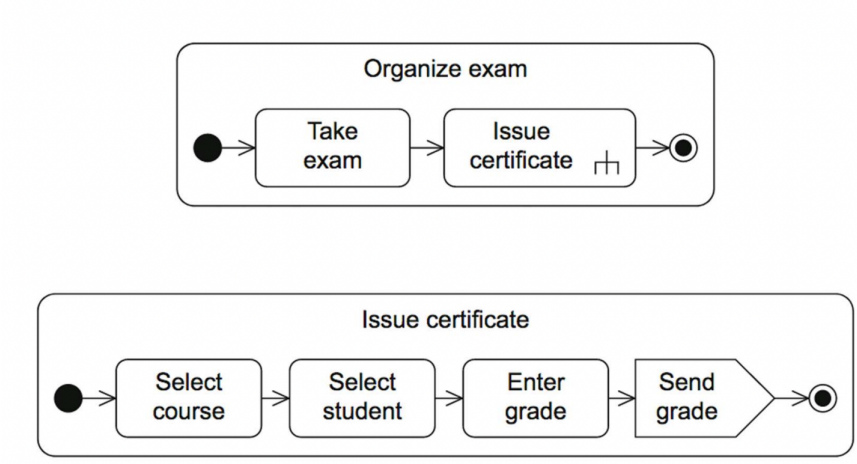


Per gli eventi esterni e temporali, gli archi entranti non sono necessari, quindi in questo caso quando si verifica quel determinato evento viene generato un token. Se invece gli archi entranti sono presenti, quando arriva il token questo attende che l'evento esterno si verifichi.

A differenza delle azioni che si usano quando le attività sono effettuate dalle entità di cui si sta descrivendo il comportamento, mentre i segnali e gli eventi si usano quando si comunica con un'entità esterna.

3.5.2 Sotto-Attività

Un diagramma può contenere il riferimento a delle attività secondarie, rappresentate come un'azione ma con in più il simbolo di un "rastrello".



3.5.3 Partizioni

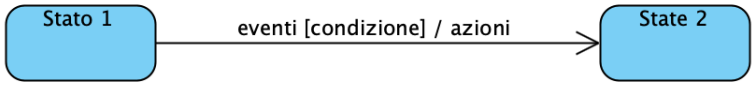
Le **partizioni** permettono di dividere le azioni, assegnandole all'entità che ne è responsabile.

3.6 Macchina a Stati

È un grafo *"stati-transizioni"* che permette di descrivere il comportamento delle istanze di una classe. A differenza dei *diagrammi delle attività* dove l'obbiettivo è mettere in ordine un insieme di azioni, qui viene mostrata l'evoluzioni degli oggetti in risposta a determinati eventi.

Ogni transizione avviene al verificarsi di un determinato evento interno, rappresentato da un'*operazione* della classe; oppure tramite messaggi inviati da altri oggetti.

In una *transizione*, gli *eventi* possono essere accompagnati anche da una *condizione* e da una sequenza di azioni.



La semantica è la seguente: se si verifica almeno uno degli eventi, e vale la condizione, si eseguono tutte le azioni e si passa nell'altro stato.

3.6.1 Eventi

Le *transizioni* possono essere *non deterministiche*, cioè da uno stato ci possono essere più transizioni con lo stesso evento.

Gli eventi possono classificarsi in:

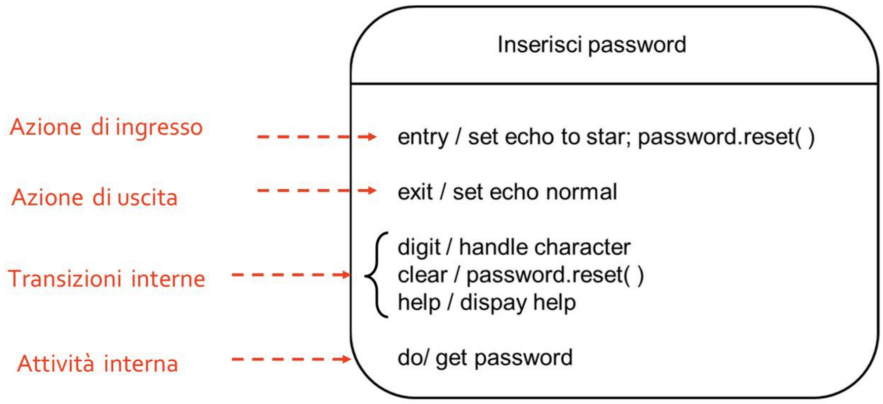
- Eventi di operazione o segnale: `op(a: T)`. Ovvero la transizione è abilitata quando riceve un segnale o avviene una chiamata di metodo con parametri `a` e tipo `T`.
- Eventi di variazione: `when(exp)`. La transizione è abilitata quando `exp` diventa vera.
- Eventi temporali: `after(time)`. La transizione è abilitata dopo che l'oggetto è stato fermo per un tempo `time` in quello stato.

Entry Anche chiamata *azione di entrata*, viene eseguita all'ingresso di uno stato.

Exit Anche chiamata *azione di uscita*, viene eseguita quando si esce da uno stato.

Transizione Interna Risposta a un determinato evento, ma si rimane nello stesso stato.

Do-Activity Azione eseguita in continuazione finchè l'oggetto si trova in quello stato. A differenza delle altre *azione* che sono *atomiche*, queste consumano del tempo e possono essere interrotte, ad esempio quando si esce dallo stato.



3.6.2 Stato Composito Sequenziale

Consiste nell'avere uno stato che contiene un'altra *macchina a stati*. Quando una transizione entrante nello *stato composito* termina sul bordo, vuol dire che si prosegue a partire dallo *stato iniziale* dello *stato composito*. Altrimenti la transizione può avere come *target* anche uno stato interno. Le transizioni di uscita, invece, possono anch'esse partire dal bordo, in tal caso significa che ci si può andare da qualsiasi stato interno, altrimenti quelle che partono da un singolo stato interno sono possibili solo da esso. Esiste anche una transizione d'uscita speciale chiamata di completamento, dalla quale ci si può passare solo una volta arrivati nello stato finale dello stato composito.

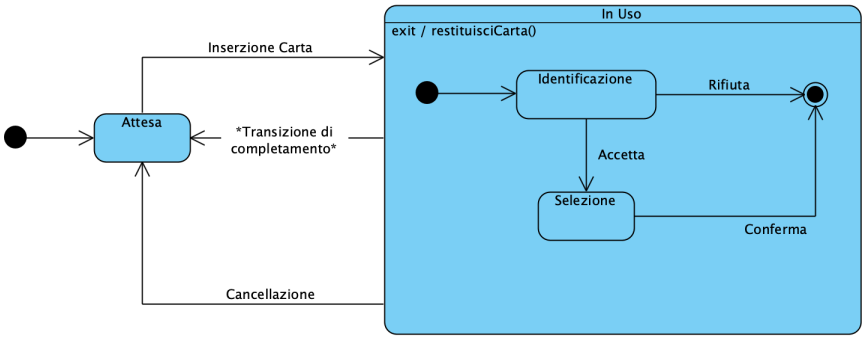
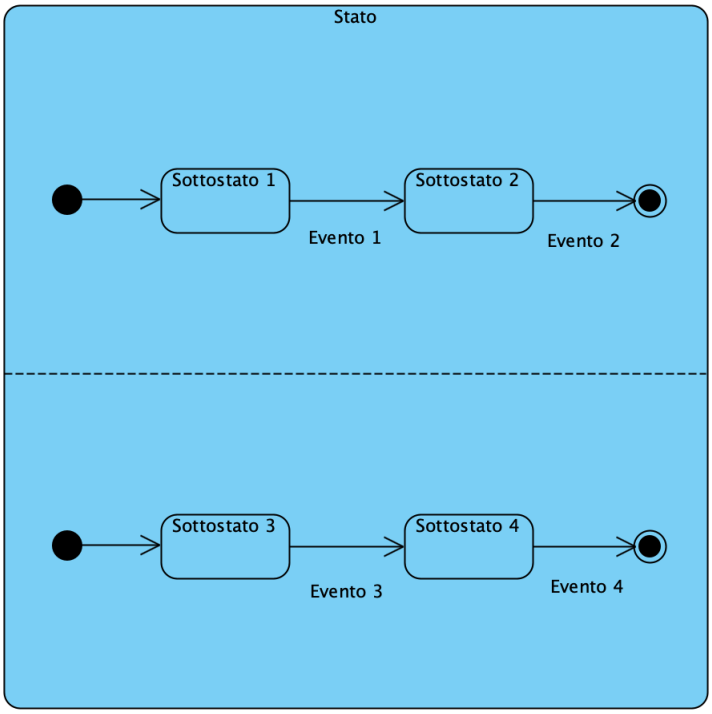


Figure 8: Esempio di macchina a stati che descrive l'uso di un sistema di pagamento con stato composito sequenziale.

3.6.3 Stato Composito Parallelo

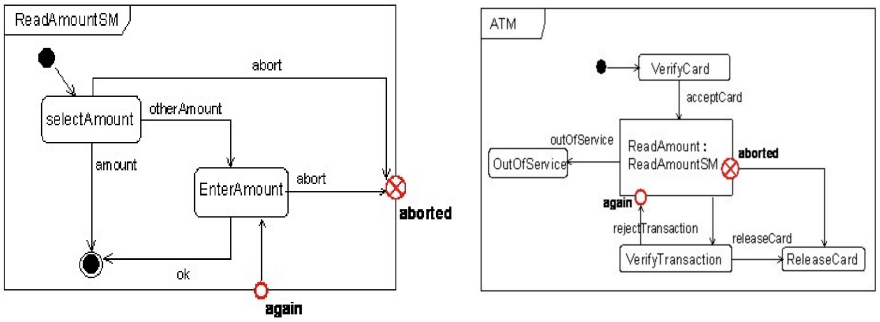
Nello *stato composito* abbiamo più sottostati che si eseguono in modo parallelo. Una transizione entrante sul bordo attiva tutti gli stati iniziali. La *transizione di completamento* può avvenire solo se si è arrivati in tutti gli stati finali. Mentre, una transizione di uscita che parte dal bordo è raggiungibile da un qualunque stato interno e fà uscire da **tutti** i sottostati. Anche un transizione che parte da un singolo stato interno fà uscire da tutti i sottostati e può avvenire solo se ci si trova in quel determinato stato.



3.6.4 Sottomacchine

Le sottomacchine si usano quando si vuole rappresentare uno stato composito in un diagramma a parte, in modo da poterlo riusare in più contesti. La *sottomacchina* ha un nome che ne definisce il `Tipo` e le sue istanze si indicano con `Nome Istanza: Tipo`.

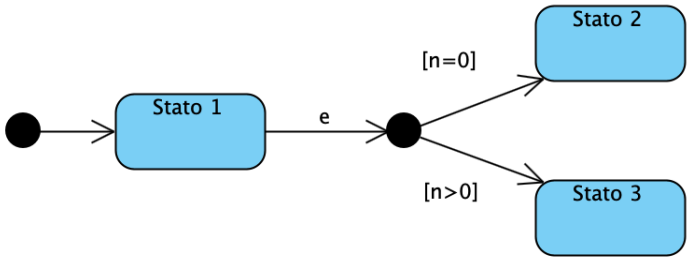
Entry & Exit Points Sono dei nodi che servono per poter collegare le transizioni della macchina principale con la sottomacchina.



In questo caso le *transizioni di completamento* possono scattare anche quando si arriva in un *exit point*.

3.6.5 Pseudostati

Giunzione Uno *pseudostato* da cui possono entrare e/o uscire due o più transizioni. Se sono presenti condizioni, queste sono valutate in modo statico, quindi prima che avvengano gli eventi interessati. Inoltre, se le condizioni non coprono tutti i casi, l'evento può essere ignorato e si rimane in quello stato.



Decisione A differenza delle *giunzioni*, questo sono valutate dinamicamente, quindi dopo che avvengono gli eventi; e devono coprire tutti i casi possibili. Ovviamente un'altra differenza con le *giunzioni* è che in questo caso la transizione entrante può essere solo una.

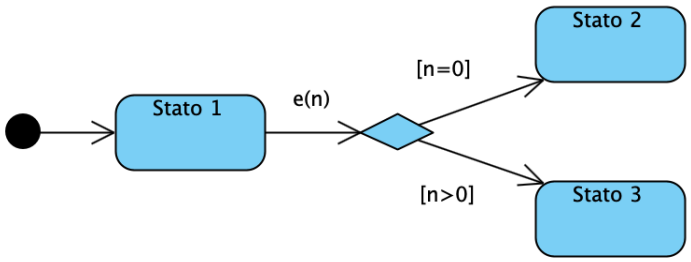


Figure 9: In questo esempio bisogna avere garanzia che $n \geq 0$.

Storia Lo stato *history* permette di memorizzare lo stato della macchina quando viene *interrotta* o *spenta*. La transizione entrante nello stato *history*, invoca il ripristino dello stato precedente, mentre se c'è una transizione uscente, questa indica lo stato in cui passare nel caso in cui la macchina non sia ancora stata mai interrotta, quindi quando lo stato *history* è vuoto.

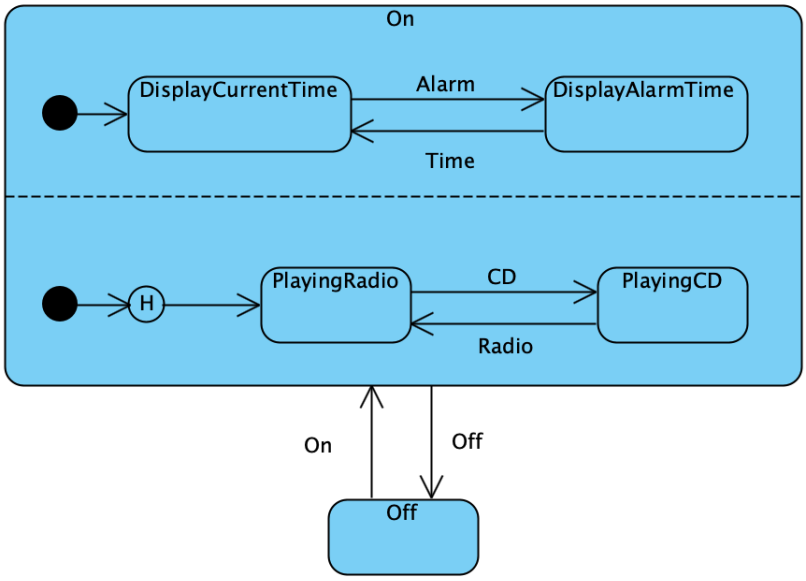


Figure 10: In questo esempio di funzionamento di un autoradio, la prima volta che si accenderà verrà riprodotta in automatico una stazione radio.

4 Architetture Software

La *progettazione* è la fase che segue la *specifica* e precede la *codifica*. Descrive in che modo deve essere realizzato il progetto. Il suo prodotto si chiama *architettura*.

Esistono due tipi di *progettazione*:

- Progettazione di alto livello: lo scopo è identificare e scomporre il sistema in tanti piccoli sottosistemi e definire le loro interconnessioni.
- Progettazione di dettaglio: si decide come ogni singolo *sottosistema* deve essere realizzato.

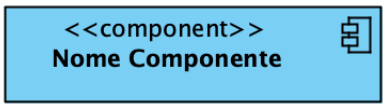
Quindi l'architettura di un sistema software è la struttura del sistema costituita dalle sue parti e dalle relazioni fra esse, più le loro proprietà visibili all'esterno. Vengono considerati anche gli aspetti *non funzionali*.

Durante questa fase il sistema viene analizzato attraverso tre punti di vista:

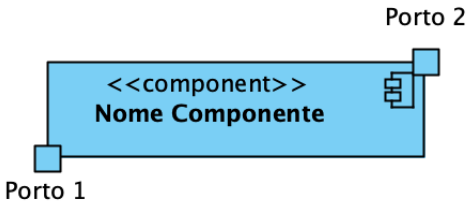
- Vista **Comportamentale** o *component-and-connector*: questa vista descrive il sistema trattandolo come composizione di altri software. Quindi si specificano tutte le componenti (i software), che presentano delle *interfacce*, successivamente si descrivono le caratteristiche dei *connettori* che collegano le varie componenti. Infine si analizza la struttura del sistema in esecuzione, infatti questa vista è molto utile per analizzare la qualità e le prestazioni del software e per documentare lo stile dell'architettura.
- Vista **Strutturale**: permette di descrivere la struttura del sistema vedendolo come un insieme di *classi* e/o *packages*. È utile per analizzare le dipendenza tra le *classi*/*packages* e per progettare i test di *unità* e di *integrazione*.
- Vista **Logica** o di *deployment*: descrive l'allocazione del software su più ambienti di esecuzione.

4.1 Vista Comportamentale

Definizione (Componente)**.** Una **componente software** è un'unità software indipendente e che può essere riusata da altri componenti. Incapsula un insieme di funzionalità e/o di dati di un sistema, restringendo l'accesso ad essi tramite la definizione di *iterfacce*.



Definizione (Porti)**.** I **porti** identificano i punti di interazione di un componente, ogni componente ne ha uno per ogni connessione che ha con altri componenti. Un *porto* può richiedere una o più interfacce dello stesso tipo.



Le *interfacce* possono essere rappresentate in modo esteso o in modo sintetico mediante l'uso di *forchette* (quando viene richiesta un'interfaccia) o *lollipop* (quando l'interfaccia è fornita).

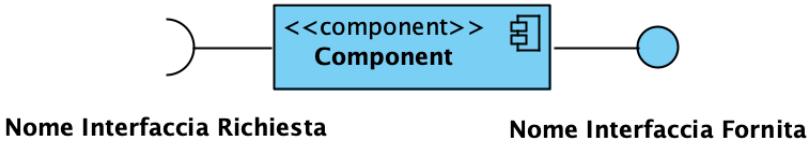


Figure 11: Modo Sintetico

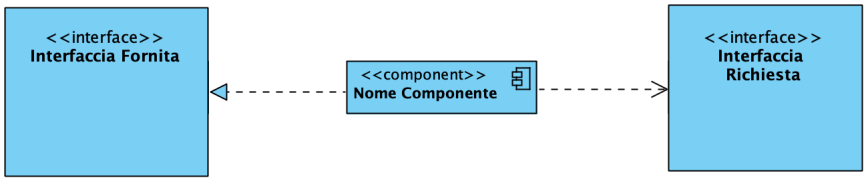


Figure 12: Modo Esteso

Definizione (Connettori)**.** I **connettori** sono i canali di comunicazione che collegano i *porti* fra componenti diverse. Sul connettore è possibile indicare il *protocollo di comunicazione* utilizzato fra <<...>>.

4.1.1 Stili Architetturali

Definizione (Stile)**.** Uno **stile** è una proprietà dell'architettura che ne caratterizza una famiglia con caratteristiche e interazioni fra componenti comuni.

Pipe & Filter Le componenti sono di tipo *filtro*, ovvero effettuano delle trasformazioni sui dati che provengono dai porti d'ingresso e li mandano sui porti d'uscita. I connettori, invece, sono di tipo *pipe*, ovvero un canale di comunicazione unidirezionale e che preserva l'ordine dei dati che entrano e che escono.

Client-Server Il sistema è formato da componenti che si comportano da *client* e altri da *server*. Il client si connette a un server tramite un porto per richiedere un servizio, il server può ricevere connessioni da parte di più client.

Master-Slave È un caso particolare del modello *client-server*, in cui lo *slave* (il server) può servire un solo *master* (client).

Peer-to-Peer Tutti i componenti agiscono sia da client che da server, quindi nella rappresentazione ci sarà un solo componente.

Publish-Subscrive I componenti interagiscono tra loro tramite l'annuncio di eventi, una componente può svolgere sia il ruolo di *Publisher* (produttore di eventi) che di *Subscriber* (consumatore di eventi). Questo stile può coinvolgere anche un intermediario chiamato *broker*, che permette alle altre componenti di non interagire direttamente fra loro e quindi di non essere consapevole delle indentià delle altre.

Model-View-Controller Permette di isolare il *modello*, ovvero le funzionalità e i dati del sistema, la *vista*, quindi come viene rappresentato il modello, e il *controllore*, che riceve l'input ed effettua le chiamate alle operazioni del modello.

Coordinatore di Processi Prevede l'uso di un componente noto come *coordinatore*, che conosce la sequenza per realizzare il processo. Una volta ricevuta la richiesta, chiama le componenti *server* nell'ordine prefissato e successivamente fornisce una risposta. I server non sono a conoscenza del loro ruolo, ma si limitano solo a fornire un servizio.

4.2 Vista Strutturale

Gli elementi di questa vista sono chiamati *moduli*, ed essendo classi o packages, le relazioni fra essi sono di ereditarietà, dipendenza o composizione. Un *package* può contenere un insieme di classi e/o di altri packages.

Questo tipo di vista è utile per fornire uno schema del codice e dei file sorgenti, non và utilizzato per le analisi dinamiche, effettuate con viste *comportamentali* e *logistiche*.

Nelle classi, rispetto alla descrizione del *dominio*, qui si specificano meglio le loro operazioni.

4.2.1 Vista Strutturale a Strati

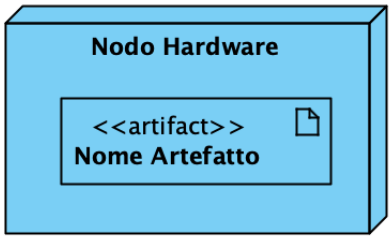
Gli elementi sono gli *strati*, ovvero un insieme di moduli, che possono essere raggruppati in *segmenti*, inoltre implementano un'interfaccia pubblica, chiamata *macchina virtuale* per i servizi che offrono. La relazione fra gli strati è di tipo `<<allowedToUse>>`, è antisimmetrica e non è transitiva.

4.2.2 Vista Strutturale di Generalizzazione

Gli elementi sono sempre i moduli, ma la relazione è solo di *generalizzazione*.

4.3 Vista Logistica

In questa vista gli elementi possono essere *hardware* (rappresentati da parallelepipedi) come l'ambiente di esecuzione, oppure *software* (con stereotipo `<<artifact>>`), chiamati *artefatti*, cioè qualsiasi file che viene utilizzato o prodotto da un software da un sistema generale. Si usa per valutare le prestazioni e per redigere una guida per l'installazione del software. Le relazioni tra i gli elementi hardware possono essere connessioni fisiche o protocolli di comunicazione.

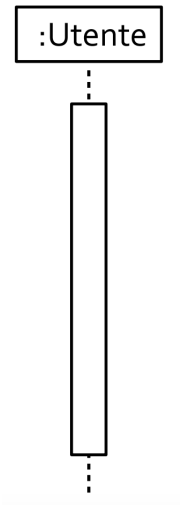


Artefatto L'artefatto è una copia di un'implementazione di un componente che è stata installata su un ambiente di esecuzione. L'artefatto si dice che "*manifesta*" (`<<manifesta>>`) un componente.

5 Diagrammi di Sequenza

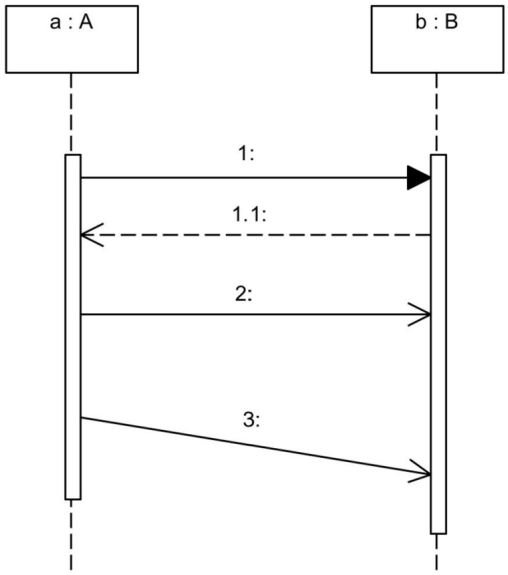
Si usano per descrivere lo scambio di messaggi e dati tra oggetti, indicando anche la sequenza temporale. Si può usare sia in *fase di analisi* per formalizzare la *sequenza principale degli eventi* nei casi d'uso, oppure in *fase di progettazione* per mostrare i messaggi scambiati dalle componenti (e anche attori) nell'architettura.

Gli oggetti e attori sono rappresentati da un rettangolo che indica il ruolo e/o il tipo (nel caso dell'oggetto). Dal rettangolo parte una linea verticale chiamata *linea di vita* dell'oggetto: è tratteggiata quando l'entità è inattiva, invece è continua e doppia quando è attiva (nel caso di un attore la sua linea di vita è sempre attiva).



Messaggi I messaggi rappresentano l'invocazione di *operazioni* o *segnali*, possono essere:

- Sincroni (1)
- Di ritorno (1.1)
- Asincroni (2)
- Asincroni con esplicito uso di tempo (3)



Inoltre i messaggi presentano la seguente sintassi:

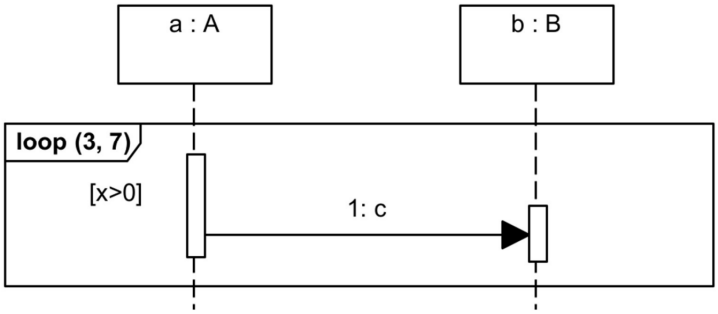
```
id: attributo =
nome_messaggio(arg1, arg2, ..., argN): valore_ritorno
```

Le entità possono essere aggiunte e rimosse anche dinamicamente all'interazione. Per la creazione si indica sulla freccia del messaggio la keyword `<<create>>`. Per la cancellazione, invece, si pone una $\times$ alla fine della sua linea di vita.

Nella linea di vita è possibile indicare anche vincoli di tempo o durata per indicare un limite inferiore e/o superiore alla durata di un intervallo.

Frame Condizionale È possibile dividere il *frame* del diagramma in più *sottoframe*, ognuno dei quali può avere una guardia (se non ne ha si assume che sia uguale a `[true]`). In base alla guardia che è vera (se ce n'è più di una la scelta è non deterministica) si eseguono le azioni corrispondenti a quel sottoframe.

Frame Iterativo Un frame può essere ripetuto più volte, si utilizza la sintassi `loop(min, max)`.



Esempi.

- `loop(0, *)[guardia]`: modella il `while(guardia){...}`.
- `loop(1, *)[guardia]`: modella il `do{...}while(guardia)`.
- `loop(n, n)` o `loop(n)`: modellano un ciclo `for` che và da 0 a n escluso.

Frame Condizionale Si specifica con la keyword `opt` ed è associato ad una guardia che, nel caso è vera fà eseguire quel determinato frame.

Frame Parallelo Le interazioni nei *sottoframe* sono eseguiti in parallelo (*interleaving*).

Ref È possibile includere in un frame un'interazione già definita tramite l'indicatore `ref` e scrivendo il nome del diagramma di sequenza da includere.

Gates Un *gate* è un punto sul bordo del diagramma a cui è collegato un messaggio, o in ingresso o in uscita. Ogni gate ha un nome, e si utilizzano principalmente quando si includono altri diagrammi.

6 Progettazione Software

Questa fase risulta molto importante, in quanto si mira a realizzare un software, che oltre ad essere perfettamente funzionante, deve essere facilmente mantenibile e riusabile in altri sistemi.

Durante la progettazione vengono assegnati i metodi alle classi e si specifica con gli oggetti devono interagire fra loro per realizzare i casi d'uso. In questa realizzazione si usano i diagrammi di interazione e i *pattern*, e si assegano le *responsabilità* alle classi. Esistono due tipi principali di responsabilità, quelle di *azione* che definiscono ciò che possono fare gli oggetti, e quelle di *conoscenza* ovvero i dati a cui hanno accesso direttamente o indirettamente (tramite calcolo).

È importante sottolineare che un *metodo* di una classe non è una *responsabilità*, ma i metodi vengono implementati per soddisfare le responsabilità.

6.1 Principi Generali

6.1.1 Information Hiding

Consiste nel separare l'*interfaccia* di un componente, che è pubblica, dalla sua *implementazione*, che rimane privata e invisibile all'esterno.

Interfaccia L'interfaccia esprime ciò che il componente offre e/o richiede all'esterno.

L'*information hiding* permette di non comprendere necessariamente i dettagli implementativi di un componente per usarlo; di essere facilmente manutenibile dato che che l'implementazione di ogni componente è a compartimento stagno; e ovviamente garantisce maggiore sicurezza.

Un esempio di *information hiding*, permesso dall'*incapsulamento* di molti linguaggi di programmazione a oggetti, è l'uso di *Getters & Setters*. Ovvero i metodi `get()` e `set()` permettono di nascondere la rappresentazione dei dati e di restituirli/modificarli effettuando dei controlli interni.

Non è sempre utile il loro utilizzo in quanto se non sono necessari e vengono introdotti lo stesso poi occorrà mantenerli.

Questa astrazione riguardante i dati, è diversa da quella che riguarda il concetto di *moduli* o *librerie*. Entrambe permettono di fare *information hiding*, ma mentre quest'ultime sono puramente funzionali, l'invocazione di un'operazione sulle prime, invece, può comportare una variazione dello stato.

6.1.2 Coesione

È una proprietà di un compomente che consiste nel raggruppare funzionalità strettamente collegate fra loro, secondo diverse classificazioni:

- Coesione funzionale: raggruppa parti che collaborano per realizzare una singola funzionalità.
- Coesione comunicativa: raggruppa parti che operano sugli stessi dati di input o che collaborano agli stessi output.
- Coesione procedurale: raggruppa le parti che realizzano i passi di un modulo (*libreria*).

Di seguito, invece, altri tipi di coesione, ma che sono fortemente **sconsigliati**:

- Coesione temporale: tra azioni che sono fatte nello stesso arco di tempo, ma in questo modo si và a realizzare un componente che è difficilmente riutilizzabile.
- Coesione logica: tra elementi che hanno una correlazione logica (per esempio nello stesso dominio) piuttosto che funzionale.

6.1.3 Disaccoppiamento

È una proprietà di un insieme di componenti (nella maggior parte dei casi di una architettura) che vengono accoppiati sulla base del loro legame, come la presenza di dipendenze o se comunicano tramite uno scambio di messaggi.

L'obbiettivo è di creare sistemi che siano **disaccopiati**, ovvero i cui componenti **non** siano fortemente legati fra loro.

L'ideale sarebbe creare sistemi che esibiscano un **alto** grado di **coesione** e uno **basso** di **disaccoppiamento**.

6.2 SOLID

SOLID definisce cinque principi base di progettazione (di dettaglio) che si applicano alla programmazione *object-oriented*:

- **S ingle Responsibility Principle**: Una classe dovrebbe avere solo un motivo per cambiare (cioè una sola responsabilità), quindi nel caso individuamo due responsabilità occorre dividere la classe in due classi. Questo perchè se la classe ha più responsabilità, le modifiche potrebbero coinvolgere altre funzionalità della classe e i moduli che le usano. Tutto questo sta a significare che la classe deve implementare una singola funzionalità. L'unica eccezione

alla regola si presenta quando non si può cambiare una delle due responsabilità senza cambiare contestualmente anche l'altra.

- **O pen Closed Principle**: Le entità software, come classi o moduli, devono poter essere **aperte** per essere **estese**, ma **chiuse** per essere **modificate**. Un esempio può essere l'uso di *classi astratte* e del principio di *ereditarietà* per poter estendere classi già esistenti senza cambiarle.
- **L iskov Substitution Principle**: Questo Principio di Sostituzione definisce che se una classe S è sottotipo di una classe T, allora per ogni oggetto o_1 di S esiste un oggetto o_2 di T tale che per ogni programma P definito su T, il suo comportamento è immutato quando o_1 è usato al posto di o_2.
- **I nterface Segregation Principle**: Si preferisce costruire interfacce che siano a grana fine e specifiche per ogni *client*, cioè un *client* non deve dipendere da interfacce di cui non usa tutti i metodi.
- **D ependency Inversion Principle**: Un modulo non deve dipendere da implementazioni concrete di una classe, ma da una sua astrazione, in questo modo sarà più facile estendere le sue funzionalità. Questo principio garantisce il *disaccoppiamento*, in quanto il *modulo* dipenderà solo dall'interfaccia senza sapere como sono costruiti i suoi servizi.

6.3 GRASP

General Responsibility Assignment Software Patterns è un'altra famiglia di principi di progettazione che si basa sull'assegnazione delle responsabilità. Si definiscono prima gli oggetti e i loro metodi, e ci si fà guidare da *pattern* (schemi) di assegnazione delle responsabilità.

6.4 Qualità del Software

Il modello di qualità del prodotto **ISO/IEC 25010** è composto da 8 caratteristiche:

- Adeguatezza funzionale: rappresenta la misura che definisce se un sistema fornisce funzioni che soddisfano le esigenze dichiarate precedentemente.
- Efficienza delle prestazioni: rappresenta le prestazioni relative alla quantità di risorse che vengono utilizzate e ai tempi di risposta.
- Compatibilità: misura in cui un sistema o componente può scambiare informazioni con altri sistemi o componenti e/o nel caso

viene eseguito nello stesso ambiente hardware dell'altro software/componente condividendo le stesse risorse.

- Usabilità: misura in cui un sistema può essere usato dagli utenti specificati in precedenza in fase di analisi, in modo che possa soddisfare i loro bisogni. Misura anche se il sistema è semplice da imparare ad usare e la qualità dell'interfaccia utente.
- Affidabilità: misura in cui si analizza il comportamento del sistema in condizioni normali e in condizioni con presenza di guasti hardware e/o software. Inoltre misura anche la possibilità di recuperare dati e di ristabilire lo stato precendente in caso di guasti.
- Sicurezza: misura in cui il sistema protegge le informazioni e i dati e dia il giusto grado di accesso in base al livello di autorizzazione dell'utente. Valuta anche se ogni azione può essere ricondotta in modo univoco a chi l'ha compiuta.
- Manutenibilità: misura che definisce l'efficienza con cui un sistema può essere migliorato o corretto. È garantita dalla *modularità* del sistema.
- Portabilità: misura l'efficacia con cui un software può essere trasferito da un ambiente a un altro, e quanto può essere efficace nella sostituzione di un altro software che si occupa dello stesso scopo.

6.5 Stili dell'Architettura

Definizione (Scalabilità)**.** La **scalabilità** di un software è definita come la sua capacità di aumentare il proprio *throughput* in proporzione all'aumento dell'hardware utilizzato.

Definizione (Scalabilità Verticale)**.** La **scalabilità verticale** si riferisce all'aggiunta di memoria e CPU su un singolo nodo (*host*). Ideale per i *database*.

Definizione (Scalabilità Orizzontale)**.** La **scalabilità orizzontale** si riferisce all'aggiunta di più nodi hardware. È ideale per le applicazioni web che condividono poca memoria e presentano tanti *thread* indipendenti.

Le architetture viste in precedenza presentano diverse caratteristiche sulla *qualità del software*:

- Client-Server e 2 o N-tier: presenta una grande *disponibilità* e *fault tolerance* in quanto se un server non è disponibile si reindirizza la richiesta ad un altro server. Presenta anche un'alta modificabilità, ovvero la misura in cui il sistema può essere modificato senza introdurre difetti o degradare la qualità, grazie all'elevata *coesione* e

disaccoppiamento. Anche La scalabilità è elevata, ma può presentare un collo di bottiglia per via dei database se si scala orizzontalmente.

- Pipes & Filters: per la *fault tolerance* occorre aspettare la riparazione del componente che rompe la catena. La *modificabilità* e la *scalabilità* sono alte.
- Publish-Subscribe: la *fault tolerance* può essere aggirata introducendo un maggior numero di *dispatcher* (anche chiamati *broker*). Anche qui la *modificabilità* e la *scalabilità* vanno bene, quest'ultima se si utilizza sempre un grande numero di *dispatcher*.
- P2P: la *fault tolerance* e la *scalabilità* ovviamente sono sempre garantite dato che tutti i nodi sono alla pari. La *modificabilità* và bene se l'architettura si occupa solo della parte di comunicazione; e la *performance* dipende dal numeri di nodi connessi e dagli algoritmi utilizzati.
- Coordinatore di Processi: per avere maggiore *fault tolerance* e *scalabilità* occorre replicare tante volte il *coordinatore*. Per la *modificabilità* si possono aggiornare i *server* se non cambiano le funzionalità che sono esportate. Le *performance* sono alte se il coordinatore riesce a gestire più richieste in modo concorrente e sono influenzate anche dal server più lento.

6.6 Design Pattern

I **design pattern** sono una serie di regole pratiche, definite molte volte grazie a secoli di esperienza, che il progettista deve seguire.

Gli autori chiamati *Gang of Four* hanno definito 23 design pattern che sono suddivisi in base al loro scopo:

- Creazionali: propongono soluzioni per la creazione di oggetti.
- Comportamentali: soluzioni per gestire al meglio la suddivisione di responsabilità fra classi o oggetti.
- Strutturali: soluzioni per come devono essere composte le classi o gli oggetti.

I livelli di astrazione nella costruzione di un *pattern* possono essere diversi, si può realizzare un design *ad hoc* per un'applicazione o per un sottosistema, quindi lavorando ad un livello molto alto di astrazione, oppure si possono cercare soluzioni per un generico problema di design nel contesto di lavoro, fino ad arrivare ad un livello più concreto (di codice), per realizzare *pattern* semplici e riusabili.

6.6.1 Strategy

Questo *design pattern* si basa sul favorire la composizione tramite la definizione di *interfacce*, preferite rispetto all'*ereditarietà*. Quindi in primo luogo si identifica una famiglia di algoritmi e ognuno di essi viene incapsulato in un'interfaccia. In questo modo si potranno alterare ed estendere le parti che variano rispetto a quelle che non lo fanno.

I partecipanti in questo pattern sono:

- Strategy: rappresenta una famiglia di algoritmi, e quindi definisce un'interfaccia comune ad essi.
- Concrete Strategy: implementa un algoritmo.
- Context: contiene un riferimento ad un'istanza di tipo *Strategy*, può definire un'interfaccia che consenta alla *Strategy* di accedere ai propri dati, oppure può passarglieli come argomenti quando viene invocato un suo metodo.

Applicabilità Si utilizza questo modello quando sono presenti molte classi simili che differiscono solo nel comportamento; quando ci sono diversi varianti di un algoritmo e quando l'algortimo utilizza dati che non devono essere a conoscenza del contesto; e quando si vuole evitare di esporre le strutture dati utilizzate.

I benefici principali di questo modello sono che elimina molte istruzioni condizionali di grandi dimensioni all'interno della classe *Context* e che fornisce proprio un'alternativa alla formazione di gerarchie nel *Context*, preferendo l'implementazione esplicita di algoritmi e funzionalità. Gli svantaggi principali invece sono l'aumento del numero di oggetti e che tutti gli algoritmi devono utilizzare la stessa interfaccia *Strategy*.

Il caso in cui, invece, l'utilizzo di questo modello è sconsigliato è quando diverse *Concrete Strategy* richiedono dati diversi: se questi vengono passati all'interfaccia generica (*Strategy*) c'è una grande probabilità che non tutte le *Concrete Strategy* li utilizzeranno. Una possibile soluzione è il rafforzamento del legame tra il *Context* e le *Concrete Strategy*, in modo tale che quest'ultime possano richiede dati al primo.

6.6.2 State

È un pattern comportamentale, che consente ad un oggetto di modificare il suo comportamento quando il suo stato interno cambia.

I partecipanti sono:

- Context: rappresenta l'interfaccia che utilizza l'utente e mantiene un'istanza di *Concrete State* che rappresenta lo stato corrente.

- State: definisce l'interfaccia degli stati; può anche essere una classe concreta o astratta nel caso ci sono comportamenti comuni che non dipendono dallo stato, o nel caso si vogliono definire comportamenti di *default*.
- Concrete State: è una sottoclasse di *State* che implementa un singolo stato. Generalmente sono i *Concrete State* ad effettuare la transizione di stato, dato che conoscono il loro *next state*, ma nei casi più semplici si può cambiare stato anche nel *Context*.

6.6.3 Factories

Una **Factory** è una classe il cui compito è quello di creare e restituire istanze di altre classi.

In questo modo viene nascosta, e resa più semplice, la costruzione degli oggetti, senza dover invocare ogni volta il proprio *costruttore*.

Infatti, ogni volta che viene utilizzato il comando `new`, viene violato il principio di progettazione di scrivere codice basandosi esclusivamente sulle *interfacce*, dato che possiamo associare come istanza di una classe una sua sottoclasse (per il *Principio di Sostituzione di Liskov*).

Inoltre, se abbiamo un frammento di codice che restituisce un'istanza di classe diversa in base al valore di determinate variabili, allora potrebbero venire violati i principi di *Open-Closed* e di *Information Hiding*, dato che ogni volta che si aggiunge o rimuove una classe, occorre anche modificare questo pezzo di codice.

Di seguito vengono trattate le tre tipologie di pattern *factories*.

Simple Factory Quando si vuole implementare una logica di creazione di un oggetto molto complessa e la si vuole separare dalle sue funzionalità, allora si delega la creazione a un oggetto chiamato **factory**.

Quindi, praticamente, viene spostata la logica di creazione dell'oggetto in un metodo della classe *Factory*, e in questo modo occorrerà solo invocare questo metodo all'esterno per ottenere un'istanza. Così se dovesse cambiare il modo in cui creaiamo gli oggetti, non servirà apportare modifiche all'esterno, ma solo nella *Factory*.

Factory Method Questo pattern si focalizza sull'uso dell'ereditarietà per decidere quale oggetto deve essere istanziato.

I partecipanti sono:

- Product: definisce l'interfaccia per il tipo di oggetti che il *factory method* crea.
- Concrete Product: implementa l'interfaccia *Product*.

- Creator: dichiara il *factory method* e restituisce un oggetto di tipo *Product*.
- Concrete Creator: sovrascrive il *factory method* per restituire un'istanza di *Concrete Product*.

Si usa questo pattern solo se una classe non può prevedere la classe di oggetti che deve creare, o se una classe richiede delle sottoclassi per specificare gli oggetti da creare.

Ovviamente i benefici solo la grande flessibilità e la riusabilità di questo pattern, ed inoltre dall'esterno è necessario interfacciarsi solo con la classe *Product*. Gli svantaggi sono che ogni volta che vogliamo istanziare uno specifico *Concrete Product* occorre creare una sottoclasse di *Creator*. Il *Creator* inoltre può essere sia astratto che concreto (nel caso si voglia implementare un *factory method*) di default.

Abstract Factory È una generalizzazione del pattern *Simple Factory*, e la differenza principale col *Factory Method*, è che mentre lì si utilizza l'ereditarietà basandosi su una sottoclasse che gestisce l'istanzazione di un oggetto, nell'*Abstract Factory*, una classe delega la responsabilità di creazione di un oggetto a un altro oggetto tramite la composizione. In realtà, l'oggetto delegato per la creazione, potrebbe utilizzare dei *factory method* per eseguire l'istanzazione, quindi applicando entrambi i pattern.

I **Pure Fabrication** sono un pattern *GRASP* che si pone l'obbiettivo di non violare l'*alta coesione* e il *basso accoppiamento*. Questo viene realizzato assegnando un insieme di responsabilità molto coese fra loro ad una classe artificiale che non rappresenta niente nel dominio del problema, di conseguenza avremo implementato che il *basso accoppiamento* che favorisce il riutilizzo. La progettazione di oggetti può essere partizionata in due gruppi, quelli decomposti sulla rappresentazione, e quelli decomposti sul comportamento. L'ultimo gruppo non rappresenta proprio niente nel dominio del problema e sono solo costruiti per convenienza del programmatore, da qui il nome *Pure Fabrication*.

Una *Factory* è un *Pure Fabrication* che ha l'obbiettivo di confinare ed incapsulare la logica di creazione, soprattutto quand'è abbastanza complessa.

6.6.4 Singleton

Questo pattern è utile quando si vuole che un metodo abbia una sola istanza e si vuole fornire un punto di accesso globale ad essa. Per prevenire le multiple istanze, si crea un'istanza della classe all'interno della stessa con accesso privato e statica; successivamente si realizza un metodo pubblico che permette di rendere disponibile l'istanza.

Inoltre si preferisce usare una *Lazy Initialization* dell'istanza, ovvero quando il metodo pubblico è invocato, se l'istanza non è stata creata si crea e si restituisce, altrimenti si restituisce solo. Infatti la creazione dell'istanza potrebbe richiedere dei parametri oppure semplicemente è troppo pesante e non avrebbe senso crearla prima che venga usata o se non verrebbe usata proprio.

Il problema si crea quando si lavora in un ambiente *multi-thread* per colpa delle *race condition*. Una soluzione potrebbe essere quella di sincronizzare il metodo che restituisce l'istanza, ma sarebbe troppo dispendioso dato che la creazione dell'istanza avviene solo una volta. In alternativa si può usare un `double-checked-locking`.

Consiste nel dichiarare l'istanza di tipo `volatile` ovvero viene garantito che quando leggiamo il valore da questa variabile si legge sempre l'ultimo scritto. In seguito nel metodo che restituisce l'istanza si fà prima un controllo se l'istanza è diversa da `NULL`, successivamente si crea un blocco `synchronized` che garantisce la *mutua esclusione* sull'intera classe, e al cui interno prima di istanziare l'oggetto si fà un altro controllo se l'istanza corrente è `NULL` (dato che il controllo precendente non era mutualmente esclusivo).

Se si vogliono creare delle sottoclassi utilizzando *Singleton*, quindi mantenendo sempre una sola istanza, si può fare in due modi, o nel metodo della superclasse determinare il tipo dell'istanza da creare attraverso un paramtro passato come argomento, anche se in questo caso non è garantito che i costruttori delle sottoclassi siano privati e in questo caso si potranno istanziare più oggetti delle sottoclassi (inoltre viene anche violato il principio di *Open-Closed* come visto prima); oppure il modo migliore è quello di spostare il metodo che restituisce l'istanza in ogni sottoclasse.

Tuttavia i vantaggi di usare *Singleton* piuttosto che una *classe statica*, sono la possibilità di passare l'istanza unica come parametro in un altro metodo, e di poter utilizzare i *factory* pattern per costruire l'istanza.

6.6.5 Adapter

Gli **adapter** sono utili quando vogliamo interagire con un'interfaccia che non è omogenea a quella utilizzata dal *client*.

Questo pattern permette di convertire l'interfaccia di una classe in un'altra interfaccia che è compatibile con quella utilizzata dal *client*.

Si usa la *delegazione* per associare un interfaccia *adapter* ad una *adattata*.

I partecipanti sono:

- Target: l'interfaccia che il *client* usa.
- Client: utilizza oggetti che siano conformi all'interfaccia *Target*.

- Adaptee: definisce un'interfaccia già esistente che necessita di essere adattata.
- Adapter: adatta l'interfaccia *Adaptee* all'interfaccia *Target*.

L'*Adapter* può anche ricoprire il ruolo di *Concrete Strategy* nello *Strategy* pattern. In questo modo implementiamo la possibilità di cambiare gli oggetti *Adapter* a runtime.

6.6.6 Proxy

Questo pattern fornisce un surrogato per un altro oggetto, ovvero lo sostituisce in modo da controllare l'accesso alle sue funzionalità. Può sembrare simile ad *Adapter* ma, a differenza di questo, il **Proxy** può eseguire pre e post-elaborazioni, inoltre esso e l'oggetto originale hanno la stessa interfaccia.

Esistono vari tipi di Proxy:

- Remote Proxy: permette l'accesso ad un oggetto remoto. Nello specifico il *Proxy* si comporta come una rappresentazione locale di un oggetto che risiede in un'altra *Java Virtual Machine*. Il client invoca un metodo del *Proxy* che lo inoltra tramite la rete all'oggetto reale, una volta ricevuta la risposta, il *Proxy* la restituisce al *client*.

  Inoltre è anche possibile introdurre un altro *Proxy* lato server, a questo punto, quello lato client lo chiameremo *Client Helper*, mentre il proxy lato server sarà il *Service Helper*. I due proxy hanno ovviamente la stessa interfaccia, che è la stessa dell'oggetto reale, il *Service Helper* in particolare aggiunge un altro *layer* di operazioni, spacchettando la richiesta del *Client Helper* ed invocando i metodi opportuni dell'oggetto reale. Al contrario, il *Service Helper* impacchetta il risultato del metodo e lo spedisce al *Client Helper* che una volta ricevuto, lo spacchetterà e lo darà al *client*.

  Nel *Remote Method Invocation* di Java, il *Client Helper* è chiamato *Stub*, mentre il *Service Helper* è lo *Skeleton*. L'unica differenza col metodo precendente è che qui lato client non possediamo già lo *Stub*, quindi il *client* deve prima effettuare una richiesta di `lookup()` nell'*RMI Registry* al server che gli restituirà l'oggetto *Stub* per il servizio richiesto, che potrà essere utilizzato per effettuare le richieste al server.

- Protection Proxy: implementa un controllo sugli accessi.
- Cache Proxy: mantiene coppie del tipo `richiesta-risposta` sgravando il *server* di questa responsabilità.
- Synchronization Proxy: gestisce gli accessi concorrenti.

- Virtual Proxy: si comporta come l'oggetto originale, mentre quest'ultimo viene costruito. Viene utilizzato principalmente quando per l'appunto l'oggetto reale richiede costi di creazione molto elevati. Una volta terminata la creazione, il *Virtual Proxy* delegherà le richieste all'oggetto originale.

6.6.7 Decorator

Questo pattern si usa per evitare di creare un'interfaccia che abbia un numero troppo elevato di classi che la implementano. Una prima soluzione sarebbe quella di avere una serie di valori booleani nella superclasse che indicano se l'oggetto presenta o no delle determinate caratteristiche, in questo modo eliminiamo un po' di sottoclassi inutili e manteniamo solo quelle principali. Ovviamente anche questa soluzione è errata, dato che nel caso si vuole aggiungere o rimuovere una caratteristica si andrebbe a violare il principio *Open-Closed*.

Questo pattern ci viene incontro offrendo una soluzione: l'idea è quella di trasformare quelle caratteristiche che erano sotto forma di valori booleani, in classi, a questo punto se vogliamo aggiungere una caratteristica al nostro oggetto basta *"wrapparla"* all'interno della caratteristica che avevamo già, e così via fino a quando non arriviamo ad una foglia, rappresentata da una delle classi che avevamo indicato come principali.

I partecipanti sono i seguenti:

- Component: è l'interfaccia generale di tutti gli altri oggetti di questo pattern.
- Concrete Component: la classe *"foglia"* di oggetti che ricevono nuove caratteristiche.
- Decorator: definisce un'interfaccia che estende *Component*, in più mantiene un riferimento ad un altro oggetto di tipo *Component*.
- Concrete Decorator: definisce una caratteristica.

Nonostante viene usata l'ereditarietà tra *Decorator* e *Component*, questa non viene utilizzata per aggiungere nuove funzionalità, ma per avere un *type matching* tra i *Decorator* e gli oggetti che vanno a *"decorare"*. Infatti l'aggiunta di nuove funzionalità avviene non grazie all'ereditarietà, ma per merito della composizione che ci permette di aggiungere *Decorator* a *runtime*.

Questo pattern è molto utile perchè ci permette anche di attaccare ad un oggetto la stessa caratteristica più di una volta.

Questo pattern può presentare alcuni problemi, ad esempio l'interfaccia di un oggetto *Decorator* deve essere uguale all'interfaccia

dell'oggetto che và a decorare. Inoltre se l'interfaccia *Decorator* implementa una sola caratteristica, l'interfaccia stessa và omessa e si accorpora la caratteristica direttamente in *Concrete Decorator*. Dato che la classe *Component* è ereditata da componenti e decoratori è necessario cercare di mantenerla il più leggera e semplice possibile; dato che se è complessa di conseguenza sarà complessa anche la classe *Decorator* che diventa costosa da usare in grandi quantità. Un'altra limitazione, è che occorre usare questo pattern solo se si vuole aggiungere uno strato all'oggetto e non si vuole modificarlo, in quel caso và usato il pattern *Strategy*.

Ad esempio a differenza di *Strategy* in *Decorator* possiamo solo aggiungere caratteristiche dinamicamente ad un oggetto, mentre nel primo possiamo anche modificarle. D'altro canto in *Strategy* quando modifichiamo la *Concrete Strategy* di un oggetto si và a modificare l'oggetto stesso, che è a conoscenza anche delle funzionalità che gli sono state aggiunte, e non sempre questa è una cosa desiderata.

7 Verifica e Validazione

Come dimostrato da Alan Turing nel 1937, non esiste alcun programma P che prende in input altri programmi e che per ogni programma preso in input decide in tempo finito se termina, e tantomeno se è corretto.

Quindi non è facile effettuare verifiche sul corretto funzionamento di un programma.

Ci sono anche alcune caratteristiche che rendono questo processo particolarmente difficile, come il fatto che sia sempre in evoluzione, la distribuzione irregolare dei guasti e soprattutto la non linearità, ad esempio se abbiamo verificato che un programma funziona correttamente con input `x = 100`, non è detto che vada bene anche per valori inferiori a `x`.

Alcuni errori, possono dipendere soprattutto dal linguaggio utilizzato, come ad esempio la presenza di *deadlock* o *race condition* o dei problemi dovuti al polimorfismo ed ereditarietà nei linguaggi *object-oriented*.

La fase di testing non è la parte finale del processo di sviluppo di un software, ma inizia non appena si decide di creare il prodotto e dura anche molto oltre la consegna di esso, ovvero per il tutto il tempo che il software è in uso.

Per capire se un prodotto è pronto per il rilascio, ci sono alcune misure di *dependability* che aiutano a capirlo:

- La Disponibilità: misura la qualità del software sulla base del tempo in cui il sistema è in esecuzione rispetto al tempo in cui il sistema è *down*.

- Il Tempo Medio tra i Guasti: misura la qualità in termini di tempo tra un guasto e il successivo.
- L'Affidabilità: indica la percentuale di operazioni che terminano con successo.

Inoltre i test eseguiti possono essere suddivisi in due categorie:

- Gli Alpha Test: sono eseguiti dagli sviluppatori o da utenti che comunque sono in un ambiente controllato dal produttore del software.
- I Beta Test: sono i test eseguiti da utenti nel loro ambiente, eseguendo operazioni senza essere monitorati.

Mentre la **Validazione** si pone l'obbiettivo di affermare che si sta costruendo il sistema di cui l'utente necessita; la **Verifica**, invece, vuole constatare che il sistema rispetti le specifiche. Si può dire che il processo di *verifica* è intrinseco a quello di *validazione*.

Un'altra distinzione importante è tra **Malfunzionamento** e **Difetto**. Il *malfunzionamento* ha una natura dinamica ovvero può essere osservato solo a tempo di esecuzione e si verifica quando il sistema non si comporta secondo le specifiche. Il *difetto*, invece, si intende come difetto nel codice ed è quello che causa il malfunzionamento, ma non sempre, in questo caso si parla di *difetto latente*, quando il suo effetto è nullo oppure è contenuto in un cammino mai percorso nei test. L'**Errore**, invece, è l'incomprensione umana che ha causato il *difetto*.

Come detto all'inizio, il *Testing* è sottoposto al problema dell'indecidibilità, ovvero effettuare un *testing esaustivo* su ogni possibile input richiederebbe tempo infinito (se il dominio degli input è anch'esso infinito).

Definizione (Tesi di Dijkstra)**.** Il test di un programma può rilevare la presenza di difetti ma non dimostrarne l'assenza.

7.1 Verifica Statica

Non prevede l'esecuzione del programma, utilizzando *metodi manuali* come la lettura del codice (*desk-check*), oppure *metodi formali* come l'analisi statica.

Ci sono due metodi pratici di *desk-check*:

- Inspection: si esegue una lettura mirata del codice, guidata da una lista di controllo con l'obbiettivo di rilevare la presenza di difetti. Si focalizza la ricerca su aspetti già ben definiti effettuando un *"error guessing"*. Questa attività è svolta da dei verificatori diversi dai programmatori. Le liste di controllo sono frutto di esperienza e contengono tipicamente aspetti che non possono essere

controllati in modo automatica. Le liste sono aggiornate ad ogni iterazione di *Inspection*. Si possono vedere come delle linee guida e buone pratiche che permettono di evitare parecchie ore di testing.

- Walkthrough: si basa su una lettura critica del codice che consiste nel percorrere il codice simulandone l'esecuzione. È svolto sia dai programmatori che dai verificatori. Mentre *Inspection* è basato sulla rilevazione di errori presupposti, *Walkthrough* è molto più completo ma è meno rapido.

I vantaggi della verifica manuale solo la praticità, l'intuitività e la convenienza economica, dato che dipendono solo dalla lunghezza del codice.

I **Metodi Formali**, invece sono basati sulla dimostrazione formale di correttezza di un modello finito; quindi dimostrare corretto il modello garantisce anche la correttezza di una sua istanza.

7.2 Verifica Dinamica

Una caratteristica della *verifica dinamica* è la ripetibilità delle prove, ovvero bisogna poter ripetere una sessione di test in condizioni immutate.

7.2.1 Progettazione Casi di Test

Un **caso di prova** o *test case* è una tripla del tipo `<input, output (atteso), ambiente>`.

La **test obligation** è una specifica o proprietà che i casi di prova devono soddisfare. Un insieme di *test obligation* è un criterio di adeguatezza.

Un insieme di casi di prova, chiamato **batteria di prove** soddisfa un *criterio di adeguatezza* se tutti i test hanno successo e ogni *test obligation* è soddisfatta da almeno un caso di test.

Le *test obligation* possono essere di quattro tipi:

- Black Box: a scatola chiusa, ovvero si guardano le specifiche del software e ci si basa sulla conoscenza delle funzionalità.
- White Box: a scatola aperta, si guarda la struttura del codice.
- Definiti guardando i modelli utilizzati nella fase di specifica o di progettazione o derivati dal codice.
- Definiti da *fault ipotetici*, ovvero si cercano dei difetti abbastanza comuni.

Una **procedura di prova**, invece, serve per eseguire, registrare e analizzare i risultati di una *batteria di prove*.

La realizzazione dell'*ambiente di prova* si chiama test scaffolding e rappresenta del codice aggiuntivo per eseguire dei test. Può includere:

- Driver di Test: sostituiscono un programma principale o di chiamata.
- Stub: sostituiscono funzionalità chiamate e utilizzate dal software in prova.
- Test Harness: sostituiscono delle parti dell'ambiente di distribuzione.
- Tool per gestire l'esecuzione del test.
- Tool per registrare i risultati.

Metodi Black Box per Generare Input Per ogni funzionalità si deriva un insieme di casi di test. Per ogni tipo di input si individuano dei valori da testare e per l'insieme dei parametri della funzionalità si utilizzano delle tecniche di *testing combinatorio* per ridurre le combinazioni.

Il metodo statistico permette di selezionare i test case in base alla distribuzione di probabilità dei dati di ingresso del programma. Quindi si progetta il test per eseguire il programma sui valori di ingresso più probabili, lo svantaggio è che questo potrebbe non corrispondere alle effettive condizioni di utilizzo del software e potrebbe essere oneroso calcolare l'output atteso.

Il dominio dei dati di input viene partizionato in *classi di equivalenza*. Due input appartengono alla stessa classe di equivalenza se dovrebbero produrre lo stesso comportamento, ma non necessariamente lo stesso output. Questo criterio è valido per tutti quei programmi in cui il numero dei possibili comportamenti è inferiore rispetto al numero delle possibile configurazioni dell'input.

Molte volte si cerca anche di testare il programma sui *valori limite* (o di frontiera), come ad esempio gli estremi delle classi di equivalenza. Si guardano spesso gli intorni dei valori limite perchè spesso è lì che si nascondono i difetti del codice.

Per ogni input si definiscono anche i *casi non validi*, ovvero quelli che devono generare un errore.

Il metodo random, invece, permette di generare in modo automatico un insieme grande di valori; la generazione costa zero, ma dato che viene generato casualmente non può essere ripetibile. È applicabile se costa poco l'esecuzione.

Un test può anche essere basato su un *catalogo*, ovvero raccogliere tutti i casi di test su un tipo di variabile che il produttore del software ha già catturato con l'esperienza in progetti passati.

7.3 Testing Combinatorio

In presenza di più dati di input, il loro prodotto cartesiano potrebbe portare ad un insieme troppo grande e poco gestibile. Si usano delle tecniche per ridurre l'esplosione combinatoria.

Le strategie a **vincoli** possono essere di tre tipi:

- Di Errore: per ridurre le combinazioni si considera un solo caso, per ogni input, quando non è valido.
- Di Proprietà: in alcuni casi si può verificare che per alcuni input i valori vengano divisi in base a delle *"property"*, mentre per altri input i valori possono essere divisi in delle *"if property"*, ciò sta a significare che se un input ha un valore che appartiene ad una *"property"*, gli altri input possono solo assumere valori che appartengono alla corrispondente *"if property"*.
- Singoletti: si può decidere per un input di fissare un solo valore.

Questa tecnica basata sui vincoli, funziona bene solo se i vincoli che introduciamo solo reali nel dominio che stiamo analizzando e non sono aggiunti solo per lo scopo di limitare le combinazioni.

7.4 Pairwise Testing

Quando non è possibile limitare il dominio, si opta per un'altra tecnica, ovvero si generano tutte le combinazioni possibili solo per i sottoinsiemi di soli 2 input ed infine si sommano i risultati. Molte volte si possono generare le combinazioni in maniera efficiente, ovvere generare solo i valori fra i due insiemi di input più grandi e poi aggiungere ad ogni coppia ottenuta un elemento di un terzo insieme in modo tale che vengano coperti anche tutti i casi di combinazione tra il primo insieme ed il terzo, e tra il secondo insieme ed il terzo.

7.5 Criteri White-Box

Sono criteri che si basano sulla struttura del codice per individuare i casi di input. Aiutano ad aggiungere altri test oltre a quelli generati con quelli di tipo *black-box*.

Si dice che un programma non è testato adeguatamente se tutti i suoi cammini non vengono esercitati dai test.

Un *grafo di flusso* definisce la struttura del codice identificandone tutte le parti collegate fra loro.

Quindi si cerca un insieme di valori per i dati di input in modo tale che vengano esercitati tutti i comandi e tutte le condizioni del programma.

La **Misura di Copertura** per i **comandi** è definita come segue:

$$Misura\ di\ Copertura = \frac{Numero\ di\ Comandi\ Esercitati}{Numero\ di\ Comandi\ Totali}$$

È importante sottolineare che la *copertura* non è monotona rispetto alla dimensione dell'insieme di test. Infatti ci possono essere insiemi più piccoli che magari garantiscono una copertura maggiore.

Se in un caso di test vengono eseguiti tutti comandi, non è detto che vengano eseguite anche tutte le condizioni, quindi si parla anche di **Misura di Copertura** per le **condizioni**:

$$Misura\ di\ Copertura = \frac{Numero\ di\ Archi\ Entranti}{Numero\ di\ Archi\ Totali}$$

Per le *condizioni composte*, bisogna anche fare in modo che i test esercitino tutti i possibili valori di verità delle singole condizioni semplici all'interno.

Copertura di Condizioni Semplici Un insieme di test per un programma copre tutte le condizioni semplici (*basic condition*) se per ogni condizione semplice nel programma, l'insieme di test ne contiene almeno uno in cui la condizione vale `true` e almeno uno in cui vale `false`.

$$Copertura\ delle\ Basic\ Condition = \frac{Numero\ di\ Valori\ di\ Verità\ Assunti}{2^{Numero\ di\ Basic\ Condition}}$$

Nella *Copertura delle Multiple Condition*, per ogni condizione multipla con n condizioni semplici, invece, si cerca di coprire tutte le 2^n possibili combinazioni. Ma grazie alla valutazione *short-circuit* [6] di molti linguaggi di programmazione ci si può ridurre a meno.

La *Copertura dei Cammini*, invece richiede di percorrere tutti i cammini, cresce in modo esponenziale col il numero di decisioni. Ovviamente in presenza di cicli il numero di cammini è potenzialmente infinito, per questo ci si limita a richiedere casi di test che esercitino ogni ciclo 0 volte, 1 volta e più di una volta.

7.6 Fault Based Testing

Ipotizza dei difetti potenziali nel codice sotto test, ovvero si valuta un insieme di test sulla base della loro capacità di rilevare i difetti ipotizzati.

La tecnica più nota è il **Test Mutazionale**, ovvero si iniettano difetti modificando il codice.

Occorre priva aver esercitato un programma P su una batteria di test T e si verifica che P sia corretto rispetto a T. Successivamente si

[6] `https://en.wikipedia.org/wiki/Short-circuit_evaluation`

introducono dei piccoli difetti in P chiamate *mutazioni*. Il programma *mutante* è indicato con P'. Quindi si eseguono su P' gli stessi test di T.

Se il test non rileva i difetti iniettati, allora significa che la batteria di test non era buona, se li rileva si ha una maggiore fiducia su essa.

Nella realtà si applicano tante piccole mutazioni a P per ottenere una sequenza di mutanti $P_1, P_2, \ldots, P_n$. Si dice che T uccide il mutante P_j se rileva un malfunzionamento in almeno un caso di test di T.

L'**Efficacia di un Test** è il rapporto $\frac{Nr.\ Mutanti\ Uccisi}{Numero\ Totale\ Mutanti}$.

Un mutante sopravvive a una test suite se per tutti i test case della batteria non si distingue l'esito del test se eseguito sul programma originale o sul mutante.

Qui si lavora sull'assunzione che iniettando difetti piccoli e artificiali è possibile individuare difetti più complessi e reali.

Praticamente si iniettano M difetti meccanici nell'originale ottenendo il mutante, si esegue il mutante sulla batteria, e si troverà un numero di difetti uguale a N. Se $N > M$ allora ho trovato difetti non artificiali che erano presenti anche nel programma originale.

Esempi di mutazioni:

- CRP: sostituzione di una costante per una costante.
- ROR: sostituzione di un operatore relazione (da $\leq$ a $<$).
- VIE: eliminazione dell'inizializzazione di una variabile.
- LRC: sostituzione di un operatore logico.
- ABS: inserimento di un valore assoluto.

Un *mutante* si dice ***invalido*** se non è sintatticamente corretto, ovvero se non passa la compilazione, altrimenti si dice ***valido***.

Un *mutante* si dice ***utile*** se è *valido* e distinguerlo dal programma originale non è facile, ovvero esiste solo un piccolo sottoinsieme di casi di test che lo consente. Invece si dice ***inutile*** se è ucciso da quasi tutti i casi di test.

Si cerca di costruire mutazioni che producano mutanti *utili* e *validi*.

Un mutante può essere ***equivalente*** al programma originale se ad esempio la mutazione non crea un vero difetto o se la batteria di test era troppo debole.

Ovviamente questa tecnica è limitata dal numero di mutanti che possono essere definiti, dato che il loro costo di realizzazione, il tempo e le risorse necessarie ad eseguire i test possono essere molto elevati.

7.7 Oracolo ed Output Attesi

Definizione (Oracolo)**.** Un **oracolo** è uno strumento utilizzato per generare i risultati attesi di una batteria di test. La sua importanza è che fornisce un punto di riferimento per la correttezza del sistema durante il testing.

L'*output atteso* può essere trovato in diversi modi:

- *Risultati ricavati dalle specifiche*.
- *Inversione delle funzioni*: quando la funzione inversa è più facile. In questo modo dall'output si calcola l'input.
- *Versioni precedenti del codice*: per le funzionalità non modificate.
- *Versioni multiple indipendenti*: programmi preesistenti, oppure semplificazioni di algortimi, che sono poco efficienti ma corrette.
- *Semplificazione dei dati d'ingresso*: si provano le funzionalità su dati semplici, oppure i risultati sono noti tramite l'utilizzo di altri mezzi, o addirittura ci si mette nell'ipotesi di avere un comportamento costante.
- *Semplificazione dei risultati*: ci si basa su risultati plausibili, ma si applicano dei vincoli fra gli input e gli output, e delle invarianti sugli output.

7.8 Test di Sistema

- Facility Test: mira a controllare che ogni funzionalità stabilita nei requisiti sia stata realizzata correttamente.
- Security Test: si cerca di accedere a dati e funzionalità che dovrebbero essere riservate.
- Usability Test: si valuta la facilità d'uso del software da parte dell'utente finale. Ci si basa sulla documentazione e sul livello di competenza dell'utenza.
- Performance Test: si valuta l'efficienza di un sistema rispetto ai tempi di elaborazione e risposta. Si sottopone il sistema a diversi livelli di carico.
- Volume Test: qui si sottopone il sistema al carico di lavoro massimo stabilito nei requisiti. Si cercano malfunzionamenti che non si presentano normalmente.
- Stress Test: si sottopone il sistema a carichi di lavoro superiori a quelli previsti nei requisiti, oppure si porta in condizioni non previste, ad esempio sottraendogli risorse di memoria e calcolo. Lo scopo è quello di controllare la capacità di *recovery* del sistema dopo un fallimento.
- Storage Use Test: è un controllo mirato alla richiesta delle risorse, in particolare la memoria, ed ha implicazioni sulla definizione dei requisiti minimi del sistema per poter installare il software.

- Configuration Test: si prova il sistema su sistemi operativi diversi e con differenti hardware installati.
- Compatibility Test: si valuta la compatibilità del software con altri, anche con una sua versione precendente che deve rimpiazzare.