Riassunti di

INGEGNERIA DEL SOFTWARE

DI D'ANGELO CARMINE

Indice

1 Introduzione 6
1.1 Definizioni 6
1.1.1 Cosa è il software 6
1.1.2 Prodotti generici vs Prodotti specifici 6
1.1.3 Programma vs prodotto 6
1.1.4 Costi 6
1.1.5 Manutenzione 6
1.1.6 Software engineering 6
1.1.7 Differenza tra software engineering e computer science 6
1.1.8 Natura del software 6
1.2 Fondamenti dell'ingegneria del software 7
1.2.1 Principi 7
1.2.2 Metodi e metodologie 7
1.2.3 Strumenti, procedure e paradigmi 7
1.3 Qualità del software 8
2 Cicli di vita del software 8
2.1 Modello di ciclo di vita del software (CVS) 8
2.2 Modello a cascata 9
2.2.1 Organizzazione sequenziale: *fasi alte* del processo 9
2.2.2 Organizzazione sequenziale: *fasi basse* del processo 9
2.2.3 Pro e contro del modello a cascata 9
2.3 Modello (V&V) e Retroazione (Feedback) 10
2.4 Modello a V 10
2.5 Modello trasformazionale 11
2.6 Modello di sviluppo basato sul riuso 11
2.6.1 Full Reuse Model 11
2.7 Modello evolutivo (a prototipazione) 11
2.8 Modello incrementale 12
2.9 Modello a spirale 12
2.10 Modelli e valutazione dei rischi 13
2.12 Scopo dell'ingegneria del software 14
3 Project management 14
3.1 Team di sviluppo 14
3.2 Stesura del piano del progetto 14
3.3 Grafo delle attività (PERT) 15

3.4 Management dei rischi 15
3.4.1 Identificazione 15
3.4.2 Analisi dei rischi 16
3.4.3 Pianificazione dei rischi 16
3.4.4 Monitoraggio dei rischi 16
4 UML (Unified Modeling Language) 16
4.1 Diagrammi dei casi d'uso (use case diagrams) 16
4.2 Diagrammi di classi (class diagrams) 17
4.2.1 Classi in UML 17
4.2.2 Attributi 17
4.2.2.1 Attributi derivati 17
4.2.2.2 Attributi ridondanti 18
4.2.3 Operazioni (Metodi/Funzioni) 18
4.2.4 Responsibility 18
4.2.5 Proprietà attributi e operazioni 18
4.2.5.1 Visibilità 18
4.2.5.2 Molteplicità 19
4.2.5.3 Specifica attributi 19
4.2.5.3 Specifica operazioni 19
4.2.6 Legami e associazioni 19
4.2.6.1 Molteplicità delle associazioni 20
4.2.7 Ruoli 20
4.2.7.1 Interface specifier 20
4.2.8 Associazione 20
4.2.8.1 Classi associative 21
4.2.9 Aggregazione 21
4.2.10 Composizione 21
4.2.11 Generalizzazione (o ereditarietà) 21
4.2.11.1 Ereditarietà multipla 21
4.2.12 Classi astratte 22
4.2.13 Root, leef elementi polimorfici 22
4.2.14 Gerarchia di classi 22
4.2.15 Dependency 22
4.2.16 Realization 23
4.2.16 Interfacce (definizione) 23
4.2.17 Modelli dinamici con UML 23
4.3 Interaction diagram 23
4.3.1 Iterazioni e messaggi 24
4.4 Diagrammi sequenziali (sequence diagrams) 24
4.4.1 Iterazioni (ricorrenze) 25
4.4.2 Cicli e condizioni 25
4.4.3 Auto-chiamata 26
4.4.4 Esprimere vincoli sul tempo di risposta 26
4.4.5 Messaggi e azioni 26
4.5 Diagramma a stati (state chart) 26
4.5.1 Elementi grafici dello state diagram 27
4.5.2 Caratteristiche dello stato 28
4.5.3 Caratteristiche delle transizioni 28

4.5.4 Attributi (opzionali: evento [condizione]/azione) ... 28
4.5.5 Stato composto ... 28
4.6 Diagrammi delle attività (activity diagrams) ... 29
4.6.1 Elementi grafici ... 30
4.6.2 Swimlanes ... 30
4.7 Raggruppamento (packages) ... 31
5 Raccolta dei requisiti (requirements elicitation) ... 31
5.1 Classificazione dei requisiti ... 32
5.1.1 Requisiti funzionali ... 32
5.1.2 Requisiti non funzionali ... 32
5.2 Validazione dei requisiti ... 32
5.2.1 Tracciabilità ... 33
5.2.2 Greenfield engineering, re-engineering, interface engineering ... 33
5.3 Attività della raccolta dei requisiti ... 33
5.3.1 Identificare gli attori ... 34
5.3.2 Identificare gli scenari ... 34
5.3.3 Identificare i casi d'uso ... 34
5.3.4 Raffinare i casi d'uso ... 35
5.3.5 Identificare le relazioni tra attori e casi d'uso ... 35
5.3.6 Identificare gli oggetti partecipanti ... 35
5.3.7 Identificare i requisiti non funzionali ... 36
5.4 Gestire la raccolta dei requisiti ... 37
6 Analisi dei requisiti ... 38
6.1 Concetti dell'analisi ... 38
6.1.1 Il modello ad oggetti ... 38
6.1.2 Il modello dinamico ... 38
6.1.3 Generalizzazione e specializzazione ... 39
6.1.4 Entity, Boundary (oggetti frontiera) e Control object ... 39
6.1.5 Ambiguità ... 39
6.2 Attività dell'analisi (trasformare un caso d'uso in oggetti) ... 39
6.2.1 Notazioni ... 40
6.2.2 Identificare gli oggetti entity ... 40
6.2.3 Identificare gli oggetti boundary ... 41
6.2.4 Identificare gli oggetti controllo ... 41
6.2.5 Mappare casi d'uso in oggetti con sequence diagrams ... 42
6.2.6 Identificare le associazioni ... 42
6.2.7 Identificare le aggregazioni ... 43
6.2.8 Identificare gli attributi ... 43
6.2.9 Modellare il comportamento e gli stati di ogni oggetto ... 43
6.2.10 Rivedere il modello dell'analisi ... 43
7 System design ... 44
7.1 Scopi criteri e architetture ... 44
7.2 Identificare gli obiettivi di design ... 44
7.3 Decomposizione del sistema in sottosistemi ... 45
7.3.1 Accoppiamento (coupling) ... 45
7.3.2 Coesione e Coupling ... 46
7.3.3 Divisione del sistema con i layer ... 46
7.3.4 Divisione del sistema con le partizioni ... 46

7.3.5 Sottosistemi e classi 47
7.3.6 Servizi e interfacce di sottosistemi 47
7.3.7 Macchina virtuale (Dijkstra) 47
7.3.7.1 Architettura chiusa (opaque layering) 47
7.3.7.2 Architettura aperta (Trasparent layering) 48
7.3.7.3 Vantaggi e svantaggi delle architetture chiuse 48
7.3.8 Basic Layer pattern 48
7.4 Architetture software 48
7.4.1 Repository 48
7.4.1.1 Vantaggi e svantaggi 48
7.4.2 Model/View/Control (MVC) 49
7.4.3 Client-Server 49
7.4.4 Peer-To-Peer 49
7.4.5 Three-Tier 49
7.4.6 Architetture a flusso di dati (pipeline) 50
7.4.7 Considerazioni finali 50
7.4.8 Euristiche per scegliere le componenti 50
7.5 Descrizione delle attività del System Design 50
7.5.1 UML Deployment Diagram 51
7.5.2 Mappare i sottosistemi su piattaforme e processori 51
7.5.3 Identificare e memorizzare i dati persistenti 52
7.5.4 Stabilire i controlli di accesso 53
7.5.5 Progettare il flusso di controllo globale 53
7.5.6 Identificare le condizioni limite 54
7.5.7 Rivedere il modello del system design 54
7.5.8 Gestione del system design 54
8 Object design 56
8.1 Concetti di riuso 56
8.1.1 Oggetti di applicazione e oggetti di soluzione 56
8.1.2 Ereditarietà di specifica ed ereditarietà di implementazione 56
8.1.3 Delegazione 57
8.1.4 Il principio di sostituzione di Liskov 57
8.1.5 Delegazione ed ereditarietà nei design pattern 57
8.2 Attività del riuso (selezionare i design pattern e le componenti) 57
8.2.1 Classificazione dei design pattern 58
8.2.2 Pattern Strutturali 58
8.2.2.1 Composite pattern 58
8.2.2.2 Facade Pattern 59
8.2.2.3 Adapter Pattern 60
8.2.2.4 Bridge Pattern 61
8.2.2.5 Proxy Pattern 62
8.2.3 Pattern comportamentali 62
8.2.3.1 Observer Pattern 62
8.2.4 Pattern Creazionali 63
8.2.4.1 Abstract Factory 63
8.2.5 Suggerimenti e Note su utilizzo design pattern 64
8.2.6 MVC 64
8.2.7 Identificare e migliorare i framework di applicazione 65

8.3 Valutare il riuso 65
8.4 Specificare le interfacce dei servizi 66
8.4.1 Tipologie di sviluppatori 66
8.4.2 Specificare le firme 66
8.4.3 Aggiungere contratti (precondizioni, postcondizioni, invarianti) 66
8.5 Documentare l'Object Design 68
9 Mappare il modello nel codice 68
9.1 Concetti di mapping 68
9.1.1 Trasformazioni 68
9.2 Attività del mapping 72
9.2.1 Ottimizzare il modello di Object Design 72
9.2.2 Mappare associazioni in collezioni e riferimenti 73
9.2.3 Mappare i contratti in eccezioni 74
9.2.4 Mappare il modello di classi in uno schema di memorizzazione 75
9.3 Responsabilità 76
9.4 Gestire l'implementazione 76
10 Testing 77
10.1 Overview 77
10.2 Concetti di testing 78
10.2.1 Test case 79
10.2.2 Correzioni 80
10.3 Attività di testing 80
10.3.1 Component inspection 81
10.3.2 *Managing testing* 81
10.3.3 Documentazione di testing 81
10.3.4 Usability testing 82
10.3.4 Unit testing 82
10.3.5 Integration testing 86
10.3.6 System testing 87
11 SCRUM 88
11 Ruoli 89
11.1.1 Il Team Scrum 89
11.1.2 Il Product Owner 89
11.1.3 Team di sviluppo 89
11.1.4 Scrum Master 90
11.1.5 Ruoli ausiliari 90
11.2 Cerimonie 90
11.2.1 Sprint 90
11.2.2 Sprint planning meeting 91
11.2.3 Daily Scrum meeting 91
11.2.4 Sprint Review 91
11.2.5 Sprint Retrospective 92
11.3 Artefatti 92
11.3.1 Product backlog 92
11.3.2 Sprint backlog 92
11.3.3 Burn down charts 92

1 Introduzione

1.1 **Definizioni**

1.1.1 Cosa è il software
Il software non è solo un insieme di linee di codice ma comprende anche tutta la documentazione, i case test e i manuali.

1.1.2 Prodotti generici vs Prodotti specifici
I prodotti generici sono dei software prodotti da aziende e utilizzati da un ampio bacino di utenza diversificato. I prodotti specifici sono software sviluppati ad hoc per uno specifico cliente, visionato dallo stesso. Il costo dei prodotti generici è maggiore rispetto a quello dei prodotti specifici.

1.1.3 Programma vs prodotto
Un programma è una semplice applicazione sviluppata, testata e usata dallo stesso sviluppatore. Il prodotto software viene sviluppato per terzi, è un software industriale che ha un costo di circa 10 volte superiore ad un normale programma. e deve essere corredato di documentazione, manuali e case test.

1.1.4 Costi
Il costo del software viene calcolato in base alle ore di lavoro, il software e hardware utilizzato e altre risorse di supporto. Il costo della manutenzione è superiore a quello di produzione.

1.1.5 Manutenzione
Il software dopo il suo rilascio, specie se lo stesso ha una vita lunga, ha bisogno di alcune fasi di manutenzione. Per manutenzione intendiamo sia la correzione di eventuali bug, sia l'estensione/modifica di alcune caratteristiche. Il costo della manutenzione è più elevato rispetto a quello di produzione.
Abbiamo tre tipi di manutenzione:

- ***Correttiva:*** eliminazione di difetti
- ***Effettiva:*** adattamento a nuovi ambienti.
- ***Evolutiva:*** miglioramenti e nuove funzionalità

1.1.6 Software engineering
Disciplina che cerca di fornire le regole per il processo di produzione del software. Lo scopo dell'ingegneria del software è di pianificare, progettare, sviluppare il software tramite lavoro di gruppo. È possibile che vengono rilasciate più versioni del prodotto software. Tale attività ha senso per progetti di grosse dimensioni e di notevole complessità ove si rende necessaria la pianificazione.

1.1.7 Differenza tra software engineering e computer science
Mentre la computer science si occupa di creare e ottimizzare algoritmi e si occupa degli aspetti teorici dell'informatica, il software engineering si occupa della pianificazione e della progettazione con la finalità di ottenere un prodotto software.

1.1.8 Natura del software

- **Intangibile:** infatti un prodotto software che creiamo non può essere toccato o visto ameno che di creazione di interfacci grafiche.

- **Malleabile:** un software può essere modificato, ma potrebbero derivarne gravi problemi.
- **Ad alta intensità di lavoro umano:** la maggior parte del lavoro nella creazione del software è ancora fatta dall'uomo.
- **Spesso costruito ad hoc invece che assemblato:** si creano ogni volta nuove parti di codice invece di riutilizzare quelle già create.

1.2 Fondamenti dell'ingegneria del software

L'ingegneria del software si occupa principalmente di tre aspetti fondamentali: i principi, i metodi, le metodologie e gli strumenti.

1.2.1 Principi

1. Rigore e formalità
2. Affrontare separatamente le varie problematiche dell'attività
3. Modularità (divide-et-impera)
4. Anticipazione del cambiamento (scalabilità)
5. Generalità (tentare di risolvere il problema nel suo caso generale)
6. Incrementalità (lavorare a fasi di sviluppo, ognuna delle quali viene terminata con il rilascio di una release, anche se piccola).

1.2.2 Metodi e metodologie

Un metodo è una particolare procedimento per risolvere problemi specifici, mentre la metodologia à un'insieme di principi e metodi che serve per garantire la correttezza e l'efficacia della soluzione al problema.

1.2.3 Strumenti, procedure e paradigmi

Uno strumento è un artefatto che viene usato per fare qualcosa in modo migliore. Una procedura è una combinazione di strumenti e metodi finalizzati alla realizzazione di un prodotto.
Processo: una particolare metodologia operativa che nella tecnica definisce le singole operazioni fondamentali per ottenere un prodotto industriale.
Processo software: un metodo per sviluppare del software, cioè un insieme organizzativo di attività che sovrintendono alla costruzione del prodotto da parte del team di sviluppo usando metodi, metodologie e strumenti. È suddiviso in varie fasi secondo uno schema di riferimento (ciclo di vita del software). È descritto da un modello (informale, semi-formale, formale).
Processo creazione software: il processo attraverso il quale le esigenze degli utenti sono tradotte in un prodotto software. Il processo prevede la traduzione delle esigenze degli utenti in requisiti software, la trasformazione dei requisiti software in progettazione, l'implementazione della progettazione in codice, la verifica del codice e, a volte, l'installazione e la verifica del software per l'uso operativo.

- **Upper-CASE:** Strumenti che supportano le attività delle fasi di analisi e specifica dei requisiti e progettazione di un processo software. Includono editor grafici per sviluppare modelli di sistema, dizionari dei dati per gestire entità del progetto.
- **Lower-CASE:** Strumenti che supportano le attività delle fasi finali del processo, come programming, testing e debugging. Includono generatori di graphical UI per la costruzione d'interfacce utente, debuggers per supportare la ricerca di program fault, traduttori automatici per generare nuove versioni di un programma.

1.3 **Qualità del software**

La qualità può essere riferita sia al prodotto che al processo applicato per ottenere il risultato finale. Un particolare modello di qualità (modello di McCall) dice che la qualità si basa sui seguenti tre aspetti principali:

1. *Revisione*: manutenibilità, flessibilità e verificabilità (deve rispettare i requisiti del cliente)
2. *Transizione*: portabilità, riusabilità, interoperabilità (capacità del sistema di interagire con altri sistemi esistenti)
3. *Operatività*: correttezza (conformità dello stesso rispetto ai requisiti), affidabilità, efficienza (tempo di risposta o uso della memoria), usabilità, integrità (capacità di sopportare attacchi alla sicurezza).

2 Cicli di vita del software

Un *processo software* è un insieme organizzato di attività finalizzate ad ottenere il prodotto da parte di un team di sviluppo utilizzando metodo, tecniche, metodologie e strumenti.
Il processo viene suddiviso in fasi in base ad uno schema di riferimento (ciclo di vita del software).

2.1 **Modello di ciclo di vita del software (CVS)**

È una caratterizzazione descrittiva o prescrittivi di come un sistema software viene sviluppato.

I modelli CVS devono avere le seguenti caratteristiche:

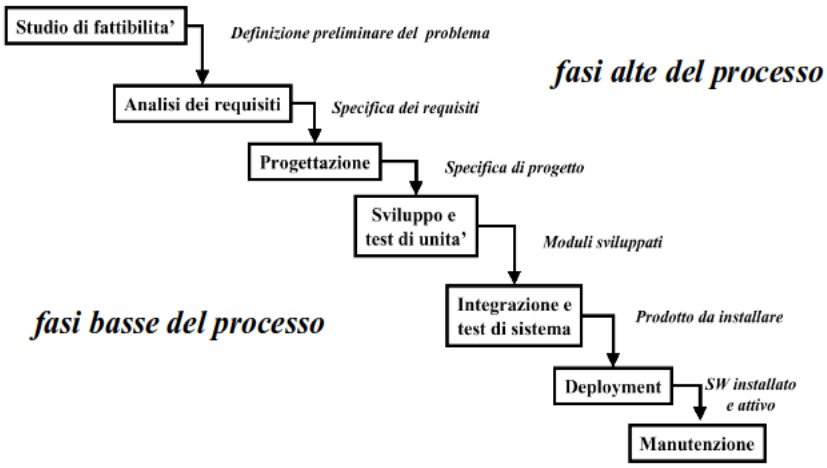
1. Descrizione dell'organizzazione del lavoro nella software-house.
2. Linee guida per pianificare, dimensionare il personale, assegnare budget, schedulare e gestire.
3. Definizione e scrittura dei manuali d'uso e diagrammi vari.
4. Determinazione e classificazione dei metodi e strumenti più adatti alle attività da svolgere.

Le fasi principali di un qualsiasi CVS sono le seguenti:

1. **Definizione** (si occupa del cosa).
   Determinazione dei requisiti, informazioni da elaborare, comportamento del sistema, criteri di validazione, vincoli progettuali.
2. **Sviluppo** (si occupa del come)
   Definizione del progetto, dell'architettura software, traduzione del progetto nel linguaggio di programmazione, collaudi.
3. **Manutenzione** (si occupa delle modifiche)
   Miglioramenti, correzioni, prevenzione, adattamenti.

2.2 **Modello a cascata**

Definisce che il processo segua una progressione sequenziale di fasi senza ricicli, al fine di controllare meglio tempi e costi. Inoltre definisce e separa le varie fasi e attività del processo in modo da minimizzare la sovrapposizione tra di esse. Ad ogni fase viene prodotto un semilavorato con la relativa documentazione e lo stesso viene passato alla fase successiva (*milestore*). I prodotti ottenuti da una fase non possono essere modificati durante il processo di elaborazione delle fasi successive.



2.2.1 Organizzazione sequenziale: *fasi alte* del processo

- **Studio di fattibilità:** Effettua una valutazione preliminare dei costi e dei requisiti in collaborazione con il committente. L'obiettivo è quello di decidere la fattibilità del progetto, valutarne i costi, i tempi necessari e le modalità di sviluppo.
  Output: documento di fattibilità.
- **Analisi e specifica dei requisiti:** Vengono analizzate le necessità dell'utente e del dominio d'applicazione del problema. *Output*: documento di specifica dei requisiti.
- **Progettazione**: Viene definita la struttura del software e il sistema viene scomposto in componenti e moduli. *Output*: definizione dei linguaggi e formalismi.

2.2.2 Organizzazione sequenziale: *fasi basse* del processo

- **Programmazione e test di unità**: Ogni modulo viene codificato nel linguaggio e testato separatamente dagli altri.
- **Integrazione e test di sistema:** I moduli vengono integrati tra loro e vengono testate le loro interazioni. Viene rilasciata una *beta release* (release esterna) oppure una *alpha release* (release interna) per testare al meglio il sistema.
- **Deployment:** Rilascio del prodotto al cliente.
- **Manutenzione:** Gestione dell'evoluzione del software.

2.2.3 Pro e contro del modello a cascata

Pro

- Facile da comprendere e applicare

Contro

- L'interazione con il cliente avviene solo all'inizio e alla fine del ciclo.
- I requisiti dell'utente vengono scoperti solo alla fine.
- Se il prodotto non ha soddisfatto tutti i requisiti, alla fine del ciclo, è necessario iniziare daccapo tutto il processo.

Rischi:

- Soltanto alla fine del progetto il nostro cliente può giudicare il progetto e quindi decidere dei

cambiamenti.

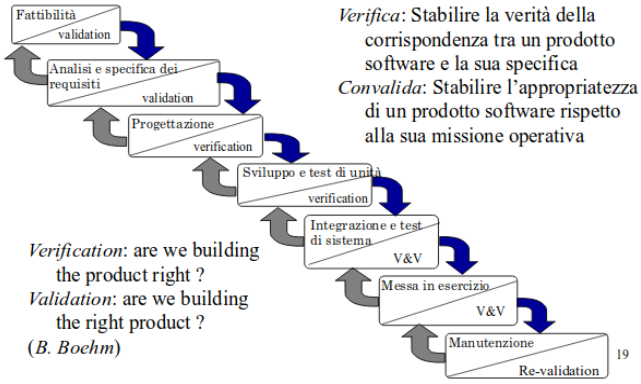
- Fino al termine del progetto non c'è comunicazione con il cliente.

2.3 **Modello (V&V) e Retroazione (Feedback)**

Uguale al modello a cascata, vengono applicati i ricicli, ovvero al completamento di ogni fase viene fatta una verifica ed è possibile tornare alla fase precedente nel caso la stessa non verifica le aspettative.

Verification: stiamo costruendo il prodotto nel modo corretto? Rispetto alle specifiche definite precedentemente?

Validation: stiamo costruendo il giusto prodotto? Cioè giusto per il nostro cliente?

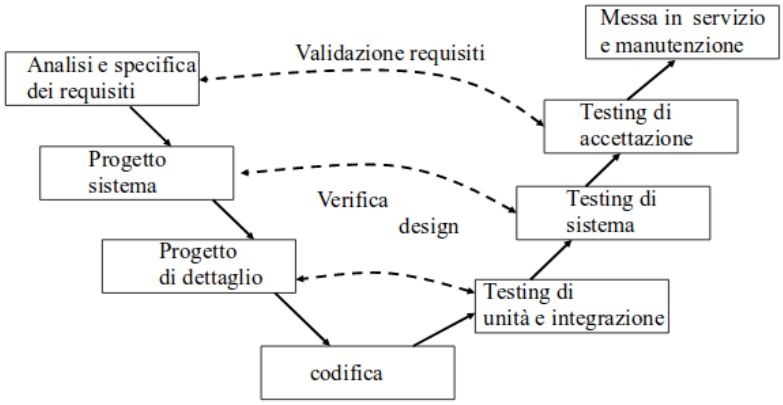


2.4 **Modello a V**

È come il modello a cascata, la sua caratteristica è che mette in correlazione alcune frasi basse con le fasi precedenti:

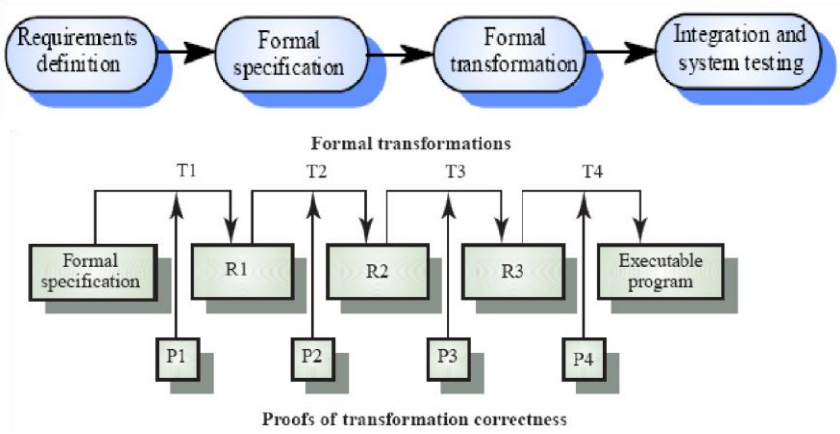
- quando si fa testing di unità e integrazione si avrà bisogno del progetto di dettaglio;
- quando si verifica la conformità del testing di sistema si ha bisogno del progetto di sistema;
- Infine per la validazione dei requisiti del testing di accettazione si ha bisogno di Analisi e specifica dei requisiti.

Visto che le attività di destra sono collegate con quelle di sinistra se si trova un errore in una fase di destra si riesegue il pezzo della V collegato.



2.5 **Modello trasformazionale**

Basato su un modello matematico che viene trasformato da una rappresentazione formale ad un'altra. Questo modello comporta problemi nel personale in quanto non è facile trovare persone con le conoscenze giuste per poterlo implementare.



Tale modello può essere utilizzato con altri modelli per migliorare il processo di sviluppo di sistemi critici, soprattutto quelli in cui la sicurezza è molto importante.

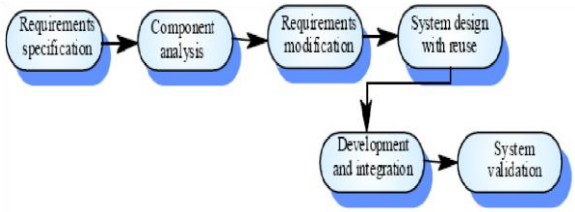
Richiede una conoscenza molto approfondita delle proprie tecniche e molta esperienza nell'applicazione della tecnica. Rende difficile specificare formalmente aspetti come le interfacce.

Si hanno pochi esempi di applicazione per sistemi complessi.

2.6 **Modello di sviluppo basato sul riuso**

È previsto un repository dove vengono depositate le componenti sviluppate durante le fasi del ciclo di vita. Le componenti vengono prese dal repository e riutilizzate quando necessario. Questo modello è particolarmente usato per sviluppare software in linguaggi Object Oriented.

2.6.1 **Full Reuse Model**



Prevede repository di componenti riusabili a diversi livelli di astrazione, prodotti durante le diverse fasi del ciclo di vita (specifiche, progetti, codice, test case, ecc.).

Durante lo sviluppo di un nuovo sistema avvengono:

- il riuso di componenti esistenti.
- il popolamento delle repository con nuove componenti.

2.7 **Modello evolutivo (a prototipazione)**

In questo modello il cliente è parte integrate del processo di sviluppo del prodotto. Il modello evolutivo si basa su due tipologie di sviluppo basate sui prototipi:

- **Prototipazione evolutiva (esplorativa)**
  Si inizia a sviluppare le parti del sistema che sono già ben specificate aggiungendo nuove caratteristiche secondo le necessità fornite dal cliente man mano. Lo sviluppo dovrebbe avviarsi con la parte dei requisiti meglio compresa. I pro sono che si può usare per sistemi di piccole dimensioni, e per sistemi con vita limitata. I contro sono che se si effettuano troppe modifiche non si avanza mai nella fase successiva, se vengono apportate molte modifiche il progetto potrebbe non avere più una buona struttura.
- **Prototipo usa e getta (throw-away)**

Lo scopo di questo tipo di prototipazione è quello di identificare meglio le specifiche richieste dall'utente sviluppando dei prototipi che sono funzionanti. Non appena il prototipo è stato verificato da parte del cliente o da parte degli sviluppatori può essere buttato via.

Abbiamo poi altri due tipi di prototipazione:

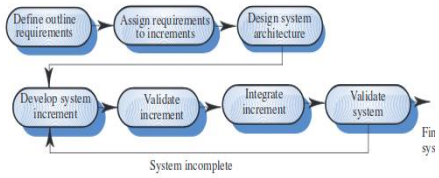
- **Mock-ups:** produzione completa dell'interfaccia utente. Consente di definire con completezza e senza ambiguità i requisiti (si può, già in questa fase, definire il manuale di utente).
- **Breadboards:** implementazione di sottoinsieme di funzionalità critiche del SS, non nel senso della fattibilità ma in quello dei vincoli pesanti che sono posti nel funzionamento del SS (carichi elevati, tempo di risposta, …), senza le interfacce utente. Produce feedbacks su come implementare la funzionalità (in pratica si cerca di conscere prima di garantire).

Questo modello si può applicare anche a quello a cascata, ma porta diversi problemi come la perdità della visibilità del processo, spesso il prodotto finito è scarsamente strutturato, si perde la visibilità del processo da parte del management. È applicabile su sistemi interattivi di taglia medio-piccola e sistemi con ciclo di vita medio-piccola.

2.8 **Modello incrementale**

Utilizzato per la progettazione di grandi software che richiedono tempi ristretti. Vengono rilasciate delle release funzionanti (*deliverables*) anche se non soddisfano pienamente i requisiti del cliente. Vi sono due tipi di modelli incrementali:

- **Modello ad implementazione incrementale**
  Le fasi alte del modello a cascata vengono realizzate e portate a termine, il software viene finito, testato e rilasciato ma non soddisfa tutte le aspettative richieste dall'utente. Le funzionalità non incluse vengono comunque implementate e aggiunte in tempi diversi. Con questo tipo di modello diventa fondamentale la parte di integrazione tra sottosistemi.



- **Modello a sviluppo e consegna incrementale**
  È un particolare modello a cascata in cui ad ogni fase viene applicato il modello ad implementazione incrementale e successivamente le singole fasi vengono sviluppate e integrate con il sistema esistente. Il sistema viene sviluppato seguendo le normali fasi e ad ogni fase l'output viene consegnato al cliente.

Vantaggio

Possibilità di anticipare da subito delle funzionalità
al committente:

- Ciascun incremento corrisponde al rilascio di una parte delle funzionalità
- I requisiti a più alta priorità per il committente vengono rilasciati per prima
- Minore rischio di un completo fallimento del progetto

Testing più esaustivo:

- I rilasci iniziali agiscono come prototipi e consentono di individuare i requisiti per i successivi incrementi
- I servizi a più alta priorità sono anche quelli che vengono maggiormente testati

2.9 **Modello a spirale**

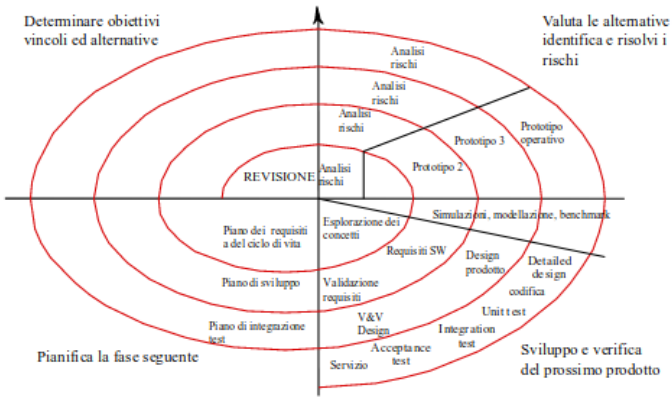
Il processo è visto come una spirale dove ogni ciclo viene diviso in quattro fasi:

- Determinazione degli *obiettivi* della fase
- Identificazione e riduzione dei *rischi*, valutazione delle alternative

- *Sviluppo e verifica* della fase
- *Pianificazione* della fase successiva

Una caratteristica importante di questo modello è il fatto che i rischi vengono presi seriamente in considerazione e che ogni fine ciclo produce una *deliverables*. In un certo senso può essere visto come un modello a cascata iterato più volte.

- **Vantaggi:** Rende esplicita la gestione dei rischi, focalizza l'attenzione sul riuso, determina errori in fasi iniziali, aiuta a considerare gli aspetti della qualità e integra sviluppo e manutenzione.
- **Svantaggi:** Richiede un aumento nei tempi di sviluppo, delle persone con capacità di identificare i rischi, una gestione maggiore del team di sviluppo e quindi anche un costo maggiore.



2.10 **Modelli e valutazione dei rischi**

Il *modello a cascata* genera alti rischi su un progetto mai sviluppato (greenfield engineering) e bassi rischi nello sviluppo di applicazioni familiari con tecnologie già note. Nel *modello a prototipazione* si hanno bassi rischi nelle nuove applicazioni, alti rischi per la mancanza di un processo definito e visibile. Nel *modello trasformazionale* si hanno alti rischi a causa delle tecnologie coinvolte e delle professionalità richieste.

2.11 **Dodici regole dell'extreme programming:**

1. Progettare con il cliente;
2. Test funzionali e unitari;
3. Refactoring (riscrivere il codice senza alterarne le funzionalità esterne);
4. Progettare al minimo (solo l'indispensabile);
5. Descrivere il sistema con una metafora, anche per la descrizione formale (cioè qualcosa che ricorda una struttura chiara del sistema);
6. Proprietà del codice collettiva (contribuisce alla stesura chiunque sia coinvolto nel progetto);
7. Scegliere ed utilizzare un preciso standard di scrittura del codice;
8. Integrare continuamente i cambiamenti al codice;
9. Il cliente deve essere presente e disponibile a verificare (sono consigliate riunioni settimanali, costanti feedback);
10. Open Workspace;
11. 40 ore di lavoro settimanali;
12. Pair Programming (due programmatori lavorano insieme su un solo computer).

Gli obiettivi dell'extreme programming sono l'aiutare la produttività e qualità dei prodotti software.

2.12 **Scopo dell'ingegneria del software**

- Migliorare la qualità del prodotto e del processo software
- Portabilità su sistemi legacy
- Eterogeneità
- Velocità di sviluppo

3 Project management

Il project management racchiude le attività necessarie per assicurare che un progetto software venga sviluppato rispettando le scadenze e gli standard.

Le entità fisiche che prendono parte al project management sono:

1. **Buiness manager**: definiscono i termini economici del progetto
2. **Project manager:** panificano, motivano, organizzano e controllano lo sviluppo, stimano il costo del progetto, selezionano il team di sviluppo, stendono i rapporti e le presentazioni.
3. **Practitioners**: hanno competenze tecniche per realizzare il sistema
4. **Customers (clienti)**: specificano i requisiti del software da sviluppare
5. **End users (utenti)**: gli utenti che interagiscono con il sistema

3.1 **Team di sviluppo**

Esistono vari tipi di team di sviluppo qui di seguito indicati:

- **Democratico decentralizzato:** Assenza di un leader permanente (possono esistere dei leader a rotazione), consenso di gruppo, organizzazione orizzontale.
  Vantaggi: individuazione degli errori, adatto a problemi difficili
  Svantaggi: difficile da implementare, non è scalabile.
- **Controllato decentralizzato:** Vi è un leader che controlla il lavoro e assegna i problemi ai gruppi a lui sottesi. I sotto gruppi hanno un leader e sono composti di 2 a 5 persone. I leader dei sottogruppi possono comunicare tra loro come anche i membri dei sottogruppi possono comunicare in maniere orizzontale.
- **Controllato centralizzato:** Vi è un leader che decide sulle soluzioni e l'organizzazione dei gruppi. Ogni gruppo ha un proprio leader che assegna e controlla il lavoro dei componenti. I leader dei gruppi non comunicano tra loro ma possono comunicare solo con il loro capo. I membri dei gruppi non comunicano tra loro ma solo con il capo gruppo.

3.2 **Stesura del piano del progetto**

- **Introduzione:** Viene definita una descrizione di massima del progetto, gli elementi che vengono consegnati con le rispettive date di consegne e vengono pianificati eventuali cambiamenti.
- **Organizzazione del progetto:** Vengono definite le relazioni tra le varie fasi del progetto, la sua struttura organizzativa, le interazioni con entità esterne, le responsabilità di progetto (le principali funzioni e chi sono i responsabili).
- **Processi gestionali:** Si definiscono gli obiettivi e le priorità, le assunzioni, le dipendenze, i vincoli, i rischi con i relativi meccanismi di monitoraggio, pianificazione dello staff.
- **Processi tecnici:** Vanno specificati i sistemi di calcolo, i metodi di sviluppo, la struttura del team, il piano di documentazione del software e viene pianificata la gestione della qualità.
- **Pianificazione del lavoro, delle risorse umane e del budget:** Il progetto viene diviso in task (attività) e a ciascuno assegnata una priorità, le dipendenze, le risorse necessarie e i costi. Le attività devono essere organizzate in modo da produrre risultati valutabili dal management. I

risultati possono essere *milestone* o *deliverables*; il primo rappresenta il punto finale di un'attività di processo, il secondo è un risultato fornito al cliente.
Ogni *task* è un'unità atomica definita specificando: nome e descrizione del lavoro da svolgere, precondizioni per poter avviare il lavoro, risultato atteso, rischi. I vari task vanno organizzati in modo da ottimizzare la concorrenza e minimizzare la forza lavoro. Lo scopo è quello di minimizzare la dipendenza tra le mansioni per evitare ritardi dovuti al completamento di altre attività.

Le attività del progetto vengono divise in task che sono caratterizzati dai tempi di inizio e fine, una descrizione, le precondizioni di partenza, i rischi possibili ed i risultati attesi.

3.3 **Grafo delle attività (PERT)**

Mostra la suddivisione del lavoro in attività evidenziando le dipendenze e il cammino.

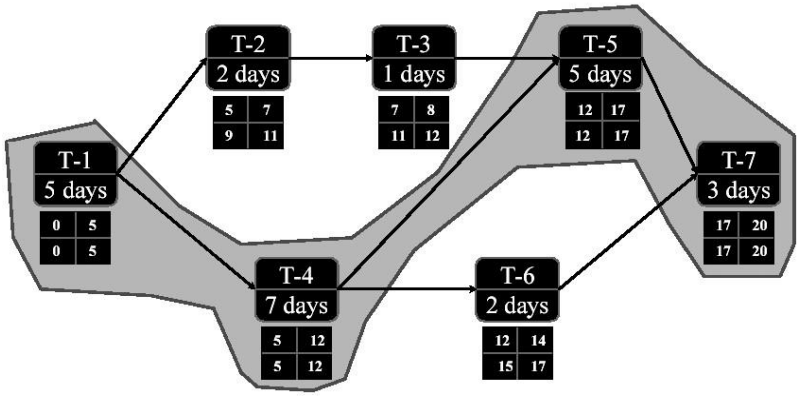


Figura 1 Cammino critico del grafo della attività

ES[1]	**EF**[2]
LS[3]	**LF**[4]

Tabella 1 Tempi relativi alle attività

3.4 **Management dei rischi**

Il management dei rischi identifica i rischi possibili e cerca di pianificare per minimizzare il loro effetto sul progetto, pianifica i rischi e li monitorizza.

3.4.1 Identificazione

I rischi da identificare sono di vari tipi tra cui: rischi tecnologici, rischi delle risorse umane, rischi organizzativi, rischi nei tools, rischi relativi ai requisiti, rischi di stima/sottostima. Le tipologie di rischi sono le seguenti:

- **Tecnologici:** Alcune tecnologie di supporto (database, componenti esterne) non sono abbastanza validi come ci aspettavamo.

[1] **ES** (earliest start time) tempo minimo di inizio dell'attività a partire dal minimo tempo in cui terminano le attività precedenti (ovvero il valore massimo degli EF precedenti).
[2] **EF** (earliest finish time) dato ES è il minimo tempo in cui l'attività può finire.
[3] **LS** (latest start time) dato LF e la durata del task quale è il giorno massimo in cui deve iniziare per evitare ritardo nei task che dipendono da lui.
[4] **LF** (latest finish time) il giorno massimo in cui quel task può finire senza portare ritardi ai successivi (ovvero il valore minimo tra gli LS dei successivi)

- **Risorse umane:**Non è possibile reclutare staff con la competenza richiesta oppure non è possibile fare formazione allo staff.
- **Organizzativi:** Cambi nella struttura organizzativa possono causare ritardi o nello sviluppo del progetto.
- **Strumenti:** Ad esempio il codice/documentazione prodotto con un determinato strumento non è abbastanza efficiente.
- **Requisiti:** Cambiamenti nei requisiti richiedono una revisione del progetto già sviluppato.
- **Stima:** Il tempo richiesto, la dimensione del progetto sono stati sottostimati.

3.4.2 Analisi dei rischi
Ad ogni rischio va assegnata una probabilità che esso si verifichi e vanno valutati gli effetti dello stesso che possono essere: catastrofici, seri, tollerabili, insignificanti.

3.4.3 Pianificazione dei rischi
Viene considerato ciascun rischio e viene sviluppata una strategia per risolverlo. Le strategie che possiamo prendere possono essere: - Evitare i rischi con una prevenzione
- Minimizzare i rischi
- Gestire i rischi con un piano di contingenza per evitarli

3.4.4 Monitoraggio dei rischi
Ogni rischio viene regolarmente valutato e viene verificato se è diventato meno o più probabile, inoltre i suoi aspetti vanno discussi con il management per valutare meglio i provvedimenti da adottare.

4 UML (Unified Modeling Language)
Il modello UML può essere visto gerarchicamente come un Sistema diviso in uno o più modelli a sua volta divisi in una o più viste. Lo scopo dei modelli UML è quello di semplificare l'astrazione di un progetto software e nascondere i dettagli non necessari alla comprensione della struttura generale.
Prima di visionare i vari tipi di diagrammi, UML definisce alcune convenzioni.
I nomi sottolineati delineano le istanze, mentre nomi non sottolineati denotano tipi (o classi), i diagrammi sono dei grafi, i nodi sono le entità e gli archi sono le interazioni tra di essi, gli attori rappresentano le entità esterne che interagiscono con il sistema. Esistono vari tipi di diagrammi UML qui di seguito descritti:

4.1 **Diagrammi dei casi d'uso (use case diagrams)**
Serve a rappresentare l'interazione del sistema con uno o più attori e descrive tutti i vari casi possibili. Ha un nome univoco, degli attori partecipanti (almeno 1), una o più condizioni di entrata e di uscita, un flusso di eventi e delle eventuali condizioni eccezionali.
Tra le entità di un use-case diagram si possono avere varie relazioni rappresentate dagli archi con la freccia rivolta verso l'entità.
Le relazioni possono essere di vario tipo: l'arco senza descrizione e senza freccia indica che l'attore può eseguire una certa funzionalità, l'arco con linea continua e con freccia indica una generalizzazione (ereditarietà) e punta verso il caso generale, l'arco con la descrizione <<extend>> rappresenta un caso eccezionale o che si verifica di rado e punta verso il caso eccezionale, quello con la descrizione <<include>> è la relazione tra due entità che indica che una determinata funzionalità implica l'esecuzione di un altra e punta verso quella che viene eseguita per implicazione.

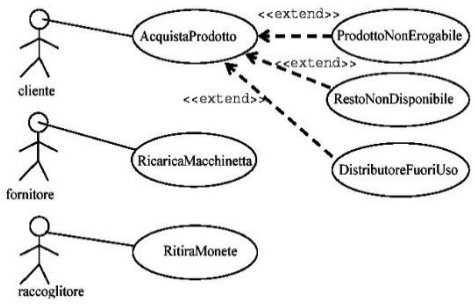


Figura 2 Casi d'uso per un distributore di snack

4.2 **Diagrammi di classi (class diagrams)**

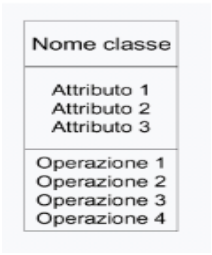
Rappresentano la struttura di un sistema. Vengono usati durante la fase di analisi dei requisiti e il system design di un modello.

È possibile definire diagrammi contenenti classi astratte e altri in cui compaiono istanze delle stesse con i relativi attributi specificati. Ogni nodo del grafo rappresenta una classe o un'istanza con un nome (nel caso di un'*istanza* il nome è sottolineato) e contiene i suoi attributi e i suoi comportamenti (le operazioni che essa svolge). Ogni attributo ha un tipo e ogni operazione ha una firma.

4.2.1 Classi in UML

In UML una classe è composta da tre parti:

1. nome
2. attributi (lo stato)
3. metodi o operazioni (il comportamneto)

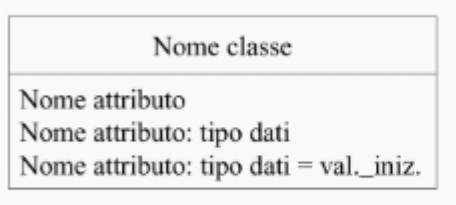


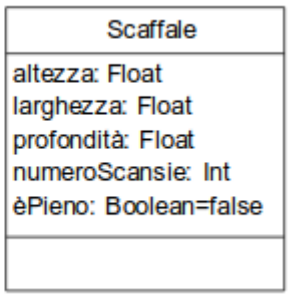
Una classe può essere rappresentata anche usando soltanto la sezione nome, un nome può essere:

- *simple name*: il solo nome della classe.
- *Path name*: il nome della classe è preceduto dal package in cui si trova.

4.2.2 Attributi

Un attributo è una proprietà statica di un oggetto, contiene un solo valore per ogni istanza. I nomi degli attributi devono essere unici all'interno di una classe. Per ciascun attributo si può specificare il tipo ed un eventuale valore iniziale. Tipicamente il nome di un attributo è composto da una parola o più parole, la prima parola ha sempre la lettera scritto in minuscolo.





4.2.2.1 Attributi derivati

Attributi calcolati e non memorizzati. Si usano quando i loro valori variano frequentemente e la correttezza (precisione) del valore è importante. Il valore viene calcolato in base ai valori di altri attributi, es: età = f(dataDiNascita, oggi).

4.2.2.2 Attributi ridondanti
Il simbolo / inserito prima del nome di un'entità indica un attributo ridondante.

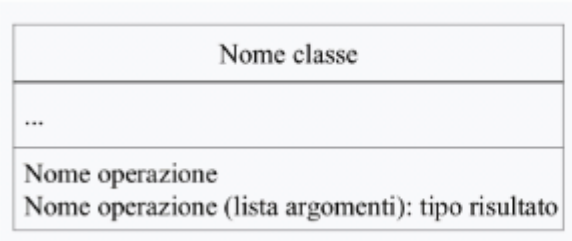
4.2.3 Operazioni (Metodi/Funzioni)
Un'operazione è un'azione che un oggetto esegue su un altro oggetto e che determina una reazione.
Tipi di operazione:

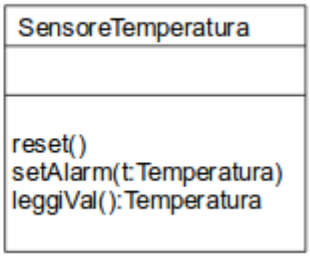
- selettore (query): accedono allo stato dell'oggetto senza alterarlo (es. "lunghezza"della classe coda).
- modificatore: alterano lo stato di un oggetto (es. "appendi"della classe coda).

Operazioni di base per una classe di oggetti (realizzate con modalità diverse a seconda dei linguaggi):

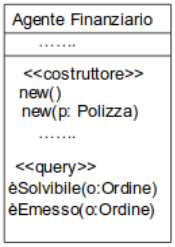
- costruttore: crea un nuovo oggetto e/o inizializza il suo stato
- distruttore: distrugge un oggetto e/o libera il suo stato

Per ciascuna operazione si può specificare il solo nome o la sua *signature*, indicando il nome, il tipo, parametri e in caso di funzione il tipo ritornato. Stesse convenzioni dette per gli attributi sull'uso di minuscolo e maiuscolo per i nomi delle operazioni.

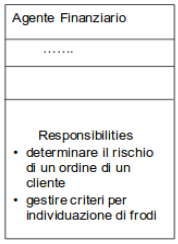




Per organizzare meglio lunghe liste di attributi/operazioni conviene raggrupparli in categorie usando gli stereotipi.



4.2.4 Responsibility
Una responsibility è un contratto o una obbligazione di una classe. Questa è definita dallo stato e comportamento della classe. Una classe può avere un qualsiasi numero di responsabilità, ma una classe ben strutturata ha una o poche responsabilità.



4.2.5 Proprietà attributi e operazioni
4.2.5.1 Visibilità
È possibile specificare la visibiltà di attributi e operazioni. In UML è possibile specificare tre livelli di visibilità:

- **+ (public):** qualsiasi altra classe con visibilità alla classe data può usare l'attributo/operazione (di default se nessun simbolo è indicato).

- **# (protected):** qualsiasi classe discendente della classe data può usare l'attributo/operazione.
- **- (private):** solo la classe data può usare l'attributo/operazione.

4.2.5.2 Molteplicità

La molteplicità è usata per indicare il numero di istanze di una classe, una molteplicità pari a zero indicherà una classe astratta, una molteplicità pari ad uno o una singleton class. Per default è assunta una molteplicità maggiore di uno. La molteplicità di una class è indicata con un numero intero posto nell'angolo in alto a destra del simbolo della class. La molteplicità si applica anche agli attributi, indicandola tra [..] (Es: consolePort [2..*]: Port).

4.2.5.3 Specifica attributi

La sintassi completa per specificare un attributo in UML è:

[visibility] name [[multiplicity]] [: type] [= inital-value] [{property-string}]

Dove property-string può assumere uno dei seguenti valori:

- *changeable:* nessuna limitazione per la modifica del valore dell'attributo
- *addOnly:* per attributi con molteplicità maggiore di 1 possono essere aggiunti ulteriori valori, ma una volta creato un valore non può essere né rimosso né modificato
- *frozen:* il valore non può essere modificato dopo la sua inizializzazione

4.2.5.3 Specifica operazioni

La sintassi completa per specificare un operation in UML è:

[visibility] name [(parameter-list)] [:return-type] [{property-string}]

Con la lista dei parametri avente questa sintassi:

[direction] name: type [=default-value]

Dove direction può assumere uno dei seguenti valori:

- **in**: parametro di input
- **out**: parametro di output
- **inout**: parametro di input/output

Property-string può assumere i seguenti valori:

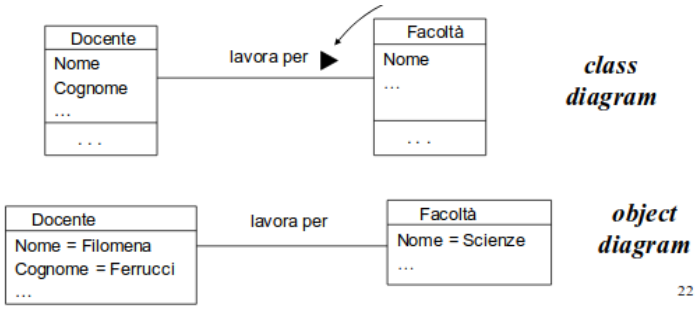
- **isQuery:** l'esecuzione dell'operazione lascia lo stato del sistema immutato.

Proprietà che riguardano la concorrenza, rilevanti solo in presenza di oggetti attivi, processi o threads

- **sequential:** la semantica e l'integrità dell'oggetto è garantita nel caso di un solo flusso di controllo per volta verso l'oggetto
- **guarded:** la semantica e l'integrità dell'oggetto è garantita in presenza di flussi di controllo multipli sequenzializzando le chiamate alle operazioni guarded
- **concurrent:** la semantica e l'integrità dell'oggetto è garantita in presenza di flussi di controllo multipli, trattando la operation come atomica

4.2.6 Legami e associazioni

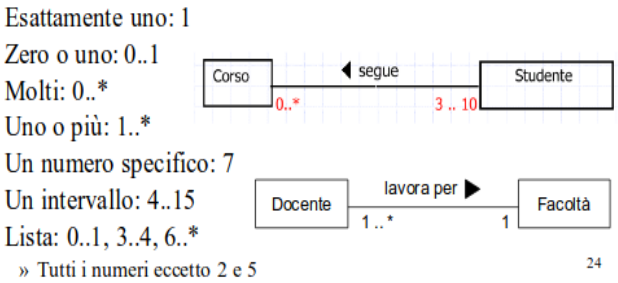
Un legame (link) rappresenta una relazione (fisica o concettuale) tra oggetti la cui conoscenza deve essere preservata per un certo periodo di tempo (Filomena Ferrucci lavora per Facoltà di Scienze). Un'associazione descrive un gruppo di legami aventi struttura e semantica comuni (Docente lavora per Facoltà). Un'associazione deve avere un nome, il nome è solitamente un verbo. Le associazioni sono bidirezionali sebbene al nome della relazione può essere associata una direzione.
I link sono istanze delle associazioni. Un link connette due oggetti (Object diagram).
Un'associazione connette due classi (class diagram).



4.2.6.1 Molteplicità delle associazioni

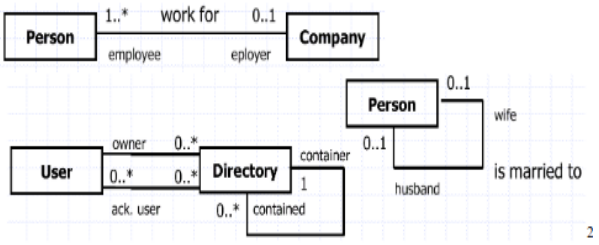
La molteplicità dice:

- Se l'associazione è obbligatoria oppure no
- Il numero minimo e massimo di oggetti che possono essere relazionati ad un altro oggetto.



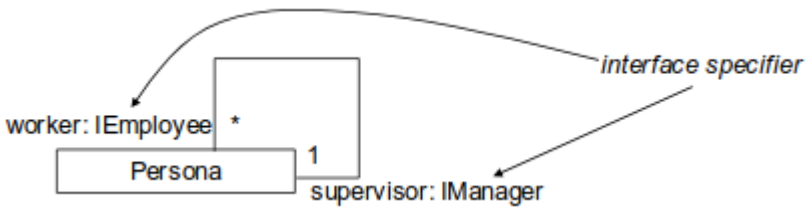
4.2.7 Ruoli

I ruoli forniscono una modalità per attraversare relazioni da una classe ad un'altra. I nomi di ruolo possono essere usati in alternativa ai nomi delle associazioni. I ruoli sono spesso usati per relazioni tra oggetti della stessa classe (associazioni riflessive).



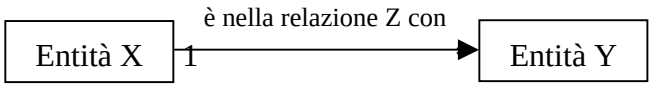
4.2.7.1 Interface specifier

Un'associazione può avere un interface specifier, per specificare quale parte dell'interfaccia di una classe è mostrata da questa nei confronti di un'altra classe della stessa associazione.
Esempio: Una classe Persona nel ruolo di supervisor presenta solo la 'faccia' IManager al worker; mentre una Persona nel ruolo di worker presenta solo la 'faccia' IEmployee al supervisor.



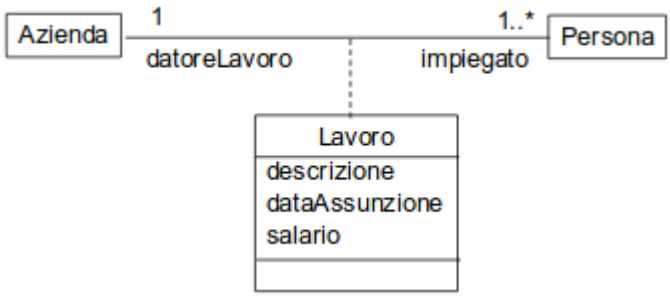
4.2.8 Associazione

L'arco semplice rappresenta una *associazione* che può essere specificata anche da un testo, inoltre è possibile indicare le molteplicità: ad esempio un'entità X può essere associata ad una o più entità Y in tal caso avremo il seguente diagramma:



4.2.8.1 Classi associative

Sono utilizzate per modellare proprietà delle associazioni. Alcune proprietà potrebbero appartenere all'associazione e non alle parti coinvolte. Un attributo di una classe associativa contiene un valore per ogni legame.

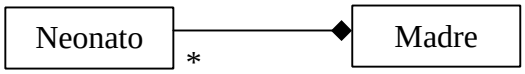


4.2.9 Aggregazione

È possibile rappresentare chi contiene cosa sotto forma di grafo (o albero) utilizzando la linea con il rombo terminatore come nella figura. Le sue proprietà sono: transitività, anti simmetria (se A è parte di B allora B non è parte di A) e dipendenza (un oggetto contenuto potrebbe non sopravvivere senza l'oggetto contenente).

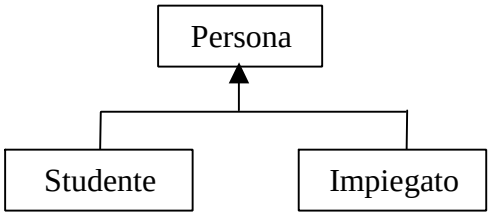
4.2.10 Composizione

Una particolare aggregazione che comporta l'esistenza di un'entità padre data un'entità figlio. In particolare:



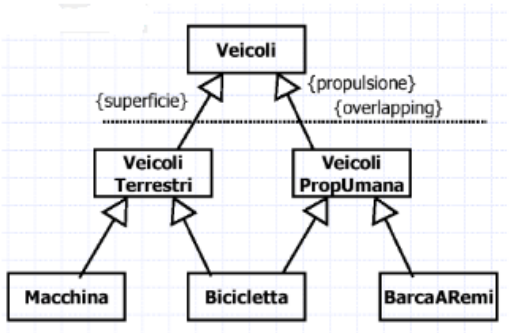
4.2.11 Generalizzazione (o ereditarietà)

Indica l'ereditarietà tra entità: la classe figlio eredità attributi e operazioni del padre semplificando il modello ed eliminando la ridondanza. Tutte le proprietà (attributi e operazioni) di una super classe possono essere applicati alle sottoclassi (sono ereditati). Generalizzazione ed ereditarietà godono della proprietà transitiva. È possibile definire nuove proprietà per le sottoclassi, ed è possibile ridefinire le proprietà ereditate (overriding). Anche le relazioni di una super classe valgono per le sottoclassi.



4.2.11.1 Ereditarietà multipla

L'overlapping può portare all'ereditarietà multipla. Una sottoclasse ha più di una super classe ed eredita le proprietà da tutte le superclassi. Comoda nella modellizzazione ma può creare conflitti nella realizzazione.

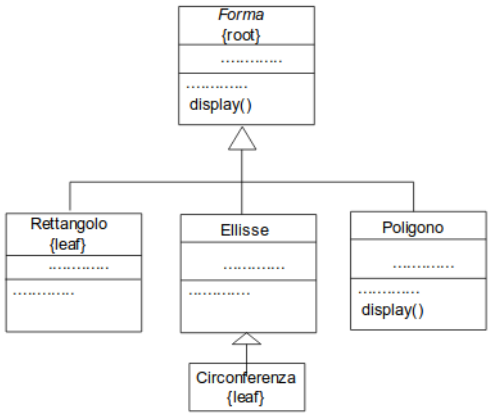


4.2.12 Classi astratte

Una classe astratta definisce un comportamento generico. Definisce e può implementare parzialmente il comportamento di un oggetto. Dettagli più specifici sono completati nelle sottoclassi specializzate. Queste ultime sono indicate scrivendo il nome in corsivo.

4.2.13 Root, leef elementi polimorfici

Per specificare che una classe non può avere discendenti si indicherà per questa la proprietà leaf sotto il nome della classe. Per specificare che una classe non può avere antenati si indicherà per questa la proprietà root sotto il nome della classe. Un'operazione per la quale esiste la stessa signature in più classi di una gerarchia è polimorfica.



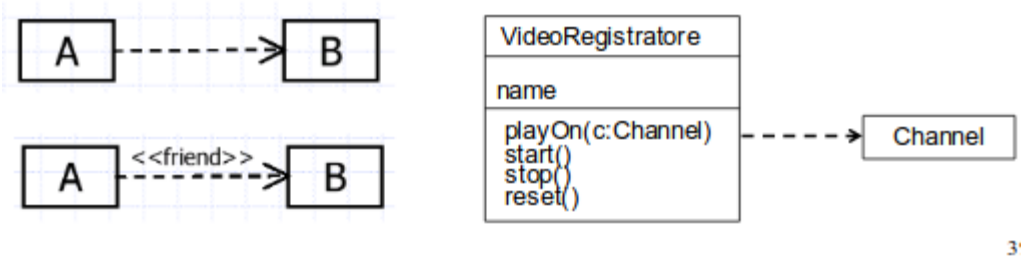
4.2.14 Gerarchia di classi

UML definisce

- 1 stereotipo:
  - **implementation:** cioè la sottoclasse eredita l'implementazione della super classe ma non rende pubblica né supporta la sua interfaccia.
- 4 vincoli:
  - **Complete:** tutte le sottoclassi sono state specificate, nessun'altra sottoclasse è permessa.
  - **Incomplete:** non tutte le sottoclassi sono state specificate, altre sottoclassi sono permesse.
  - **Disjoint:** oggetti del genitore possono avere non più di un figlio come tipo.
  - **Overlapping:** oggetti del genitore possono aver più di un figlio come tipo.

4.2.15 Dependency

Relazione semantica in cui un cambiamento sulla classe indipendente può influenzare la semantica della classe dipendente. La freccia punta verso la classe indipendente. UML defisce 17 stereotypes (organizzati in 6 gruppi) che possono essere applicati a dependency.

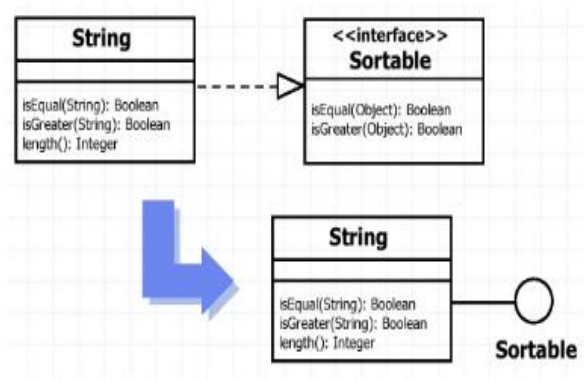


4.2.16 Realization

Una relazione tra classi in cui una specifica un contratto che l'altra garantisce di compiere. La freccia punta alla classe che definisce il contratto. È un incrocio tra dependency e generalization, usata principalmente in due circostanze: contesto delle interface e delle collaboration.

4.2.16 Interfacce (definizione)

Specifica il comportamento di una classe senza darne l'implementazione.



Nota: Un'interfaccia si può rappresentare anche solo con un cerchio.

4.2.17 Modelli dinamici con UML

Abbiamo due modelli dinamici che possono essere rappresentati con UML:

- *Interaction diagram:* descrive il comportamento dinamico tra gli oggetti
- *Statechart:* descrive il comportamento dinamico di un singolo oggetto

Gli iteraction diagrams si suddividono in:

- *Sequence diagram: c*omportamento dinamico di un insieme di oggetti disposti in sequenza temporale.
- *Collaboration diagram:* Mostra la relazione tra gli oggetti. Non mostra il tempo.

Invece uno statechart è: una macchina a stati che descrive la risposta di un oggetto di una data classe alla ricezione di stimoli esterni (Eventi). L'activity diagram è un tipo speciale di diagramma di diagramma di stato, in cui tutti gli stati sono stati di azione.
Possiamo dire quindi che la modellazione dinamica è un insieme di statechart, cioè uno statechart per ogni classe di oggetti.

4.3 **Interaction diagram**

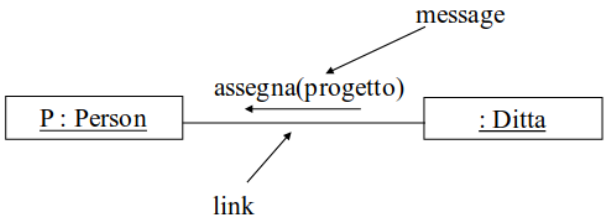
Descrivono il comportamento dinamico di un gruppo di oggetti che "interagiscono" per risolvere un problema. Un' interazione è un comportamento che comprende un insieme di messaggi scambiati tra un insieme di oggetti nell'ambito di un contesto per raggiungere uno scopo. UML propone due diversi tipi di sequence diagram:

- Sequence diagram
- Collaboration diagram

Sequence e Collaboration diagrams esprimono informazioni simili, ma le evidenziano in modo diverso.

4.3.1 Iterazioni e messaggi

Una interazione, tipicamente, avviene tra oggetti tra cui esiste un link (istanza di un'associazione). Un messaggio è una specificazione di una comunicazione tra oggetti che trasmette informazione con l'aspettativa che ne conseguirà una attività. La ricezione di un messaggio può essere considerata una istanza di un evento.



4.4 Diagrammi sequenziali (sequence diagrams)

Descrivono le sequenze di azioni e le interazioni tra le componenti. Servono a dettagliare gli use case durante la fase di analisi dei requisiti oppure durante il system design per definire le interfacce del sottosistema e per trovare tutti gli oggetti partecipanti al sistema ("chi" fa "che cosa").
I diagrammi di sequenza possono essere utilizzati nei seguenti modi:

- Per modellare le interazioni ad alto livello tra oggetti attivi all'interno di un sistema
- Per modellare l'interazione tra istanze di oggetti nel contesto di una collaborazione che realizza un caso d'uso
- Per modellare l'interazione tra oggetti in una collaborazione che realizza una operazione

In più:

- Evidenziano la sequenza temporale delle azioni
- Non si vedono le associazioni tra oggetti
- Le attività svolte dagli oggetti sono mostrate su linee verticali
- La sequenza dei messaggi scambiati tra gli oggetti è mostrata su linee orizzontali
- Possono corrispondere a uno scenario specifico o a un intero caso d'uso (aggiungendo salti e iterazioni)
- Si possono annotare con vincoli temporali

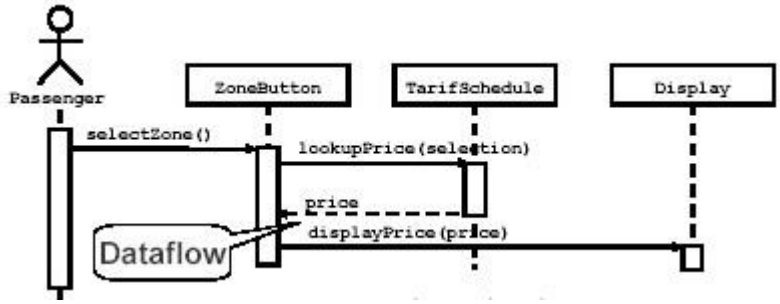


Figura 3 Diagramma sequenziale di un passeggero che acquista un biglietto

- Sull'asse X abbiamo gli oggetti, sull'asse T (asse del tempo) abbiamo il flusso del tempo.
- Un messaggio sincrono si disegna con una freccia chiusa, normalmente è etichettata con il nome del metodo invocato opzionalmente con i suoi parametri e valori di ritorno.
- Il *life-time* (durata, vita) di un metodo è rappresentato da un rettangolino che collega la freccia di invocazione con quella di ritorno. Il ritorno è disegnato con una freccia tratteggiata ed è opzionale, se viene omessa la fine del metodo è decretata dalla fine del life-time.
- I messaggi asincroni si usano per descrivere interazioni concorrenti, si rappresenta con una freccia aperta.
- L'esecuzione del metodo può essere assoggettata ad una condizione: [cond]: nomeMetodo()
- La parola *new (create)* sulla freccia rappresenta la costruzione del nuovo oggetto non presente fino a quel momento nel sistema.

- La life-line di un oggetto è la line tratteggiata che rappresenta l'esistenza di un oggetto per un periodo di tempo.
- La distruzione di un oggetto si rappresenta con una X alla fine della life-line dell'oggetto.

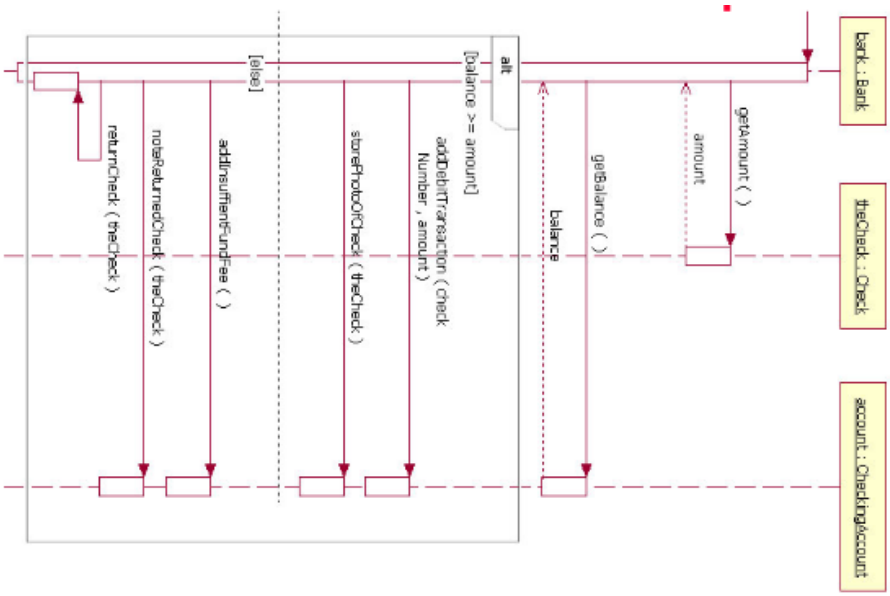
4.4.1 Iterazioni (ricorrenze)

Rappresenta l'esecuzione ciclica di più messaggi. Si disegna raggruppando con un blocco i messaggi su cui si vuole iterare. Si può aggiungere la condizione che definisce l'iterazione sull'angolo in alto a sinistra del blocco. La condizione si rappresenta al solito tra parentesi quadre.

4.4.2 Cicli e condizioni

Cicli e condizioni si indicano con un riquadro (frame) che racchiude una sotto sequenza di messaggi. Nell'angolo in alto è indicato il costrutto. Tra i costrutti possibili:

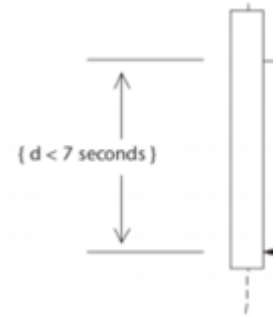
- Loop (ciclo while-do o do-while): la condizione è indicata tra parentesi quadra all'inizio o alla fine;
- Alt (if-then-else): la condizione si indica in cima; se ci sono anche dei rami else allora si usa una linea tratteggiata per separare la zona then dalla zona else indicando eventualmente un'altra condizione accanto alla parola else;
- Opt (if-then): racchiude una sottosequenza che viene eseguita solo se la condizione indicata in cima è verificata.



4.4.3 Auto-chiamata

Descrive un oggetto che invoca un proprio metodo.

4.4.4 Esprimere vincoli sul tempo di risposta



4.4.5 Messaggi e azioni

Ad un messaggio possono corrispondere diversi tipi di azioni

- **Call**: invoca una operazione di un oggetto; un oggetto può inviare un messaggio a se stesso
- **Return**: restituisce un valore al chiamante
- **Send**: invia un segnale ad un oggetto
- **Create**: crea un oggetto
- **Destroy**: distrugge un oggetto; un oggetto può distruggere se stesso

I messaggi possono essere preceduti da condizioni

- [x > 0] messaggio()

I messaggi possono indicare iterazioni

- * messaggio()

4.5 **Diagramma a stati (state chart)**

Descrivono una sequenza di stati di un oggetto in risposta a determinati eventi. Rappresentano anche le transizioni causate da un evento esterno e l'eventuale stato in cui esso viene portato.

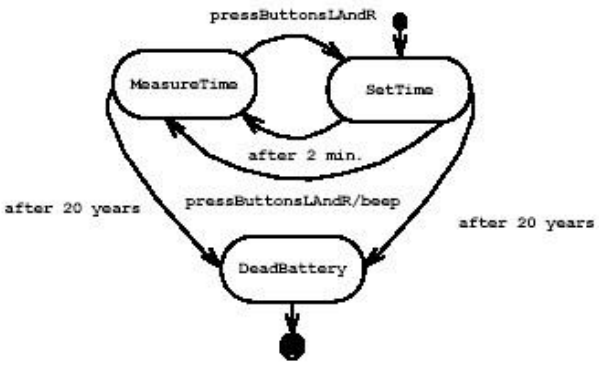
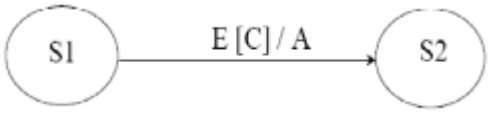


Figura 4 Diagramma a stati di un orologio

Uno stato rappresenta una situazione in cui un oggetto ha un insieme di proprietà considerate stabili.
Una transizione modella un ca,biamento di stato ed è denotata da: **Evento [Condizione] / Azione**.

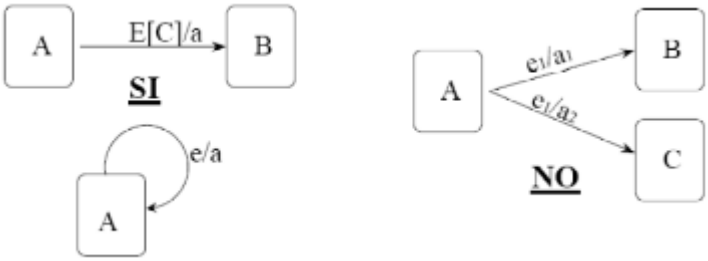


Il significato di una transizione del tipo di quella qui mostrata è:

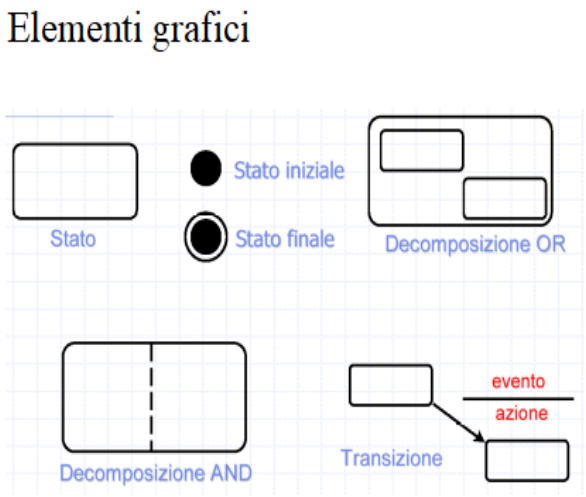
- Se l'oggetto si trova nello stato S1, e
- Se si verifica l'evento E, e
- Se la condizione C è verificata
- Allora viene eseguita l'azione A e l'oggetto passa nello stato S2

Per indicare uno stato inziale una freccia parte da "stato inziale" ed entra in uno stato, viceversa invece per lo stato finale.

Ogni transizione connette due stati. Il diagramma corrsiponde ad un automa **deterministico**, in cui un evento è un input, mentre un'azione è un'output. La condizione è detta anche "guardia", L'evento è quasi sempre presente mentre condizione e azione sono opzionali.

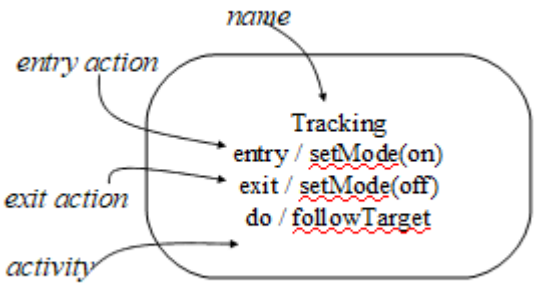


4.5.1 Elementi grafici dello state diagram



Decomposizione OR: solo uno stato viene eseguito.
Decomposizione AND: gli stati vengono eseguiti concorrentemente. Se uno stato raggiunge lo stato finale prima dell'altro, il controllo aspetta lo stato finale dell'altro. Quando avviene una transizione il flusso di controllo subisce una fork per ciascuno stato in parallelo, alla fine si ricompone in un unico flusso con un join.

4.5.2 Caratteristiche dello stato



- Gli attributi sono opzionali, e sono:
- **Nome:** una stringa; uno stato può essere anonimo.
- **Entry/ Exit actions:** eseguite all'ingresso/uscita dallo stato (non sono interrompibili, hanno durata instantanea).
- **Transizioni interne:** non causano un cambiamento di stato.
- **Attività** dello stato (interrompibile, durata significativa).
- **Sottostati:** strutttura innestata di stati; disgiunti (sequenzialmente attivi) o concorrenti (concorrentemente attivi).

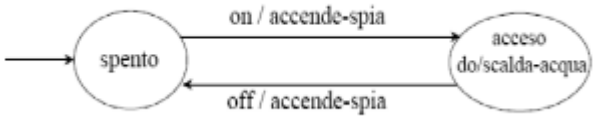
4.5.3 Caratteristiche delle transizioni

- transizione esterna (stato finale diverso dallo stato iniziale)
- interna (stato finale uguale allo stato iniziale)

4.5.4 Attributi (opzionali: evento [condizione]/azione)

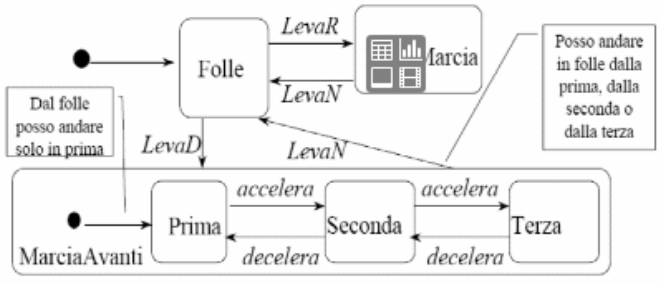
- **Evento:** segnale, messaggio da altri oggetti, passaggio del tempo, cambiamento
- **Condizione o guardia:** la transizione occorre se l'evento accade e la condizione di guardia è vera
- **Azione:** è eseguita durante la transizione e non è interrompibile (durata istantanea)
- **Transizioni senza eventi (triggerless):** scattano secondo le seguente condizioni:
  - **con guardia:** se la condizione di guardia diventa vera
  - **senza guardia:** se l'attività interna allo stato iniziale è completata

Alcune volte vogliamo rappresentare dei processi che l'oggetto esegue senza cambiare stato. Questi processi si chiamano **Attività**, e si mostrano negli stati con la notazione: **do/attività**.



4.5.5 Stato composto

Uno stato composto (macro-stato) è uno stato che ha un nome, e che contiene a sua volta un diagramma. Esiste uno stato iniziale del macro-stato. I sottostati **ereditano** le transazioni in uscita del macro-stato.



4.6 **Diagrammi delle attività (activity diagrams)**

È un particolare diagramma a stati in cui però al posto degli stati vi sono delle funzioni. Gli archi rappresentano la motivazione di esecuzione della funzione a cui punta.

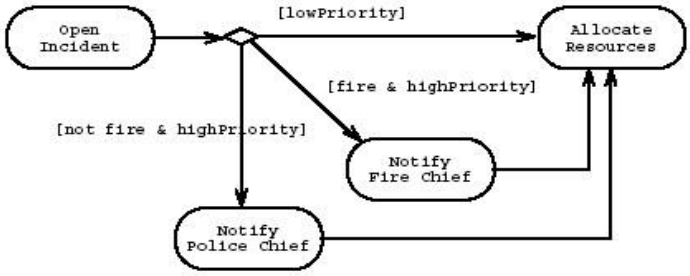


Figura 5 Diagramma delle attività per l'apertura di un incidente

È possibile modellare situazioni di concorrenza in cui varie funzioni vengono eseguite simultaneamente utilizzando la seguente forma di grafico:

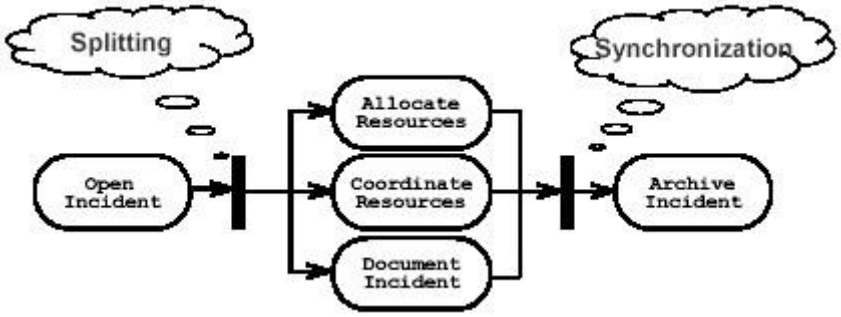


Figura 6 Diagramma delle attività con concorrenza

Un Activity diagram può essere associato:

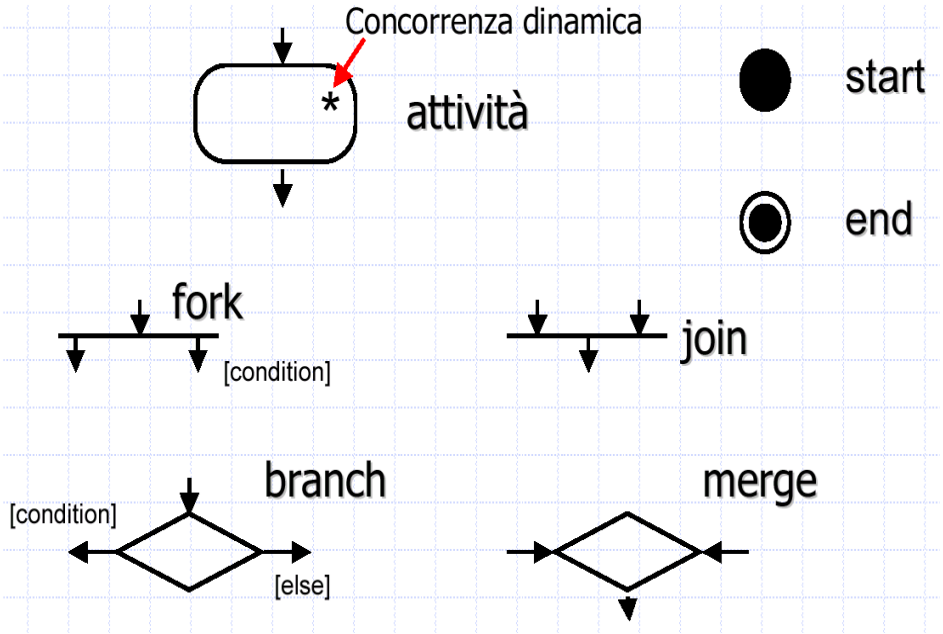
- a una classe
- All'implementazione di un'operazione
- Ad uno Uso Case

Utili per modellare

- comportamenti sequenziali
- non determinismo
- concorrenza
- sistemi distribuiti
- business workflow
- operazioni

Sono ammessi stereotipi per rappresentare le azioni.

4.6.1 Elementi grafici



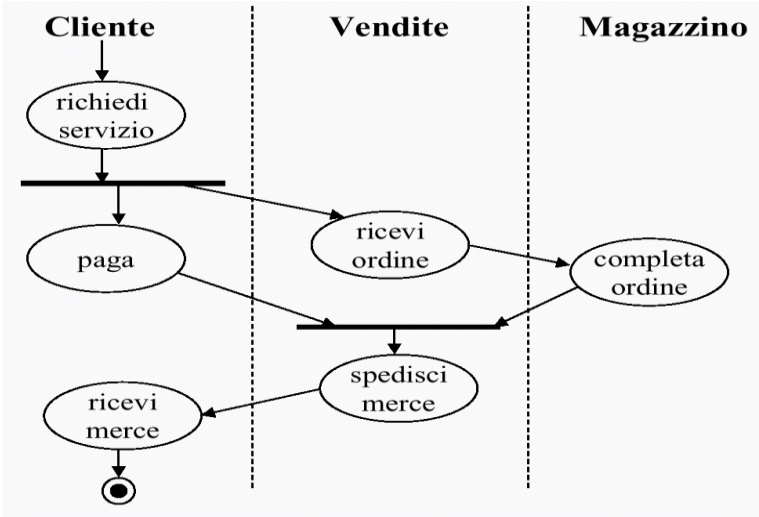
- *Activity*: una esecuzione non atomica entro uno state machine
  - Una activity è composta da action, elaborazioni atomiche comportanti un cambiamento di stato del sistema o il ritorno di un valore
- *Transition*: flusso di controllo tra due action successive
- *Guard expression*: espressione booleana (condition) che deve essere verificata per attivare una transition
- *Branch*: specifica percorsi alternativi in base a espressioni booleane; un branch ha una unica transition in ingresso e due o più transition in uscita
- *Synchronization bar*: usata per sincronizzare flussi concorrenti
  - *fork*: per splittare un flusso su più transition verso action state concorrenti
  - *join*: per unificare più transition da più action state concorrenti in una sola
    - il numero di fork e di join dovrebbero essere bilanciati

4.6.2 Swimlanes

lo swimlanes raggruppa in modo omogeneo le attività, formisce anche una descrizione ad alto livello di come è organizzato l'ambito lavorativo del sistema.

- Costrutto grafico rappresentante un insieme partizionato di action/activity;
- Identificano le responsabilità relative alle diverse operazioni
  - Parti di un oggetto
  - Oggetti diversi
- In un Business Model identificano le unità organizzative
- Per ogni oggetto responsabile di action/activity nel diagramma è definito un swimlane, identificato da un nome univoco nel diagramma
  - le action/activity state sono divise in gruppi, ciascun gruppo è assegnato allo swimlane dell'oggetto responsabile per esse
  - l'ordine con cui gli swimlane si succedono non ha alcuna importanza

- le transition possono attraversare swimlane per raggiungere uno state in uno swimlane non adiacente a quello di start della transition



4.7 **Raggruppamento (packages)**

Si può cercare migliorare la semplicità di un sistema raggruppando elementi del modello in packages. Ad esempio è possibile raggruppare use case o attività.

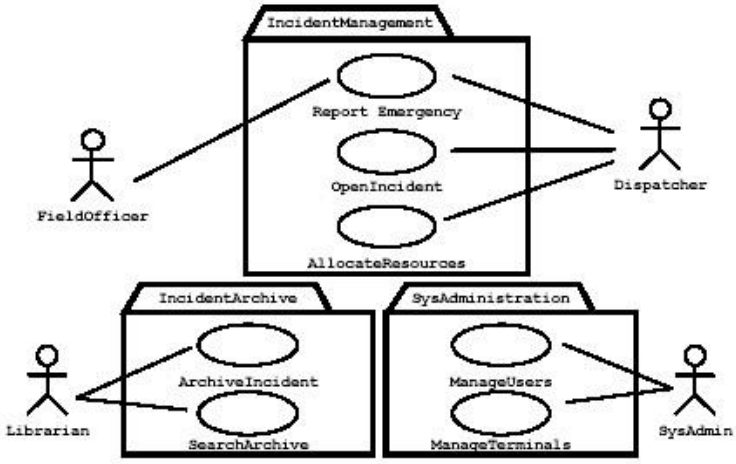


Figura 7 Raggruppamento

5 Raccolta dei requisiti (requirements elicitation)

L'ingegneria dei requisiti coinvolge due attività: *raccolta dei requisiti* e *analisi dei requisiti*.

La raccolta dei requisiti richiede la collaborazione tra più gruppi di partecipanti di tipologie e conoscenze diversificate. Gli errori commessi durante questa fase sono difficili da correggere e vengono spesso notati nella fase di consegna. Alcuni errori possono essere: funzionalità non specificate o incorrette o interfacce poco intuitive.

Utenti e sviluppatori devono collaborare per scrivere il **documento di specifica dei requisiti** che è scritto in linguaggio naturale per poi essere successivamente formalizzato e strutturato (in UML o altro) durante la fase di analisi per produrre il **modello di analisi**.

Il primo documento (la specifica dei requisiti) è utile al fine di favorire la comunicazione con il cliente e gli utenti, il documento prodotto nell'analisi è usato dagli sviluppatori.

La raccolta dei requisiti e l'analisi dei requisiti si focalizzano sul punto di vista dell'utente e definiscono i confini del sistema da sviluppare, in particolare vengono specificate:

- Funzionalità del sistema
- Interazione utente-sistema
- Errori che il sistema deve gestire
- Vincoli e condizioni di utilizzo

5.1 **Classificazione dei requisiti**

Le specifiche dei requisiti sono una sorta di contratto tra il cliente e gli sviluppatori e deve essere curata con attenzione in ogni suo dettaglio. Inoltre le parti del sistema che comportano un maggior rischio devono essere prototipate e provate con simulazioni per controllare la loro funzionalità e ed ottenere un riscontro dall'utente.
Esistono varie tipologie di requisiti qui di seguito specificati:

5.1.1 Requisiti funzionali

Descrivono le interazioni tra il sistema e l'ambiente esterno (utenti e sistemi esterni) indipendentemente dall'implementazione.
Un modo per scrivere in modo corretto i requisiti funzionali è il seguente:
[condition][subject][action][object][constraint] = [condizione] [soggetto] [azione] [oggetto] [vincolo]

5.1.2 Requisiti non funzionali

Descrivono aspetti del sistema che non sono legati direttamente alle funzionalità del sistema. Ad esempio sono requisiti non funzionali dettagli implementativi tipo timeout e altro.
Altri requisiti non funzionali sono parte dello standard **FURPS** e sono di qualità e di vincoli.

Qualità

- Usabilità (help in linea, documentazione a livello utente)
- Attendibilità (robustezza, coerenza delle funzionalità richieste)
- Performance (tempo di risposta, throughput, disponibilità)
- Supportabilità (manutenzione, portabilità, adattabilità)

Vincoli

- Implementazione (uso di tool, linguaggi, piattaforma hardware)
- Interfacce (vincoli imposti da sistemi esterni tra cui sistemi legacy e formato di interscambio di dati)
- Operativi (vincoli di management e amministrativi)
- Packaging (riguardano i tools che sono richiesti all'utente al fine del funzionamento del software)
- Legali (licenza, certificazione e regolamento)

5.2 **Validazione dei requisiti**

I requisiti devono essere continuamente validati con il cliente e l'utente, la validazione degli stessi è un aspetto molto importante perché ha lo scopo di non tralasciare nessun aspetto.
I requisiti devono rispettare le seguenti caratteristiche:

- Completezza (devono essere presi in considerazione tutti i possibili scenari, inclusi i comportamenti eccezionali)
- Consistenza (non devono contraddire se stessi)

- Non ambiguità (deve essere definito un unico sistema e non deve essere possibile interpretare la specifica in modi differenti)
- Correttezza (deve rappresentare il sistema di cui il cliente ha bisogno con accuratezza)
- Realistiche (se il sistema può essere implementato in tempi ragionevoli)
- Verificabili (se una volta che il sistema è stato implementato è possibile effettuare dei test)
- Tracciabili (se ogni requisito può essere mappato con una corrispondente funzionalità del sistema)

5.2.1 Tracciabilità

Esistono diversi tipi di tracciabilità:

- **Source tracciability:** da quale fonte arrivano i requisiti scelti dagli stakeholder, chi ha proposto i requisiti e il fondamento logico per questi requisiti. Quando viene proposta una modifica, si utilizzano queste informazioni per scoprire le parti interessate in modo che possano essere consultate in merito al cambiamento.
- **Le informazioni sulla tracciabilità dei requisiti** collegano i requisiti dipendenti all'interno del documento dei requisiti. Si utilizzano queste informazioni per valutare quanti requisiti sono interessati da una possibile modifica.
- **Le informazioni sulla tracciabilità del design** collegano i requisiti al design dei moduli in cui questi requisiti sono implementati. Si usano queste informazioni per valutare l'impatto delle modifiche proposte sui requisiti, la progettazione e l'implementazione del sistema.
- **Il test delle informazioni di tracciabilità** collega i requisiti ai casi di test dove vengono testati questi requisiti.

5.2.2 Greenfield engineering, re-engineering, interface engineering

Altri requisiti possono essere specificati in base alla sorgente delle informazioni. Il *greenfield engineering* avviene quando lo sviluppo di un'applicazione parte da zero, senza alcun sistema preesistente.
Il *re-engineering* è un tipo di raccolta dei requisiti dove c'è un sistema preesistente che deve essere riprogettato a causa di nuove esigenze o nuove tecnologie.
L'interface *engineering* avviene quando è necessario riprogettare un sistema per farlo lavorare in un nuovo ambiente. Un esempio possono essere i sistemi legacy che vengono lasciati inalterati nelle interfacce.

5.3 Attività della raccolta dei requisiti

1. Identificare gli attori: durante questa attività, gli sviluppatori identificano i diversi tipi di utenti che il sistema futuro supporterà.
2. Identificare gli scenari: durante questa attività, gli sviluppatori osservano gli utenti e sviluppano una serie di scenari dettagliati per le funzionalità tipiche fornite dal sistema futuro. Gli scenari sono esempi concreti del futuro sistema in uso. Gli sviluppatori utilizzano questi scenari per comunicare con l'utente e approfondire la conoscenza del dominio dell'applicazione.
3. Identificare i casi d'uso: una volta che gli sviluppatori e gli utenti concordano su una serie di scenari, gli sviluppatori traggono dagli scenari una serie di casi d'uso che rappresentano completamente il sistema futuro.
4. Raffinare i casi d'uso: durante questa attività, gli sviluppatori assicurano che la specifica dei requisiti sia completa specificando ciascun caso d'uso e descrivendo il comportamento del sistema in presenza di errori e condizioni eccezionali.
5. Identificare le relazioni tra gli attori e i casi d'uso: durante questa attività, gli sviluppatori identificano le dipendenze tra i casi d'uso definendone le funzionalità comuni. Ciò garantisce che la specifica dei requisiti sia coerente.
6. Identificare gli oggetti partecipanti (verranno ripresi nella fase di analisi)

7. Identificare le richieste non funzionali: durante questa attività, sviluppatori, utenti e clienti concordano su aspetti visibili all'utente, ma non direttamente correlati alla funzionalità. Questi includono vincoli sulle prestazioni del sistema, la sua documentazione, le risorse che consuma, la sua sicurezza e la sua qualità.

5.3.1 Identificare gli attori

Un attore è un'entità esterna che comunica con il sistema e può essere un utente, un sistema esterno o un ambiente fisico.

Ogni attore ha un nome univoco ed una breve descrizione sulle sue funzionalità (es. Teacher: una persona; Satellite GPS: fornisce le coordinate della posizione).

Un modo molto semplice per identificare gli attori di un sistema e porsi le seguenti domande:

- Quali gruppi di utenti sono supportati dal sistema per svolgere il proprio lavoro, quali eseguono le principali funzioni del sistema, quali eseguono le funzioni di amministrazione e mantenimento?
- Con quale sistema hardware o software il sistema interagisce?

5.3.2 Identificare gli scenari

Uno scenario è una descrizione informale, concreta e focalizzata di una singola caratteristica di un sistema e descrive cosa le persone fanno e sperimentano mentre provano ad usare i sistemi di elaborazione e le applicazioni.

Ogni scenario deve essere caratterizzato da un nome, una lista dei partecipanti e un flusso di eventi.

Esistono vari tipi di scenari:

- As-is-scenario: sono usati per descrivere una situazione corrente. Vengono di solito usati nella raccolta dei requisiti di tipo re-engineering.
- Visionary-Scenario: utilizzato per descrivere funzionalità future del sistema.
- Evalutation-Scenario: descrivono funzioni eseguite dagli utenti rispetto alle quali poi viene testato il sistema.
- Training-Scenario: sono tutorial per introdurre nuovi utenti al sistema.

L'identificazione degli scenari è una fase che avviene in stretta collaborazione con il cliente e l'utente.

Per poter formulare gli scenari bisogna porsi e porre all'utente le seguenti domande:

- Quali sono i compiti primari che l'attore vuole che svolga il sistema?
- Quali dati saranno creati/memorizzati/cambiati/cancellati o aggiunti dall'utente nel sistema?
- Di quali cambiamenti esterni l'attore deve informare il sistema?
- Di quali eventi/cambiamenti deve essere informato l'attore?

5.3.3 Identificare i casi d'uso

Un caso d'uso descrive una serie di interazioni che avvengono dopo un'inizializzazione da parte di un attore e specifica tutti i possibili scenari per una determinata funzionalità (visto in altri termini uno scenario è un'istanza di un caso d'uso). Ogni caso d'uso contiene le seguenti informazioni:

- Un *nome* del caso d'uso che dovrebbe includere dei verbi
- I *nomi degli attori* partecipanti che dovrebbero essere sostantivi
- Le *condizioni di ingresso/uscita* da quel caso d'uso.
- Un *flusso di eventi* in linguaggio naturale
- Le *eccezioni* che possono verificarsi quando qualcosa va male descritte in modo distinto e separato
- I *requisiti speciali* che includono i requisiti non funzionali e i vincoli

Nel flusso di eventi del caso d'uso vengono distinti gli eventi iniziati dagli attori da quelli iniziati dal sistema in quanto quelli del sistema sono più a destra (con un tab) rispetto a quelli dell'attore.

Nome Use Case	ReportEmergency
Partecipanti	Inizializzato dal *FieldOfficer* Comunica con il *Dispatcher*
Flusso degli eventi	1. Il *FieldOfficer* attiva la funzione "ReportEmergency" dal suo terminale 2. *FRIEND* risponde presentando un form al FieldOfficer 3. Il *FieldOfficer* completa il form selezionando il livello di emergenza, il tipo, la località, e una breve descrizione della situazione. Il *FieldOfficer* descrive anche possibili risposte alla situazione di emergenza. Quando il form è completo, il *FieldOfficer* sottomette il form 4. *FRIEND* riceve il form e notifica il *Dispatcher* 5. Il *Dispatcher* rivede le informazioni sottomesse e crea un *Incident* nel database invocando lo use case *OpenIncident*. Il *Dispatcher* seleziona una risposta e comunica il report 6. *FRIEND* visualizza il report e la risposta selezionata per il *FieldOfficer*

Figura 8 Esempio di caso d'uso per un ReportEmergency

5.3.4 Raffinare i casi d'uso

Vengono dettagliati gli elementi che sono manipolati dal sistema, dettagliate le interazioni a basso livello tra l'attore e il sistema, specificati i dettagli su chi può fare cosa, aggiunte eccezioni non presenti, le funzionalità comuni tra i casi d'uso vengono rese distinte.

5.3.5 Identificare le relazioni tra attori e casi d'uso

Esistono vari tipi di relazioni tra attori e casi d'uso (vedi anche lezione su UML):

- **Comunicazione:** Bisogna distinguere due tipi di relazione di comunicazione tra attori e casi d'uso. La prima detta <<initiate>> viene usata per indicare che un attore può iniziare un caso d'uso, la seconda <<participate>> invece indica che l'attore (che non ha iniziato il caso d'uso) può solo comunicare (es. ottenere informazioni) con lo stesso. In questo modo è possibile specificare già in questa fase dettagli sul controllo di accesso in quando vengono indicate le procedure e i passi per accedere a determinate funzioni del sistema.
- **Extend**: È usato per indicare un caso d'uso eccezionale in cui si viene a finire quando si sta eseguendo un altro caso d'uso. L'arco è tratteggiato con l'etichetta <<extend>> e con la linea rivolta verso il caso eccezionale.
- **Include**: Usata per scomporre un caso d'uso in dei casi d'uso più semplici. La freccia dell'arco è tratteggiata, etichettata con <<include>> ed è rivolta verso il caso d'uso che viene usato di conseguenza da quelli che puntano.

5.3.6 Identificare gli oggetti partecipanti

Durante la fase di raccolta dei requisiti utenti e sviluppatori devono creare un glossario di termini usati nei casi d'uso. Si parte dalla terminologia che gli utenti hanno (quella del dominio dell'applicazione) e successivamente si negoziano cambiamenti. Il glossario creato è lo stesso che viene incluso nel manuale utente finale.

I termini possono rappresentare oggetti, procedure, sorgenti di dati, attori e casi d'uso. Ogni termine ha una piccola descrizione e deve avere un nome univoco e non ambiguo.

Euristica:

- Termini che gli sviluppatori o gli utenti devono chiarire per comprendere il caso d'uso
- Nomi ricorrenti nei casi d'uso (ad es. Incidente)
- Entità del mondo reale che il sistema deve tracciare (ad es. FieldOfficer, Resource)
- Processi reali che il sistema deve tracciare (ad esempio, EmergencyOperationsPlan)
- Casi d'uso (ad esempio, ReportEmergency)
- Sorgenti o link di dati (ad es., Stampante)

- Manufatti con cui l'utente interagisce (ad esempio, Station)
- Utilizzare sempre i termini del dominio dell'applicazione.

5.3.7 Identificare i requisiti non funzionali

I requisiti non funzionali descrivono aspetti del sistema che non sono direttamente correlati al suo comportamento funzionale. I requisiti non funzionali coprono una serie di problemi, dall'aspetto dell'interfaccia utente ai requisiti di tempo di risposta ai problemi di sicurezza. I requisiti non funzionali sono definiti contemporaneamente ai requisiti funzionali perché hanno lo stesso impatto sullo sviluppo e sui costi del sistema.

Categoria	**Cosa chiedersi?**
Usabilità	• Qual è il livello di abilità dell'utente? • Quali standard di interfaccia sono familiari all'utente? • Quale documentazione dovrebbe essere fornita all'utente?
Affidabilità	• Quanto affidabile, disponibile e robusto dovrebbe essere il sistema? • Il riavvio del sistema è ammissibile se si verifica un errore? • Quanti dati il sistema può perdere? • Come dovrebbe gestire le eccezioni il sistema?Ci sono requisiti safe nel sistema? (es. una centrale nucleare deve essere safe). • Ci sono requisiti secure nel sistema? (es. la lunghezza di una password è secure).
Prestazione	• Quanto reattivo dovrebbe essere il sistema? • Ci sono tempi critici per le attività degli utenti? • Quanti utenti concorrenti dovrebbe supportare? • Quanto è grande un tipico archivio di dati per dei sistemi comparabili? • Qual è il ritardo accettabile per gli utenti?
Supportabilità (include manutenzione e sopportabilità)	• Quali sono le estensioni previste nel sistema? • Chi si occupa della manutenzione del sistema? • Ci sono dei piani per trasferire il sistema in ambienti hardware o software differenti?

Implementazione	• Ci sono vincoli sulla piattaforma hardware? • Ci sono vincoli imposti dal team di manutenzione? • Ci sono vincoli imposti dal team di testing?
Interfaccia	• Il sistema dovrebbe interagire con sistemi già esistenti? • Come i dati vengono importati/esportati nel sistema? • Quali standard utilizzati dal cliente dovrebbero essere supportati dal sistema?
Operazione	• Chi gestisce il sistema attuale?
Impacchettamento	• Chi installa il sistema? • Quante istallazioni sono previste? • Ci sono limiti di tempo sull'istallazione?
Legale	Come dovrebbe essere autorizzato il sistema? • Ci sono problemi di responsabilità associati ai fallimenti del sistema? • Ci sono eventuali diritti d'autore o costi di licenza associati all'uso di algoritmi o componenti specifiche?

Euristiche:

- Un requisito rispetta il seguente formato:
  - [Condition][Subject][Action][Object][Constraint]
  - oppure: [Condiction] [Action or Constraint][Value]
  - oppure [Subject][Action][Values]
- Il requisito indica COME il sistema dovrebbe essere?
- Un requisito deve essere scritto utilizzando espressioni positivo
- Un requisito deve essere scritto evitando l'uso di linguaggi soggetto (es: user friendly oppure easy to use)
- Un requisito non deve essere ambiguo
- Un requisito non deve essere scritto in forma passiva

5.4 **Gestire la raccolta dei requisiti**

Uno dei metodi per negoziare le specifiche con il cliente è il Joint Application Design (**JAD**), sviluppato da IBM, che si compone di cinque attività:

- *Definizione del progetto*: vengono interpellati il cliente e il project manager e vengono determinati gli obiettivi del progetto
- *Ricerca*: vengono interpellati utenti attuali e futuri e vengono raccolte informazioni sul dominio di applicazione e descritti ad alto livello i casi d'uso
- *Preparazione*: si prepara una sessione, un documento di lavoro che è un primo abbozzo dei documenti finale, un agenda della sessione e ogni altro documenti cartaceo utile che rappresenta informazioni raccolte durante la ricerca

- *Sessione*: viene guidato il team nella creazione della specifica dei requisiti, lo stesso definisce e si accorda sugli scenari, i casi d'uso e interfaccia utente mock-up.
- *Documento finale*: viene rivisto il documento lavoro e messe insieme tutte le documentazioni raccolte; il documento rappresenta una completa specificazione del sistema accordato durante l'attività di sessione.

Un altro aspetto importante è la **tracciabilità** del sistema che include la conoscenza della sorgente della richiesta e gli aspetti del sistema e del progetto. Lo scopo della tracciabilità è quello di avere una visione chiara del progetto e rendere meno complessa e lunga un'eventuale fase di modifica ad un aspetto del sistema. A tale supporto è possibile creare dei collegamenti tra i documenti per meglio identificare le dipendenze tra le componenti del sistema.
Il documento dell'analisi dei requisiti (**RAD**) contiene la raccolta dei requisiti e l'analisi dei requisiti ed è il documento finale del progetto, serve come base contrattuale tra il cliente e gli sviluppatori.

6 Analisi dei requisiti

L'analisi dei requisiti è finalizzata a produrre un modello del sistema chiamato modello dell'analisi che deve essere corretto completo consistente e non ambiguo.
La differenza tra la raccolta dei requisiti e l'analisi è nel fatto che gli sviluppatori si occupano di strutturare e formalizzare i requisiti dati dall'utente e trovare gli errori commessi nella fase precedente (raccolta dei requisiti).
L'analisi, rendendo i requisiti più formali, obbliga gli sviluppatori a identificare e risolvere caratteristiche difficili del sistema già in questa fase, il che non avviene di solito. Il modello dell'analisi è composto da tre modelli individuali:
- Il *modello funzionale* rappresentato da casi d'uso e scenari
- Il *modello ad oggetti* dell'analisi rappresentato da diagrammi di classi e diagrammi ad oggetti
- Il *modello dinamico* rappresentato da diagrammi a stati e sequence diagram

6.1 **Concetti dell'analisi**

6.1.1 Il modello ad oggetti

Il modello ad oggetti è una parte del modello dell'analisi basato e si focalizza sui concetti del sistema visti individualmente, le loro proprietà e le loro relazioni. Viene rappresentato con un diagramma a classi di UML includendo operazioni, attributi e classi.

6.1.2 Il modello dinamico

Il modello dinamico si focalizza sul comportamento del sistema utilizzando sequence diagram e diagrammi a stati. I sequence diagram rappresentano l'interazioni di un insieme di oggetti nell'ambito di un singolo caso d'uso. I diagrammi a stati rappresentano il comportamento di un singolo oggetto. Lo scopo del modello dinamico è quello di assegnare le responsabilità ad ogni singola classe, identificare nuove classi, nuove associazioni e nuovi attributi.

Nel modello ad oggetti dell'analisi e nel modello dinamico le classi che vengono descritte non sono quelle che in realtà poi verranno implementate nel software ma rappresentano ancora un punto di vista dell'utente. Spesso infatti ogni classe del modello viene mappata con una o più classi del codice sorgente, e gli attributi e tutte le sue caratteristiche sono specificate in modo minimale.

6.1.3 Generalizzazione e specializzazione

La generalizzazione è l'attività di modellazione che identifica i concetti astratti da quelli di livello inferiore.
La specializzazione è l'attività che identifica concetti più specifici da uno di alto livello -
In entrambi i casi, la generalizzazione e la specializzazione determinano la specificazione delle relazioni di ereditarietà tra concetti.

6.1.4 Entity, Boundary (oggetti frontiera) e Control object

Il modello ad oggetti è costituto da oggetti di tipo *entity, boundary* e *control*.
Gli oggetti *entity* rappresentano informazioni persistenti tracciate dal sistema; gli oggetti *boundary* rappresentano l'interazione tra attore e sistema; gli oggetti di *controllo* realizzano i casi d'uso. Gli stereotipi entity control e bondary possono essere inclusi nei diagrammi e attaccati agli oggetti con le notazioni <<entity> <<control>> <<boundary>>.

6.1.5 Ambiguità

Un modello di analisi non deve contenere ambiguità, queste sono causate:

- inaccuratezza inerente nel linguaggio naturale
- assunzioni fatte dagli autori della specifica e non esplicitate
  - Una quantità specificata senza l'unità di misura
  - Un'ora senza la zona a cui si riferisce

Il processo di formalizzazione aiuta ad individuare

- aree di ambiguità
- inconsistenze
- omissioni

L'analisi dei requisiti ha come obiettivo quello di tradurre le specifiche dei requisiti in un modello del sistema formale o semi formale:

- Formalizzando e strutturando i requisiti si acquisisce maggiore conoscenza ed è possibile scoprire errori nelle richieste.
- La formalizzazione costringe a risolvere subito questioni difficili che altrimenti sarebbero rimandate.

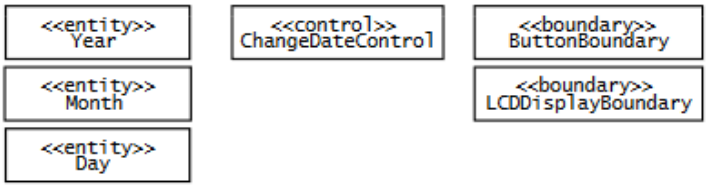
Poiché il modello di analisi può non essere comprensibile al cliente e all'utente è necessario modificare di conseguenza anche il documento di specifica delle richieste.

6.2 **Attività dell'analisi (trasformare un caso d'uso in oggetti)**

Le attività che andremo a descrivere sono le seguenti:

- Identificare gli oggetti entity
- Identificare gli oggetti boundary
- Identificare gli oggetti control
- Mappare i casi d'uso in oggetti con i sequence diagram
- Identificare le associazioni
- Identificare le aggregazioni
- Identificare gli attributi
- Modellare il comportamento e gli stati di ogni oggetto
- Rivedere il modello dell'analisi

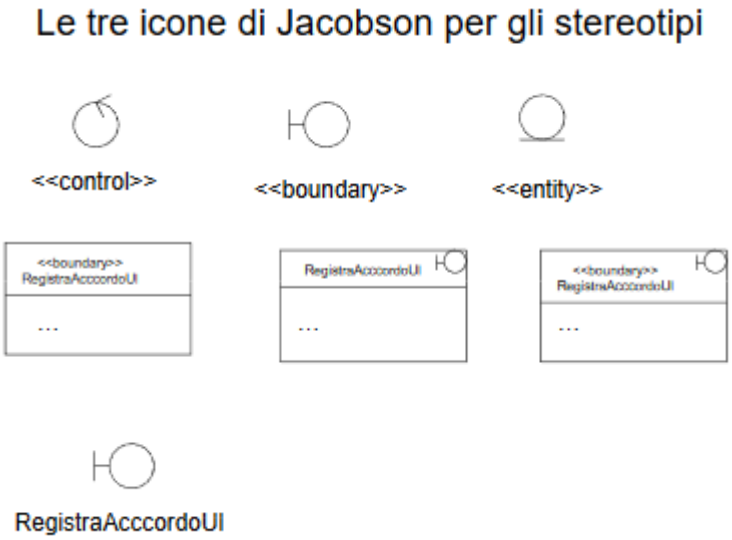
Nota: lo stereotipo sono uno strumento che consente a UML di poter estendere la sua capacità espressiva, infatti il suo obiettivo è di essere utilizzato per qualsiasi cosa. Si possono riconoscere dal fatto che sono rappresentati con le parentesi angolarei (<< >> un esempio è << include >>). Alcuni stereotipi sono già predefiniti ma alcuni li possiamo definire noi:



6.2.1 Notazioni

Sono possibili tre diverse notazioni:

- utilizzo dei simboli di stereotipi nei simboli di classe
- collasso del riquadro della classe nel solo simbolo di stereotipo
- uso combinato del nome di stereotipo e del simbolo



6.2.2 Identificare gli oggetti entity

Per identificare gli oggetti partecipanti al modello dell'analisi bisogna prendere in considerazione quelli identificati durante la specifica di requisiti.

Visto che il documento della specifica dei requisiti è scritto in linguaggio naturale, è frequente riscontrare imprecisioni nel testo, oppure dei sinonimi sulle notazioni che possono indurre gli sviluppatori a considerare male gli oggetti.

Gli sviluppatori possono limitare gli errori usando delle euristiche come quella di Abbott che detta le seguenti regole:

Testo	Modello ad oggetti
Nomi propri	Istanze
Nomi comuni	Classi
Verbi di fare	Operazioni
Verbi essere	Ereditarietà
Verbi avere	Aggregazioni
Verbi modali (es. deve essere)	Costanti
Aggettivi	Attributi

Vantaggio:

- Ci si focalizza sui termini dell'utente

Svantaggi:

- Il linguaggio naturale è impreciso, anche il modello ad oggetti derivato rischia di essere impreciso
- La qualità del modello dipende fortemente dallo stile di scrittura dell'analista
- Ci possono essere molti più sostantivi delle classi rilevanti, corrispondenti a sinonimi o attributi. Va bene per generare una lista iniziale di candidati a partire da una descrizione breve come il flusso di eventi di uno scenario o use case.

Oltre a quella di Abbott è possibile usare questa ulteriore euristica che suggerisce di dare peso alle seguenti caratteristiche dell'analisi dei requisiti:

- Termini che gli sviluppatori usano per comprendere meglio il caso d'uso
- Notazioni ricorrenti nei casi d'uso (sostantivi ricorrenti)
- Entità del mondo reale che il sistema deve tracciare
- Attività del mondo reale che il sistema deve tracciare
- Sorgenti o destinazioni di dati

Per ogni oggetto identificato,

- si assegna un nome (univoco) e una breve descrizione, per gli oggetti Entity è opportuno utilizzare gli stessi nomi utilizzati dagli utenti e dagli specialisti del dominio applicativo
- Si individuano attributi e responsabilità (non tutti, soprattutto non quelli ovvii)
- Il processo è iterativo e varie revisioni saranno richieste: quando il modello di analisi sarà stabile sarà necessario fornire una descrizione dettagliata di ogni oggetto

6.2.3 Identificare gli oggetti boundary

Gli oggetti boundary rappresentano l'interfaccia del sistema con l'attore. Ogni attore dovrebbe interagire con almeno un oggetto boundary.
Gli oggetti boundary raccolgono informazioni dall'attore e le traducono in un formato che può essere usato dagli oggetti entity e control.
Essi non descrivono in dettaglio gli aspetti visuali dell'interfaccia utente: ad esempio specificare scr oll-bar o menu-item può essere troppo dettagliato.
Per scovare gli oggetti boundary è possibile anche in questo caso usare un'euristica:

- Identificare il controllo dell'interfaccia utente di cui l'utente ha bisogno per iniziare un caso d'uso
- Identificare i moduli di cui gli utenti hanno bisogno per inserire dati nel sistema
- Identificare messaggi e notifiche che il sistema deve fornire all'utente
- Non modellare aspetti visuali dell'interfaccia con oggetti boundary
- Usare sempre il termine utente finale per descrivere le interfacce.

6.2.4 Identificare gli oggetti controllo

Gli oggetti Control sono responsabili del coordinamento tra gli oggetti entity e boundary e lo scopo è quello di prendere informazioni dagli oggetti boundary e inviarli agli oggetti entità. Un oggetto control viene creato all'inizio di un caso d'uso e termina alla fine di questo. Anche questo come gli oggetti entità e boundary si basa su delle euristiche:

- Identificare un oggetto control per ogni caso d'uso
- Identificare un oggetto control per ogni attore nel caso d'uso
- La vita di un oggetto control deve corrispondere alla durata di un caso d'uso o di una sessione utente.

6.2.5 Mappare casi d'uso in oggetti con sequence diagrams

Mappare i casi d'uso in sequence diagram serve per mostrare il comportamento tra gli oggetti partecipanti e ne mostrano l'interazione.
I sequence diagram non sono comprensibili all'utente ma sono uno strumento più preciso di supporto agli sviluppatori.

In un sequence diagram:

- Le colonne rappresentano gli oggetti che partecipano al caso d'uso
- La prima colonna rappresenta l'attore che inizia il caso d'uso
- La seconda colonna è l'oggetto boundary con cui l'attore interagisce per iniziare il caso d'uso
- La terza colonna è l'oggetto control che gestisce il resto del caso d'uso
- Gli oggetti control creano altri oggetti boundary e possono interagire con altri oggetti Control
- Le frecce orizzontali tra le colonne rappresentano messaggi o stimoli inviati da un oggetto ad un altro
- La ricezione di un messaggio determina l'attivazione di un'operazione
- L'attivazione è rappresentata da un rettangolo da cui altri messaggi possono prendere origine
- La lunghezza del rettangolo rappresenta il tempo durante il quale l'operazione è attiva
- Il tempo procede verticalmente dall'alto al basso
- Al top del diagramma si trovano gli oggetti che esistono prima del 1° messaggio inviato
- Oggetti creati durante l'interazione sono illustrati con il messaggio <<create>>
- Oggetti distrutti durante l'interazione sono evidenziati con una croce
- La linea tratteggiata indica il tempo in cui l'oggetto può ricevere messaggi

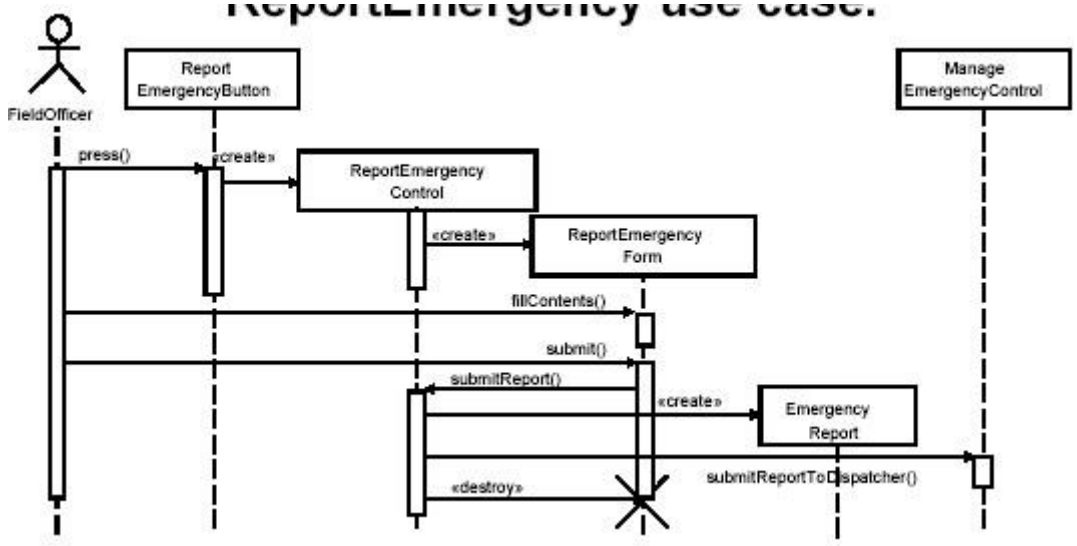


Figura 9 Esempio di sequence diagram

Mediante i sequence diagram è possibile trovare comportamenti o oggetti mancanti. Nel caso manchi qualche entità è necessario ritornare ai casi d'uso, ridefinire le parti mancanti e tornare a questa fase per ricreare il sequence diagram.

6.2.6 Identificare le associazioni

Un'associazione è una relazione tra due o più oggetti/classi. Ogni associazione ha un nome, un ruolo ad ogni capo dell'arco che identifica la funzione di ogni classe rispetto all'associazione e una molteplicità che indica ad ogni capo il numero di istanze possibili (vedi UML per dettagli).
Un'utile euristica per trovare le associazioni è la seguente:

- Esaminare i verbi nelle frasi
- Nominare in modo preciso i nomi delle associazioni e i ruoli
- Eliminare associazioni che possono essere derivate da altre associazioni
- Troppe associazioni rendono il modello degli oggetti “illeggibile”

6.2.7 Identificare le aggregazioni
Identifica che un oggetto è parte di un altro oggetto o lo contiene (vedi UML per dettagli).

6.2.8 Identificare gli attributi
Gli attributi sono proprietà individuali degli oggetti/classi. Ogni attributo ha un nome, una breve descrizione e un tipo che ne descrive i possibili valori.
L’euristica di Abbott indica che gli attributi possono essere identificati nel linguaggio naturale del documento delle specifiche prendendo in considerazione gli aggettivi.

6.2.9 Modellare il comportamento e gli stati di ogni oggetto
Gli oggetti che hanno un ciclo di vita più lungo dovrebbero essere descritti in base agli stati che essi possono assumere. Per fare ciò vengono usati i diagrammi a stati.

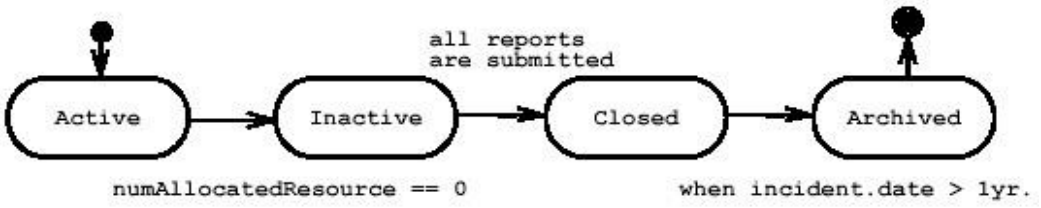


Figura 10 Un esempio di diagramma a stati

6.2.10 Rivedere il modello dell’analisi
Una volta che il modello dell’analisi non subisce più modifiche o ne subisce raramente è possibile passare alla fase si revisione del modello. La revisione del modello deve essere fatta prima dagli sviluppatori e poi insieme dagli sviluppatori e gli utenti.
L’obiettivo di questa attività di revisione è stabilire che la specifica risulta essere: corretta, completa, consistente e chiara,

Domande da porsi per assicurarsi della correttezza:
– Il glossario è comprensibile per gli utenti?
– Le classi astratte corrispondono a concetti ad alto livello?
– Tutte le descrizioni concordano con le definizioni degli utenti?
– Oggetti Entity e Boundary hanno nomi significativi?
– Oggetti control e use case sono nominati con verbi significativi del dominio?
– Tutti gli errori/eccezioni sono descritti e trattati?

Domande da porsi per assicurarsi della completezza:
– Per ogni oggetto: è necessario per uno use case? In quale use case è creato? modificato? distrutto? Può essere acceduto da un oggetto boundary?
– Per ogni attributo: quando è settato? Quale è il tipo?
– Per ogni associazione: quando è attraversata? Perché ha una data molteplicità?
– Per ogni oggetto control: ha le associazioni necessarie per accedere agli oggetti che partecipano nel corrispondente use case?

Domande da porsi per assicurarsi della consistenza:
– Ci sono classi o use case con lo stesso nome?
– Ci sono entità con nomi simili e che denotano concetti simili?

Domande da porsi per assicurarsi della chiarezza:
– Le richieste di performance specificate sono state assicurate?
– Può essere costruito un prototipo per assicurarsi della fattibilità?

7 System design

La progettazione di un sistema (System Design) è la trasformazione del modello di analisi nel modello di progettazione del sistema progettazione del sistema.

7.1 **Scopi criteri e architetture**

Gli scopi del system design sono quelli di definire gli obiettivi di progettazione del sistema, decomporre il sistema in sottosistemi più piccoli in modo da poterli assegnare a team individuali e selezionare alcune strategie quali:

- Scelte hardware e software
- Gestione dei dati persistenti
- Il flusso di controllo globale
- Le politiche di controllo degli accessi
- La gestione delle condizioni boundary (startup, shutdown, eccezioni)

Il system design si focalizza sul dominio di implementazione, prende in input il modello di analisi e dopo averlo trasformato da in output un modello del sistema.
Le attività del system design globalmente possono essere divise in tre fasi:

7.2 **Identificare gli obiettivi di design**

In questa fase gli sviluppatori definiscono le priorità delle qualità del sistema. Molti obiettivi possono essere ricavati utilizzando requisiti non funzionali o dal dominio dell'applicazione, altri vengono forniti direttamente dal cliente. Per ottenere gli obiettivi finali vanno seguiti dei criteri di progettazione tenendo presente performance, affidabilità, costi, mantenimento e utente finale.

- **Criteri di performance:** In questi criteri vengono inclusi: tempo di risposta, throughput (quantità di task eseguibili in un determinato periodo di tempo) e memoria.
- **Criteri di affidabilità:** L'affidabilità include criteri di robustezza (capacità di gestire condizioni non previste), attendibilità (non ci deve essere differenza tra il comportamento atteso e quello osservato), disponibilità (tempo in cui il sistema è disponibile per l'utilizzo), tolleranza ai fault (capacità di operare in condizioni di errore), sicurezza, fidatezza (capacità di non danneggiare vite umane).
- **Criteri di costi:** Vanno valutati i costi di sviluppo del sistema, alla sua installazione e al training degli utenti, eventuali costi per convertire i dati del sistema precedente, costi di manutenzione e costi di amministrazione.
- **Criteri di mantenimento:** Tra i criteri di mantenimento troviamo: estendibilità, modificabilità, adattabilità, portabilità, leggibilità e tracciabilità dei requisiti.
- **Criteri di utente finale:** In criteri dell'utente finale da tenere in considerazione sono quelli di utilità (quando bene il sistema dovrà facilitare e supportare il lavoro dell'utente) e quelli di usabilità.

Spesso quando si progetta un sistema non è possibile rispettare tutti i criteri di qualità contemporaneamente, viene quindi utilizzata una strategia di trade-off (compromesso) e data una priorità maggiore ad alcuni criteri tenendo presente scelte manageriali quali scheduling e budget.

7.3 **Decomposizione del sistema in sottosistemi**

Utilizzando come base i casi d'uso e l'analisi e seguendo una particolare *architettura software* (MVC, Client-Server ecc.) il sistema viene decomposto in sottosistemi.
Lo scopo è quello di poter assegnare a un singolo sviluppatore o ad un team parti software semplici.
In questa fase viene descritto come i sottosistemi sono collegati alle classi.

Un sottosistema è caratterizzato dai servizi (insieme di operazioni) che esso offre agli altri sottosistemi. L'insieme di servizi che un sistema espone viene chiamato interfaccia (API: *application programming interface*) che include, per ogni operazione: i parametri, il tipo e i valori di ritorno. Le operazioni che essi svolgono vengono descritte ad alto livello senza entrare troppo nello specifico.

Il sistema va diviso in sottosistemi tenendo presente queste due proprietà:

7.3.1 Accoppiamento (coupling)

Misura quanto un sistema è dipendente da un altro.
Due sistemi si dicono **loosely coupled** (leggermente accoppiati) se una modifica in un sottosistema avrà poco impatto nell'altro sistema, mentre si dicono **strongly coupled** (fortemente accoppiati) se una modifica su uno dei sottosistemi avrà un forte impatto sull'altro.
La condizione ideale di accoppiamento è quella di tipo loosely in quanto richiede meno sforzo quando devono essere modificate delle componenti.
Se ad esempio, tre componenti usano lo stesso servizio esposto da una componente che potrebbe essere modificata spesso, conviene frapporre tra di essere una nuova componente che ci permette di evitare una modifica alle tre componenti che usufruiscono del servizio.

Ad esempio una decomposizione di questo tipo

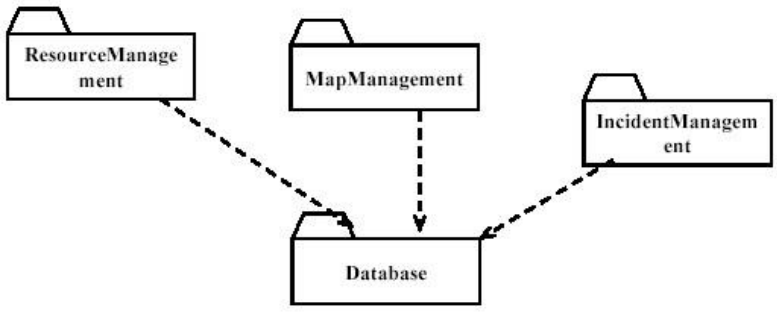


Figura 11 Composizione dei sottosistemi prima dell'aggiunta di una componente

potrebbe diventare

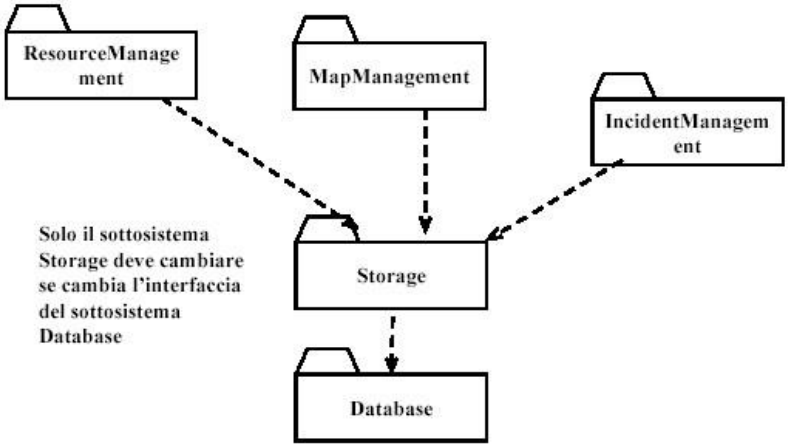


Figura 12 Composizione dopo l'aggiunta della componente Storage

Ovviamente ove non sono presenti componenti che si pensa debbano essere modificate spesso, non conviene utilizzare questa strategia in quanto aggiungerebbe complessità di sviluppo e di calcolo al sistema.

7.3.2 Coesione e Coupling

Misura la dipendenza tra le classi contenute in un sottosistema.
La coesione è alta se due componenti di un sottosistema realizzano compiti simili o sono collegate l'una con l'altra attraverso associazioni (es. ereditarietà), è invece bassa nel caso contrario. L'ideale sarebbe quello di avere sottosistemi con coesione interna alta.
Il coupling, misura la dipendenza fra due sottosistemi. Se due sistemi sono scarsamente accoppiati (loosely coupled) sono relativamente indipendenti: modifiche su uno dei sottosistemi avranno probabilmente pochi impatti sull'altro.
Se due sistemi sono fortemente accoppiati (strongly coupled), una modifica su di un sottosistema probabilmente avrà impatti sull'altro.

La decomposizione del sistema avviene utilizzando layer e/o partizioni.

7.3.3 Divisione del sistema con i layer

Con la decomposizione in layer il sistema viene visto come una gerarchia di sottosistemi. Un layer è un raggruppamento di sottosistemi che forniscono servizi correlati. I layer per implementare un servizio potrebbero usare a sua volta servizi offerti dai layer sottostanti ma non possono usare servizi dei livelli più alti.
Con i layer si possono avere due tipi di **architettura**: **chiusa** e **aperta**.
Con l'architettura chiusa un layer può accedere solo alle funzionalità del layer immediatamente a lui sottostante, con quella aperta il layer può accedere alle funzionalità del layer sottostante e di tutti gli altri sotto di esso.
Nel primo caso si ottiene un alta manutentibilità e portabilità, nel secondo una maggiore efficienza in quanto si risparmia l'overhead delle chiamate in cascata.

7.3.4 Divisione del sistema con le partizioni

Il sistema viene diviso in sottosistemi paritari (peer), ognuno dei quali è responsabile di diverse classi di servizi.

In generale una decomposizione di un sistema avviene utilizzando ambedue le tecniche. Infatti il sistema viene prima diviso in sottosistemi tramite le partizioni e successivamente ogni partizione viene organizzata in layer finché i sottosistemi non sono abbastanza semplici da essere sviluppati da un singolo sviluppatore o team.

7.3.5 Sottosistemi e classi

Per ridurre la complessità della soluzione, decomponiamo il sistema in parti più piccole, chiamate sottosistemi. Un sottosistema è costituito da un certo numero di classi del dominio della soluzione dominio della soluzione.
Decomponendo il sistema in sottosistemi relativamente indipendenti, i team di progetto possono lavorare sui sottosistemi individuali con un minimo overhead di comunicazione (è importante che siano definiti gli obiettivi di design chiari a tutti i sottoteam). Nel caso di sottosistemi complessi, applichiamo ulteriormente questo principio e li decomponiamo in sottosistemi più semplici.

7.3.6 Servizi e interfacce di sottosistemi

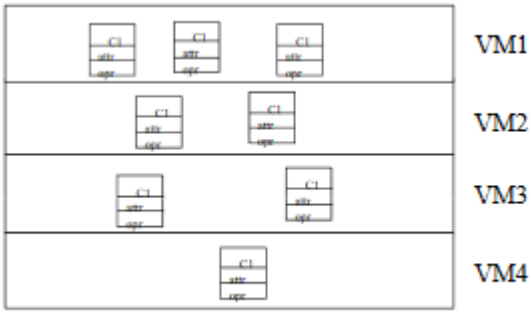
Un sottosistema è caratterizzato dai servizi che fornisce agli altri sottosistemi. Un servizio è un insieme di operazioni correlate fornite dal sottosistema per uno specifico scopo. L'insieme di operazioni di un sottosistema che sono disponibili agli altri sottosistemi forma l'interfaccia del sottosistema. L'interfaccia di un sottosistema include il nome delle operazioni, i loro parametri, il loro tipo ed i loro valori di ritorno.
Il system design si focalizza sulla definizione dei servizi forniti da ogni sotto sistema in termini di:

- Operazioni
- Loro parametri
- Loro comportamento ad alto livello

L'Object design si focalizza sulle operazioni, l'Application Programmer Interface (API), che raffina ed estende le interfacce, definendo i tipi dei parametri ed i valori di ritorno di ogni operazione.

7.3.7 Macchina virtuale (Dijkstra)

Un sistema dovrebbe essere sviluppato da un insieme di macchine virtuali, ognuna costruita in termini di quelle al di sotto di essa. Una macchina virtuale è un'astrazione che fornisce un insieme di attributi e operazioni. Le macchine virtuali possono implementare due tipi di architetture software: architetture chiuse e architetture aperte.



7.3.7.1 Architettura chiusa (opaque layering)

Una macchina virtuale può solo chiamare le operazioni dello strato sottostante. Design goal: alta manutenibilità. Esempio TCP/IP

7.3.7.2 Architettura aperta (Trasparent layering)

Una macchina virtuale può utilizzare i servizi delle macchine dei layer sottostanti. Design goal: efficienza relativa al tempo di esecuzione (runtime).

7.3.7.3 Vantaggi e svantaggi delle architetture chiuse

Vantaggi

- Basso accoppiamento tra le componenti
- Integrazione e testing incrementale
- Integrazione e testing incrementale

Svantaggi

- Ogni livello aggiunge overhead in termini di tempo e memoria
- Diventa difficile soddisfare alcuni obiettivi di design (performance)
- Aggiungere funzionalità al sistema può essere difficile

7.3.8 Basic Layer pattern

Un sistema di Base contiene tipicamente i sottosistemi

- Interface: Il sottosistema Interface contiene gli oggetti boundary/interface, si scompone ulteriormente in user interface e system interface.
- Function: Il sottosistema Function contiene gli oggetti control e la logica dell'applicazione.
- Model: Il sottosistema Model contiene gli oggetti entity del dominio di applicazione.

7.4 Architetture software

Un'architettura software include scelte relative alla decomposizione in sottosistemi, flusso di controllo globale, gestione delle condizioni limite e i protocolli di comunicazione tra i sottosistemi. È da notare che la decomposizione dei sottosistemi è una fase molto critica in quando una volta iniziato lo sviluppo con una determinata decomposizione è complesso ed oneroso dover tornare indietro in quanto molte interfacce dei sottosistemi dovrebbero essere modificate. Alcuni stili architetturali che potrebbero essere usati sono:

7.4.1 Repository

Con questo stile tutti i sottosistemi accedono e modificano i dati tramite un oggetto repository. Il flusso di controllo viene dettato dal repository tramite un cambiamento dei dati oppure dai sottosistemi tramite meccanismi di sincronizzazione o lock.

I vantaggi di questo stile si vedono quando si implementano applicazioni in cui i dati cambiano di frequente poiché si evitano incoerenze. Considerando i problemi che questo stile può dare sicuramente si può notare che il repository può facilmente diventare un collo di bottiglia in termini di prestazioni e inoltre il coupling tra i sottosistemi e il repository e altissimo: una modifica all'API del repository comporta la modifica di tutti sottosistemi che lo utilizzano.

7.4.1.1 Vantaggi e svantaggi

Vantaggi:

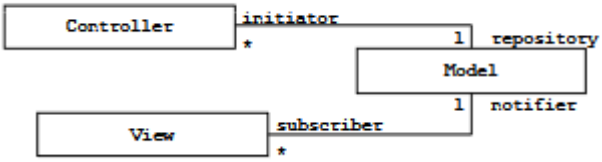
- I repository sono adatti per applicazioni con task di elaborazione dati che cambiano di frequente.
- Una volta che un repository centrale è stato definito, nuovi servizi nella forma di sottosistemi addizionali possono essere definiti facilmente.

Svantaggi:

- il repository centrale può facilmente diventare un collo di bottiglia per aspetti sia di prestazione, sia di modificabilità.
- Il coupling fra ogni sottosistema ed il repository è alto, così è difficile cambiare il repository senza avere un impatto su tutti i sottosistemi.

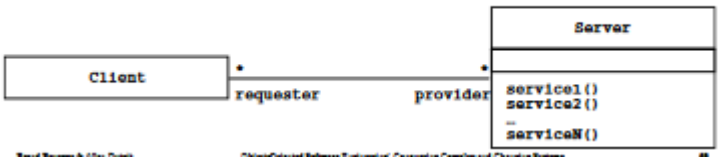
7.4.2 Model/View/Control (MVC)

Il sistema viene diviso in tre sottosistemi: Model, View e Control. Il sottosistema model implementa la struttura dati centrale, il controller gestisce il flusso di controllo (si occupa di prendere l'input dall'utente e di inviarlo al model), il view è la parte di interazione con l'utente.
Uno dei vantaggi di MVC si vede quando le interfacce utente (view) vengono modificate più di frequente rispetto alla conoscenza del dominio dell'applicazione (model). Per questo motivo MVC è l'ideale per sistemi interattivi e quando il sistema deve avere viste multiple.



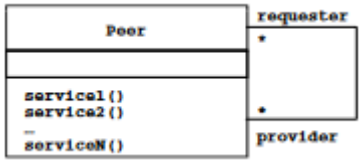
7.4.3 Client-Server

Il sottosistema server fornisce servizi ad una serie di istanze di altri sottosistemi detti client i quali si occupano dell'interazione con l'utente. La maggior parte della computazione viene svolta a lato server. Questo stile è spesso usato in sistemi basati su database in quanto è più facile gestire l'integrità e la consistenza dei dati.
I Client conoscono l'interfaccia del Server (i suoi servizi), mentre i Server non conoscono le interfacce dei Client. Gli utenti interagiscono solo con il Client. Il flusso di controllo nei client e nei server è indipendente.



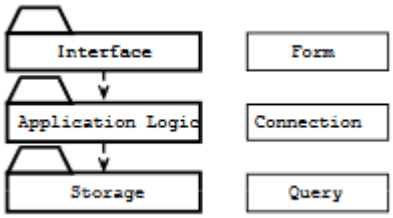
7.4.4 Peer-To-Peer

È una generalizzazione dello stile client-server in cui però client e server possono essere scambiati di ruolo ed ognuno dei due può fornire servizi.



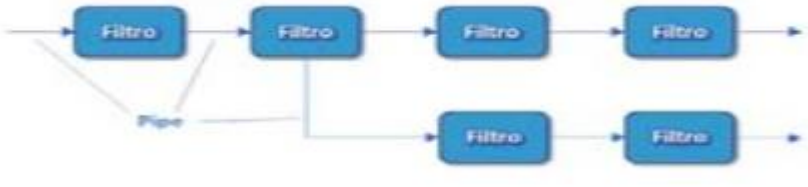
7.4.5 Three-Tier

I sottosistemi vengono organizzati in tre livelli hardware: *interface*, *application* e *storage*. Il primo conterrà tutti gli oggetti boundary di interazione con l'utente, il secondo include gli oggetti relativi al controllo e alle entità, il terzo effettua l'interrogazione e la ricerca di dati persistenti.



7.4.6 Architetture a flusso di dati (pipeline)

Questo stile architetturale si rivela efficace nel caso in cui si abbia un insieme di dati in input da trasformare attraverso una catena di componenti e di filtri software al fine di ottenere una serie di dati in output. Ogni componente della catena lavora in modo indipendente rispetto agli altri, trasforma i dati in input in dati in output e delega ai componenti successivi le ulteriori trasformazioni. Ogni parte è inconsapevole delle modalità di funzionamento dei componenti adiacenti.



7.4.7 Considerazioni finali

Quando si decidono le componenti di un sottosistema bisognerebbe tenere presente che la maggior parte dell'interazione tra le componenti dovrebbe avvenire all'interno di un sottosistema allo scopo di ottenere un'alta coesione.

7.4.8 Euristiche per scegliere le componenti

Le euristiche per scegliere le componenti dei sottosistemi sono le seguenti:

- Gli oggetti identificati in un caso d'uso dovrebbero appartenere ad uno stesso sottosistema.
- Bisogna creare dei sottosistemi che si occupano di trasferire i dati tra i sottosistemi
- Minimizzare il numero di associazioni tra i sottosistemi (devono essere loosely coupled)
- Tutti gli oggetti di un sottosistema dovrebbero essere funzionalmente correlati

7.5 **Descrizione delle attività del System Design**

Le attività del system design sono le seguenti:

- Mappare i sottosistemi su piattaforme e processori
- Identificare e memorizzare informazioni persistenti
- Stabilire i controlli di accesso
- Progettare il flusso di controllo globale
- Identificare le condizioni limite
- Rivedere il modello del system design

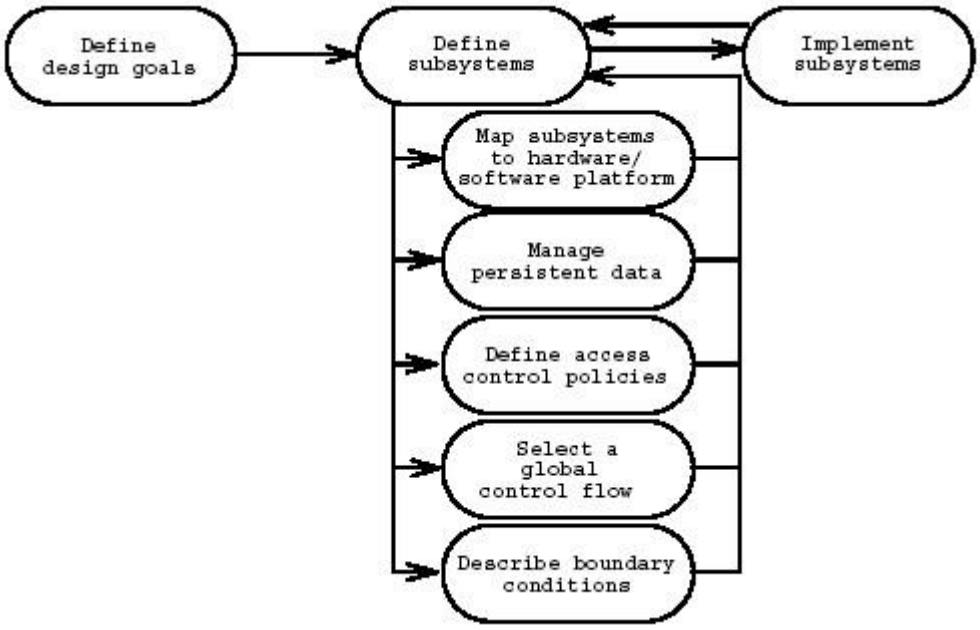


Figura 13 Attività del System Design

7.5.1 UML Deployment Diagram

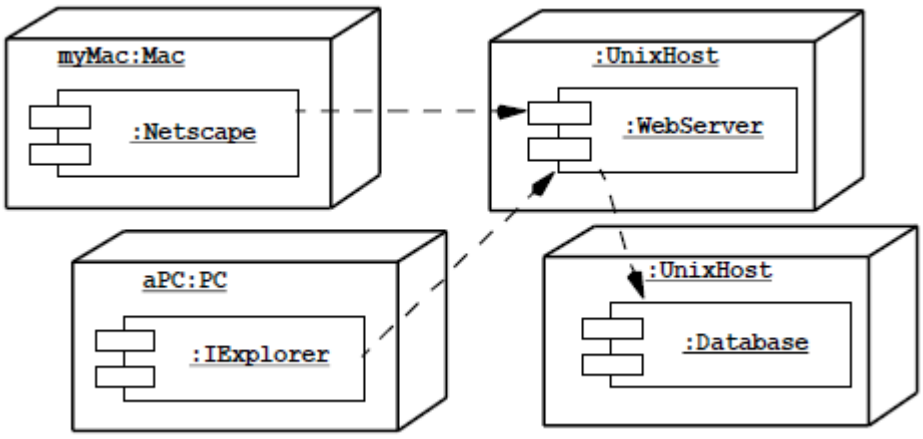
Sono utili per mostrare il progetto del sistema dopo che le seguenti decisioni sono state prese:

- **Decomposizione in sottosistemi**
- **Concorrenza**
- **Mapping Hardware/Software**

Il deployment diagram è un grafo di nodi connessi attraverso associazioni di comunicazione.

- **I nodi sono mostrati come box 3-D**
- **I nodi possono contenere istanze di componenti.**
- **Le componenti possono contenere oggetti (indica che un oggetto fa**
- **parte di una componente)**

Sono utilizzati per descrivere le relazioni tra le componenti runtime e i nodi hardware. Le componenti sono entità autonome (indipendenti) che forniscono servizi ad altre componenti o attori.



7.5.2 Mappare i sottosistemi su piattaforme e processori

Molti sistemi complessi necessitano di lavorare su più di un computer interconnessi da rete. L'uso di più computer può ottimizzare le performance e permettere l'utilizzo del sistema a più utenti distribuiti sulla rete.

In questa fase vanno prese alcune decisioni per quanto riguarda le piattaforme hardware e software su cui il sistema dovrà girare (es Unix vs Windows, Intel vs Sparc etc).
Una volta decise le piattaforme è necessario mappare le componenti su di esse. Questa operazione potrebbe portare all'introduzione di nuove componenti per interfacciare i sottosistemi su diverse piattaforme (es. una libreria per il collegamento ad un DBMS).
Sfortunatamente, da una parte, l'introduzione di nuovi nodi hardware distribuisce la computazione, dall'altro introduce alcune problematiche tra cui la sincronizzazione, la memorizzazione, il trasferimento e la replicazione di informazioni tra sottosistemi. Se l'architettura è distribuita abbiamo bisogno di descrivere la rete (e quindi il sottosistema di comunicazione).

7.5.2.1 Allocare oggetti e sistemi a nodi
Consente di distribuire le funzionalità e l'elaborazione dove è necessario. Consente di identificare nuovi oggetti e sottosistemi per spostare informazioni tra i nodi. Di contro, questo implica che bisogna fare scelte di memorizzazione, di trasferimento, e di sincronizzazione dell'informazione tra i vari sottosistemi.

7.5.3 Identificare e memorizzare i dati persistenti
Il modo in cui i dati vengono memorizzati può influenzare l'architettura del sistema (vedi lo stile architetturale repository) e la scelta di uno specifico database. In questa fase vanno identificati gli oggetti persistenti e scelto il tipo di infrastruttura da usare per memorizzarli (dbms, file o altro).
Gli oggetti entity identificati durante l'analisi dei requisiti sono dei buoni candidati a diventare persistenti. Non è detto però che tutti gli oggetti entità debbano diventare persistenti. In generale i dati sono persistenti se sopravvivono ad una singola esecuzione del sistema. Il sistema dovrà memorizzare i dati persistenti quando questi non servono più e ricaricarli quando necessario.
Una volta decisi gli oggetti dobbiamo decidere come questi oggetti devono essere memorizzati. Principalmente potremmo avere a disposizione tre mezzi: file, dbms relazionale e dbms ad oggetti.

7.5.3.1 File
La prima tipologia da una parte richiede una logica più complessa per la scrittura e lettura, dall'altra permette un accesso ai dati più efficiente.

7.5.3.2 DBMS relazionale
Un DMBS relazione fornisce un'interfaccia di più alto livello rispetto al file. I dati vengono memorizzati in tabelle ed è possibile utilizzare un linguaggio standard per le operazioni (SQL). Gli oggetti devono essere mappati sulle tabelle per poter essere memorizzati.

7.5.3.3 DBMS ad oggetti
Un database orientato ad oggetti è simile ad un DBMS relazionale con la differenza che non è necessario mappare gli oggetti in tabelle in quanto questi vengono memorizzati così come sono. Questo tipo di database riduce il tempo di setup iniziale (si risparmia sulle decisioni di mapping) ma sono più lenti e le query sono di più difficile comprensione.

7.5.3.4 Considerazioni e trade-offs
La scelta tra una tecnologia o un'altra per la memorizzazione può essere influenzata da vari fattori.
In particolare conviene usare un file in questi casi:
- Dimensione elevata dei dati (es. immagini, video ecc.)
- Dati temporanei e logging

Conviene invece usare un DBMS (relazionale e ad oggetti) in casi di:

- Accessi concorrenti (i DBMS effettuano controlli di consistenza e concorrenza bloccando i dati quando necessario)
- Uso dei dati da parte di più piattaforme
- Particolari politiche di accesso a dati

7.5.4 Stabilire i controlli di accesso

In un sistema multi utenza è necessario fornire delle politiche di accesso alle informazioni. Nell'analisi sono stati associati casi d'uso ad attori, in questa fase vanno definite in modo più preciso le operazioni e le informazioni effettuabili da ogni singolo attore e come questi si autenticano al sistema. È possibile rappresentare queste politiche tramite una matrice in tre modi:

- **Tabella di accesso globale**
  Ogni riga della matrice contiene una tripla (attore, classe, operazione). Se la tupla è presente per una determinata classe e operazioni l'accesso è consentito altrimenti no.
- **Access control list (ACL)**
  Ogni classe ha una lista che contiene una tupla (attore, operazione) che specifica se l'attore può accedere a quella determinata operazione della classe a cui la ACL appartiene.
- **Capability**
  Una capability è associata ad un attore ed ogni riga della matrice contiene una tupla (classe, operazione che l'attore a cui è associata può eseguire.

Scegliere una o l'altra soluzione impatta sulle performance del sistema. Ad esempio scegliere una tabella di accesso globale potrebbe far consumare molta memoria. Le altre vanno usate in base al tipo di controllo che vogliamo effettuare: se vogliamo rispondere più velocemente alla domanda "chi può accedere a questa classe?" useremo una ACL, se invece vogliamo rispondere più velocemente alla domanda "a quale operazione può eccedere questo attore?" useremo una capability.

7.5.5 Progettare il flusso di controllo globale

Un flusso di controllo è una sequenza di azioni di un sistema. In un sistema Object Oriented una sequenza di azioni include prendere decisioni su quali operazioni eseguire ed in che ordine. Queste decisioni sono basate su eventi esterni generati da attori o causati dal trascorrere del tempo. Esistono tre tipi di controlli di flusso:

7.5.5.1 Procedure-driven control

Le operazioni rimangono in attesa di un input dell'utente ogni volta che hanno bisogno di elaborare dati. Questo tipo di controllo di flusso è particolarmente usato in sistemi legacy di tipo procedurale.

7.5.5.2 Event-driven control

In questo controllo di flusso un ciclo principale aspetta il verificarsi di un evento esterno. Non appena l'evento diventa disponibile la richiesta viene direzionata all'opportuno oggetto. Questo tipo di controllo ha il vantaggio di centralizzare tutti gli input in un ciclo principale ma ha lo svantaggio di rendere complessa l'implementazione di sequenze di operazioni composte di più passi.

7.5.5.3 Threads

Questo controllo di flusso è una modifica del procedure-driven control che aggiunge la gestione della concorrenza. Il sistema può creare un arbitrario numero di threads (processi leggeri), ognuno assegnato ad un determinato evento.

Se si sceglie di usare un control-flow di tipo threads bisogna stare attenti a gestire situazioni di concorrenza in quando più thread possono accedere contemporaneamente alle stesse risorse e creare situazioni non previste.

7.5.6 Identificare le condizioni limite

Le condizioni limite del sistema includono lo **startup**, lo **shutdown**, l'inizializzazione e la gestione di fallimenti come corruzione di dati, caduta di connessione e caduta di componenti.
A tale scopo vanno elaborati dei casi d'uso che specificano la sequenza di operazioni in ciascuno dei casi sopra elencati.
In generale *per ogni oggetto persistente*, si esamina in quale caso d'uso viene creato e distrutto. Se l'oggetto non viene creato o distrutto in nessun caso d'uso deve essere aggiunto un caso d'uso invocato dall'amministratore.
Per ogni componente vanno aggiunti tre casi d'uso per l'avvio, lo shutdown e per la configurazione.
Per ogni tipologia di fallimento delle componenti bisogna specificare come il sistema si accorge di tale situazione, come reagisce e quali sono le conseguenze.
Un'eccezione è un evento o errore che si verifica durante l'esecuzione del sistema. Una situazione del genere può verificarsi in tre casi:

- Un fallimento hardware (dovuto all'invecchiamento dell'hardware)
- Un cambiamento nell'ambiente (interruzione di corrente)
- Un fallimento del software (causato da un errore di progettazione)

Nel caso in cui un errore dipenda da un input errato dell'utente, tale situazione deve essere comunicata all'utente tramite un messaggio così che lo stesso possa correggere l'input e riprovare. Nel caso di caduta di un collegamento il sistema dovrebbe salvare lo stato del sistema in modo da poter riprendere l'esecuzione non appena il collegamento ritorna.

7.5.7 Rivedere il modello del system design

Un progetto di sistema deve raggiungere degli obbiettivi e bisogna assicurarsi che rispetti i seguenti criteri:

- **Correttezza:** Il system design è corretto se il modello di analisi può essere mappato su di esso.
- **Completezza:** La progettazione di un sistema è completa se ogni requisito e ogni caratteristica è stata portata a compimento.
- **Consistenza**: Il system design è consistente se non contiene contraddizioni.
- **Realismo:** Un progetto è realistico se il sistema può essere realizzato ed è possibile rispettare problemi di concorrenza e tecnologie.
- **Leggibilità:** Un system design è leggibile se anche sviluppatori non coinvolti nella progettazione possono comprendere il modello realizzato.

7.5.8 Gestione del system design

La gestione del system design coinvolge le seguenti attività:

- Documentazione del System Design (SDD)
- Assegnazione delle responsabilità
- Iterazione delle attività

La documentazione del system design può essere fatta seguendo questo template:

1. Introduction
    - 1.1. Purpose of the system
    - 1.2. Design Goals
    - 1.3. Definition, acronyms, and abbreviations
    - 1.4. References
    - 1.5. Overview
2. Current software architecture
3. Proposed software architecture
    - 3.1. Overview
    - 3.2. Subsystem decomposition
    - 3.3. Hardware/software mapping
    - 3.4. Persistent data management
    - 3.5. Access control and security
    - 3.6. Global software control
    - 3.7. Boundary conditions
4. Subsystems services

Glossary

La suddivisione in sottosistemi effettuata nel system design influenza le decisioni sulla quantità di personale necessario per portare a termine il progetto e su come dividere i team di sviluppo. Un team particolare, l'*architecture team* si occupa di suddividere il sistema in sottosistemi e di assegnare le responsabilità ai singoli team o ai singoli sviluppatori.
Un'altra figura importante è quella dell'*architetto* che deve assicurare la consistenza delle decisioni e dello stile delle interfacce.
La stesura del system design è effettuata in modo iterativo. Infatti alla fine di ogni stesura è possibile effettuare delle modifiche al modello relativamente a:

- Decomposizione in sottosistemi
- Interfacce
- Condizioni eccezionali

Descrizioni breve dell template:

1. Introduction: breve paronamica sull'architettura del software e i design goals
- **1.1. Purpose of the system**
- **1.2. Design Goals**
- **1.3. Definition, acronyms, and abbreviations**
- **1.4. References:** riferimenti agli altri documenti e tracciabilità delle informazioni
- **1.5. Overview**

2. Current software architecture: l'architetttura con cui il sistema sarà sostituita, se è la prima volta che viene creato il sistema indicare un'architettura simile.

3. Proposed software architecture: documentazione del modello del nuovo system design del sistema
- **3.1. Overview:** descrizione ad alto livello dell'architettura e delle funzionalità dei sottosistemi
- **3.2. Subsystem decomposition:** descrizione dei sottosistemi
- **3.3. Hardware/software mapping:** descrizione di quali sotto sistemi sono assegnati ai componenti hardware e software.
- **3.4. Persistent data management:** decrizione di come saranno memorizzati i dati persistenti.

3.5. Access control and security: descrive il sistema in termine di matrice di accesso.
3.6. Global software control: descrive come il sistema software sarà controllato, in particolare sincronizzazione e concorrenza.
3.7. Boundary conditions: descrive lo start-uo, shutdowns e errori del sistema.
4. Subsystems services: descrive i servici offerti da ogni sottosistema in termini di operazioni.
4.1. Glossary

8 Object design

L'Object design è il processo che si occupa di:
- aggiungere dettagli all'analisi dei requisiti e
- prendere decisioni di implementazione

L'object designer deve scegliere fra diversi modi di implementare il modello di analisi con l'obiettivo di minimizzare il tempo di esecuzione, la memoria ed altri costi. Con l'Object Design: Iteriamo nel processo di assegnazione delle operazioni al modello ad oggetti, infatti serve come base dell'implementazione.
Le fasi dell'object design sono le seguenti:
- **Applicare i concetti di riuso:** Include l'uso di componenti off-the-shelf (riutilizzate), l'applicazione di design pattern specifici per risolvere problematiche comuni e la creazione di relazioni di ereditarietà.
- **Specificare le interfacce dei servizi:** I servizi dei sottosistemi identificati nel system design vengono dettagliati in termini di interfacce di classe, includendo operazioni, argomenti, firme di metodi ed eccezioni. Vengono anche aggiunte eventuali operazioni o oggetti necessari a trasferire i dati tra i sottosistemi.
- **Ristrutturare il modello ad oggetti:** La ristrutturazione manipola il modello del sistema per incrementare il riuso di codice, trasforma associazioni N-arie in binarie e associazioni binarie in semplici riferimenti, trasforma classi semplici in attributi di tipo predefinito ecc.
- **Ottimizzare il modello ad oggetti:** L'ottimizzazione ha lo scopo di migliorare le performance del sistema, aumentando la velocità, riducendo l'uso di memoria e diminuendo la molteplicità delle associazioni.

8.1 Concetti di riuso

8.1.1 Oggetti di applicazione e oggetti di soluzione

Gli oggetti possono essere divisi in due tipologie:
- **Oggetti di applicazione** (chiamati anche oggetti del dominio)
  Rappresentano concetti del dominio dell'applicazione che sono rilevanti nel sistema (la maggior parte degli oggetti entity sono di questo tipo)
- **Oggetti di soluzione**
  Sono oggetti che non sono mappabili su concetti relativi al dominio dell'applicazione e servono a rendere funzionale il sistema. Un esempio di oggetti di questo tipo identificati durante l'analisi sono gli oggetti boundary e control.

8.1.2 Ereditarietà di specifica ed ereditarietà di implementazione

L'ereditarietà usata al solo scopo di riusare codice è detta di *implementazione*. Con questo tipo di ereditarietà gli sviluppatori possono riusare codice in modo veloce estendendo una classe esistente e rifinendo il suo comportamento. Un esempio di questo tipo di ereditarietà può essere la definizione di una collezione come un insieme ereditando da una tabella hash.

L’ereditarietà di *specifica* crea una gerarchia di concetti mappabili in una vera gerarchia (es. un insieme è una collezione ma un insieme non è una tabella hash).

8.1.3 Delegazione

La delegazione è un’alternativa all’ereditarietà di implementazione applicabile quando si vuole riusare codice. Nell’esempio precedente in cui un insieme eredita da una tabella hash, è possibile usare la delegazione eliminando la relazione di ereditarietà e includendo nella classe insieme un’istanza dell’oggetto tabella hash.
In generale ogni qual volta ci troviamo di fronte ad ereditarietà di implementazione è meglio usare la delegazione. L’uso della delegazione porta alla scrittura di codice più robusto e non interferisce con le componenti già esistenti.

8.1.4 Il principio di sostituzione di Liskov

Il principio di Liskov asserisce che se un oggetto di tipo S può sostituire un oggetto di tipo T in qualunque posto in cui ci si aspetta di trovate T, allora S può essere definito un sottotipo di T.
In altre parole l’oggetto di tipo S è sottotipo di T se esso non sovrascrive metodi di T cambiandone il comportamento atteso.

8.1.5 Delegazione ed ereditarietà nei design pattern

In generale non è sempre chiaro e facile da interpretare quando conviene usare la delegazione o l’ereditarietà. I design pattern sono di template di soluzioni a problemi comuni, raffinate nel tempo dagli sviluppatori.
Un design pattern ha quattro elementi: un nome, una descrizione del problema che risolve, una soluzione, delle conseguenze a cui porta.

8.2 **Attività del riuso** (selezionare i design pattern e le componenti)

Anticipare i cambiamenti del sistema è una caratteristica molto importante della progettazione. Spesso i cambiamenti tendono ad essere gli stessi per più tipologie di sistemi. Alcuni cambiamenti possibili includono:

- **Nuovo produttore o nuova tecnologia:** Spesso le componenti usate per costruire il sistema vengono sostituite da altre di diversi venditori o da componenti che rispecchiano il cambiamento del trend del mercato.
- **Nuove implementazioni:** Quando i sistemi vengono integrati è possibile che ci si renda conto che il sistema è non è abbastanza performante.
- **Nuove interfacce grafiche:** La poca usabilità del software rende necessario riprogettare l’intera interfaccia grafica.
- **Nuova complessità nel dominio di applicazione:** Nel dominio di applicazione è necessario aggiungere alcune caratteristiche che rendono il sistema più complesso (es. passare da un sistema a singolo utente ad un sistema multi utente).
- **Errori:** Molti errori vengono trovati solo quando gli utenti iniziano ad usare il prodotto software.

I design pattern attraverso l’uso di delegazione, ereditarietà e classi astratte, possono aiutare ad anticipare questo tipo di cambiamenti.
Ogni design pattern che andremo ad esporre risolve uno dei problemi prima esposti.

8.2.1 Classificazione dei design pattern

- Pattern strutturali: Si occupano delle modalità di composizione di classi e oggetti per formare strutture complesse. Riducono l'accoppiamento tra due o più classi, incapsulano strutture complesse, introducono una classe astratta per permettere estensioni future.
- Pattern comportamentali: Si occupano di algoritmi e dell'assegnazione delle responsabilità tra oggetti che collaborano tra loro (*chi fa che cosa?).* Caratterizzano i flussi di controllo complessi difficili da seguire a run-time.
- Pattern creazionali: Forniscono un'astrazione del processo di creazione degli oggetti. Aiutano a rendere un sistema indipendente dalle modalità di creazione, composizione e rappresentazione degli oggetti utilizzati.

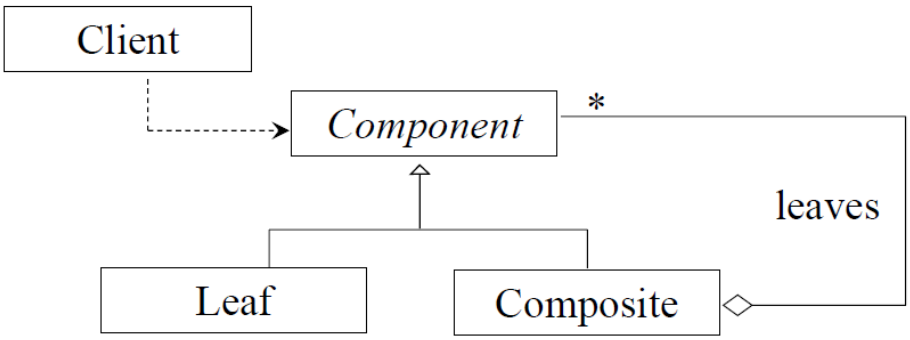
8.2.2 Pattern Strutturali

I pattern strutturali risolvono problematiche inerenti la struttura delle classi e degli oggetti.

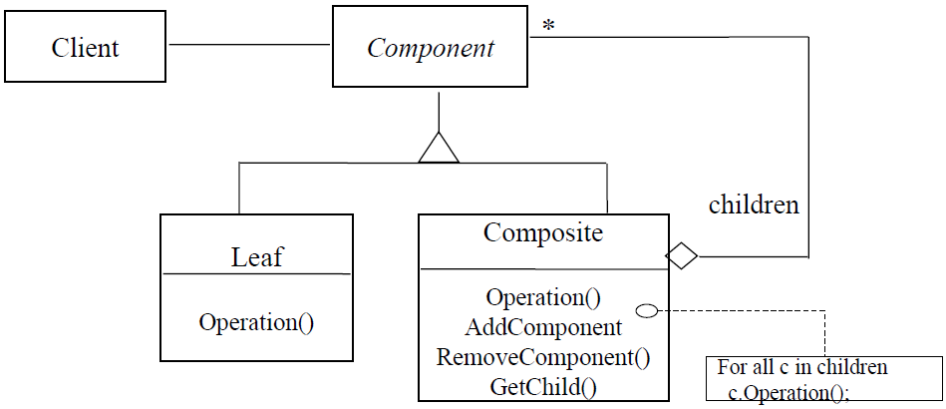
8.2.2.1 Composite pattern

Nome: Composite.

Descrizione del problema: compone gli oggetti in strutture ad albero di ampiezza e profondità variabile per rappresentare gerarchie tutto-parte; permette al client di trattare gli oggetti individuali (foglie) e gli oggetti composti (nodi interni) in maniera uniforme attraverso un'interfaccia comune. (È una struttura ad albero per rappresentare oggetti complessi).

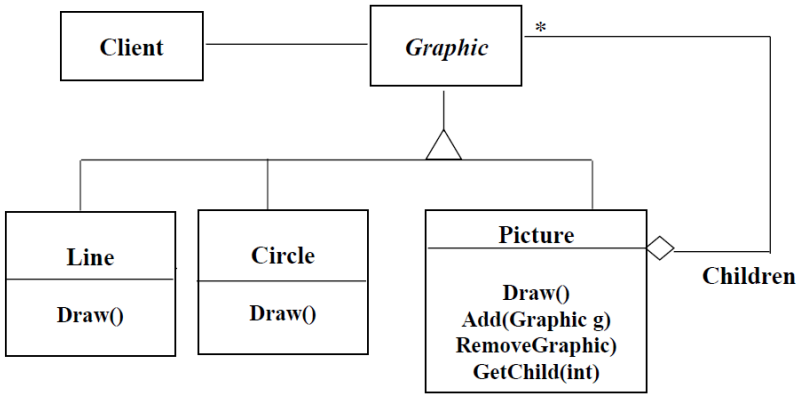


Soluzione: l'interfaccia *Component* specifica i servizi che sono condivisi tra *Leaf* e *Composite*; un Composite ha un'associazione di aggregazione con Component e implementa ogni servizio iterando su ogni Component che contiene. I Leaf fanno il lavoro effettivo.

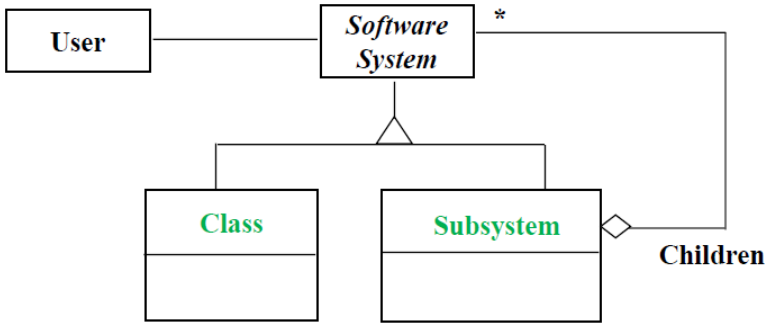


Conseguenze: il Client usa lo stesso codice per gestire i Leaf o i Composite; i comportamenti specifici di un Leaf possono essere cambiati senza cambiare la gerarchia; nuove classi di Leaf possono essere aggiunte senza cambiare la gerarchia.

Esempio: le applicazioni grafiche usano il pattern composite.



- **Vogliamo modellare un sistema software con il pattern composite:**
  - ***Definizione:*** A software system consists of subsystems which are either other Subsystems or collection of Classes
  - ***Leaf node:*** **Class**
  - ***Composite:*** **Subsystem**

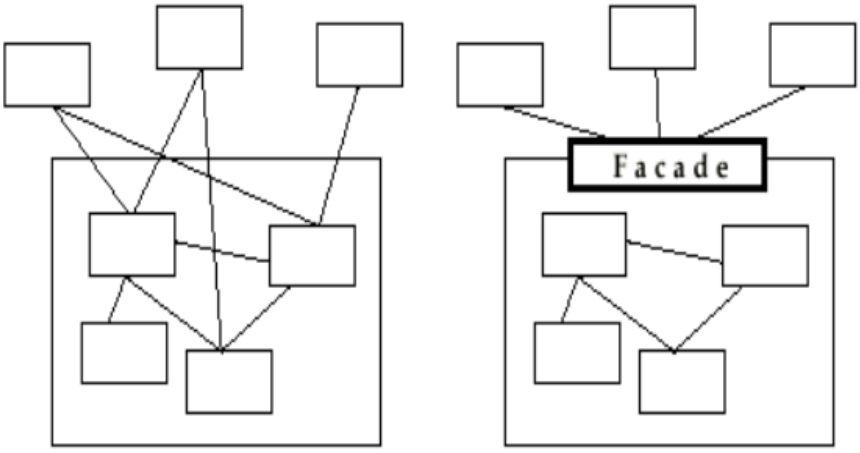


8.2.2.2 Facade Pattern

Nome: Facade.

Descrizione del problema: Fornisce un'unica interfaccia per accedere ad un insieme di oggetti che compongono un sottosistema.

Possibili applicazioni: Un facade pattern dovrebbe essere utilizzato da tutti i sottosistemi in un sistema software. Il facade definisce tutti i servizi del sottosistema. Il facade ci permette di fornire un'architettura chiusa.



Conseguenze: Fornisce un'interfaccia unificata per interfacciarsi in un sottosistema. Definisce un'interfaccia ad alto livello che rende il sottosistema più facile da utilizzare. Protegge il client dalle componenti del sottosistema. Crea accoppiamento debole tra client e le componenti del sistema.

8.2.2.3 Adapter Pattern

Nome: Adapter (Wrapper).

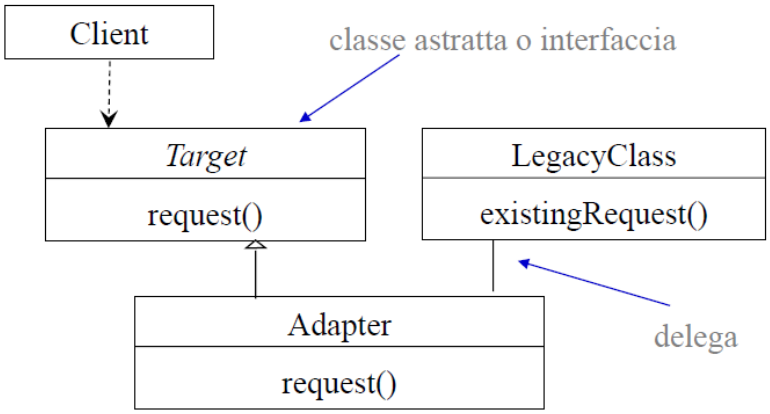
Descrizione del problema: Convertire l'interfaccia di una classe in un'interfaccia diversa che il cliente si aspetta, in maniera tale che classi diverse possano operare insieme nonostante abbiamo interfacce incompatibili. (Unisce le interfacce di differenti classi).

Possibili applicazioni:

- Incapsulare componenti esistenti per riutilizzare componenti da progetti precedenti o usare componenti off-the-shelf (COTS).
- Fornire una nuova interfaccia a componenti legacy esistenti (interface engineering, reengineering).

Esempio: Come utilizzare un Web Browser per accedere ad un sistema di information retrieval esistente?

Soluzione: Ogni metodo dell'interfaccia *Target* è implementato in termini di richieste alla classe *Legacy*. Ogni conversione tra strutture dati o variazioni del comportamento sono realizzate dalla classe *Adapter*.



Conseguenze: Se il Client utilizza il *Target* allora può utilizzare qualsiasi istanza dell'*Adapter* in maniera trasparente senza dover essere modificato. L'adapter lavora con la classe *Legacy* e con tutte le sue sottoclassi. L'Adapter pattern viene utilizzato quando l'interfaccia (es: *Target*) e la sua implementazione (es: *LegcyClass*) esistono già e non possono essere modificate.

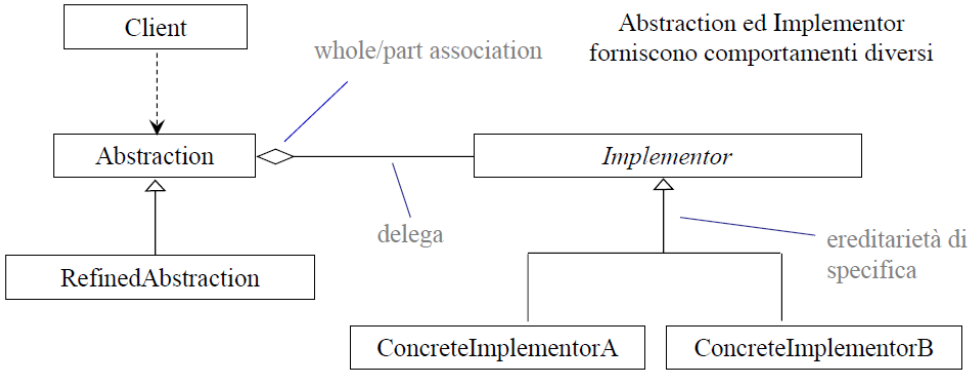
8.2.2.4 Bridge Pattern

Nome: Bridge (Handle/Body).

Descrizione del problema: Separare un'astrazione da una implementazione così che una diversa implementazione possa essere sostituita, eventualmente a runtime. (Separa l'interfaccia di un oggetto dalla sua implementazione).

Possibili applicazioni: Utile per interfacciare un insieme di oggetti; quando l'insieme non è ancora completamente noto (ad es. in fase di analisi, design, testing); quando c'è necessità di estendere un sottosistema dopo che il sistema è stato consegnato ed è in esecuzione (estensione dinamica).

Soluzione: Una classe Abstraction definisce l'interfaccia visibile al codice Client. L'Implementor è una interfaccia astratta che definisce i metodi di basso livello disponibili ad *Abstraction*. Un'istanza di *Abstraction* mantiene un riferimento alla corrispondente istanza di *Implementor***.** *Abstraction* e *Implementor* possono essere raffinate indipendentemente.




```
// Client
Event e = new Event();
AbstractEventEngine aee = new
    RefinedAbstractEventEngine();
EventEngine ee = aee.getEventEngineImplementor();
ee.notify(e);
```

```
// Implementor
public interface EventEngine{
    public void notify(Event e);
}
```

```
// Abstraction
public abstract class AbstractEventEngine {
  protected EventEngine imp;

  public abstract EventEngine
    getEventEngineImplementor();
}
```

```
// Concrete Implementor
public class ConcreteEventEngine implements
    EventEngine {
    public void notify(Event e) {
      /* Implementazione concreta */
    }
}
```

```
// RefinedAbstraction
public class RefinedAbstractEventEngine extends
    AbstractEventEngine{

  public EventEngine getEventEngineImplementor(){
    //imp = new EventEngineStub();
    imp = new ConcreteEventEngine();
    return imp;
  }
}
```

```
//ConcreteImplementor (Stub)
public class EventEngineStub implements
EventEngine {

  public void notify(Event e){

    System.out.println(
      "Ho inviato l'evento " + e);

  }
}
```

Conseguenze:

Disaccoppiamento tra interfaccia ed implementazione:

- Un'implementazione non è più legata in modo permanente ad un'interfaccia.
- L ' implementazione di un ' astrazione può essere configurata durante l'esecuzione.
- La parte di alto livello di un sistema dovrà conoscere soltanto le classi Abstraction e Implementor.

Maggiore estendibilità:

- Le gerarchie Abstraction e Implementor possono essere estese indipendentemente.

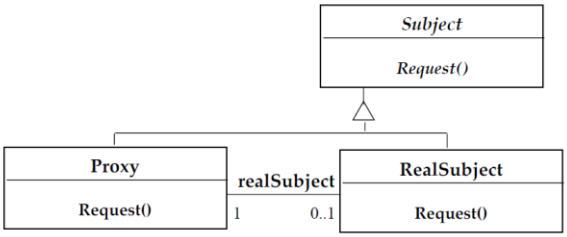
Mascheramento dei dettagli dell'implementazione ai client:

- I client non devono preoccuparsi dei dettagli implementativi.

8.2.2.5 Proxy Pattern

La creazione di oggetti e l'inizializzazione di questi ultimi è spesso molto costosa.

Il proxy pattern riduce il costo di accesso agli oggetti. Utilizza un altro oggetto, il proxy che agisce come l'oggetto reale. Il proxy crea l'oggetto reale sono se viene richiesto dall'utente.



L'ereditarietà dell'interfaccia è utilizzata per specificare l'interfaccia condivisa dal Proxy e dal RealSubject. La delegazione viene utilizzata per catturare e inoltrare un qualsiasi accesso al RealSubject (se richiesta).

Il proxy pattern può essere utilizzato per stime lente e per invocazioni remote. Il proxy può essere implementato con un'interfaccia Java.

Tipi di applicazione del proxy:

- Proxy remoto ➔ rappresentazione locale di un oggetto che si trova in uno spazio diverso di indirizzamento.
- Proxy virtuale ➔ L'oggetto è così costoso da creare o così costoso da scaricare.
- Proxy di protezione ➔ fornisce un controllo degli accessi agli oggetti reali. Utile quando oggetti diversi dovrebbero avere accessi diversi e giuste visioni per gli stessi documenti.

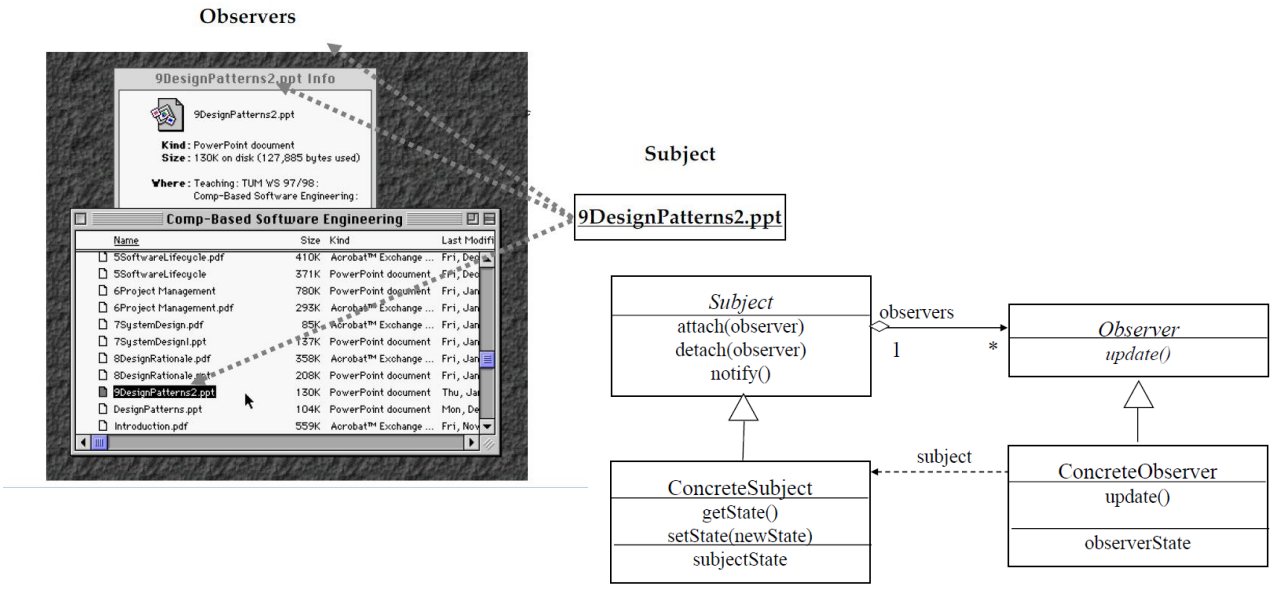
8.2.3 Pattern comportamentali

I pattern comportamentali forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti.

8.2.3.1 Observer Pattern

Definisce una dipendenza uno a molti tra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti che dipendono da esso vengono notificati e aggiornati automaticamente. È anche detto "Publish e Subscribe".

Uso: Gestire la consistenza tra stati ridondanti. Ottimizzare la quantità dei cambiamenti per gestire la consistenza.



Il Subject rappresenta lo stato attuale, gli Observers rappresentano diverse viste dello stato. L' Observer può essere implementato come un'interfaccia Java. Il Subject è una super classe, non un'interfaccia.

8.2.4 Pattern Creazionali

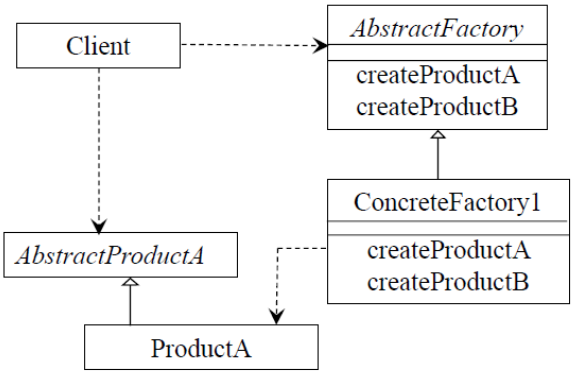
I pattern creazionali risolvono problematiche inerenti l'istanziazione degli oggetti

8.2.4.1 Abstract Factory

Nome: Abstract Factory

Descrizione del problema: Fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete.

Soluzione: Una piattaforma è rappresentato con un insieme di *AbstractProducts*, ciascuno dei quali rappresenta un concetto. Una classe *AbstractFactory* dichiara le operazioni per creare ogni singolo prodotto. Una piattaforma specifica è poi realizzata da un *ConcreteFactory* ed un insieme di *ConcreteProducts.*



Conseguenze:

- Isola le classi concrete ➔ Abstract Factory incapsula la responsabilità e il processo di creazione di oggetti Product e rende il client indipendente dalle classi effettivamente utilizzate per l'implementazione degli oggetti.
- Permette di sostituire facilmente famiglie di prodotti a runtime ➔ una factory concreta compare solo quando deve essere istanziata.
- Promuove la coerenza nell'utilizzo dei prodotti ➔una famiglia di prodotti correlati è solitamente progettata per essere utilizzata insieme. Il Client può creare prodotti solo utilizzando la AbstractFactory.
- Aggiungere nuove tipologie di prodotti è difficile ➔ devono essere create nuove realizzazioni per ogni factory.

8.2.5 Suggerimenti e Note su utilizzo design pattern

I requisiti non funzionali potrebbero dare una mano.
Argomenti: "produttore indipendente", "dispositivo indipendente", "devono supportare una famiglia di prodotti". ➔ Abstract Factory Pattern.
Argomenti: "devono interfacciarsi con oggetti esistenti" ➔ Adapter Pattern.
Argomenti: "si deve trattare con l'interfaccia di diversi sistemi, alcuni di loro potrebbero essere sviluppati in futuro, "un primo prototipo potrebbe essere mostrato. ➔ Bridge Pattern.
Argomenti: "strutture complesse", "devono avere profondità e larghezza variabile". ➔ Composite Pattern.
Argomenti: "devono interfacciarsi con un insieme di oggetti esistenti" ➔ Facade Pattern.
Argomenti: "deve essere localizzato in maniera trasparente. ➔ Proxy Pattern.
Argomenti: "deve essere estensibile", "deve essere scalabile" ➔ Observer Pattern.
Argomenti: "deve fornire una policy indipendente dal meccanismo" ➔ Strategy Pattern.

8.2.6 MVC

CONTESTO: L'applicazione deve fornire una interfaccia grafica (GUI) costituita da più schermate, che mostrano vari dati all'utente. Inoltre, le informazioni che devono essere visualizzate devono essere sempre quelle aggiornate

PROBLEMA: L'applicazione deve avere una natura modulare e basata sulle responsabilità, al fine di ottenere una vera e propria applicazione component - based.

SOLUZIONE E STRUTTURA: L'applicazione deve separare i componenti software che implementano il modello delle funzionalità di business, dai componenti che implementano la logica di presentazione e di
controllo che utilizzano tali funzionalità. Vengono quindi definiti tre tipologie di componenti che soddisfano tali requisiti:

- il Model, che implementa le funzionalità di business.
- la View: che implementa la logica di presentazione.
- il Controller: che implementa la logica di controllo.

8.2.7 Identificare e migliorare i framework di applicazione

8.2.7.1 Framework

Un framework di applicazione è un' applicazione parziale riusabile che può essere specializzata per produrre applicazioni personalizzate. A differenza delle librerie di classe i framework sono diretti a specifiche applicazioni (es: comunicazione cellulare o applicazioni real-time).
I framework contengono dei metodi hook (uncino) che devono essere implementati dalla specifica applicazione che si sta sviluppando. I framework possono essere classificati in:

- **Framework di infrastruttura**
  Esempi di questo tipo sono i sistemi operativi, debugger, interfacce utente (es Swing) che vengono inclusi in un progetto software ma non vengono presentati al cliente. - **Framework middleware**
  Vengono usati per integrare applicazioni distribuite con componenti (es RMI, CORBA).
- **Enterprice application frameworks**
  Sono applicazioni specifiche che si focalizzano su particolari domini di applicazione (comunicazione cellulare o altro).

I framework possono inoltre essere classificati in base alle tecnologie usate per estenderli.

- **Whitebox framework**
  Per ottenere l'estensibilità si affidano all'ereditarietà e alla riscrittura di metodi hook.
- **Blackbox framework**
  L'estendibilità è affidata solo all'implementazione delle interfacce e all'intergazione di tali implementazioni usando la delegazione.

8.2.7.2 Differenza tra design pattern e framework

La principale differenza tra design pattern e framework è che i framework si focalizzano sul riuso degli algoritmi e sull'implementazione in un particolare linguaggio, mentre i patterns di focalizzano sul riuso di particolari strategie astratte.

8.2.7.3 Differenza tra librerie di classe e framework

Le librerie di classe sono il più generiche possibile e non si focalizzano su un particolare dominio di applicazione fornendo solo un riuso limitato mentre i framework sono specifici per un dominio o famiglia di domini.

8.2.7.4 Differenza tra componenti e framework

Le componenti sono un raggruppamento di classi funzionanti anche da sole. Messe insieme, le componenti formano un'applicazione completa. In termini di riuso una componente è una scatola nera di cui non conosciamo l'implementazione ma solo alcune delle operazioni effettuabili su di esse. Le componenti offrono il vantaggio che le applicazioni le possono usare senza dover effettuare relazioni di ereditarietà.

8.3 **Valutare il riuso**

Il riuso ha alcuni vantaggi tra cui:

- Permettere un minor sforzo di sviluppo
- Minimizzare i rischi
- Ampio utilizzo di termini standard (i nomi dei design pattern denotano concetti precisi che gli sviluppatori danno per scontati)
- Aumento dell'affidabilità (si riduce la necessità di fare testing per componenti riusate) Alcuni svantaggi del riuso sono:

- Sindrome del Not Invented here (NIH)
- Necessità di supporto nel processo di riuso delle soluzioni
- Necessità di una fase di training

8.4 **Specificare le interfacce dei servizi**

La seconda fase dell'object design è la specifica delle interfacce. L'obbiettivo di questa fase è quello di produrre un modello che integri tutte le informazioni in modo coerente e preciso. Questa fase è composta dalle seguenti attività:

- Identificare gli attributi
- Specificare le firme e la visibilità di ogni operazione
- Specificare le precondizioni
- Specificare le postcondizioni
- Specificare le invarianti

8.4.1 Tipologie di sviluppatori

Precedentemente abbiamo parlato di sviluppatori in modo generico. E' possibile dividere gli sviluppatori in base al loro punto di vista.

- **Class implementor** Scrivono delle classi del sistema.
- **Class user:** Usano classi già create da altri sviluppatori.
- **Class extender:** Estendono classi già create da altri sviluppatori.

8.4.2 Specificare le firme

La firma di un metodo è la specifica completa dei tipi di parametri che un metodo prende in input e di quelli presi in output.

La visibilità di un metodo può essere di tre tipi: Private, Protected, Public. In UML questi tre tipi di rappresentano rispettivamente con i simboli - # + fatti precedere alla firma del metodo o di un attributo.

Nel progettare le interfacce di classe bisogna valutare bene quali sono le informazioni da rendere pubbliche. Come regola bisognerebbe esporre in modo pubblico solo le informazioni strettamente necessarie.

8.4.3 **Aggiungere contratti** (precondizioni, postcondizioni, invarianti)

Spesso l'informazione sul tipo di un attributo o parametro non bastano a restringere il range di valori di quell'attributo. Ai class user, implementor ed extendor serve condividere le stesse assunzioni sulle restrizioni.

I contratti possono essere di tre tipi:

- **Invariante:** è un predicato che è sempre vero per tutte le istanze di una classe.
- **Precondizione:** Sono predicati associati ad una specifica operazione e deve essere vera prima che l'operazione sia invocata. Servono a specificare i vincoli che un chiamante deve rispettare prima di chiamare un'operazione.
- **Postcondizione:** Sono predicati associati ad una specifica operazione e devono essere soddisfatti dopo che l'operazione è stata eseguita. Sono usati per specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione.

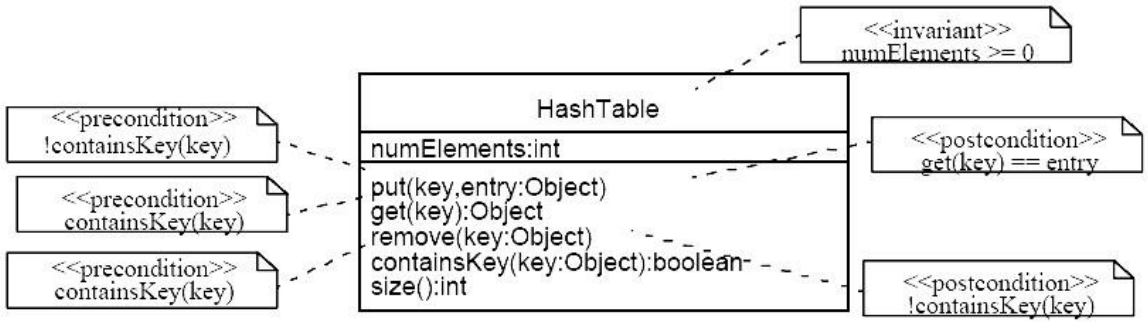


Figura 20 Esempio di contratti in UML

8.4.3.1 Object Constraint Language (OCL)

Per esprimere contratti in modo più formale è possibile usare l'OCL. In OCL un contratto è una espressione che ritorna un valore booleano vero quando il contratto è soddisfatto.

Le espressioni hanno tutte questo template: **context *NomeClasse::firmaMetodo() tipoContratto*: *espressione***. Per esprimere un contratto di classe e non di metodo si può omettere la firma del metodo e il doppio due punti dal template sopra (es. context NomeClasse pre: nomeAttributo = 0).

Nell'espressione è possibile usare nomi di metodi. (es. context Hashtable::put(key,value) pre: !containsKey(key)).

Nelle espressioni di postcondizione è possibile indicare il valore restituito da un'operazione o il valore di un attributo prima della chiamata del metodo usando @pre.nomeMetodo() oppure @pre.nomeAttributo (es. context Hashtable::put(key,value) post: getSize() = @pre.getSize() + 1). È possibile esprimere vincoli che coinvolgono più di una classe mettendo una freccia verso la classe che fa parte dell'espressione.

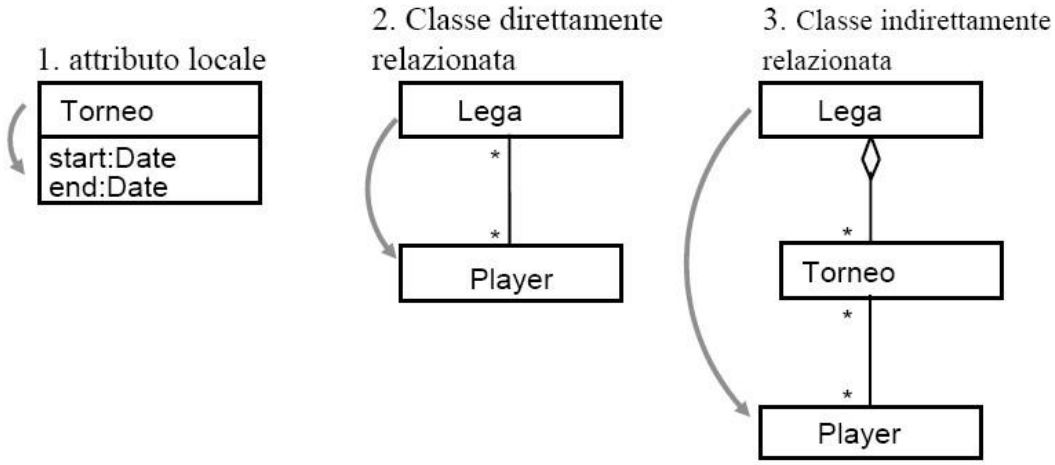


Figura 21 OCL in caso di vincoli che coinvolgono più classi

OCL così come è stato descritto non permette espressioni che coinvolgono collezioni di oggetti. OCL mette a disposizione tre tipi di collezioni:

- **Sets:** Insieme non ordinato di oggetti esprimibile con la forma {elemento1, elemento2, elemento3}
- **Sequence:** Insieme ordinato di oggetti esprimibile con la forma [elemento1, elemento2, elemento3]
- **Bags:** Insiemi multipli di oggetti. La differenza con Sets è che gli oggetti possono essere presenti più volte o l'insieme può essere vuoto. (es. {elemento1, elemento2, elemento2, elemento3}.

Per accedere alle collezioni OCL fornisce delle operazioni standard usabili con la sintassi collezione ->operazione(). Le più usate sono:

- **size()**: restituisce la dimensione della collezione
- **includes(oggetto)**: restituisce true se l'oggetto appartiene all'insieme
- **select(expression)**: restituisce una collezione che contiene gli oggetti su cui l'espressione è vera.
- **union(collection)**: restituisce una collezione che è l'unione della collezione in input e quella su cui viene eseguita l'operazione
- **intersection(collection)**: restituisce una collezione che contiene gli elementi in comune tra i due insiemi
- **asSet(collection)**: restituisce un insieme contenente gli elementi della collezione.

È possibile inoltre usare dei quantificatori come l'espressione *collezione->forAll(condizione)* che restituisce true se la condizione è vera per tutti gli elementi della collezione oppure l'espressione *collezione->exists(condizione)* che restituisce true se esiste un elemento nella collezione che rispetta quella condizione.

8.5 **Documentare l'Object Design**

È possibile usare un template per documentare l'Object Design che è il seguente:

1. Introduzione
    1.1 Object Design Trade-offs
    1.2 Linee Guida per la Documentazione delle Interfacce
    1.3 Definizioni, acronimi e abbreviazioni
    1.4 Riferimenti
2. Packages
3. Class interfaces

Glossario

9 Mappare il modello nel codice

9.1 **Concetti di mapping**

9.1.1 Trasformazioni

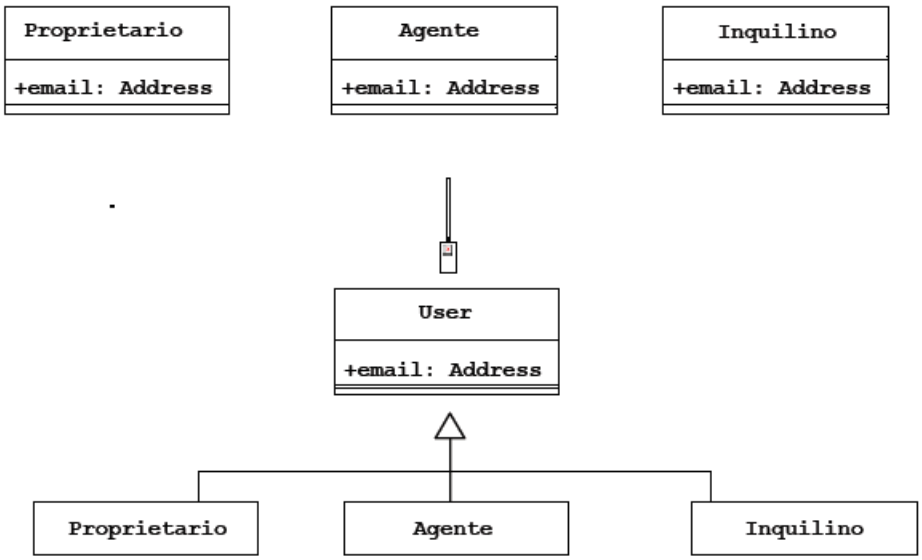
Una **trasformazione** ha lo scopo di migliorare un aspetto del modello (ad es. la sua modularità) e preservare allo stesso tempo tutte le altre proprietà (ad es. le funzionalità).
Generalmente una trasformazione:

- è localizzata
- impatta su un numero ristretto di classi, attributi e operazioni
- viene eseguita in una serie di semplici passi.

Esistono quattro tipi di trasformazioni:

9.1.1.1 Trasformazioni del modello

Questo tipo di trasformazioni operano sul modello del sistema e non sul codice (es. convertire una stringa che rappresenta un indirizzo in una classe contenente i campi città, via, cap ecc).
L'input e **l'output** di questa trasformazione è il modello ad oggetti. Lo **scopo** di questa trasformazione è quello di ottimizzare o semplificare il modello originale. Una trasformazione di questo tipo potrebbe aggiungere rimuovere o rinominare classi, attributi, associazioni e operazioni.



L' attributo ridondante email viene eliminato creando una superclasse.

9.1.1.2

Refactoring

I refactoring sono trasformazioni che operano sul ccodice sorgente. Effettuano un miglioramento della leggibilità del codice o la sua modifica senza intaccare le funzionalità del sistema. Lo scopo di questa trasformazione è quello di aumentare la leggibilità e la modificabilità. Si focalizza sulla trasformazione di un singolo metodo o classe. Le operazioni di trasformazione, onde evitare di intaccare le funzionalità del sistema ed introdurre errori, vengono fatte eseguendo piccoli passi incrementali intervallati da test; l'utilizzo di test driver per ogni classe incoraggia lo sviluppatore a modificare il codice per migliorarlo.

Esempio:

La trasformazione del modello di oggetti vista prima corrisponde ad una sequenza di tre refactoring:

Passo 1: spostare il campo **email** dalle sottoclassi alla superclasse:

1. **Assicurarsi che i campi email siano equivalenti nelle tre classi.**
2. **Creare una classe pubblica User.**
3. **Rendere User il padre di Agente, Inquilino, Proprietario.**
4. **Aggiungere un campo email alla classe User.**
5. **Rimuovere il campo email da Agente, Inquilino, Proprietario.**
6. **Compilare e testare.**

Prima del refactoring

```
public class Proprietario {
   private String email;
   //…
}
public class Agente {
    private String email;
    //…
}
public class Inquilino {
    private String email;
    //…
}
```

Dopo il refactoring

```
public class User {
      private String email;
      //…
}
public class Proprietario extends User{
      //…
}
public class Agente extends User{
     //…
}
public class Inquilino extends User{
      //…
}
```

Passo 2: spostare il codice di inizializzazione del campo **email** dalle sottoclassi alla superclasse:

1. Aggiungere il costruttore *User(String email)* alla classe *User*
2. Assegnare al campo *email* il valore passato nel parametro
3. Aggiungere la chiamata *super(email)* al costruttore della classe Proprietario
4. Compilare e testare
5. Ripetere i passi 2-4 per le classi *Agente* e *Inquilino*

Prima del refactoring

```
public class User {
      private String email;
}
public class Proprietario extends User{
   public Proprietario(String email) {
                this.email=email;
                 //…
      }
}
public class Agente extends User{
   public Agente(String email) {
                    this.email=email;
                    //…
      }
}
public class Inquilino extends User{
   public Inquilino(String email) {
                    this.email=email;
                    //…
      }
}
```

Dopo il refactoring

```
public class User {
   private String email;
   public User (String email) {
   this.email=email
   }
}
public class Proprietario extendsUser{
   public Proprietario(String email){
            super(email);
             //…
      }
}
public class Agente extends User{
      public Agente(String email)
   {super(email);//…}
}
public class Inquilino extends User{
      public Inquilino(String email) {
          super(email);
        //…
       }
}
```

Passo 3: spostare i metodi che manipolano il campo **email** dalle sottoclassi alla superclasse:

1. Esaminare i metodi della sottoclasse *Proprietario* che usano il campo *email* e selezionare quelli che non usano campi o operazioni specifiche di *Proprietario*.
2. Copiarli nella superclasse *User* e ricompilarla.
3. Cancellare questi metodi dalla sottoclasse *Proprietario*.
4. Compilare e testare.
5. Ripetere i passi 1-4 per le sottoclassi *Agente* e *Inquilino*.

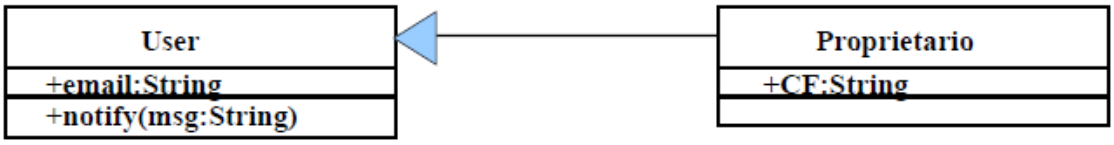
Si può notare che il refactoring include molti più passi della corrispondente trasformazione del modello ad oggetti; alterna attività di testing con attività di trasformazione.
La motivazione è che il codice sorgente include molti più dettagli e quindi più possibilità di introdurre errori.

9.1.1.3 Forward enginering

Input: un insieme di elementi del modello.
Output: un insieme di istruzioni di codice sorgente.
Obiettivi:

1. mantenere una forte corrispondenza fra il modello di design ad oggetti ed il codice;
2. ridurre il numero di errori introdotti durante l'implementazione;
3. ridurre gli sforzi di implementazione.

Ogni **classe** del diagramma **UML** è mappata in una **classe JAVA**. La **relazione di generalizzazione** UML è mappata in **una istruzione extends** (della classe Proprietario). Ogni **attributo** del modello **UML** è mappato in **un campo privato** della classe Java e **due metodi pubblici** per settare e visualizzare i valori del campo. Gli sviluppatori possono raffinare il risultato della trasformazione con comportamenti aggiuntivi (es: controllare se un campo è positivo). Il codice risultante da una trasformazione di questo tipo è sempre lo stesso (eccetto i nomi degli attributi). Se le classi sono progettate in modo adeguato si introducono meno errori.



Codice sorgente ottenuto con forward engineering

```
public class User {
    private String email;
    public String getEmail() {
          return email;
    }
    public void setEmail(String value){
          email=value;
    }
    public void notify(String msg) {
    //…
    }
//…
}
```

```
public class Proprietario extends User{
    private String CF;
    public String getCF() {
         return CF;
    }
    public void setCF (String value) {
         CF=value;
    }
    //…..
}
```

9.1.1.4 Reverse enginering

Input: un insieme di elementi di codice sorgente.
Output: un insieme di elementi del modello.
Obiettivo: ricreare il modello per un sistema esistente.
Motivazioni:

- Il modello è andato smarrito o non è mai stato realizzato.
- Il modello non è più allineato con il codice sorgente.

La trasformazione di tipo Reverse Engineering è la trasformazione inversa del Forward Engineering:

- crea una classe UML per ciascuna classe;
- aggiunge un attributo per ciascun campo;
- aggiunge un'operazione per ciascun metodo.

Il reverse engineering non crea necessariamente il modello originario (forward engineering può far perdere informazioni come le associazioni). È supportata da CASE tool ma richiede comunque l'intervento dello sviluppatore per ricostruire un modello il più possibile vicino a quello originario.

9.1.1.5 Principi di trasformazione

Per evitare che le trasformazioni inducano in errori difficili da trovare e riparare, le trasformazioni dovrebbero seguire questi semplici principi:

- Ogni trasformazione deve soddisfare un solo design goal per volta.
- Ogni trasformazione deve essere locale e dovrebbe cambiare solo pochi metodi e poche classi.
- Ogni trasformazione deve essere applicata singolarmente e non devono essere effettuate trasformazioni simultaneamente.
- Ogni trasformazione deve essere seguita da una fase di validazione/testing.

9.2 Attività del mapping

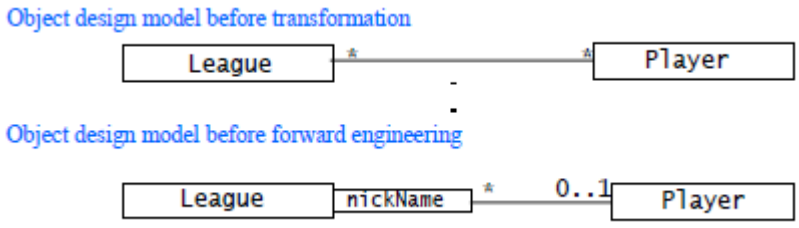
Le attività riguardanti il mapping del modello sul codice coinvolgono tutte una serie di trasformazioni. Le trasformazioni che verranno trattate sono:

- **Ottimizzare il modello di Object Design**
  Quest'attività ha lo scopo di migliorare le performance del sistema. Questo può essere ottenuto riducendo la molteplicità delle associazioni per velocizzare le query.
- **Realizzare le associazioni**
  Durante quest'attività mappiamo le associazioni in riferimenti o collezioni di riferimenti.
- **Mappare i contratti in eccezioni**
  In questa fase descriviamo le operazioni che il sistema deve effettuare quando vengono rotti i contratti.
- **Mappare il modello di classi in uno schema di memorizzazione**
  In questa attività mappiamo il modello delle classi in uno specifico schema di memorizzazione (es. definire le tabelle).

9.2.1 Ottimizzare il modello di Object Design

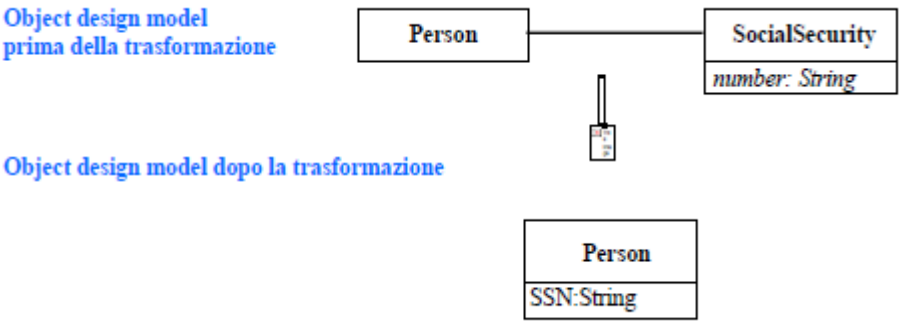
Le fasi di ottimizzazione sono le seguenti:

- **Ottimizzare i cammini di accesso alle informazioni**
  Se per richiedere una informazione bisogna attraversare troppe associazioni (es. metodo().metodo2().metodo3().metodo(4)) bisognerebbe aggiungere un'associazione diretta tra i due oggetti richiedente e fornitore.
  Un'altra ottimizzazione può essere ottenuta riducendo le associazioni di tipo "molti" in "uno" oppure ordinando gli oggetti lato "molti" per velocizzare il tempo di accesso.



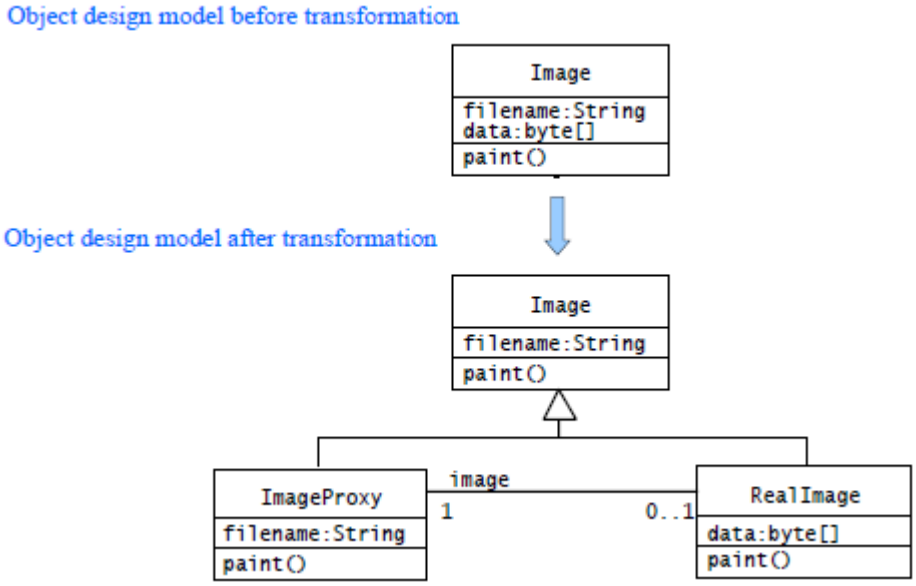
  Gli attributi, nella fase di analisi, potrebbero essere stati collocati male nelle classi se vengono solo acceduti tramite i metodi get e set. Si potrebbero spostare direttamente nella classe che usa i metodi get e set per velocizzarne l'accesso.

- **Collassare gli oggetti in attributi**
  Dopo le fasi di ottimizzazione alcuni oggetti potrebbero contenere pochi attributi e operazioni. In questi casi può essere utile trasformare queste classi in attributi semplici.



- **Ritardare le elaborazioni costose**
  Alcuni oggetti sono costosi da creare, la loro creazione può spesso essere ritardata finché il loro contenuto è effettivamente necessario.

• Esempio: considerare un oggetto che rappresenti un'immagine memorizzata come file, infatti caricare tutti i pixel che costituiscono l'immagine dal file è costoso. Si utilizza il design pattern Proxy: Un oggetto ImageProxy sostituisce l'oggetto Image e fornisce la stessa interfaccia dell'oggetto Image. Operazioni semplici come larghezza() ed altezza() saranno gestite da ImageProxy. Quando Image deve essere disegnato, ImageProxy carica i dati dal disco e crea l'oggetto RealImage. Se il client non invoca la paint(), l'oggetto RealImage non è creato, risparmiando un notevole tempo di calcolo.



- **Mantenere in una struttura temporanea il risultato di elaborazioni costose ritardandone l'elaborazione**
  A volte caricare grosse quantità di dati che non vengono usati totalmente può essere inutile. Conviene, talvolta, caricare in memoria solo la parte di informazione che è strettamente necessaria e mettere il resto in una struttura temporanea.

9.2.2 Mappare associazioni in collezioni e riferimenti

Le associazioni viste nei diagrammi sono concetti astratti UML che nei linguaggi di programmazione tipo Java vengono mappate in riferimenti (nel caso di associazioni uno a uno) e collezioni (nel caso di associazioni uno-a-molti).

Se l'associazione tra due classi è *uno-ad-uno unidirezionale*, solo la classe che utilizza le operazioni dell'altra avrà il riferimento all'altra classe. Se invece l'associazione è *uno-ad-uno bidirezionale* ambedue le classi avranno un riferimento all'altra. Nel caso in cui l'associazione sia *molti-a-molti* si usano delle collezioni come liste, insiemi o tabelle hash.
Per quanto riguarda le *classi di associazione* viste in UML, queste possono essere mappate nel codice convertendole in normali classi e inserendo le classi dell'associazione come riferimenti.

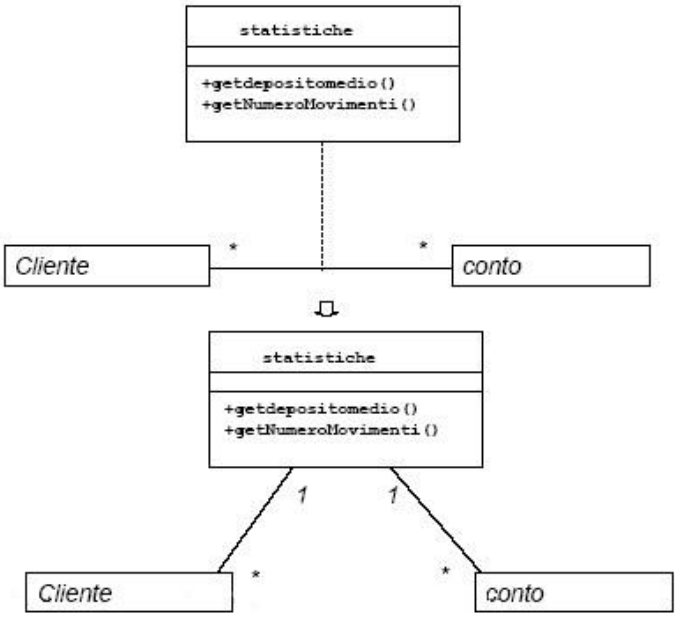


Figura 22 Mapping di una classe di associazione

9.2.3 Mappare i contratti in eccezioni

Quando viene violato un contratto è buona norma che il sistema software lanci un'eccezione. Lanciare un'eccezione è un'operazione che provoca un'interruzione nel normale flusso del programma. Tale eccezione risale lo stack delle chiamate fino a quando non viene colta in qualche blocco di codice.
Le precondizioni vanno controllate prima dell'inizio del metodo e nel caso sia falsa va lanciata un'eccezione. Le post-condizioni vengono controllate alla fine del metodo e deve essere lanciata un'eccezione se non vengono soddisfatte. Le invarianti vanno controllate nello stesso momento delle post-condizioni.
È buona norma incapsulare il codice di controllo di un contratto in un metodo soprattutto se tale controllo deve essere effettuato da sottoclassi.
Alcune euristiche da seguire sono le seguenti:

- Si può omettere il codice di controllo per post-condizioni e invarianti: è ridondante inserirlo insieme al codice che realizza la funzionalità della classe, inoltre non individua molti bug a meno che non venga scritto da un altro sviluppatore.
- Si può omettere il codice di controllo per metodi privati e protetti se è ben definita l'interfaccia del sottosistema.
- Concentrarsi sulle componenti che hanno una lunga durata: oggetti Entity, non oggetti boundary associati all'interfaccia utente.
- Riusare codice per il controllo dei vincoli:
    - molte operazioni hanno precondizioni simili;
    - incapsulare il codice per il controllo degli stessi vincoli in metodi così possono condividere le stesse classi di eccezioni.

9.2.4 Mappare il modello di classi in uno schema di memorizzazione

Gli oggetti vengono mappati in strutture dati che rispettano le scelte fatte nel system design (file o database). Se si scelgono i file bisogna scegliere uno schema di memorizzazione standard che non induca in ambiguità.

Le classi vengono mappate in tabelle che hanno lo stesso nome della classe e gli attributi vengono mappati in una colonna della tabella con lo stesso nome dell'attributo. Ogni riga della tabella corrisponde ad un'istanza della classe.

Nelle tabelle bisogna scegliere una chiave primaria che identifichi univocamente un'istanza della classe.

Per quanto riguarda le associazioni uno-ad-uno o uno-a-molti è necessario usare una chiave esterna che collega le due tabelle. Se l'associazione è uno-a-molti la chiave esterna è nella classe del lato "molti" in quanto si collega alla classe che la contiene, se è uno-ad-uno vengono mappate usando la chiave esterna in una qualsiasi delle due tabelle. Le associazioni molti-a-molti sono implementate usando una tabella aggiuntiva che ha due colonne contenenti le chiavi esterne delle tabelle relative alle classi in relazione.

9.2.4.1 Mappare le relazioni di ereditarietà

È possibile mappare l'ereditarietà in due modi:

9.2.4.1.1 Mapping verticale

La superclasse e la sottoclasse sono mappate in tabelle distinte. La tabella della superclasse mantiene una colonna per ogni attributo definito nella superclasse e una colonna che indica quale tipo di sottoclasse rappresenta quell'istanza. La sottoclasse include solo gli attributi aggiuntivi rispetto alla superclasse e una chiave esterna che la collega alla tabella della superclasse.

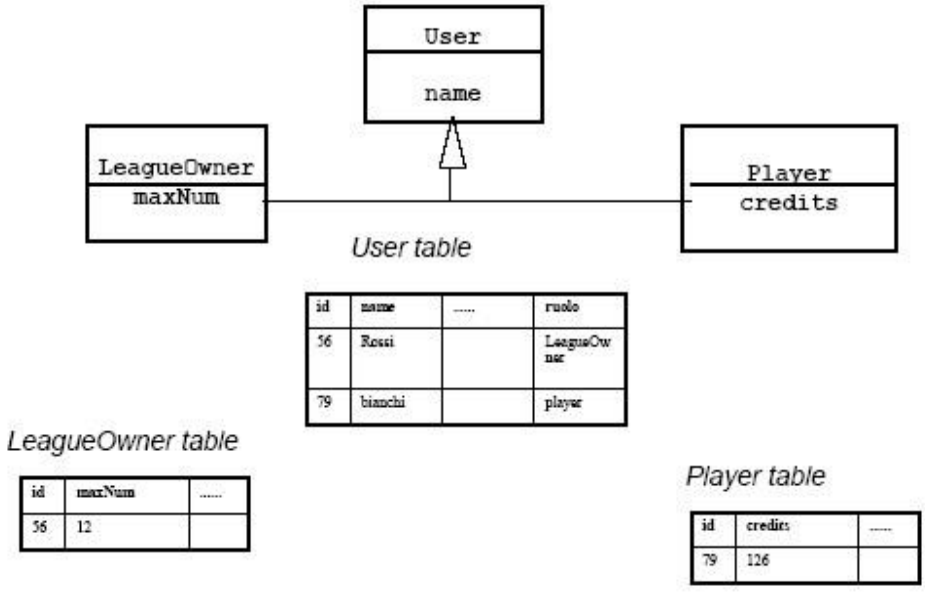
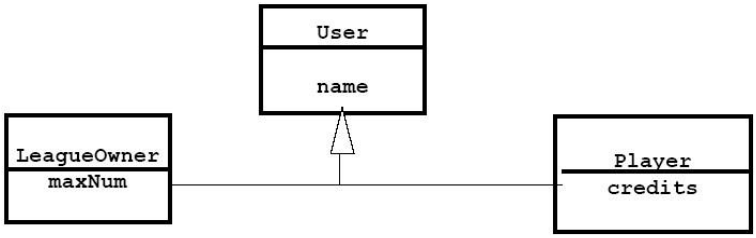


Figura 23 Esempio di mapping di ereditarietà verticale

9.2.4.1.2 Mapping orizzontale

Nel mapping orizzontale non esiste una tabella per la superclasse ma una tabella per ciascuna sottoclasse. In questo modo c'è ripetizione di attributi nel senso che tutte e due le tabelle avranno una colonna per ogni attributo contenuto nella superclasse.



LeagueOwner table

id	name	maxNum	...
56	Rossi		

Player table

id	name	credits	...
79	Bianchi	126	

Figura 24 Esempio di mapping di ereditarietà orizzontale

9.2.4.1.3 Trade off tra mapping orizzontale e verticale

I trade-off tra i due meccanismi riguardano la modificabilità e il tempo di accesso. Nella prima soluzione la modificabilità è più semplice in quanto aggiungere un attributo alla super classe richiede l'aggiunta di una colonna solo in una tabella e aggiungere una sottoclasse richiede di aggiungere una tabella che contenente solo gli attributi della sottoclasse. D'altra parte scegliere un mapping virtuale porta ad una bassa efficienza in quanto caricare una sottoclasse dal database richiede l'accesso a due tabelle anziché una.
La soluzione di mapping orizzontale consente un più rapido accesso alle informazioni ma una più bassa modificabilità ed estensibilità.
Per scegliere una o l'altra soluzione bisogna trovare dei compromessi tra efficienza e modificabilità.

9.3 **Responsabilità**

Nelle fasi di trasformazione ci sono vari ruoli che cooperano.
Il c*ore-architect* seleziona le trasformazioni che devono essere applicate in maniere sistematica. L'*architecture liason* è responsabile di documentare i contratti associati alle interfacce dei sottosistemi. Quando questi contratti cambiano è responsabile di comunicare i cambiamenti ai class user.
Lo *sviluppatore* deve seguire le convenzioni dettate dal core-architect e convertire il modello in codice sorgente.

9.4 **Gestire l'implementazione**

Le **trasformazioni** ci consentono di:

- migliorare aspetti specifici del modello e di convertirlo in codice sorgente;
- ridurre lo sforzo complessivo ed il numero totale di errori nel codice sorgente.
- È necessario documentare le trasformazioni così che possano essere riapplicate in caso di cambiamenti nel modello di object design o nel codice sorgente.

10 Testing

Il testing è l'attività che cerca le differenze tra le specifiche/requisiti e il comportamento osservato del sistema implementato.

10.1 **Overview**

Diamo ora alcune definizioni che verranno usate in seguito:

- **Affidabilità**: Misura della conformità del sistema con il comportamento osservato.
- **Affidabilità del software:** Probabilità che il sistema software non causi fallimenti per un certo tempo e sotto determinate condizioni.
- **Fallimento**: Una deviazione del comportamento osservato rispetto a quello atteso.
- **Difetto** (chiamato anche bug): Causa algoritmica o meccanica che ha portato a un comportamento errato.
- **Errore:** Il sistema è in uno stato in cui qualsiasi operazione porta ad un fallimento.
- **Prova di correttezza:** argomentare sistematicamente (in modo formale o informale) che il programma funziona correttamente per *tutti i possibili dati di ingresso.*
- **Testing:** particolare tipo di attività sperimentale fatta mediante esecuzione del programma, selezionando alcuni dati di ingresso e valutando risultati dà un riscontro *parziale: programma provato solo per quei dati.* È una tecnica *dinamica rispetto alle verifiche statiche fatte dal compilatore.*

Test e casi di test

Un programma è esercitato da un caso di test (insieme di dati di input)**.** Un ***test*** è formato da un insieme di casi di test**.** L'esecuzione del test consiste nell'esecuzione del programma per tutti i casi di test**.** Un test ha **successo** se rileva uno o più malfunzionamenti del programma.

Ci sono varie tecniche per aumentare l'affidabilità di un sistema software:

- **Fault avoidance** (prevenzione degli errori): Questa tecnica prova a prevenire l'inserimento di errori nel sistema prima che lo stesso venga rilasciato.
- **Fault detection:** Fanno parte di questa tecnica il *debugging*, il *testing* e il *review* svolti durante la fase di sviluppo del software allo scopo di trovare errori. Questa tecnica non cerca di risolvere i difetti ma ha il solo scopo di identificarli (es. la scatola nera negli aeroplani). In seguito verranno descritti il review, il debugging e il testing.
- **Fault tolerance** (tolleranza agli errori): Questa tecnica assume che un sistema può essere rilasciato con errori e che i fallimenti del sistema possono essere gestiti a runtime. In questi casi è possibile assegnare la stessa attività a più componenti che la eseguono contemporaneamente e alla fine confrontano il risultato ottenuto.

Il **review** (revisione) è un controllo di parti o di tutti gli aspetti del sistema senza mandarlo in esecuzione e ne esistono di due tipi:

- **Walkthrought** (attraversamento): Gli sviluppatori presentano il codice corredato di API e documentazione di una componente da testare al team di revisione. Il team di revisione commenta il codice aggiungendo dettagli riguardanti il mappaggio dell'analisi e del object design usando come spunto i casi d'uso e gli scenari della fase di analisi.
- **Inspection** (ispezione): Un'ispezione è simile al walkthrought ma la presentazione del codice non viene fatta dagli sviluppatori. Il team di revisione controlla le interfacce e il codice delle componenti rispetto ai requisiti. Il team è anche responsabile di controllare l'efficienza degli

algoritmi rispetto ai requisiti non funzionali e di controllare se i commenti inseriti sono coerenti rispetto al comportamento del codice.

Il **debugging** assume che un errore possa essere scovato partendo da un comportamento che non era stato previsto (es. se ci si accorge che una funzionalità non esegue bene il suo compito si inizia a fare debugging). Esistono due tipi di debugging: si cercano di trovare gli errori, l'altro cerca di valutare l'efficienza.

Il **testing** cerca di creare fallimenti o stati erronei in modo pianificato.

Algorithmic Fault: oppure l'implementazione sbagliata della specifica da parte di un team.
Mechanical Fault: Un fallimento della virtual machine di un sistema software è un esempio di fallimento meccanico: anche se lo sviluppatore ha implementato correttamente, cioè mappato correttamente l'object model nel codice, il comportamento osservato può deviare ancora da quello atteso.

Gestione degli errori

- Verification: Assumere che l'ambiente ipotetico non corrisponde all'ambiente reale. La prova potrebbe essere buggata (omette vincoli importanti).
- Modular redundancy (ridondanza modulare): molto costosa, un modulo è ripetuto più volte per evitare errori, è molto utile quando il livello dia ffidabilità è alto.
- Declaring a bug to be a "feature": cattiva pratica (**Every bug is a feature**)
- Patching: rallentamento delle performance
- Testing (this lecture): non è sempre perfetto.

10.2 **Concetti di testing**

- **Componente**: È una parte del sistema che può essere isolata per fare testing. Un componente può essere un oggetto, un gruppo di oggetti o uno o più sottosistemi.
- **Stato di errore**: È una manifestazione di un errore durante l'esecuzione del sistema (può essere causato da uno o più errori e può portare ad un fallimento).

Testing *esaustivo (esecuzione per tutti i possibili ingressi) dimostra la correttezza*, non è sempre possibile realizzarlo.

Terminazione del testing
Quando il programma si può ritenere analizzato a sufficienza

- **Criterio temporale: periodo di tempo predefinito**
- **Criterio di costo: sforzo allocato predefinito**
- **Criterio di copertura:** percentuale predefinita degli elementi di un modello di programma, è legato ad un criterio di selezione dei casi di test
- **Criterio statistico:** MTBF (mean time between failures) predefinito e confronto con un modello di affidabilità esistente

Definizioni basi del test
Errors: le persone commettono errori.
♦ **Fault:** un fault è il risultato di un errore nel codice, documentazione, ecc.
♦ **Failure:** Un fallimento si verifica quando è in esecuzione un fault.
♦ **Incident:** Conseguenza di un fallimento, il verificarsi di un falliemnto può o non può essere evidente all'utente.
♦ **Testing:** Testare il software con i casi di test per trovare difetti o dare più sicurezza al sistema.
♦ **Test case:** Insieme di dati di input e risultati attesi (oracle) che testano una componente allo scopo di causare dei guasti.
♦ **Test suite (or test):** un insieme di test case.

10.2.1 Test case
È un insieme di input e un risultato atteso che potrebbero portare una componente a causare un fallimento. Un test case ha cinque attributi:
- **Name:** Un nome univoco rispetto ai test da effettuare
- **Location:** Descrive dove può essere trovato il programma che effettua il test.
- **Input:** Descrive l'insieme di comandi che devono essere immessi nel programma di test.
- **Oracle:** L'output che il programma dovrebbe dare. Abbiamo 2 tipo di oraolo:
  - ♦ **Oracolo umano:** si basa sulle specifiche o sul giudizio
  - ♦ **Oracolo automatico:** è generato dalle specifiche (formali), è lo stesso software ma sviluppato da altri, si basa su una versione precedente (test di regressione).
- **Log:** Memorizza varie esecuzioni del test e determina le differenze tra il comportamento atteso e quello osservato.

I test case possono avere le associazioni di *aggregazione* (un test case può essere composto di più sotto test) e *precedenza* (un test case deve essere eseguito prima di un altro).
Inoltre i test case possono essere classificati in *whitebox* e *blackbox*. Le blackbox si focalizzano solo sull'input e l'output del programma senza andare ad indagare nel codice, le whitebox si concentra sulla struttura interna della componente.

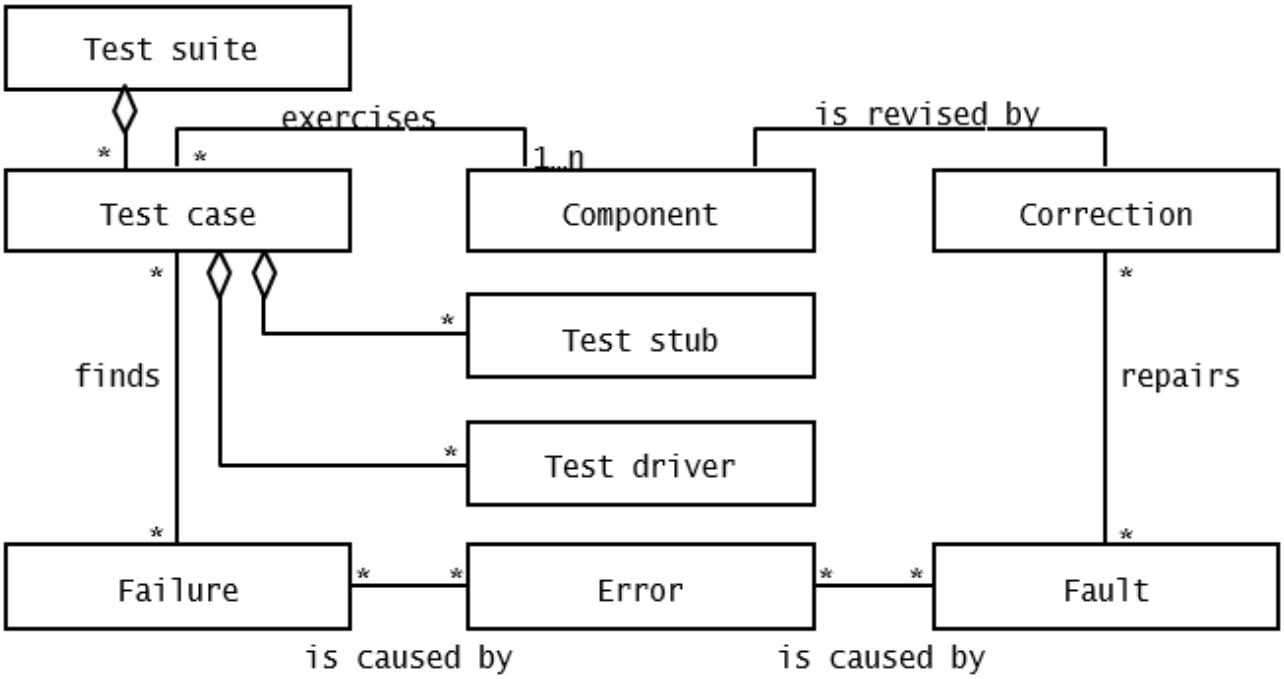
I test stub e driver sono usati per sostituire e simulare parti mancanti del sistema.
- **Test stub**: È una particolare implementazione di una componente *dal quale dipende* una componente sotto testing.
- **Test driver**: È una particolare implementazione di una componente *che fa uso* della componente sotto testing.

Un test stub deve fornire la stessa API del metodo della componente simulata e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specificato nella signature. Se l'interfaccia di una componente cambia anche il corrispondente test driver e test stub devono cambiare → gestione delle configurazioni. L'implementazione di un test stub non è una cosa semplice. Non è sufficiente scrivere un test stub che semplicemente stampa un messaggio attestante che il test stub è iniziato. La componente chiamata deve fare un qualche lavoro, se il test stub non simula il giusto comportamento la componente testata potrebbe avere un failure.

10.2.2 Correzioni

Una correzione è un cambiamento di una componente effettuato allo scopo di risolvere un errore. Una correzione potrebbe, in certi casi, introdurre nuovi errori. Per evitare questo possiamo utilizzare alcune tecniche:

- **Problem tracking:** Mantiene traccia degli errori riscontrati e delle relative soluzione tramite una documentazione.
- **Regression testing:**Vengono rieseguiti i test precedenti non appena viene corretta una componente.
- **Rational maintenance:** Include una documentazione che specifica i motivi di un cambiamento o le motivazioni dello sviluppo di una componente.



10.3 **Attività di testing**

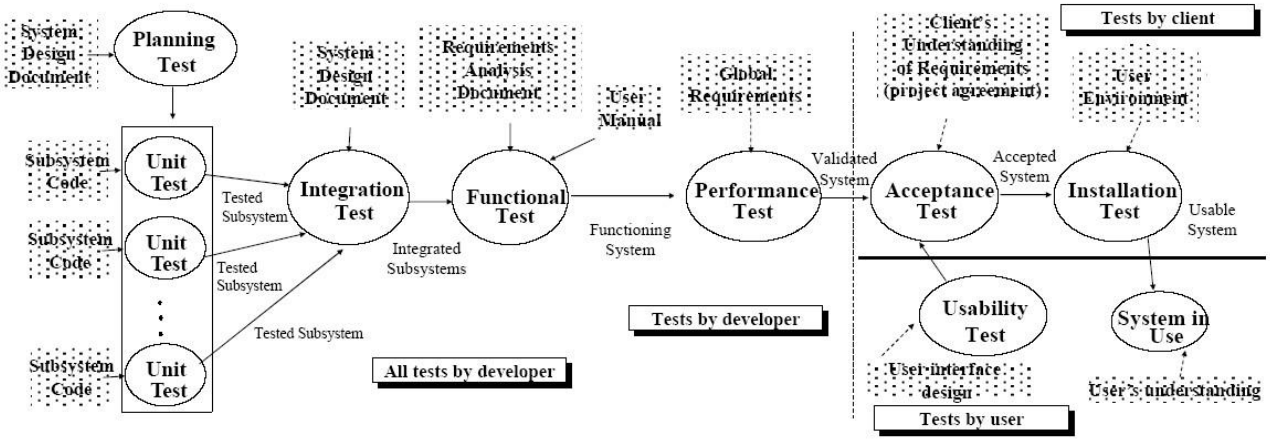


Figura 25 Attività del testing

- **Component inspection**. Trova i fault in una componente individuale attraverso l'ispezione manuale del codice sorgente
- **Usability testing**. Trova le differenze tra il sistema e l'attesa dell'utente per quanto riguarda l'uso del sistema
- **Unit testing**. Trova fault isolando una componente individuale usando test stub e test driver, e esercitando la componente tramite un test case
- **Integration testing**. Trova fault integrando differenti componenti insieme
- **System testing**. Si focalizza sul sistema completo, i suoi requisiti funzionali e non funzionali, e il suo ambiente.

10.3.1 Component inspection

Trova gli errori in singole componenti attraverso l'ispezione del codice sorgente. L'ispezione è condotta da un team di sviluppatori incluso l'autore della componente attraverso un meeting formale.
Un metodo proposto da Fagan è quello che divide l'ispezione in varie fasi:

- **Overview:** L'autore della componente presenta le finalità della componente e gli obbiettivi dell'ispezione.
- **Preparazione:** I revisori diventano familiari con l'implementazione della componente
- **Meeting di ispezione:** Una persona legge il codice della componente e il team di ispezione elabora proposte. Un moderatore tiene traccia delle cose dette.
- **Rework:** L'autore rivede e modifica la componente.
- **Follow-up:** Il moderatore controlla la qualità della revisione e determina la componente che necessita di essere reispezionata.

L' Active Design Review: processo di ispezione rivisitato proposto da Parnas:

- **Preparation.** I revisori analizzano l'implementazione della componente e individuano i fault. Compilano un questionario che testa la loro comprensione della componente
- **Non c'è Inspection meeting.** L'autore incontra separatamente ogni reviewer per collezionare feedback sulla componente

I metodi di ispezione sono molto efficaci: Circa 85% dei fault può essere individuato.

10.3.2 *Managing testing*

Serve a capire come gestire le testing activity per minimizzare le risorse necessarie. Alla fine, gli sviluppatori dovrebbero rilevare e quindi riparare un numero sufficiente di bug tale che il sistema soddisfi i requisiti funzionali e non funzionali entro un limite accettabile da parte del cliente. La pianificazione delle testing activity avviene tramite il diagramma di PERT che mostra le dipendenze.
Le attività sono documentate in 4 tipi di documenti:

- il **Test Plan** che si focalizza sugli aspetti manageriali;
- Ogni test è documentato attraverso un **Test Case Specification**;
- Ogni esecuzione di un test case è documentata attraverso un **Test Incident Report**;
- Il **Test Report Summary** elenca tutti i fallimenti rilevati durante i test che devono essere investigati

10.3.3 Documentazione di testing

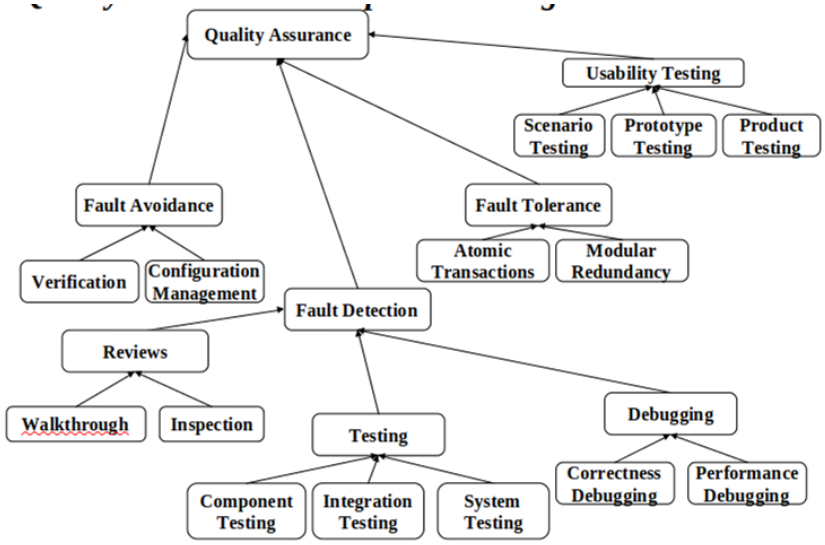
Pianificazione (planning)

- Piano di test: si concentra sugli aspetti manageriali del test. Documenta lo scopo, l'approccio, le risorse, il programma delle attività di test. I requisiti e i componenti da testare sono identificati in questo documento.
- Specifica del caso di test: documenta ciascun test. Questo documento contiene gli input, i driver, gli stub e gli output previsti dei test, nonché le attività da eseguire

Execution documenting

- Test incident report: per qualsiasi test non riuscito, fornisce dettagli sul risultato effettivo rispetto a quello atteso e altre informazioni intese a chiarire il motivo del fallimento di un test. Questo documento viene deliberatamente denominato come rapporto sugli incidenti e non come rapporto di errore. Il motivo è che una discrepanza tra i risultati attesi e quelli effettivi può verificarsi per una serie di ragioni diverse da un errore nel sistema. Tra questi, i risultati attesi sono errati, il test viene eseguito in modo errato o l'incoerenza dei requisiti, il che significa che è possibile effettuare più di un'interpretazione.

Test summary report: fornisce uno stato generale dei test. Da questo documento gli sviluppatori analizzano e assegnano la priorità a ciascun errore e pianificano le modifiche nel sistema e nei modelli.



10.3.4 Usability testing

Speso le interfacce utente potrebbero risultare poco intuitive. Per questo motivo viene effettuato questo tipo di test. Gli sviluppatori determinano degli obbiettivi e osservano gli utenti sul campo prendendo nota del tempo che hanno impiegato per eseguire determinate operazioni e accettano eventuali commenti e suggerimenti.

Ci sono tre tipi di usability testing:

- **Scenario test:** Viene presentato uno scenario agli utenti e viene valutato in quanto tempo gli utenti lo comprendono. È possibile usare anche dei mock-up.
    - Vantaggi: sono economici da realizzare e da ripetere
    - Svantaggi: gli utenti non possono interagire direttamente con il sistema, i dati sono fissi
- **Prototype testing:** Viene presentato un prototipo all'utente che rappresenta i punti chiave del sistema. Il prototype testing è di tipo vertical se implementa solo un caso d'uso, è orizzontal se implementa un livello del sistema che coinvolge più casi d'uso.
    - **Vantaggi**. Forniscono una vista realistica del sistema all'utente e il prototipo può essere concepito per collezionare informazioni dettagliate.
    - **Svantaggi**. Richiede un impegno maggiore nella costruzione rispetto agli scenari cartacei.
- **Product test:** Viene usata una versione funzionale del sistema e fatta visionare all'utente.

Usability test – basics elements

Tutti e tre le tecniche prevedono: Sviluppo degli obiettivi del test (confronto tra due stili di interazione, qualche help necessario, quale tipo di training è richiesto, …). Selezionare un campione rappresentativo degli utenti finali. Una simulazione (o il reale) dell'ambiente di lavoro. Interrogazioni controllate ed estensive degli utenti alle prese con l'utilizzo del sistema attraverso i test. Collezione e analisi dei risultati qualitativi e quantitavi. Raccomandazioni su come migliorare il sistema.

10.3.4 Unit testing

Nell'unit testing i singoli sottosistemi o oggetti vengono testati separatamente. Questo comporta il vantaggio di ridurre il tempo di testing testando piccole unità di sistema singolarmente. I candidati da sottoporre al testing vengono presi dal modello ad oggetti e dalla decomposizione in sottosistemi.

Gli unit testing possono essere divisi principalmente in due tipologie: whitebox testing e blackbox testing. Il ***blackbox*** testing guarda solo l'input e l'output senza curare il codice, il *whitebox* testing invece cura solo il codice e la struttura senza guardare l'input e output. Per quanto riguarda il blackbox testing è possibile effettuarlo in due modi:

- **Equivalence testing:** Gli input vengono divisi in classi di equivalenza. Ad esempio tutti gli input di numeri negativi e tutti gli input di numeri positivi.
- **Boundary testing:** Si selezionano degli input che sono limite per le classi di equivalenza (es. per i numeri positivi si testa il numero 1 o il più grande numero positivo rappresentabile).

Black-box ***funzionale:*** casi di test determinati in base a ciò che il componente deve fare la sua *specifica.*
White-box ***strutturale:*** casi di test determinati in base a che come il componente è implementato il codice.

Esistono delle *euristiche* per scegliere le classi di equivalenza degli input del test.
Se l'input valido è un *range*, creiamo tre classi di equivalenza corrispondenti ai valori sotto, dentro e sopra il range di valori. Se invece l'input valido è un *insieme discreto* vengono create due classi di equivalenza corrispondenti ai valori dentro e fuori l'insieme.
Uno degli *svantaggi* di queste tecniche è che non vengono testate combinazioni miste di input ma solo quelle interne alle classi di equivalenza.

Partizionamento sistematico: Si cerca di partizionare il dominio di input in modo tale che da tutti i punti del dominio ci si attende lo stesso comportamento (e quindi si possa prendere come rappresentativo un punto qualunque in esso).

Weak equivalence class testing (WECT): si prende una variabile da ogni classe di equivalenza.
Strong equivalence class testing (SECT): si prende una variabile per ogni partizione del prodotto cartesiano delle classi di equivalenza.

Category Partition Test: il category partition test è un metodo per generare test funzionali da specifiche informali. Si compone di tre passi:

1. **Decomporre le specifiche in feauture testabili indipendentemente**: in questo passo, il test designer deve identificare le feauture che devono essere testate in modo separato, identificando i parametri e qualunque altro elemento dell'ambiente di esecuzione da cui dipende (ad esempio un database). Per ciascun parametro e elemento dell'ambiente si identificano le **caratteristiche** (elementari) **del parametro**, dette **categorie**.
2. **Identificare Valori Rappresentativi**: questo passo, prevedere che il test designer identifichi un'insieme di valori rappresentativi (classe di valori, detta **choices**) per ciascuna caratteristica di ogni parametro definito nella fase precedente. Questa attività è detta "partizionare le categorie in choices". I valori dovrebbero essere identificati per ciascuna categoria indipendentemente dai valori delle altre categorie. L'identificazione viene eseguita applicando delle regole note col nome di *boundary value testing* o *erroneous condition testing*. La regola di boundary value testing prevede la selezione di valori estremi all' interno di una classe (valori massimi o minimi), valori esterni ma molto vicini alla classe e valori interni (non estremi). I valori vicini al confine di una classe sono utili nel rilevare i casi d'errore del programma.
3. **Generare Specifiche di Casi di Test**: In questa fase, il test designer, stabilisce dei vincoli semantici sui valori per indicare le combinazioni non valide e ridurre il numero di quelle valide. Una specifica di caso di test per una feauture è data da una combinazione di classi di valori, una per ciascuna caratteristica dei parametri. Il metodo del category partition permette di eliminare alcune combinazioni indicando classi di valori che non hanno bisogno di esser combinate con tutti gli altri valori. L'etichetta **[error]** indica una classe di valori che può essere combinata una sola volta nelle combinazioni con valori non errati di altri parametri. I

vincoli property e if-property sono utilizzati insieme; **property** raggruppa valori di una singola caratteristica di un parametro per identificare sottoinsiemi di valori con proprietà comuni. Il vincolo è indicato con la stringa *[property PropertyName]*. Il vincolo **if-property** limita le combinazioni di un valore di una caratteristica di un parametro con particolari valori selezionati per una caratteristica di un parametro diverso. Il vincolo è indicato con la stringa *[if PropertyName]*. Infine il vincolo **single** si comporta come error, limitando il numero di occorrenze di un valore nella combinazione a 1.

Boundary value testing:
motivazioni
Abbiamo suddiviso i domini di input in classi appropriate, partendo dal presupposto che il comportamento del programma è "simile"

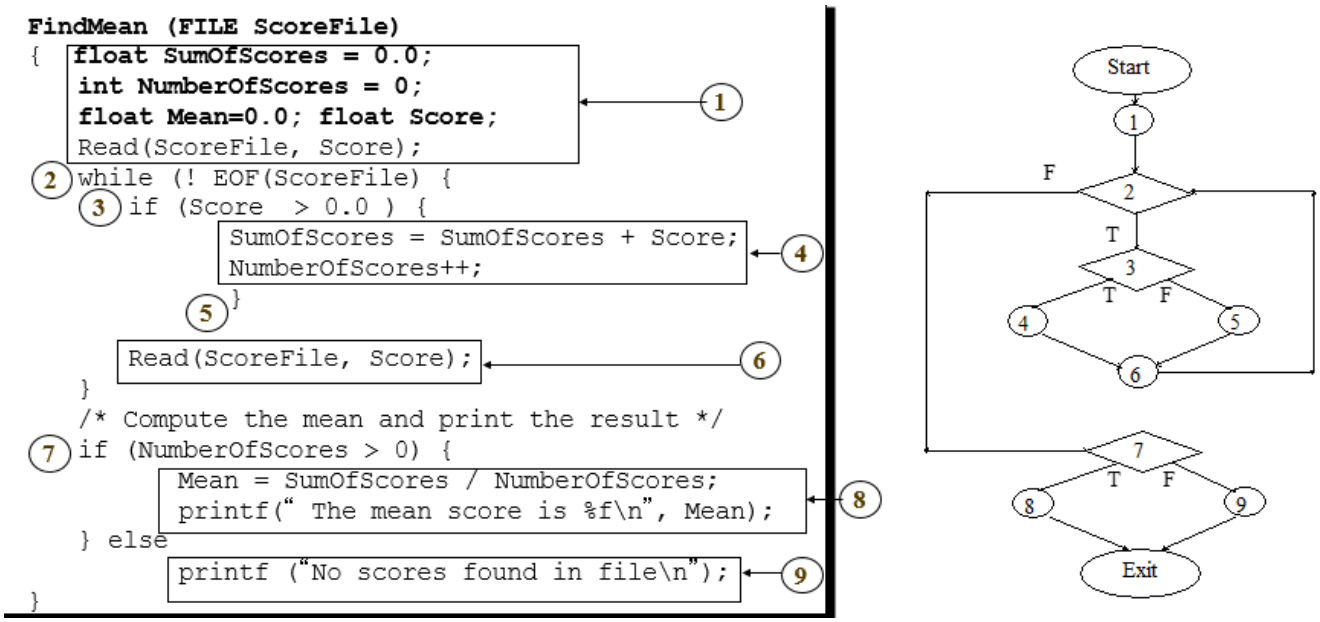
- Alcuni tipici errori di programmazione si trovano al confine tra le diverse classi
- Questo è il test si concentra sui valori al limite
- Più semplice ma complementare alle tecniche precedenti

Si prendono come valori quelli al minimo, un pò sopra al minimo, normale, un po sotto al massimo e quello massimo: min, min +, nom, max-, max.

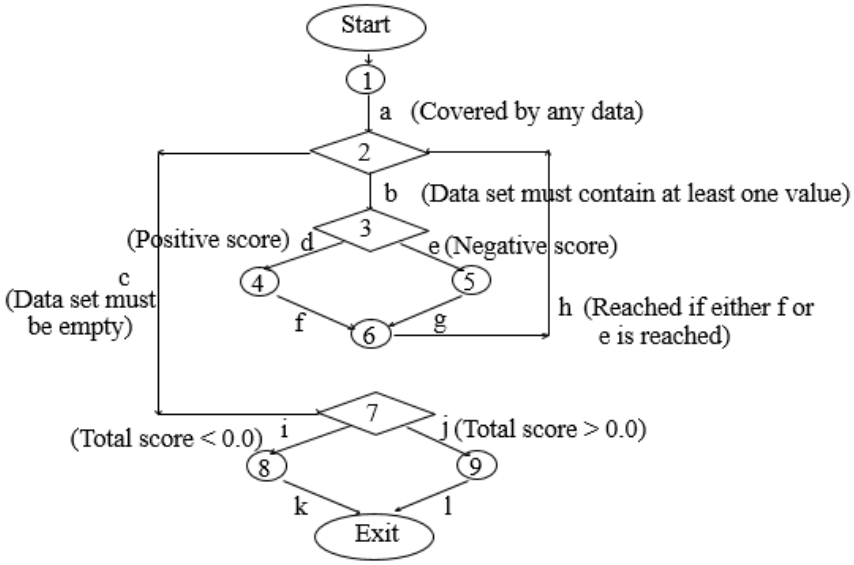
Il whitetesting è diviso in quattro sottotipi:

- **Statement testing:** vengono testati i singoli statement del codice
- **Loop testing:** vengono dati vari input in modo da effettuare ciascuna delle seguenti operazioni: saltare un ciclo, entrare in un ciclo una sola volta e ripetere un ciclo più volte.
- **Path testing:** viene costruito un diagramma di flusso del programma e si controlla se tutti i blocci del diagramma vengono attraversati. Vengono individuati dei test case che possano essere attraversati tutti i blocchi del programma.
- **Branch testing**: ci si assicura che ogni uscita di un'istruzione condizionale sia testa almeno una volta (es. il blocco if o il blocco else devono essere attraversati almeno una volta).

Esempio:



Finding the Test Cases



White-box vs Black-box:

- **White-box:** numero potenzialmente infinito di percorsi da testare. I test white-box spesso verificano ciò che viene fatto, invece di ciò che dovrebbe essere fatto. Impossibile rilevare i casi d'uso mancanti.
- **Black-box:** Potenziale esplosione combinatoria di casi di test (dati validi e non validi). Spesso non è chiaro se i test test selezionati rivelano un particolare errore. Non scopre casi d'uso estranei ("caratteristiche").

Sono necessari entrambi i tipi di test. I test white-box e black box sono gli estremi di un continuum di test. Qualsiasi scelta del caso di prova si trova in mezzo e dipende da quanto segue:

- Numero di possibili percorsi logici
- Natura dei dati di input
- Quantità di calcolo
- Complessità di algoritmi e strutture dati

4 fasi del testing

- Seleziona cosa deve essere misurato:
  - Completezza dei requisiti
  - Codice testato per affidabilità
  - Design testato per la coesione
- Decidi come è fatto il test:
  - Ispezione del codice
  - Black-box, White-box,
  - Seleziona la strategia di test di integrazione (big bang, bottom up, top down, sandwich)
- Sviluppa casi di test:
  - Un test case è un insieme di dati di test o situazioni che verranno utilizzati per esercitare l'unità (codice, modulo, sistema) in fase di test.

- Crea il test oracle:
    - Un oracolo contiene i risultati previsti per una serie di casi di test. Il test oracle deve essere annotato prima che avvenga il test vero e proprio

Usi dei vari tipi di test

- Black-box test: testa i casi d'uso e il modello funzionale
- White-box test: testa il modello dinamico
- Data-structure test: testa il modello a oggetti

10.3.5 Integration testing

L'utilità di questo testing è quello di rilevare errori che non sono stati rilevati con l'unit testing, Un piccolo gruppo di componenti realizzate vengono messe insieme. Non appena il piccolo sottoinsieme è perfettamente funzionante e non vengono evidenziati errori è possibile aggiungere componenti all'insieme. Esistono varie strategie che decidono in che modo vengono scelti i sottoinsiemi di unità.

10.3.5.1 Big bang testing

Le componenti vengono testate prima singolarmente e poi messe insieme in un unico sistema. Il vantaggio è che non è necessario sviluppare stub o driver per rendere funzionale un sottoinsieme. È però difficile, in sistemi complessi, individuare la componente responsabile di un errore.

10.3.5.2 Bottom-up testing

Viene testata ogni componente del livello più basso, vengono integrate e successivamente unite con le componenti del livello superiore. In questo caso non è necessario avere dei test stub in quanto si inizia ad integrare dal livello più basso a salire. Un vantaggio di questa tecnica è che gli errori nelle interfacce grafiche vengono trovati subito in quanto, quando si testa l'interfaccia si ha già un sistema sottostante funzionante e ben definito.

10.3.5.3 Top-down testing

Contrariamente al testing Bottom-up si inizia ad integrare le componenti del livello più alto e successivamente si intergrano quelle del livello inferiore. Non sono necessari test driver ma solo test stub per simulare le componenti inferiori. Uno dei problemi di questo tipo di testing è che è necessario scrivere molti stub.

10.3.5.4 Sandwitch testing

Questa strategia combina bottom-up e top-down insieme cercando di usare il meglio di queste. Il sistema viene diviso in tre livelli: il livello target, un livello sopra il target e uno sotto. In questo modo possiamo effettuare il testing top-down e bottom-up in parallelo con lo scopo di arrivare ad integrare il target level. Un problema di questo testing è che le componenti non vengono testate separatamente prima di integrarle. Esiste infatti una modifica del sandwitch testing che testa le singole componenti prima di intergrarle.

10.3.5.5 Step dell'integration testing

1. Sulla base della strategia di integrazione, selezionare una componente da testare. Unit test di tutte le classi della componente.
2. Unisci insieme agli altri componenti; eseguire eventuali correzioni preliminari necessarie per rendere operativo il test di integrazione (driver, stub)
3. Effettuare test funzionali: definire i casi di test che esercitano tutti i casi di utilizzo con il componente selezionato

4. Eseguire test strutturali: definire i casi di test che esercitano il componente selezionato
5. Eseguire test delle prestazioni
6. Tenere un registro dei casi di test e delle attività di test.
7. Ripetere i passaggi da 1 a 7 fino a quando non viene testato il sistema completo.

L'obiettivo principale del test di integrazione è identificare gli errori nella configurazione del componente corrente.

10.3.6 System testing

Una volta che le componenti sono state integrate è testate prima con unit testing e poi integration testing è necessaria una fase di testing globale. Le attività di questa fase sono le seguenti:

10.3.6.1 Test funzionale (o test dei requisiti)

Vengono controllate le differenze tra i requisiti funzionali e il sistema. I test case vengono presi dai casi d'uso. La differenza con usability testing è che mentre negli usability testing vengono trovate differenze tra il modello dei casi d'uso e le aspettative dell'utente, qui vengono trovate le differenze tra il modello dei casi d'uso e il comportamento osservato.

10.3.6.2 Performance testing

Questo test trova le differenze tra gli obbiettivi di design specificati e il sistema. Vengono effettuati i seguenti test:

- **Stress testing:** controlla se il sistema può rispondere a molte richieste simultanee.
- **Volume testing:** cerca errori quando il sistema elabora una grossa quantità di dati
- **Security testing:** cerca di trovare falle nella sicurezza del sistema usando tipici errori di sicurezza.
- **Timing testing:** prova a valutare il tempo di risposta del sistema.
- **Recovery test:** valuta l'abilità del sistema di ripristinarsi dopo una condizione di errore.

10.3.6.3 Pilot testing

Durante questo testing il sistema viene installato e fatto usare da una selezionata nicchia di utenti. Successivamente gli utenti vengono invitati a dare il loro feedback agli sviluppatori. Un alpha test è un test fatto nell'ambiente di sviluppo. Un beta test è un test fatto nell'ambiente di utilizzo.

10.3.6.4 Acceptance testing

L'utente può valutare in tre modi un test di acceptance:

- **Benchmark:** il cliente prepara una serie di test case su cui il sistema deve operare.
- **Competitor:** il sistema viene confrontato e testato rispetto ad un altro sistema concorrente.
- **Shadow testing:** viene testato il sistema legacy e il sistema attuale e gli output vengono messi a confronto.

10.3.6.5 Installation testing

Dopo che il sistema è stato accettato viene installato nel suo ambiente. In molti casi il test di installazione ripete i test case eseguiti durante il function testing e performance testing. Quando il cliente è soddisfatto, il sistema viene formalmente rilasciato, ed è pronto per l'uso.

10.3.6.5 Managing testing

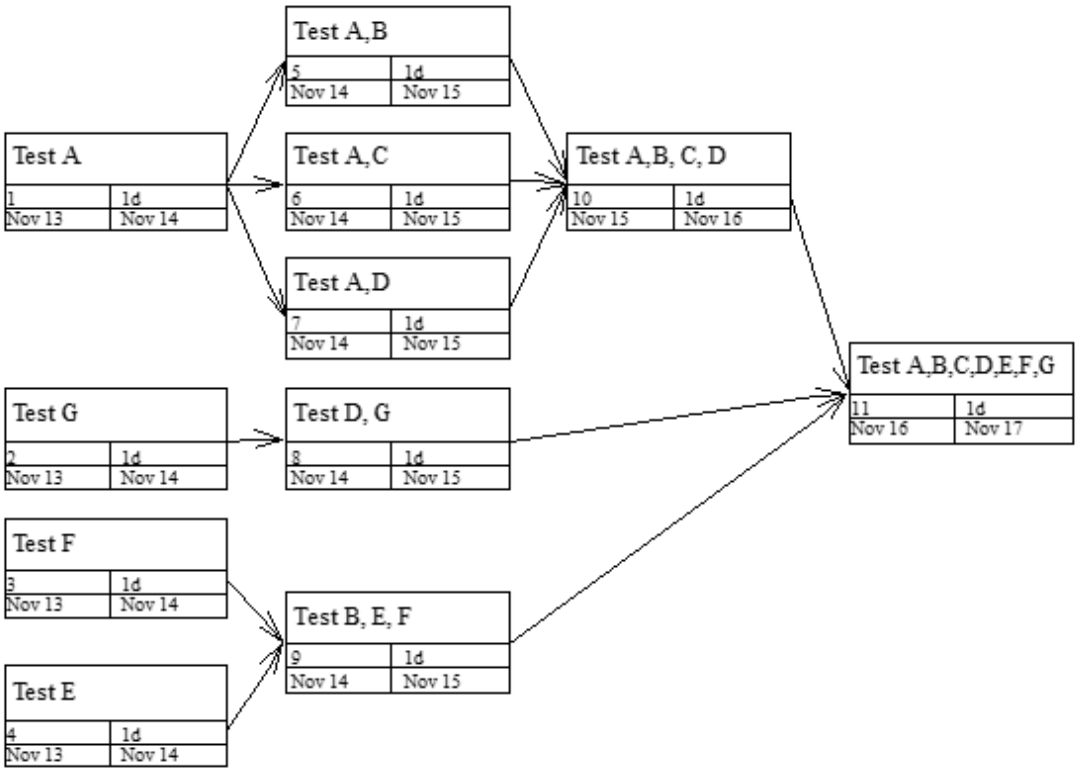
- Come gestire le testing activity per minimizzare le risorse necessarie**.** Alla fine, gli sviluppatori dovrebbero rilevare e quindi riparare un numero sufficiente di bug tale che il

sistema soddisfi i requisiti funzionali e non funzionali entro un limite accettabile da parte del cliente**.**

- Pianificazione delle testing activity (diagramma di PERT che mostra le dipendenze)**:** le attività sono documentate in 4 tipi di documenti:
    1. il Test Plan che si focalizza sugli aspetti manageriali;
    2. Ogni test è documentato attraverso un Test Case Specification;
    3. Ogni esecuzione di un test case è documentata attraverso un Test Incident Report;
    4. Il Test Report Summary elenca tutti i fallimenti rilevati durante i test che devono essere investigati

I ruoli sono assegnati durante il testing.

Esempio: diagramma di PERT per un sandwich teasting



11 SCRUM

Scrum è un framework agile per la gestione del ciclo di sviluppo del software, iterativo ed incrementale, concepito per gestire progetti e prodotti software o applicazioni di sviluppo creato e sviluppato da Ken Schwaber e Jeff Sutherland.

[Storia. Nel 1986, Hirotaka Takeuchi e Ikujiro Nonaka descrissero un nuovo approccio allo sviluppo di prodotti commerciali che avrebbe aumentato la velocità e la flessibilità, basato su casi di studio presi dall'industria automobilistica e quella relativa alla realizzazione di fotocopiatrici e stampanti. Essi lo chiamarono *approccio olistico o rugby*, in quanto l'intero processo viene eseguito da un team interfunzionale su più fasi che si sovrappongono, dove la squadra "cerca di raggiungere l'obiettivo come unità, passando la palla avanti e indietro".

Il termine *Scrum* è mutuato dal termine del rugby che indica il pacchetto di mischia ed è evidentemente una metafora del team di sviluppo che deve lavorare insieme in modo che tutti gli attori del progetto spingano nella stessa direzione, agendo come un'unica entità coordinata.**]**

Scrum si basa sulla teoria dei controlli empirici di analisi strumentale e funzionale di processo o empirismo. L'empirismo afferma che la conoscenza deriva dall'esperienza e che le decisioni si

basano su ciò che si conosce. Scrum utilizza un metodo iterativo ed un approccio incrementale per ottimizzare la prevedibilità ed il controllo del rischio.

Sono tre i pilastri fondamentali:

- **Trasparenza:** Gli aspetti significativi del processo devono essere visibili ai responsabili del lavoro. La trasparenza richiede che quegli aspetti siano definiti da uno standard comune in modo tale che gli osservatori condividano una comune comprensione di ciò che viene visto.
- **Ispezione:** Chi utilizza Scrum deve ispezionare frequentemente gli artefatti prodotti ed i progressi realizzati verso il conseguimento degli obiettivi prestabiliti, individuando in tal modo eventuali difformità rispetto a quanto si intende realizzare. La frequenza delle ispezioni non deve essere tale da determinare un'interruzione del lavoro in corso. Le ispezioni devono essere eseguite in modo accurato e da persone qualificate.
- **Adattamento:** Se chi ispeziona verifica che uno o più aspetti del processo di produzione sono al di fuori dei limiti accettabili e che il prodotto finale non potrà essere accettato, deve intervenire sul processo stesso o sul materiale prodotto dalla lavorazione. L'intervento deve essere portato a termine il più rapidamente possibile per ridurre al minimo l'ulteriore scarto rispetto agli obiettivi prestabiliti.

Scrum è un framework di processo che prevede di dividere il progetto in blocchi rapidi di lavoro (Sprint) alla fine di ciascuno dei quali creare un incremento del software.
Esso indica come definire i dettagli del lavoro da fare nell'immediato futuro e prevede vari meeting con caratteristiche precise per creare occasioni di ispezione e controllo del lavoro svolto.
Il framework Scrum è costituito dai Team Scrum e dai ruoli, eventi, artefatti e regole a essi associati. Ogni parte del framework serve a uno specifico scopo ed è essenziale per il successo e l'utilizzo di Scrum.

11 Ruoli

11.1.1 Il Team Scrum

Il Team Scrum è formato dal **Product Owner**, Il **Team di sviluppo** (**Development Team**) e da uno **Scrum Master**. I Team Scrum sono auto-organizzati e cross-funzionali: scelgono come meglio compiere il lavoro organizzandosi e coordinandosi al proprio interno e hanno tutte le competenze necessarie per realizzare il lavoro senza dover dipendere da nessuno al di fuori del team. Il modello di team in Scrum è progettato per ottimizzare la flessibilità, la creatività e la produttività.

I Team Scrum rilasciano i prodotti in modo iterativo e incrementale, massimizzando le opportunità di feedback. I rilasci incrementali di prodotto “Fatto” garantiscono che una versione potenzialmente utile del prodotto funzionante sia sempre disponibile.

11.1.2 Il Product Owner

Il Product Owner rappresenta gli stakeholders ed è la voce del cliente. È responsabile per assicurare che il team fornisca valore al business. Il Product Owner definisce gli item centrati sui bisogni dei clienti, assegna loro la priorità, e li aggiunge al product backlog. I team Scrum debbono avere un Product Owner e si raccomanda che questo ruolo non sia combinato con quello dello Scrum Master.

11.1.3 Team di sviluppo

Il Team di sviluppo è responsabile della consegna del prodotto, con incrementi di caratteristiche, che sia potenzialmente rilasciabile alla fine di ogni Sprint.

Un Team di sviluppo è composto da 3-9 persone, che realizzano il lavoro effettivo (analisi, progettazione, sviluppo, test, comunicazione tecnica, documentazione…).

11.1.4 Scrum Master

Lo ScrumMaster è responsabile della rimozione degli ostacoli che limitano la capacità del team di raggiungere l'obiettivo dello Sprint e i deliverable previsti. Sebbene sia un ruolo manageriale, lo ScrumMaster non è il team leader, ma piuttosto colui che facilita una corretta esecuzione del processo. Lo ScrumMaster detiene l'autorità relativa all'applicazione delle norme, spesso presiede le riunioni importanti e pone sfide alla squadra per migliorarla.
Una parte fondamentale del ruolo di ScrumMaster è quello di proteggere il Team di sviluppo e tenerlo concentrato sui compiti fungendo da cuscinetto verso qualsiasi influenza di distrazione.
Lo ScrumMaster differisce da un Project Manager in quanto quest'ultimo può avere responsabilità di gestione del personale che invece non sono a carico dello ScrumMaster.

11.1.5 Ruoli ausiliari

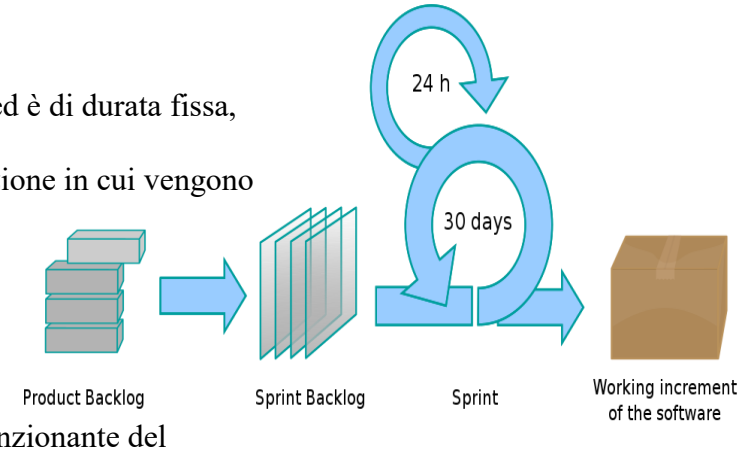
I ruoli ausiliari nei team Scrum sono quelli che non hanno alcun ruolo formale o coinvolgimento frequente nel processo Scrum ma che comunque devono essere presi in considerazione.

- **Stakeholder:** I principali stakeholder sono clienti, venditori. Sono le persone che permettono il progetto e per i quali il progetto produce i benefici concordati che ne giustificano la produzione. Sono coinvolti direttamente nel processo solo durante le Sprint Review.
- **Manager:** Persone che controllano l'ambiente di lavoro.

11.2 Cerimonie

11.2.1 Sprint

Lo sprint è un'unità di base dello sviluppo in Scrum ed è di durata fissa, generalmente da una a quattro settimane.
Ogni Sprint è preceduto da una riunione di pianificazione in cui vengono identificati gli obiettivi e vengono stimati i tempi.
Durante uno sprint non è permesso cambiare gli obiettivi, quindi le modifiche sono sospese fino alla successiva riunione di pianificazione, e potranno essere prese in considerazione nel successivo Sprint.
Al termine di ogni sprint il team di sviluppo consegna una versione potenzialmente completa e funzionante del prodotto, contenente gli avanzamenti decisi nella riunione di pianificazione dello sprint.



Nel corso di ogni sprint, il team crea porzioni complete di un prodotto. L'insieme delle funzionalità che vengono inserite in un determinato sprint provengono dal **product "backlog"**, che è una lista ordinata di requisiti. La selezione di quali item del backlog verranno effettivamente inseriti nello sprint (a costituire l'obiettivo dello sprint o **"sprint goal"**) viene effettuata durante lo sprint planning meeting. Durante questo meeting, il Product Owner comunica al team quali item del product backlog vorrebbe che fossero completati (quelli con più alta priorità). Il team determina quindi quanti di questi pensa di poter completare durante il prossimo sprint e registra questo dato nello **sprint backlog.** Lo sprint backlog è di esclusiva proprietà del team di sviluppo, quindi durante uno sprint la modifica dello sprint backlog non è consentita a nessun altro ad eccezione del team di sviluppo. Lo sprint goal non dovrebbe essere cambiato durante lo sprint. Lo sviluppo è di durata fissa, in modo tale che lo sprint termini alla data prefissata; se i requisiti non sono stati completati per una qualsiasi ragione vengono esclusi dalla review e reinseriti nel product backlog, a discrezione del Product Owner.
Le “cerimonie” previste sono utilizzate in Scrum per creare regolarità e ridurre al minimo la necessità di riunioni non definite da Scrum stesso. Scrum utilizza cerimonie time-box, in modo che

ogni cerimonia abbia una durata massima. Questo assicura che una quantità appropriata di tempo è trascorsa pianificando senza permettere l'introduzione di sprechi nel processo di pianificazione. Lo Sprint contiene e consiste dello **Sprint Planning meeting**, del **Daily Scrum**, del lavoro di sviluppo, dello **Sprint Review** e della **Sprint Retrospective**.

11.2.2 Sprint planning meeting

All'inizio di ogni ciclo di sprint (ogni 7–30 giorni), viene tenuto uno "Sprint planning meeting".
Il lavoro da svolgere nello Sprint è pianificato durante lo Sprint Planning Meeting.
Lo Sprint Planning include i seguenti elementi:

- Selezionare il lavoro da fare.
- Preparare lo Sprint Backlog che dettagli il tempo necessario per fare quel lavoro, con tutta la squadra
- Identificare e comunicare la maggior parte del lavoro che probabilmente verrà effettuato durante l'attuale sprint.

Questo incontro è frequentato dal Product Owner, Scrum Master, e dall'intero Team di sviluppo. Possono partecipare anche tutti i manager del caso interessati o i rappresentanti della clientela.

11.2.3 Daily Scrum meeting

Ogni giorno durante lo Sprint, viene tenuta una riunione di comunicazione del team di progetto. Questo meeting viene chiamato **"daily scrum"**, o **"daily standup"** ed ha un insieme di regole specifiche:

- Tutti i membri del Team di sviluppo vengono preparati con gli aggiornamenti per la riunione.
- L'incontro inizia puntualmente anche se qualche membro del team è assente.
- Il meeting dovrebbe avvenire ogni giorno nello stesso luogo e allo stesso tempo per ridurre la complessità (tipicamente davanti alla macchinetta del caffè).
- La durata del meeting è fissata al tempo massimo di 15 minuti.
- Si partecipa rimanendo in piedi, per non dare modo ai partecipanti di distrarsi ed isolarsi come accade nelle riunioni "tradizionali".
- Tutti sono benvenuti, ma normalmente solo i ruoli principali possono parlare.

Durante l'incontro quotidiano, ogni membro del team risponde a tre domande:

- Che cosa è stato ieri?
- Che cosa verrà fatto oggi?
- Quali sono stati i problemi incontrati?

11.2.4 Sprint Review

Alla fine dello Sprint si tiene l'incontro di Sprint Review. Durante la riunione di Sprint Review il Team di Sviluppo e gli stakeholder collaborano su ciò che è stato fatto durante lo Sprint. In conformità a questo e dei cambiamenti al Product Backlog fatti durante lo Sprint, i partecipanti collaborano sulle prossime cose che potrebbero esser fatte. Si tratta di un incontro informale e la presentazione dell'Incremento ha lo scopo di suscitare commenti e promuovere la collaborazione.
La Sprint Review include i seguenti elementi:

- Il Product Owner identifica ciò che è stato "Fatto" e ciò che non è stato "Fatto";
- Il Team di Sviluppo discute su cosa è andato bene durante lo Sprint, quali problemi si sono incontrati e come questi problemi sono stati risolti;
- Il Team di Sviluppo mostra il lavoro che ha "Fatto" e risponde alle domande sull'incremento;

- Il Product Owner discute il Product Backlog così com'è. Questi progetta la possibile data di completamento in base alla misura del progresso fino ad oggi;
- L'intero gruppo collabora su cosa fare dopo, così la Sprint Review fornisce un prezioso contributo alle successive riunioni di Sprint Planning.

11.2.5 Sprint Retrospective

Dopo la Sprint Review e prima del prossimo incontro Sprint Planning, il Team Scrum si riunisce per lo Sprint Retrospective.
Lo scopo della Sprint Retrospective è di:

- Esaminare come l'ultimo Sprint è andato per quanto riguarda le persone, le relazioni, I processi e gli strumenti;
- Identificare e ordinare i maggiori elementi che son andati bene e il potenziale di miglioramento;
- Creare un piano per attuare i miglioramenti al modo di lavorare dello Scrum Team.

11.3 Artefatti

Gli artefatti definiti da Scrum sono specificatamente progettati per massimizzare la trasparenza delle informazioni chiavi necessarie ad assicurare ai Team Scrum il successo nella realizzazione di un Incremento “Fatto”.

11.3.1 Product backlog

Il product backlog è una lista ordinata dei "requisiti" relativi ad un prodotto. Contiene i Product Backlog Item (PBI) a cui viene assegnata dal Product Owner una priorità in base a considerazioni quali il rischio, il valore di business, le date in cui devono essere realizzati. Il product backlog rappresenta “cosa” deve essere fatto, organizzato in base all'ordine relativo in cui dovrà essere realizzato. È aperto e modificabile da tutti, ma il Product Owner è il responsabile ultimo della sua gestione e delle priorità da dare alle storie nel backlog per il Team di sviluppo. Il Product Backlog, e il valore di business associato a ciascun item è responsabilità del Product Owner.

11.3.2 Sprint backlog

Lo Sprint backlog è la lista del lavoro che il team di sviluppo deve effettuare nel corso dello sprint successivo. Questa lista viene generata selezionando una quantità di storie/funzionalità a partire dalla cima del product backlog determinata da quanto il Team di sviluppo ritiene possa realizzare durante lo sprint: ovvero avere una quantità di lavoro tale da riempire lo sprint.
Lo sprint backlog è di proprietà del Team di sviluppo, e tutte le stime incluse sono effettuate dal team stesso. Spesso viene utilizzata una task board per visualizzare i cambiamenti di stato dei task nello sprint corrente, come ad esempio **“to do”, “in progress”** e **“done”.**

11.3.3 Burn down charts

Un burn down chart è una rappresentazione grafica del lavoro da fare su un progetto nel tempo. Utile per prevedere quando avverrà il completamento del lavoro.