Riassunti di

INGEGNERIA DEL SOFTWARE

DI AMELIO FRANCESCO

Indice

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.

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. E' 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.

1.2 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.

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 su i 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)

E' 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.

2.3 Modello verification e validation (V&V)

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.

2.4 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.

2.5 Modello di sviluppo basato sul riuso

E' 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 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**
  Si inizia a sviluppare le parti del sistema che sono già ben specificate aggiungendo nuove caratteristiche secondo le necessità fornite dal cliente man mano.
- **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.

2.7 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**
  E' 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.

2.8 **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.

2.9 **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.

2.10 **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 le 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.

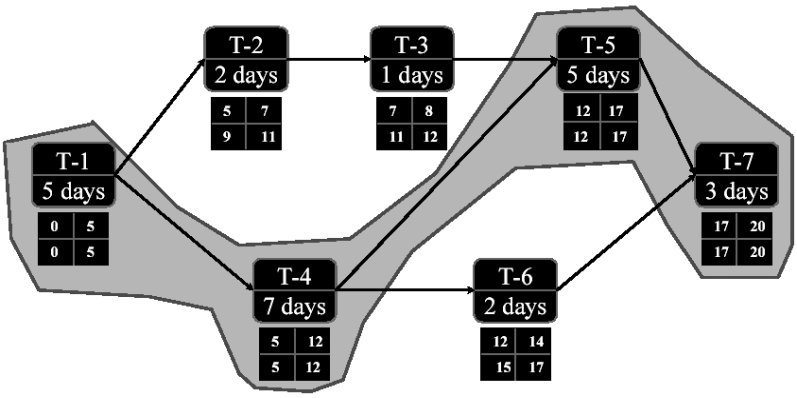


Figura 1 Cammino critico del grafo della attività

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.
- **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**

[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)

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.

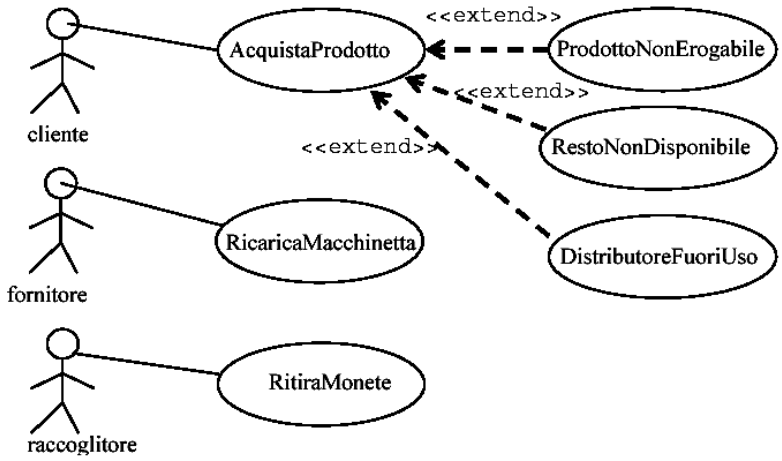


Figura 2 Casi d'uso per un distributore di snack

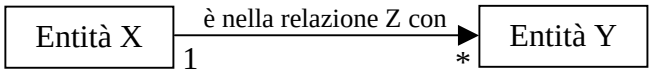
4.2 Diagrammi di classi (class diagrams)

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 il suoi comportamenti (le operazioni che essa svolge). Ogni attributo ha un tipo e ogni operazione ha una firma.

Le entità del grafo possono essere interconnesse tramite archi con freccia, senza freccia o tratteggiati ed avere le seguenti relazioni:

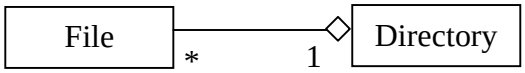
4.2.1 Associazione

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:



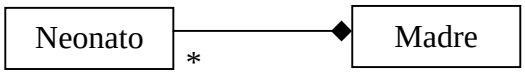
4.2.2 Aggregazione

E' possibile rappresentare chi contiene cosa sotto forma di grafo (o albero) utilizzando la linea con il rombo terminatore come nella figura.



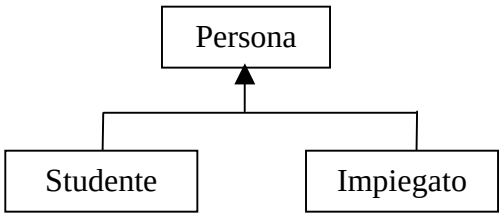
4.2.3 Composizione

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



4.2.4 Generalizzazione (o ereditarietà)

Indica l'ereditarietà tra entità: la classe figlio eredità attributi e operazioni del padre semplificando il modello ed eliminando la ridondanza.



4.3 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.

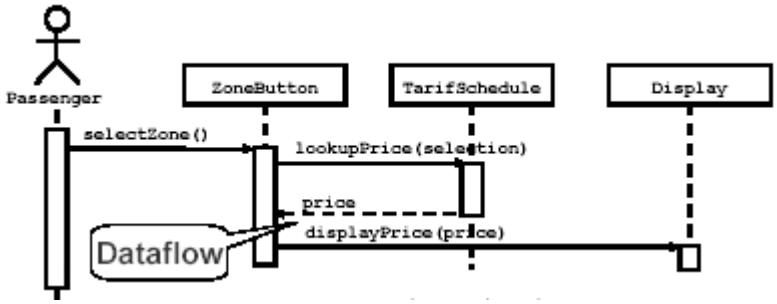


Figura 3 Diagramma sequenziale di un passeggero che acquista un biglietto

4.4 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.

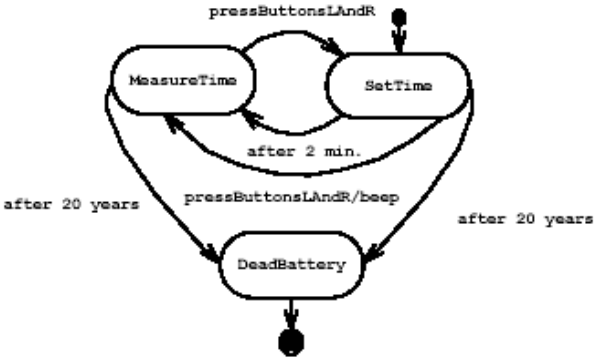


Figura 4 Diagramma a stati di un orologio

4.5 Diagrammi delle attività (activity diagrams)

E' 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.

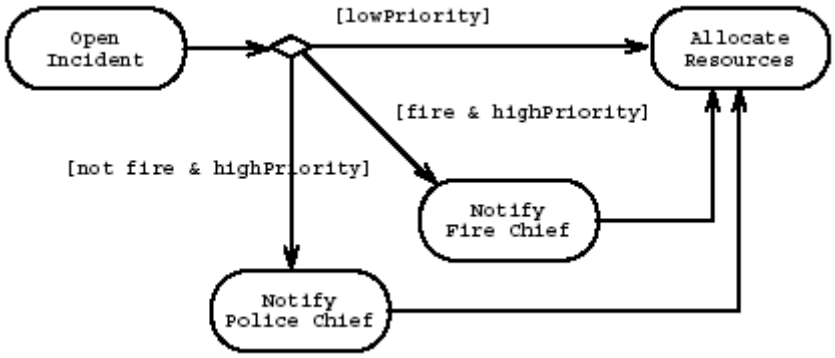


Figura 5 Diagramma delle attività per l'apertura di un incidente

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

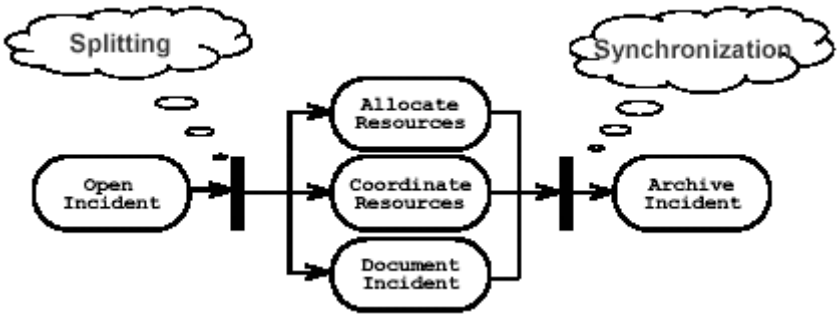


Figura 6 Diagramma delle attività con concorrenza

4.6 **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.

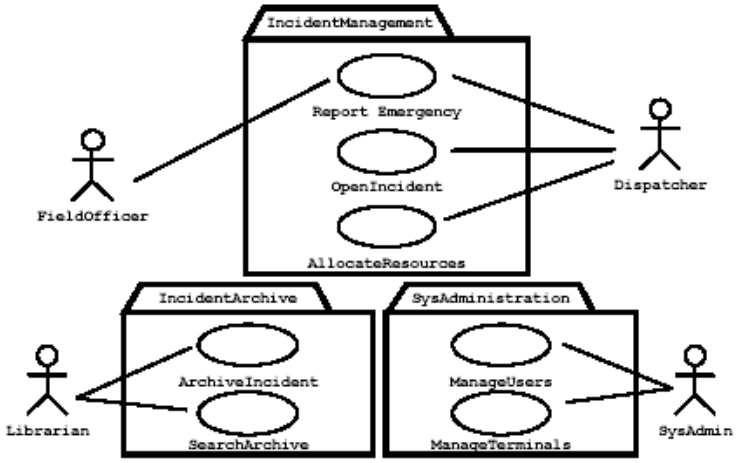


Figura 7 Raggruppamento

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.

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 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
2. Identificare gli scenari
3. Identificare i casi d'uso
4. Raffinare i casi d'uso
5. Identificare le relazioni tra gli attori e i casi d'uso
6. Identificare gli oggetti partecipanti (verranno ripresi nella fase di analisi)
7. Identificare le richieste non funzionali

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**
  E' 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 fare 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.

5.3.7 Identificare i requisiti non funzionali
--- *Vedi nelle pagine precedenti sulla classificazione dei requisiti* --

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 del 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 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>>.

6.2 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

6.2.1 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 sulla 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

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
- Entità del mondo reale che il sistema deve tracciare
- Attività del mondo reale che il sistema deve tracciare
- Sorgenti di dati

6.2.2 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.

6.2.3 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.

6.2.4 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

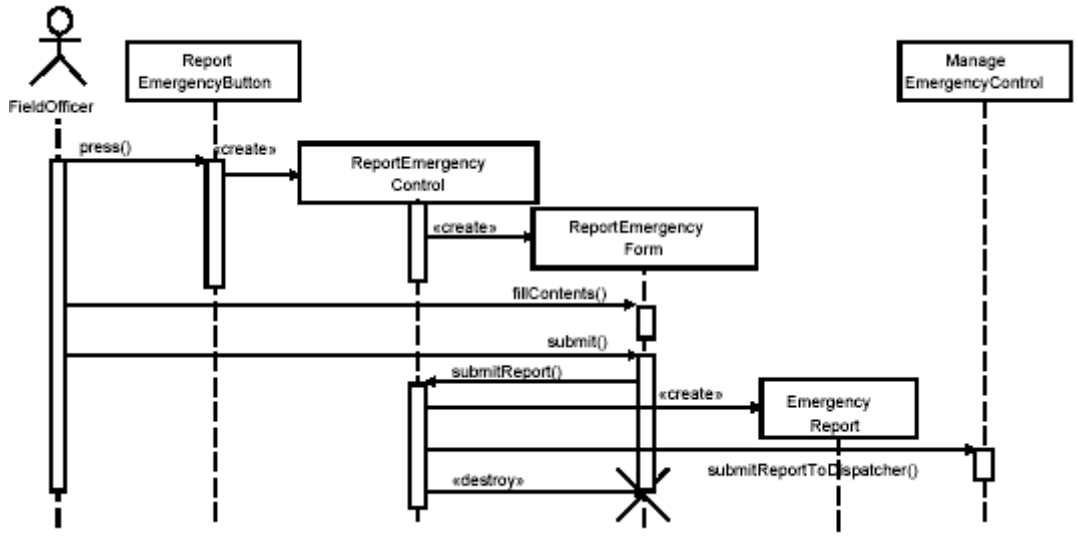


Figura 9 Esempio di sequence diagram

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.

6.2.5 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 associazione 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”

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

6.2.7 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.

6.2.8 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.

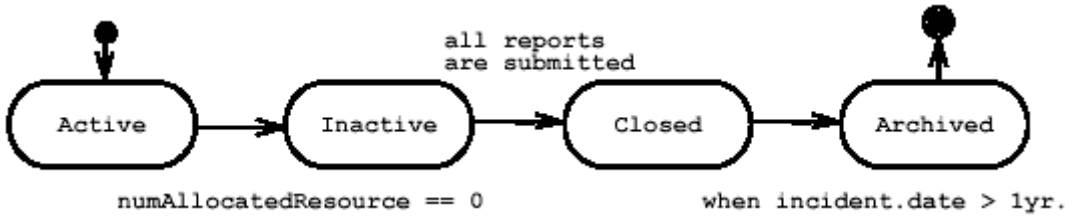


Figura 10 Un esempio di diagramma a stati

6.2.9 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

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

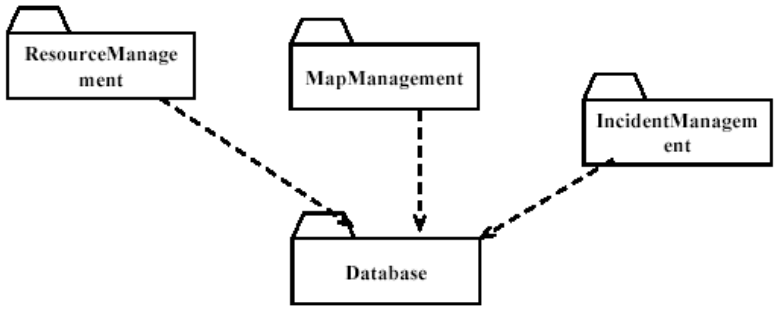


Figura 11 Composizione dei sottosistemi prima dell'aggiunta di una componente

potrebbe diventare

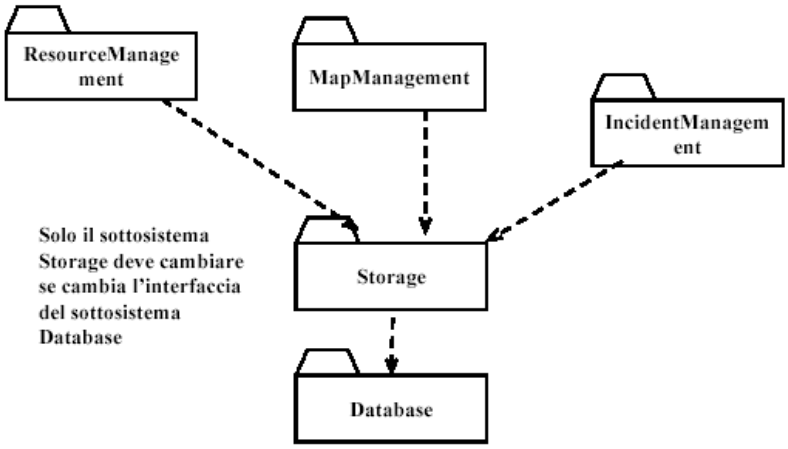


Figura 12 Composizione dopo l'aggiunta della componente Storage

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

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.

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.

7.4 Architetture software

Un architettura software include scelte relative alla decomposizione in sottosistemi, flusso di controllo globale, gestione delle condizione limite e i protocolli di comunicazione tra i sottosistemi.
E' 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.

7.4.2 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 .

7.4.4 Peer-To-Peer

E' 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 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.

7.4.7 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

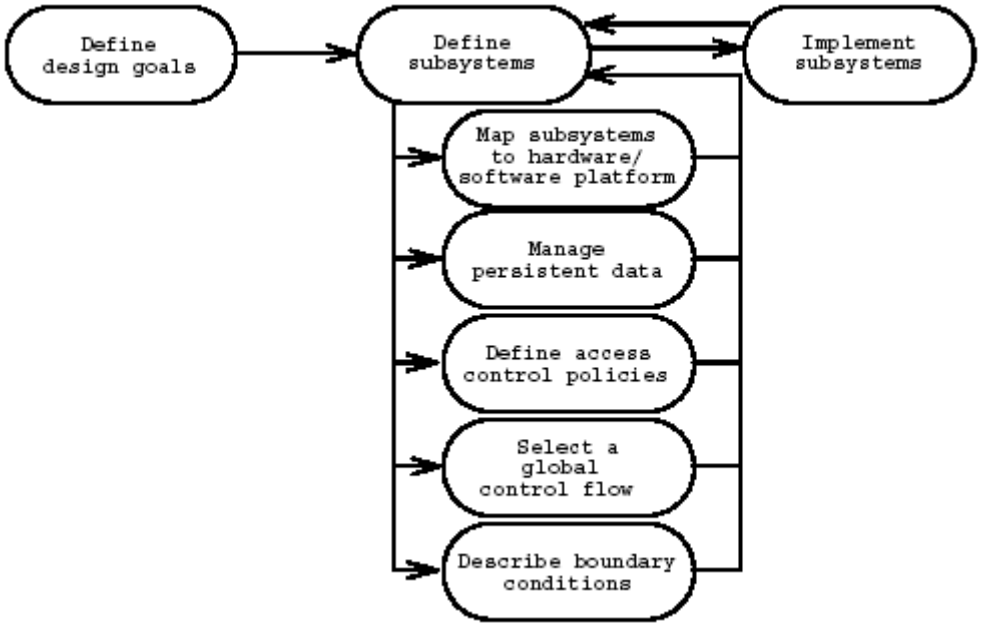


Figura 13 Attività del System Design

7.5.1 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.

7.5.2 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.

7.5.2.1 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.

7.5.2.2 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.

7.5.2.3 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.

7.5.2.4 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

7.5.3 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. E' 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.

7.5.4 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:

7.5.4.1 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.

7.5.4.2 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.

7.5.4.3 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.

7.5.5 Identificare le condizioni limite

Le condizioni limite del sistema includono lo **startup**, lo **shutdown**, l'inizializzazione e le 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.

7.5.6 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.

7.5.7 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

8 Object design

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 cambiamenti 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 Bridge pattern

Quando si usa uno sviluppo di tipo incrementale con testing e integrazione di sottosistemi creati da diversi sviluppatori, spesso si hanno dei ritardi nell'integrazione dei sottosistemi. Un problema simile si ha quando si vogliono avere diverse implementazioni dello stesso sottosistema (ad esempio una ottimizza la memoria e l'altra la complessità).
Per evitare ciò è possibile usare il bridge pattern che fa uso di un interfaccia comune (*Implementor*) ereditata da tutti i sottosistemi che implementano la stessa funzionalità (ConcreteImplementorA e ConcreteImplementorB).
Gli oggetti ConcreteImplementorX vengono creati e inizializzati dalla classe Abstraction che ne mantiene il riferimento.
In java questo può essere fatto creando un interfaccia (es *Memorizzazione*) che viene implementata da diverse classi (es MemorizzazioneSuFile e MemorizzazioneSuDBMS).

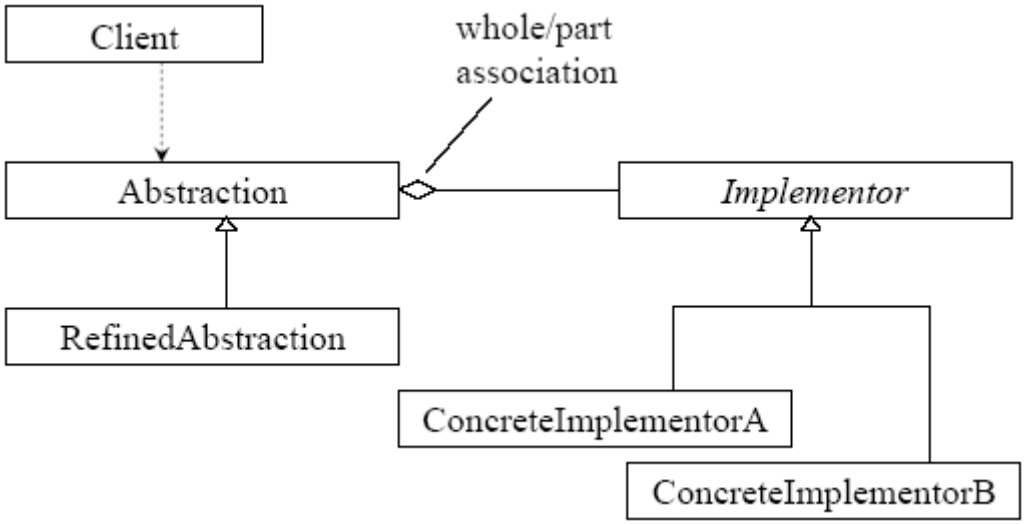


Figura 14 Class diagram di Bridge Pattern

8.2.2 Adapter pattern

Il pattern adapter viene usato quando è necessario inglobale componenti esistenti nel sistema. Questo tipo di soluzione viene adottata spesso nei sistemi interattivi, dove si usano finestre, dialog e bottoni. Tipicamente quando si riusano componenti legacy o off-the-shelf gli sviluppatori devono lavorare con codice che non possono modificare e che solitamente non è stato progettato per il loro sistema.
Il pattern adapter converte l'interfaccia esistente di un sistema nell'interfaccia che gli sviluppatori si aspettano di trovare. Una classe Adapter fa da connettore tra la classe che userà l'applicazione (ClientInterface) e la classe legacy.

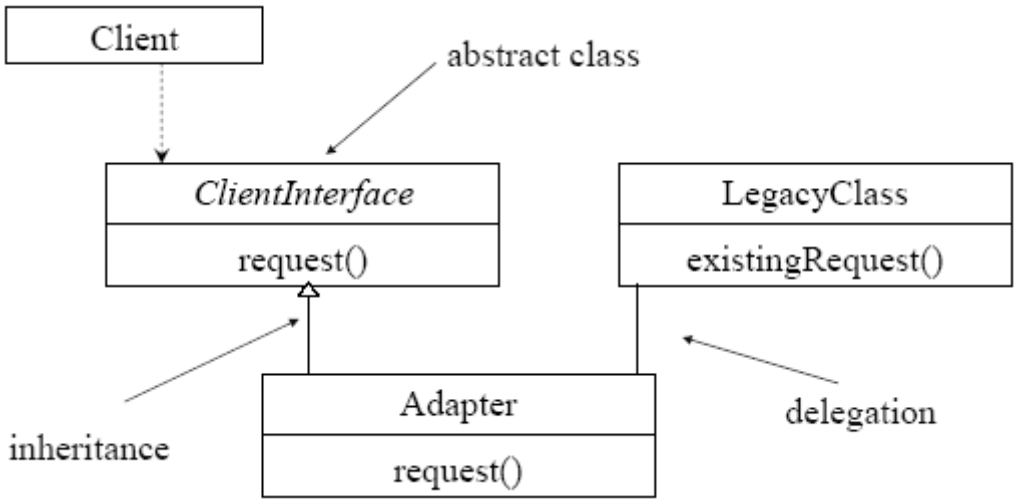


Figura 15 Class diagram di Adapter Pattern

8.2.3 Strategy pattern

Consideriamo una situazione in cui ci sono delle scelte (strategie) da effettuare in base ad una situazione corrente (es il livello di traffico su una rete). Questo pattern fornisce l'abilità di funzionamento al sistema anche in situazioni che modificano a runtime (dovute a cambiamenti dell'ambiente).
Il pattern coinvolge una classe Policy che si occupa di identificare i cambiamenti nell'ambiente, una classe Stategy che si occupa di fornire un interfaccia generica verso le attuali e future implementazioni del servizio e una classe Context che viene opportunamente configurata dal

Policy. In questo pattern la classe Policy seleziona la strategia concreta da utilizzare e la configura sul Context, scegliendola in base alla situazione attuale.
A prima vista potrebbe sembrare simile al bridge pattern ma in questo caso l'ambiente esterno cambia a runtime (a differenza del bridge in cui la scelta viene fatta al momento dell'inizializzazione) e quindi il Policy seleziona la strategia da seguire settando lo strategy sul Context il quale si occuperà di chiudere la strategia vecchia ed attuare la nuova.

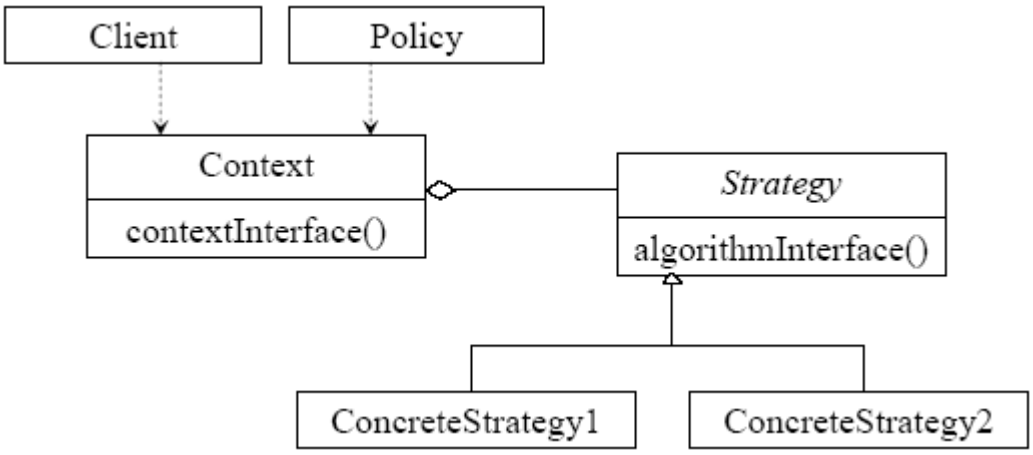


Figura 16 Class diagram di Strategy Pattern

8.2.4 Abstract factory pattern

In una situazione in cui un unico sistema deve interagire con delle componenti esterne sviluppate da diversi produttori (es condizionatore Daikin e condizionatore Samsung) non è facile ottenere l'interoperabilità delle componenti con il sistema in quanto ogni componente/produttore offrono le stesse funzionalità (es regolazione temperatura o spegnimento) ma con delle interfacce differenti.
In una situazione simile è possibile usare l'abstract factory pattern in cui sono presenti più factory (una per ogni casa produttrice), una classe astratta per ogni tipo di prodotto (es AbstractConditioner) e una implementazione di tale classe per ogni prodotto concreto (es DaikinConditioner e SamsungConditioner).
Tutte le factory presenti in realtà implementano una interfaccia comune *AbstractFactory* che permette loro un comportamento standard.
L'applicazione (Client) in questo modo può usare, oltre ad un interfaccia standard per accedere alle factory, anche un interfaccia generica per usare i prodotti.

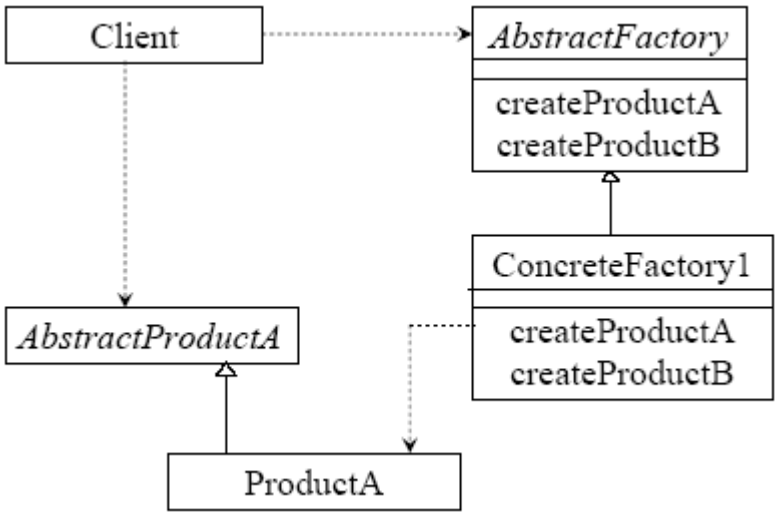


Figura 17 Class diagram di Abstract Factory pattern

8.2.5 Command pattern

In sistemi interattivi spesso si vuole che delle azioni siano registrabili, annullabili o rieseguibili. Supponiamo che esistano più tipi di azioni (es. annulla digitazione; annulla cancellazione; ripeti cancellazione ecc). Se vogliamo che un applicazione usi in modo standard le operazioni eseguibili su queste operazioni possiamo creare una interfaccia generica *Command* che viene implementata dai comandi concreti. Tutte le operazioni vengono registrate all'interno di un Invoker ed eseguite su un Receiver (immaginiamo che un Invoker è la memoria del calcolatore e il Receiver è l'interfaccia grafica che visualizza i comandi effettuati). Tra Invoker e ConcreteCommandX, Receiver e ConcreteCommandX vi è delegazione in quanto ConcreteCommand deve avere la possibilità di comunicare al receiver l'esecuzione/annullamento di un comando. Un Invoker deve inoltre memorizzare la lista dei comandi.

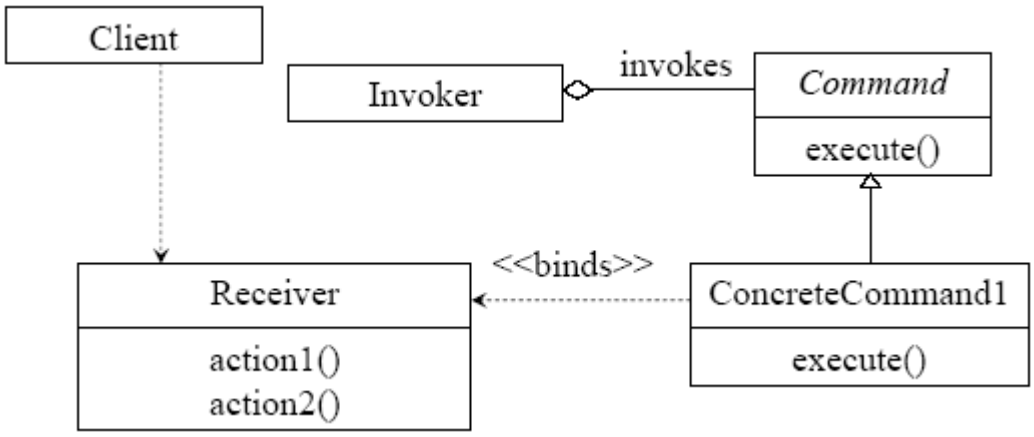


Figura 18 Class diagram di Command Pattern

8.2.6 Composite design pattern

Quando si ha a che fare con interfacce utente spesso si devono includere molte componenti grafiche all'interno di una finestra. Gestire le operazioni di ridimensionamento e spostamento senza alcun raggruppamento sarebbe una operazione complessa in quanto si dovrebbe interagire con molte componenti.

Ambienti per lo sviluppo di interfacce come Swing di Java utilizzano il composite design pattern. Le componenti grafiche vengono raggruppate in più Panel (es. parte superiore, centrale e inferiore di una finestra) lasciando ad ogni sottopannello il compito di gestire il layout delle componenti che contiene.

Nelle Swing alla radice della gerarchia di classi c'è un interfaccia *Component* che fornisce un comportamento generico per tutte le componenti grafiche (es. spostamento o ridimensionamento). Al di sotto di questa classe troviamo componenti come bottoni o label di testo e una particolare componente grafica Composite che rappresenta un contenitore di *Component* (es. può contenere bottoni, label, checkbox ecc).

Un panel ad esempio è un particolare Composite in cui è possibile inserire dei Button delle Label e qualsiasi altra classe che implementa *Component*. La strana relazione tra *Component* e Composite (ereditarietà + delegazione) è il cuore del composite design pattern e permette la realizzazione di interfacce grafiche complesse.

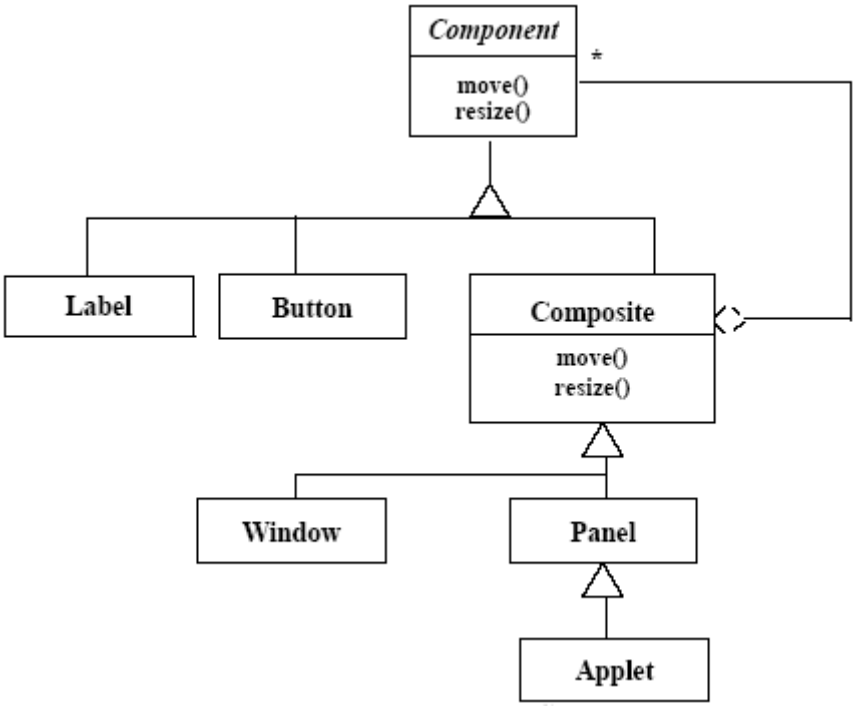


Figura 19 Class diagram di composite design pattern

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**
  E' 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 soddisfati dopo che l'operazione è stata eseguita. Sono usati per specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione.

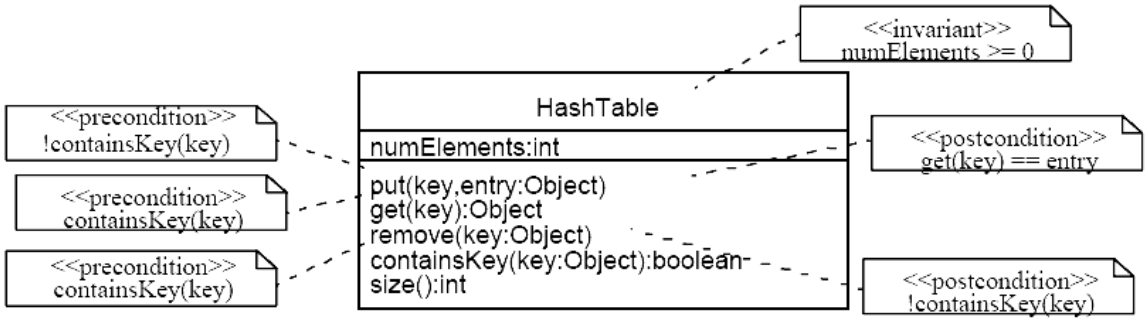


Figura 20 Esempio di contratti in UML

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).
E' possibile esprimere vincoli che coinvolgono più di una classe mettendo una freccia verso la classe che fa parte dell'espressione.

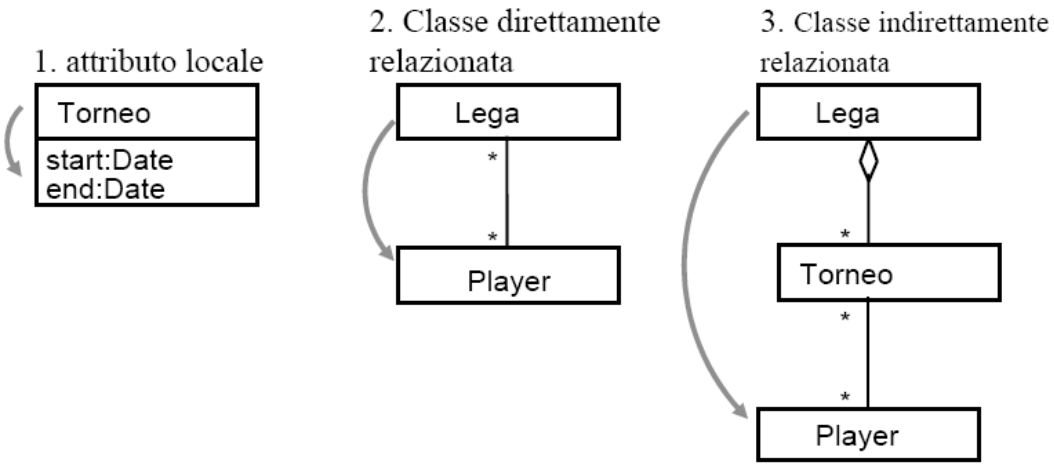


Figura 21 OCL in caso di vincoli che coinvolgono più classi

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.

E' possibile inoltre usare dei quantificatori come l'espressione *collezione->forAll(condizione)* che restituisce true se la condizione è vera per tutti gli elementi delle 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

E' 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

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. 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 .

9.1.1.2 Refactoring

I refactoring sono trasformazioni che operano sul sorgente. Effettuano un miglioramento del codice 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.

9.1.1.3 Forward enginering

A partire da un modello ad oggetti produce un template di codice sorgente. Gli attributi e le operazioni possono essere mappate facilmente nel codice mentre il corpo dei metodi verrà aggiunto dagli sviluppatori.
Ogni attributo privato può essere mappato in un campo privato della classe più due metodi pubblici getAttributo e setAttributo rispettivamente per leggere e modificare l'attributo.

9.1.1.4 Reverse enginering

A partire dal codice si produce un modello del sistema. Questo viene fatto ad esempio quando il design del sistema viene perduto o quando non è mai stato creato.

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 mappaggio 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.
- **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.

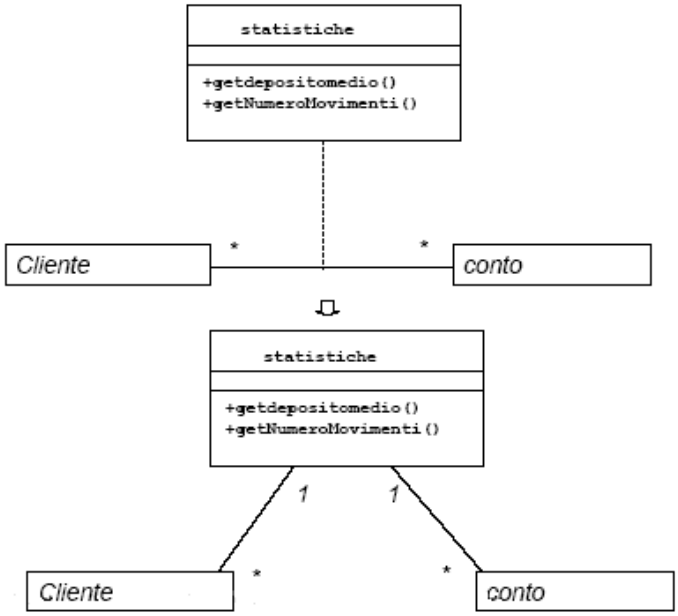


Figura 22 Mapping di una classe di associazione

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 postcondizioni vengono controllate alla fine del metodo e deve essere lanciata un eccezione se non vengono soddisfatte. Le invarianti vanno controllate nello stesso momento delle postcondizioni.

E' 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:

Evitare il codice di controllo per le postcondizioni, invarianti, metodi privati e protetti e limitarlo su componenti di breve durata.

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

E' 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.

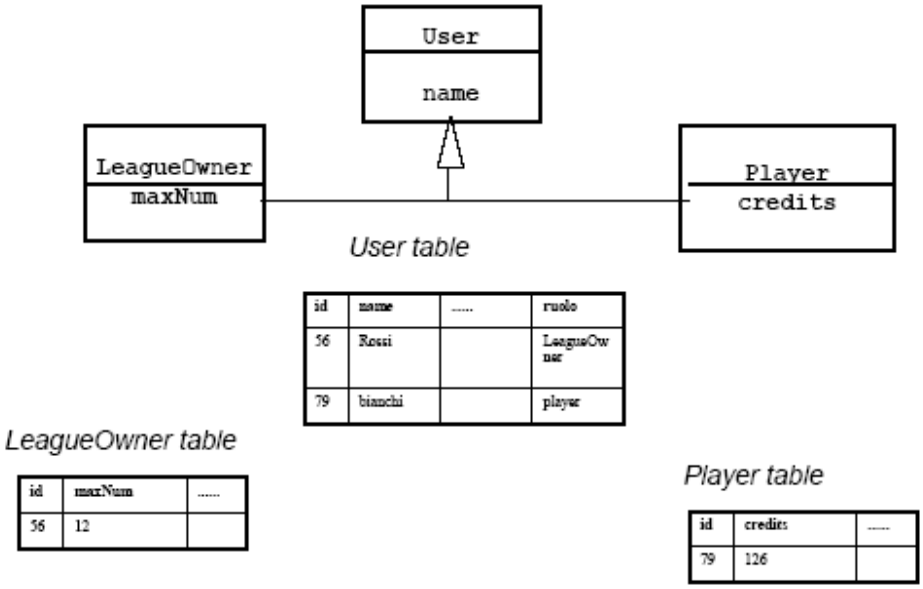


Figura 23 Esempio di mapping di ereditarietà verticale

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.

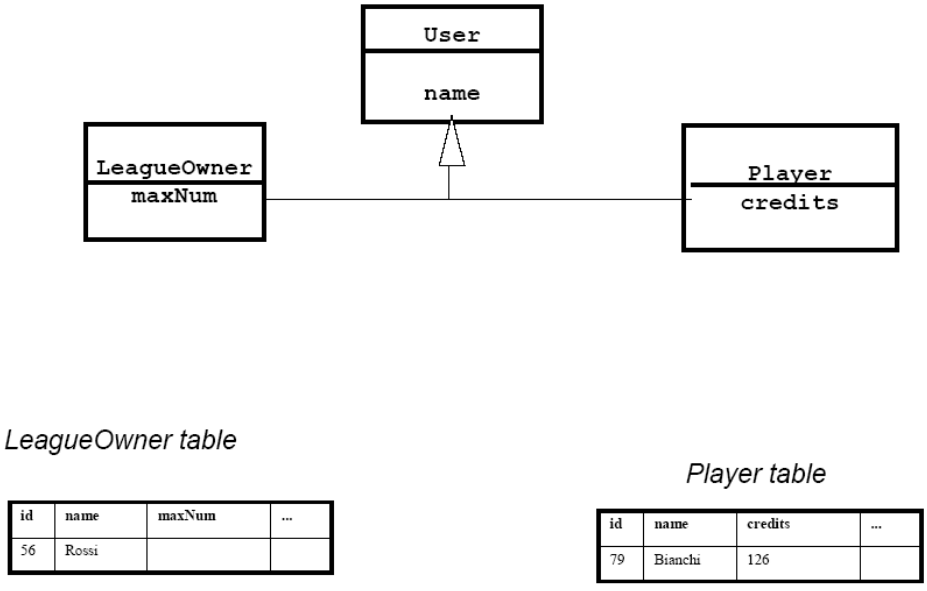


Figura 24 Esempio di mapping di ereditarietà orizzontale

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.

10 Testing
Il testing è l'attività che cerca le differenze tra le specifiche/requisiti e il comportamento osservato.

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.

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: uno cerca di trovare gli errori, l'altro cerca di valutare l'efficienza.

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

10.2 Concetti di testing

- **Componente**
  E' 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**
  E' una manifestazione di un errore durante l'esecuzione del sistema (può essere causato da uno o più errori e può portare ad un fallimento).

I test stub e driver sono usati per sostituire e simulare parti mancanti del sistema.

- **Test stub**
  E' una particolare implementazione di una componente *dal quale dipende* una componente sotto testing.
- **Test driver**
  E' una particolare implementazione di una componente *che fa uso* della componente sotto testing.

10.2.1 Test case

E' 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.
- **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.

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**

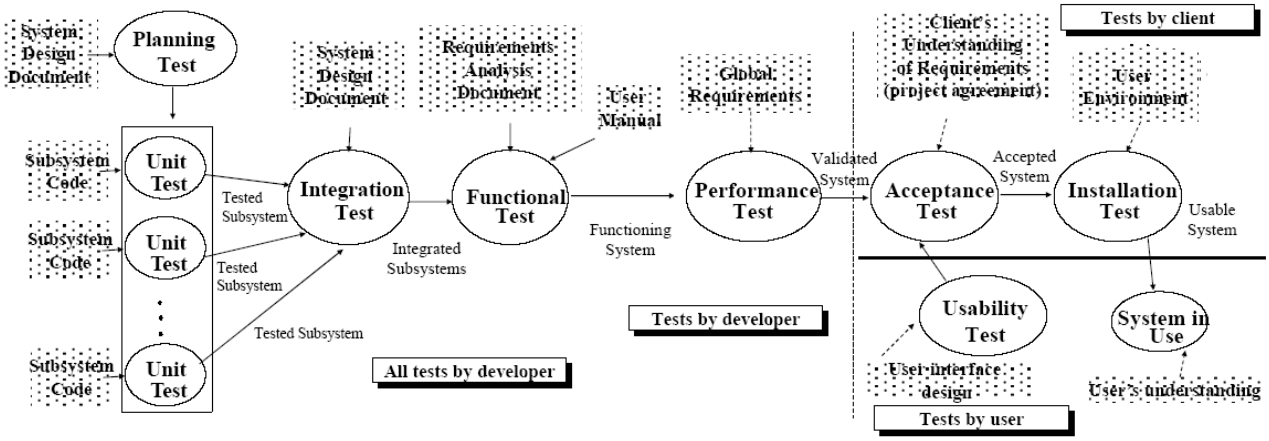


Figura 25 Attività del testing

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

.

10.3.2 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. E' possibile usare anche dei mock-up.
- **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.

- **Product test**
  Viene usata una versione funzionale del sistema e fatta visionare all'utente.

10.3.3 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).

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.

Il whitebox testing 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).

10.3.4 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.

10.3.4.1 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. E' però difficile, in sistemi complessi, individuare la componente responsabile di un errore.

10.3.4.2 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.

10.3.4.3 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.

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

10.3.5.1 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.

10.3.5.2 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.

10.3.5.3 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.

10.3.5.4 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.

10.3.5.5 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 il performance testing. Quando il cliente è soddisfatto, il sistema viene formalmente rilasciato, ed è pronto per l'uso.