Appunti (G) di Ingegneria del Software

prof. A. De Lucia, dott. F. Pecorelli

a.a. 2021/2022

Questi appunti potrebbero essere incompleti o imprecisi.
Maneggiare con cura.

Indice dei contenuti

1 Introduzione all'Ingegneria del Software **2**

2 Modelli del ciclo di vita del software **4**
2.1 Modello a cascata 4
2.2 Modello a V 5
2.3 Modello a prototipi 5
2.4 Modelli evolutivi, incrementali ed iterativi 5

3 Sistemi e notazioni grafiche **6**
3.1 Use Case Diagrams 7
3.2 Class Diagrams 7
3.3 Sequence Diagrams 8
3.4 State Chart Diagrams 8

4 Requirement Analysis Document **9**
4.1 Functional Modeling 9
4.2 Object Modeling 10
4.3 Dynamic Modeling 11
4.4 Conclusioni 12

5 System Design Document **12**
5.1 System Design Model 12
5.2 Software Pattern 14
5.3 Attività del System Design 15
5.4 Revisione del System Design 16

6 Object Design Document **16**
6.1 Design Pattern 17
6.1.1 Composite Pattern 18
6.1.2 Facade Pattern 18
6.1.3 Adapter Pattern 19
6.1.4 Bridge Pattern 19
6.1.5 Strategy Pattern 19
6.1.6 Abstract Factory Pattern 19
6.1.7 Command Pattern 20
6.1.8 Proxy Pattern 20
6.2 Object Constraint Language 20

7 Testing **22**
7.1 Unit Testing 23
7.1.1 Black-Box Testing 24
7.1.2 White-Box Testing 24
7.1.3 Equivalence Class Testing 24
7.2 Integration Testing 25
7.3 System Testing 26
7.4 Object-Oriented Testing 26

1 Introduzione all'Ingegneria del Software

In un **programma**, l'autore è anche l'utente; non è documentato, quasi mai è testato e non c'è un progetto dietro. In altre parole, non serve un approccio formale al suo sviluppo.

Un **prodotto software**, invece, è usato da persone diverse da chi lo ha sviluppato. Si tratta di un prodotto industriale, il cui costo è circa 10 volte il costo del corrispondente programma, e richiede un approccio formale allo sviluppo. Si divide in generici e specifici. S'intende quindi l'insieme completo di programmi, procedure e documentazioni allegati.

Il **software** non è solo il codice, ma anche tuti gli artefatti che lo accompagnano, dunque documentazione, casi di test, specifiche di progetto, manuali, etc.

I problemi che si possono incontrare nella produzione del software dipendono da vari fattori: costi, ritardi e abbandoni, affidabilità (malfunzionamenti). Lo scopo dell'Ingegneria del Software (IdS) riguarda la costruzione di software di grandi dimensioni, di notevole complessità e sviluppati in gruppo. Software sviluppati in questo modo hanno tipicamente versioni multiple.

Uno degli obiettivi principali dell'IdS è contestualizzare il software, poiché il software è tipicamente collocato all'interno di un sistema ibrido software/hardware. L'obiettivo finale è creare un sistema che rispetti i requisiti dell'utente.

In generale, l'IdS si occupa dei metodi, metodologie, processi e strumenti per la gestione professionale del software. Si basa su principi quali rigore, for-

malità, separazioni di aspetti diversi, modularità, astrazione, anticipazione del cambiamento, generalità e incrementalità.

Un **metodo** è un procedimento generale per risolvere classi di problemi.

Una **metodologia** è l'insieme di principi, metodi ed elementi di cui una o più discipline si servono per garantire la correttezza e l'efficacia del lavoro.

Uno **strumento** è un artefatto o un sistema di supporto pratico allo sviluppo.

Una **procedura** è una combinazione di strumenti e metodi.

Un **paradigma** è un particolare approccio nel fare qualcosa.

Un **processo** è un particolare metodo per fare qualcosa, costituito da una sequenza di passi che coinvolgono attività, vincoli e risorse.

Un **processo di sviluppo software** è il processo mediante il quale le richieste dell'utente vengono tradotte in prodotto software.

Col **Forward Engineering** si crea il codice dal modello, usando una progettazione detta a Greenfield, cioè che parte da zero e può spaziare ovunque.

Col **Reverse Engineering** si crea il modello dal codice e la progettazione è detta a Interfacce o Re-ingegneristica.

Col **Roundtrip Engineering** si realizza un misto delle prime due. Utile quando requisiti, etc., variano frequentemente.

I **componenti di un progetto** sono prodotto, schedule, partecipanti, task, definizioni del progetto formale o informale, tipo del progetto, dimensione, stati del progetto, tipo di comunicazione (pianificata o non), organizzazione della squadra (in team o gerarchie, con comitati di revisione, e ruoli).
- Sistema: collezione di sottosistemi che forniscono servizi. - Verifica: è l'equivalente del controllo tra la trasformazione tra due modelli. Validazione: si confronta un modello con la realtà, che potrebbe anche essere un sistema artificiale. Basata su principi di correttezza, completezza, consistenza, ambiguità, realismo.

Un **task** è un assegnamento ben definito di un lavoro per un ruolo. Gruppi di task sono detti attività.

L'elemento tangibile, risultato di una task, è detto **work product**, soggetto a deadlines e suggerisce altre task. Ogni work product sviluppato per il cliente è detto deliverable, altrimenti sono detti internal work product.
Organizzazione task in funzione di schedule. Tutta la specifica di un lavoro attorno a una task è detta work package, organizzato in base alle schedule. Meccanismo di comunicazione sincrono/asincrono (segnali di fumo). Comunicazione event-driven (request for change, clarification, issue resolution) oppure

schedulata (release, project review, walkthrough, status review).

Il **ciclo di vita del software** è il periodo di tempo che inizia quando il software viene concepito, e che termina quando il prodotto non è più disponibile per l'uso.

Il **ciclo di sviluppo del software** è il periodo di tempo che inizi con la decisione di sviluppare un prodotto software e termina quando il prodotto è completato.

2 Modelli del ciclo di vita del software

Un **modello del ciclo di vita del software** è una caratterizzazione descrittiva di come un sistema software dovrebbe essere sviluppato. Vari modelli: a cascata, a spirale, etc. Ad alto livello, un CVS consiste in definizione (si occupa del cosa – requisiti, funzioni, comportamenti del sistema), sviluppo (si occupa del come, cioè codice) e manutenzione.

2.1 Modello a cascata

Il **modello a cascata**, diventato popolare negli anni 70 grazie a Royce, è un modello sequenziale lineare, cioè con una progressione in cascata di fasi, senza ricicli per meglio controllare tempi e costi. Definisce e separa le varie fasi e attività (non c'è, o è minimo, overlap fra fasi).

Prevede elaborati intermedi da usare come input per le fasi successive. Ogni fase raccoglie attività omogenee per metodi e tecnologie, è caratterizzata da task e risultano in un deliverable; la fine di ogni fase è una milestone; i prodotti di una fase vengono congelati.

Si declina nelle seguenti fasi:

- **Studio di fattibilità.** Valutazione preliminare di costi e benefici. Output: documento di fattibilità.
- **Analisi dei requisiti.** Analisi completa dei bisogni dell'utente e del dominio del problema; descrive le funzionalità. Output: documento di specifica dei requisiti.
- **Progettazione.** Definizione di una struttura per il software, scomponendo il sistema in componenti e moduli. Output: documento di specifica del progetto.
- Programmazione, Unit Testing, Integrazione, System Testing, Deployment, Manutenzione, etc.

Tra i pro del modello a cascata riconosciamo la facilità di comprensione e applicabilità. Di contro, l'interazione col committente è solo all'inizio e alla fine – non vi sono giudizi intermedi.

Una variante del modello a cascata è il modello VeV (Verification e Validation) (stiamo costruendo il prodotto nella giusta maniera? e Stiamo costruendo il prodotto giusto)? In questa variante, si possono inviare feedback alle fasi precedenti.

2.2 Modello a V

Nel modello a V, le attività di sinistra sono collegate a quelle di destra intorno alla codifica. Se si trova un errore in una fase a destra, si riesegue il pezzo collegato. Si può iterare migliorando requisiti, progetto, etc.

2.3 Modello a prototipi

Un **prototipo** aiuta a comprendere i requisiti o a valutare la fattibilità di un approccio. Si realizza una prima implementazione incompleta da considerare come prova. Il prototipo è un mezzo tramite il quale si interagisce col committente per accertarsi di aver ben compreso le sue richieste.

Un **modello a prototipo** fa uso di mockup (produzione della UI), breadboards (implementazione di sottosistemi critici), e feedback. In generale, si definisce la prototipazione throw-away (lo sviluppo del prototipo parte dai requisiti meno compresi) ed esplorativa (lo sviluppo parte dai requisiti meglio compresi). Il prototipo è uno strumento di identificazione dei requisiti: dopo il suo utilizzo dev'essere gettato.

2.4 Modelli evolutivi, incrementali ed iterativi

Nello **sviluppo evolutivo**, si parte da una descrizione di massima e si sviluppa una prima versione, che viene migliorata e raffinata. Il problema è che alla fine il prodotto potrebbe essere scarsamente strutturato e si può avere perdita di visibilità del processo da parte del management. Utile per sistemi di taglia medio-piccola, o sistemi con un breve ciclo di vita.

Nelle trasformazioni formali, ci si basa sulle trasformazioni di una specifica matematica in programmi eseguibili attraverso mutami che permettono di passare da uno stato ad un altro. Richiede skill specifiche. Difficoltà nelle specifiche formali.

Il **modello di sviluppo a componenti** (riuso) prevede repository di componenti da poter riutilizzare a diversi livelli di astrazione. Utili nell'ambito dello sviluppo di software object-oriented.

I requisiti sono spesso soggetti a modifiche nel corso dello sviluppo. Questo comporta iterazioni di rework, soprattutto nelle fasi iniziali. Tali iterazioni possono essere applicate a qualsiasi modello di processo di sviluppo. Due approcci fondamentali in tal senso sono lo Sviluppo incrementale e lo Sviluppo a spirale.

Lo **sviluppo incrementale** prevede che il software venga consegnato con più rilasci. Il sistema viene decomposto in sottosistemi (detti incrementi) che

vengono implementati, testati, rilasciati, installati e posti in manutenzione secondo un piano di priorità in tempi diversi.

Diventa fondamentale la fase di integrazione di nuovi sottosistemi con quelli già in esercizio. Vantaggio dovuto alla possibilità di anticipare subito dei prodotti al committente, dunque minore rischio di fallimento. Testing più esaustivo. I modelli iterativi si basano su questa filosofia: ad ogni istante dopo il primo rilascio esiste un sistema versione N in esercizio, e un sistema N+1 in sviluppo. Ogni versione ha nuove funzionalità.

Con il **modello a spirale**, si ha la formalizzazione del concetto di iterazione: il riciclo è fondamentale per questo modello. Il processo viene rappresentato come una spirale, piuttosto che come una sequenza di attività, e ogni giro della spirale rappresenta una fase del processo. Le fasi non sono predefinite, ma vengono scelte in accordo al tipo di prodotto e ognuna di esse prevede la scoperta, la valutazione e il trattamento esplicito dei rischi.

Si tratta di un meta-modello, cioè si possono usare più modelli. Ogni quarto della spirale prevede fasi di determinazione di obiettivi, valutazione di alternative, sviluppo, verifica e pianificazione.

L'**extreme programming** è un approccio recente allo sviluppo software basato su iterazioni veloci che rilasciano piccoli incrementi delle funzionalità.

3 Sistemi e notazioni grafiche

I diagrammi e le notazioni grafiche introdotte in questa sezione sono approfonditi in seguito.

Un **sistema** è un insieme organizzato di parti comunicanti, ognuna delle quali è un sottosistema.

Un **modello** è un'astrazione che descrive un sottoinsieme di un sistema. Una vista descrive particolari aspetti di un modello.

Una **notazione** è un insieme di regole grafiche o testuali per descrivere viste.

Un **fenomeno** è un oggetto nel mondo di un dominio che possiamo percepire.

Un **concetto** descrive le proprietà dei fenomeni che sono comuni e sono 3-tuple (Nome, Obiettivo – proprietà che determinano se un fenomeno è un membro di un concetto, Membri – insieme dei fenomeni che sono parte del concetto).

Definiamo dunque l'**astrazione** come una classificazione di fenomeni in concetti e la modellazione come lo sviluppo di astrazioni per rispondere a necessità. Ad esempio, nella programmazione i "Tipi" sono astrazioni. Un membro di un Tipo è detto Istanza.

Il **Dominio Applicativo** è l'ambiente in cui opera il sistema.

Il **Dominio delle Soluzioni** è l'insieme delle tecnologie disponibili per costruire il sistema (SDD, ODD).

L'UML (**Unified Modeling Language**) è uno standard per la modellazione del software orientato agli oggetti.

Il modello UML può essere visto gerarchicamente come un sistema diviso in uno o più modelli asua 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, l'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.

Permette di costruire i seguenti diagrammi.

3.1 Use Case Diagrams

I **Use Case Diagrams** descrivono le funzionalità del sistema dal punto di vista dell'utente.

Sono usati durante la fase di raccolta dei requisiti per rappresentare il comportamento esterno. L'attore rappresenta i tipi di utenti del sistema, mentre i casi d'uso rappresentano una sequenza di interazioni per un tipo di funzionalità.

Il **modello dei casi d'uso** è l'insieme di tutti i casi d'uso, ed è una completa descrizione delle funzionalità del sistema e del suo ambiente. Un attore, in particolare, modella un'entità esterna che comunica col sistema (es. utenti, sistemi esterni, etc).

Un **caso d'uso**, dunque, rappresenta una classe di funzionalità fornite dal sistema mediante un flusso di eventi e consiste di un nome unico, attori partecipanti, condizioni di entrata e uscita, flusso di eventi e requisiti speciali.

I casi d'uso possono estenderne altri aggiungendo eventi.

La relazione *extends* rappresenta casi invocati eccezionalmente o raramente (es. errori).

La relazione *includes* rappresenta il comportamento che è tratto dal caso d'uso per il riuso, non perché è un'eccezione. Serve a eliminare ridondanza.

3.2 Class Diagrams

I **Class Diagrams** descrivono la struttura statica del sistema (oggetti, attributi, associazioni).

Vengono usati durante l'analisi dei requisiti per modellare i concetti del dominio del problema; durante la progettazione del sistema per modellare sottosistemi e interfacce; durante l'object design per modellare le classi. Una classe

rappresenta un concetto e incapsula stati e comportamenti. Ogni attributo ha un tipo, ogni operazione ha una signature.

Definiamo la differenza fra attore, classe e istanza: un **attore** è un'entità esterna al sistema da modellare e che interagisce con esso. Una **classe** è un'astrazione che modella un'entità del dominio del problema e dev'essere modellata all'interno del problema. Un **oggetto** è una specifica **istanza** di una classe.

Per indicare relazioni fra classi si usano le **associazioni** (mapping bidirezionale 1-a-1, n-a-1, etc) e link (connessione fra due oggetti). Ogni lato di un'associazione può essere etichettato con un ruolo dotato di nome (li si usa necessariamente per associazioni fra oggetti della stessa classe). Un'**aggregazione** è un caso speciale di associazione che denota una gerarchia di "*consiste di*". L'aggregato è la classe padre e le componenti sono le classi figlie. Forme forti di aggregazione sono dette **composizione**.

Un **package** è un meccanismo UML per organizzare gli elementi in gruppi (package = sottosistema).

3.3 Sequence Diagrams

I **Sequence Diagrams** descrivono il comportamento dinamico tra gli attori e il sistema, e tra gli oggetti e il sistema, mediante interazioni. Sono usati durante l'analisi dei requisiti per raffinare le descrizioni dei casi d'uso e trovare nuovi oggetti, e durante il system design per raffinare le interfacce dei sottosistemi.

Le classi sono rappresentate da colonne, i messaggi da frecce e le lifeline da linee verticali. La sorgente di una freccia indica l'attivazione che ha inviato il messaggio, che è lunga quanto tutte le attivazioni annidate. Le linee orizzontali indicano il flusso di dati. Le condizioni si esprimono nelle parentesi tonde. Si può esprimere la distruzione della vita utile di un oggetto con un marcatore.

3.4 State Chart Diagrams

Gli **State Chart Diagrams** descrivono il comportamento dinamico di un oggetto individuale mediante stati e transazioni.

Gli **Activity Diagrams** modellano i comportamenti dinamici di un sistema, in particolare il suo workflow.

È il caso speciale di uno State Chart Diagram in cui gli stati sono le attività. Tipi: *Action State* (non decomponibile e rappresenta avvenimenti istantanei) e *Activity State* (decomponibile ulteriormente e attività modellata da altro activity diagram). Con l'Activity Diagram si modellano le decisioni e la concorrenza (splitting del flusso di controllo in thread multipli).

4 Requirement Analysis Document

Il **Problem Statement** è sviluppato dal cliente come una descrizione del problema che dovrebbe affrontare il sistema. Descrive la situazione corrente, le funzionalità che il sistema dovrebbe supportare, l'ambiente in cui il sistema sarà distribuito, le consegne attese dall'utente, le date di rilascio e criteri di accettazione. Contiene anche scenari, project schedule e target environnement (ambiente in cui il software sviluppato verrà sottoposto a system testing).

Tre modi di gestire complessità: **Astrazione**, **Decomposizione** (tecnica Divide et Impera, gestibile con 'orientazione a oggetti' e 'decomposizione funzionale', che comporta codice non mantenibile) e **Gerarchia** (tecnica Layering). Il primo passo nello stabilire i requisiti è identificare il sistema; si risponde a due domande: come posso identificare gli obiettivi del sistema e come possono distinguere i confini del sistema?

Le risposte sono da cercarsi nel processo dei requisiti, che consiste di due attività: **Requirement Elicitation** (fase di raccolta dei requisiti), cioè definizione del sistema in termini comprensibili dal cliente; e **Requirement Analysis**, specifica tecnica del sistema in termini comprensibili agli sviluppatori.

L'output della fase di Requirement Elicitation è il documento di specifica dei requisiti, che usa un linguaggio naturale ed è derivato dal Problem Statement. Questo documento viene formalizzato per produrre il Modello di Analisi che usa notazioni formali, ad esempio UML.

Durante la fase di *Requirement Elicitation*, due metodi per raccogliere informazioni: Joint Application Design (JAD) – focalizza l'attenzione sull'ottenere un consenso fra sviluppatori, utenti e clienti; usato quindi in un contesto di stackholding con sviluppo partecipato – oppure Traceability, si pone l'attenzione sul registrare, strutturare, collegare e raggruppare le dipendenze tra i requisiti e gli altri work product.

In sintesi, la fase di Requirements Elicitation serve a costruire un modello funzionale del sistema che sarà poi usato durante l'analisi dei requisiti per costruire un object model e un dynamic model. Le attività della Requirements Elicitation sono identificare gli attori, gli scenari, i casi d'uso, le relazioni fra questi, raffinare i casi d'uso, identificare requisiti non funzionali e gli oggetti partecipanti.

4.1 Functional Modeling

I requisiti sono di tre tipi: **funzionali** (descrivono le interazioni tra il sistema e i suoi ambienti, indipendentemente dall'implementazione); **non funzionali** (descrivono aspetti del sistema non direttamente legati al suo comportamento funzionale) e **vincoli** (imposti dal cliente o dall'ambiente in cui si opera). I requisiti devono essere validati, e cioè essere corretti, completi, consistente, chiari,

realistici, tracciabili e verificabili.

Gli **scenari** sono esempi di utilizzo del sistema in termini di sequenze di interazioni tra utente e sistema. I **casi d'uso** sono astrazioni che descrivono una classe di scenari. Gli scenari sono una descrizione narrativa di cosa fanno le persone e mostrano come queste ultime provano ad usare l'applicazione e i sistemi computerizzati; sono descrizioni concrete e informali e coinvolgono, in genere, un singolo attore. Invece, i casi d'uso sono utili poiché modellano il sistema dal punto di vista dell'utente, risultando strumenti molto utili.

Nella pratica, dopo aver definito uno scenario bisogna trovare un caso d'uso che specifica tutte le possibili istanze dell'operazione che descrive lo scenario. Il **Modello dei Casi d'Uso** è l'insieme di tutti i casi d'uso che specificano le complete funzionalità del sistema.

I casi d'uso sono composti da nome, attori, entry ed exit condition, flusso di eventi, eccezioni, requisiti speciali (requisiti non funzionali/vincoli). Il Modello dei Casi d'Uso consiste nelle relazioni fra i casi d'uso, espresse tramite Dipendenze o Generalizzazioni.

La **Generalizzazione** indica la situazione in cui un caso d'uso astratto ha differenti specializzazioni. Le Dipendenze si dividono in Inclusioni ed Estensioni. *includes* realizza decomposizioni funzionali (funzione nel problem statement troppo complessa per essere risolta immediatamente) oppure riuso di funzionalità già esistenti; *extends* realizza comportamenti comuni tra i casi d'uso e si vuole fattorizzarli usando un'associazione di generalizzazione; i casi d'uso figli ereditano il comportamento e il significato del caso d'uso padre e aggiungono ulteriori comportamenti.

[Non importante] Il Modello FURPS è usato per suddividere in categorie i requisiti non funzionali (in usabilità, affidabilità, lealtà, performance, supportabilità, manutenibilità) e i vincoli, detti anche requisiti di qualità (di implementazione, di interfacce, di operazioni, di packaging, legali).

4.2 Object Modeling

L'obiettivo principale dell'Object Modeling è trovare le astrazioni più importanti. Si identificano le classi (importante farlo bene e cioè entro i confini del sistema stabiliti dagli obiettivi del sistema), si trovano gli attributi, i metodi e le associazioni fra classi, iterando questo processo. Per identificare obiettivi del sistema, si usano scenari e casi d'uso.

In generale, le componenti di un Object Model sono classi, relazioni, attributi e operazioni. Gli **Object Diagrams** sono una notazione grafica per modellare gli oggetti, classi e relazioni fra loro. Esempi di Object Diagram sono i **Class Diagrams** (utili per pattern, schemi) e **Instance Diagram** (utili per descrivere scenari).

Identificare le classi è il punto focale della modellazione a oggetti. Per farlo vi sono vari approcci: approccio al *dominio applicativo* (si richiede agli esperti del dominio di identificare le astrazioni più rilevanti); *sintattico* (si parte dai casi d'uso e si vedono verbi/sostantivi tramite la **tecnica di Abbot**; *design pattern*; *sui componenti*.

Gl oggetti possono essere classificati come segue: **Entity** (rappresentano informazione persistente tracciata dal sistema), **Boundary** (interazione fra utente e sistema), **Control** (operazioni eseguite dal sistema). Nel MVC, il dominio applicativo è rappresentato dai Model object, visualizzato dai View object e manipolato dai Control object.

Dunque, i Class Diagrams hanno come obiettivo principale la descrizione delle proprietà statiche del sistema.

4.3 Dynamic Modeling

I diagrammi per la modellazione dinamica sono di due tipi: **Interaction Diagram**, che descrivono il comportamento dinamico tra gli oggetti, e si dividono in **Sequence Diagram** (che mostrano il comportamento dinamico di un insieme di oggetti organizzati in sequenze di tempo, utili per complessi scenari in real-time) e **Collaboration Diagram** (che mostrano le relazioni tra gli oggetti); e gli **Statechart Diagram** che, invece, sono macchine a stati che descrivono le risposte di un oggetto di una determinata classe agli stimoli percepiti dall'esterno (eventi). Un tipo speciale di questi diagrammi sono gli **Activity Diagram**, in cui ogni stato è uno di tipo *action*.

In generale, un **modello dinamico** è una collezione di più Statechart Diagrams, uno per ogni classe di comportamento dinamico rilevante. L'obiettivo di questo modello è quello di individuare e fornire metodi per il modello a oggetti. Per ottenerlo, si comincia dai casi d'uso o dagli scenari e si modellano le interazioni tra gli oggetti mediante i Sequence Diagrams, per poi modellare i comportamenti dinamici di ogni singolo oggetto mediante gli Statechart Diagram.

I **Sequence Diagrams** sono descrizioni grafiche degli oggetti che partecipano al caso d'uso o allo scenario, usando la notazione dei grafici aciclici direzionati. Un'euristica per ottenere un sequence diagram è basata sulle seguenti osservazioni: un evento ha sempre un trasmettitore e un ricevitore; la rappresentazione degli eventi è a volte chiamata messaggio; trovare questi eventi per ogni messaggio significa trovare gli oggetti partecipanti al loro caso d'uso.

I Sequence Diagrams sono costruiti con un layout: la prima colonna dovrebbe corrispondere all'attore che ha iniziato il caso d'uso, la seconda ai *Boundary Object* e la terza ai *Control Object*; i *Control Object* sono creati nella fase di inizializzazione di un caso d'uso, mentre i *Boundary* sono creati dal control.

I Boundary objects e i Control Objects accedono agli Entity Objects. I

Sequence Diagrams possono essere strutturati come segue: **Fork Diagram** (i comportamenti dinamici in un singolo oggetto, in genere nel control object) e **Stair Diagram** (comportamento dinamico; ogni oggetto delega alcune responsabilità ad altri e ha solo poche informazioni sugli altri oggetti, ma sa quale di questi può aiutarlo in uno specifico comportamento).

Gli **Statechart Diagrams** possono essere mappati in una macchina a stati finiti. Si tratta di grafi in cui i nodi sono gli stati e gli archi direzionati sono transizioni etichettate dai nomi degli eventi. C'è sempre uno stato iniziale e uno finale. Una transazione rappresenta un cambio di stato a seguito di un evento.

Si determinano due tipi di operazioni in uno Statechart Diagram: *attività*, cioè operazioni associate agli stati che richiedono tempo per essere completate, e *azioni*, cioè operazioni istantanee e atomiche.

Uno **stato** è l'astrazione degli attributi di una classe ed ha una durata. Insieme di *sottostati* denotano un *superstato*. Sono usati anche per **modellare la concorrenza**, che può essere *di sistema* (ogni diagramma di stato è eseguito concorrentemente ad altri) e *di oggetti* (stato dell'oggetto consiste in un insieme di stati, uno stato da ogni sottodiagramma).

Riassumendo, gli Statechart Diagrams aiutano a identificare i cambiamenti in un oggetto individuale, mentre i Sequence Diagrams aiutano a identificare le relazioni temporali tra oggetti nel tempo e la sequenza delle operazioni come risposta a uno o più eventi.

4.4 Conclusioni

Il *Requirement Analysis Document* risponde alle seguenti domande:

- **Quali sono le trasformazioni?** Modellazione funzionale: creare scenari e casi d'uso parlando col cliente, osservando e facendo esperimenti.
- **Qual è la struttura del sistema?** Modellazione a oggetti: creare i Class Diagrams, dunque identificare classi, associazioni e molteplicità.
- **Qual è il suo comportamento?** Modellazione dinamica: creare i Sequence Diagrams, dunque mostrare la sequenza degli eventi scambiati tra gli oggetti, identificare dipendenze e concorrenza tra gli eventi, e creare Statechart Diagrams.

5 System Design Document

5.1 System Design Model

Il **System Design** (progettazione del sistema) è la trasformazione del modello di analisi nel **System Design Model** (modello di progettazione del sistema). A differenza dell'Analisi, che si focalizza sul dominio applicativo, il System Design si focalizza sul dominio delle soluzioni.

Lo scopo del System Design è quello di costruire un ponte fra il sistema desiderato e quello esistente usando la tecnica Divide and Conquer in maniera da modellare il sistema come un insieme di sottosistemi.

Il System Design è costituito da una serie di attività: identificare gli obiettivi di design, progettazione la decomposizione del sistema in sottosistemi e raffinare quest'ultima per rispettare gli obiettivi di design.

I prodotti del System Design sono obiettivi di design, architettura software e boundary use case.

Gli obiettivi di design sono organizzati in cinque gruppi: criteri di performance, affidabilità, costo, mantenimento e di end user. Aspetto centrale: trade-off (es. funzionalità vs usabilità).

Un **sottosistema** (package in UML) è una collezione di classi, associazioni, operazioni, eventi e vincoli che sono in relazione tra loro. I servizi dei sottosistemi sono gruppi di operazioni fornite dai sottosistemi. Trovano origine dai casi d'uso dei sottosistemi e sono specificati dalle Interfacce dei sottosistemi. Queste interfacce dovrebbero essere piccole e ben definite, e sono spesso chiamate API.

In generale, sono definiti i *servizi* (insieme di operazioni correlate che condividono un obiettivo comune) e le *interfacce* dei sottosistemi (insieme di operazioni correlate, con signature completamente specializzata). Per ridurre la complessità di un sistema, lo si decompone in più sottosistemi. Un *oggetto di interfaccia di sistema* fornisce un servizio: si tratta di un insieme di metodi pubblici forniti dal sottosistemi e l'interfaccia del sottosistema descrive tutti i metodi dell'oggetto interfaccia del sottosistema.

Meccanismi come *Coupling* e *Cohesion* hanno l'obiettivo è ridurre la complessità mentre avvengono cambiamenti. La **Coesione** misura le dipendenze tra classi, quindi **Alta Coesione** significa che le classi nel sottosistema eseguono task simili e sono in relazione con ogni altra classe mediante associazioni, mentre **Bassa Coesione** significa che la maggioranza delle classi songo ausiliare ed eterogenee con nessuna associazione.

Il **Coupling**, invece, misura le dipendenze tra i sottosistemi. **Alto Coupling** significa che i cambiamenti a un sottosistema avranno alto impatto sugli altri, mentre **Basso Coupling** significa che un cambiamento in un sottosistema non incide sugli altri.

Il **partizionamento** e la **stratificazione** sono tecniche usate per ottenere un basso coupling. Le partizioni dividono verticalmente un sistema in svariati sottosistemi indipendente (o debolmente accoppiati) che forniscono servizi allo stesso livello di astrazione. Uno strato è un sottosistema che fornisce servizi di sottosistema a uno strato più alto. Uno strato può dipendere solo da quelli più bassi e non ha conoscenza di quelli più alti.

Un esempio è dato dalla *macchina virtuale*, cioè un'astrazione che fornisce un insieme di attributi e operazioni. Le macchine virtuali posono implementare due tipi di architetture software: aperta (ogni strato può invocare operazioni a ogni strato inferiore, motivo per cui la stratificazione è detta trasparente – più efficiente) e chiusa (ogni strato può invocare solo operazioni allo strato immediatamente inferiore, motivo per cui la stratificazione è detta opaca – più portabile).

I sistemi stratificati sono detti gerarchici.

5.2 Software Pattern

Alcuni pattern software che possono essere usati come base di architetture software, sono *Client/Server*, *Peer-to-Peer*, *MVC*, *Pipe e Filtri*, e *Repository*.

- **Stile Architetturale Client e Server.** Uno o più server forniscono servizi a istanze di altri sottosistemi detti client. I client chiamano il server che realizzare alcuni servizi e restituisce un risultato. Usata per sistemi basati su database, dove il front-end è l'applicazione utente e il back-end gestisce l'accesso e la manipolazione del database. Obiettivi di design per sistemi client e server sono portabilità, trasparenza, performance, scalabilità, flessibilità e affidabilità.

- **Stile Architetturale Peer-to-Peer**. Generalizzazione di client e server, dove i client possono essere server e viceversa. Complessa, possibilità di deadlock. Esempio di P2P è Modello OSI di ISO, dove vengono definiti i sette strati di rete (fisico... applicazione). Esistono poi middleware che implementano più strati del modello, in modo da permettere di focalizzare maggiormente sul livello di applicazione.

- **Stile Architeturale a Repository**. i sottosistemi accedono e modificano i dati in una singola struttura chiamata repository, e c'è basso coupling. Il flusso di controllo è detto repository centrale. I repository sono adatti per applicazioni con task di elaborazioni dati che cambiano frequentemente.

- **Stile Architetturale Model/View/Controller**. Sottosistemi di tre tipi differenti: Model: mantiene la conoscenza del dominio applicativo; view: visualizza all'utente gli oggetti del dominio applicativo; controller: responsabile della sequenza di interazioni con l'utente e di notificare ai view i cambiamenti del modello. MVC caso speciale di architettura repository: il model implementa la struttura dati centrale, il controller gestisce escplicitamente il flusso di controllo (ottiene gli input e manda messaggi al modello) e il viewer visualizza il modello.

- **Stile Architetturale con Pipe e Filtri**. L'architettura UNIX si basa su questo stile. I filtri ricevono e inviano dati attraverso le loro pipe di input e output, ignorando l'esistenza e l'identità di ogni altro filtro.

5.3 Attività del System Design

Quando si decompone il sistema, capita spesso di dover far fronte a determinate scelte che si presentano quando il sistema viene decomposto. Ognuna di queste 'attività del System Design' correggono la decomposizione del sistema, in modo da risolvere particolari problematiche. Una volta completate queste attività, possono essere definite le interfacce dei sottosistemi.

- **Identificare la concorrenza.** In questa attività bisogna identificare i processi concorrenti. Un thread di controllo è un path attraverso un insieme di diagrammi di stato in cui, in ogni istante, un solo oggetto è attivo. Un thread rimane in un diagramma di stato fino a quando un oggetto invia un evento a un altro oggetto e attende un altro evento.

  Quando un oggetto effettua un invio non bloccante di un evento, si utilizza un **Thread Splitting**. Due oggetti sono concorrenti se possono ricevere eventi nello stesso istante senza interagire. Questi due oggetti dovrebbero essere assegnati a thread di controllo differenti.

- **Mapping hardware/software.** "Qual è la connessione tra le unità fisiche? Protocollo di comunicazione? Qual è il tempo di risposta desiderato? Se l'architettura è distribuita, in questa attività di descrive al meglio l'architettura di rete, e quindi il sottosistema di comunicazione.

  Durante il System Design dobbiamo modellare la struttura statica e dinamica. A tal proposito, si utilizzare per le struture statiche i Component Diagram che mostrano la struttura al design time, o a al compile time, e per le strutture dinamiche i Demployment Diagram, che mostrano la struttura al runtime.

  Un **Component Diagrams** è un grafo di componenti connesse mediante relazioni di dipendenza che mostra le dipendenze tra componenti software: codice sorgente, librerie, eseguibili.

  I **Deployment Diagrams**, invece, sono utili per mostrare il progetto di un sistema dopo aver effettuato scelte in merito alla decomposizione a alla concorrenza.

- **Gestione dati persistenti.** Si sceglie il tipo di storaging. Un oggetto persistente può essere realizzato tramite strutture dati, file, database (Object-Oriented, relazionali).

- **Gestione risorse e sicurezza**. Durante l'analisi si modellano accessi differenti associando differenti casi d'uso a differenti attori. A seconda dei requisiti di sicurezza del sistema, possiamo anche definire il modo in cui gli attori vengono autenticati al sistema e come certi dovrebbero essere crittografati.

  Gli accessi vengono modellati tramite una matrice di accesso: le righe rappresentano gli attori del sistema e le colone rappresentano le classi di cui vogliamo controllare l'accesso, stabilendo una correlazione chiamata

diritto di accesso. Varie implementazioni di questa matrice: tabella globale degli accesso, lista di controllo degli accessi, capability (associamento coppia [classe, operazione] ad un attore).

- **Stabilire il controllo software**. Si stabilisce la modalità tramite la quale il sistema viene controllo. Può essere controllato implicitamente (linguaggi di chiarativi) o esplicitamente (linguaggi procedurali). Il controllo centralizzato può essere procedure-driven o event-driven.

  Il controllo decentralizzato può far uso di vari oggetti indipendenti. Per stabilire quale tipo di controllo usato, si guardano i Sequence Diagrams: se hanno una struttura fork, si usa un controllo decentralizzato; in alternativa, se hanno una struttura stair, si utilizza un design decentralizzato.

- **Boundary Condition**. Si risponde a domande del tipo come si avvia il sistema? I singoli sottosistemi possono terminare? Come si comporta il sistema quando un nodo fallisce?

5.4 Revisione del System Design

Come l'analisi dei requisiti, il System Design è un'attività evolutiva ed iterativa. A differenza dell'analisi, però, non ci sono agenti esterni, come il cliente, a revisionare le successive iterazioni.

Dunque, ci si pone il problema di come revisionare il System Design, assicurandosi che sia corretto (se il modello di analisi può essere mappato su di esso), completo (se ogni requisito è stato perseguito), consistente (se non contiene contraddizioni), realistico (se il sistema può essere implementato) e leggibile (se gli sviluppatori possono comprenderlo).

Inoltre, si tenga noto che il System Design nasce attraverso successive **iterazioni** e cambiamenti. I cambiamenti sono controllati per prevenire il caos, specialmente il progetti complessi. Vi sono tre tipi di iterazioni: decisioni importanti che incidono sulla decomposizione in sottosistemi; revisioni delle interfacce dei sottosistemi per valutare specifici problemi; errori e sviste scoperti tardi.

6 Object Design Document

L'**Object Design** è il processo che si occupa di aggiungere dettagli all'analisi dei requisiti e di prendere decisioni di implementazione. L'object designer deve scegliere diversi tra diversi modi di implementare il modello di analisi.

Nell'analisi dei requisiti, i casi d'uso, il modello funzionale e quello dinamico definiscono le operazioni per il modello a oggetti. Nell'Object Design, iteriamo nel processo di assegnazione di queste operazioni per il modello a oggetti. In altre parole, l'Object Design serve come base per l'implementazione e durante il suo sviluppo chiude il gap tra gli oggetti di applicazioni e le componenti off-the-shelf, identificando oggetti soluzione e raffinando gli oggetti esistenti.

Le principali attività dell'Object Design sono applicare i concetti di riuso, specificare le interfacce dei servizi, ristrutturazione del modello a oggetti e sua ottimizzazione. Queste attività di solito non sono sequenziali, ma realizzate in maniera concorrente.

Le attività più difficile nello sviluppo di un sistema sono l'identificazione degli oggetti e la decomposizione del sistema in oggetti. L'analisi dei requisiti è concentrata sul dominio applicativo; il System Design è concentrato sia sul dominio applicativo che implementativo; infine, l'Object Design si focalizza sul dominio implementativo.

Durante l'analisi dei requisiti, le tenciche per trovare gli oggetti si basano sui casi d'uso e gli oggetti partecipanti, per poi fare una sorta di analisi testuale del flusso di eventi.
Durante lil System Design, le tecniche per trovare gli oggetti si basano sulla decomposizione in sottosistemi e sul tentativo di identificare strati e partizioni.
Durante l'Object Design, le tecniche per trovare gli oggetti sono incentrate sulla ricerca degli oggetti addizionali, applicando le conoscenze del dominio implementativo.

Gli **Application Object**, detti anche Domain Object, rappresentano concetti del dominio che sono rilevanti per il sistema. I **Solution Object**, invece, rappresentano concetti che non hanno una controparte nel dominio applicativo.

Ricapitolando, durante l'analisi identifichiamo gli entity object e le loro relazioni, attributi e operazioni. Identifichiamo anche solution object visibili all'utente, come i boundary e i control object. Durante il system Design, invece, identifichiamo altri solution object in termini di piattaforme hardware e software. Durante l'object design, raffiniamo e dettagliamo tutti questi application e solutiono object, e identifichiamo ulteriori solution object per chiudere il gap di object design.

6.1 Design Pattern

Un'ulteriore sorgente per trovare gli oggetti sono i **Design Pattern**: quest'ultimi descrivono un problema ricorrente nel nostro ambiente, e il cuore della soluzione a tale problema, in modo tale da poter usare questa soluzione ancora un milione di volte, senza mai doverla ricercare.

Ricordiamo che, in generale, l'**Ereditarietà** è usata per raggiungere due differenti obiettivi: descrivere le tassonomie e specificare le interfacce.

Nel caso dell'Object Design, e in particolare nel caso in cui si vuole aumentare il riuso, si applica la Composizione (**Black Box Reuse**), per la quale la nuove funzionalità è ottenuta mediante aggregazione; oppure l'Ereditarietà (**White Box Reuse**), per la quale la nuova funzionalità è ottenuta mediante ereditarietà. Ha come vantaggi la chiarezza d'uso e la semplicità di implementare nuove

funzionalità; di contro ha la dipendenza rispetto alla classe padre.

Con *ereditarietà di interfaccia*, detta anche subtyping, si ereditano da una classe tutte le operazioni specificate, ma non ancora implementate.
Con *ereditarietà implementativa*, o di classe, si ha l'obiettivo di estendere le funzionalità dell'applicazione mediante il riuso di funzionalità nella classe padre.

La **Delegazione**, infine, è un modo di fare composizione in maniera potente per il riuso. Sono coinvolti due oggetti nella gestione di una richiesta: un oggetti ricevente delega operazioni al suo delegato: Lo sviluppatore può essere sicuro che l'oggetto ricevente non permette al client di usare impropriamente l'oggetto delegato. Ha come vantaggio la flessibilità, ma di contro l'inefficienza.

Le migliori euristiche di design indicano le seguenti regole: mai usare l'ereditarietà implementativa, ma quella di interfaccia, e se si vuole usare la prima piuttosto usare la delegazione; una sottoclasse non dovrebbe mai nascondere le operazioni implementate in una superclasse.

6.1.1 Composite Pattern

Il **Composite Pattern** compone gli oggetti in una struttura ad albero, per rappresentare intere parti di gerarchie con profondità e altezze arbitrarie. Con i Composite Pattern si può modellare anche lo sviluppo software. In particolare, si può modellare il ciclo di vita del software come insieme di attività di sviluppo o collezioni di task.

Un sottosistema consiste di un oggetto interfaccia (Entity Object), di un insieme di oggetti del dominio applicativo che modellano entità reali o sistemi esistenti, e uno o più Control Object. La realizzazione delle interfacce avviene mediante i Facade, la realizzazione dei sistemi esisteni tramite Adapter o Bridge.

6.1.2 Facade Pattern

I **Facade Pattern** forniscono un'interfaccia unificata per un insieme di oggetti in un sottosistema. Un facade definisce un'interfaccia di livello più alto che rende il sottosistem più semplice da usare e permettere di costruire architetture chiuse. In un'architettura aperta, ogni client ha visione dell'interno del sottosistema e può chiamare tutte le operazioni delle componenti o delle classi che vuole.

Il sistema è efficiente, ma non ci si può aspettare che il chiamante sappia come lavorare col sottosistema, dando luogo a codice non portabile. Si può quindi realizzare un'architettura chiusa con un Facade: il sottosistema, in questo caso, decide esattamente come essere acceduto, affidando l'integrazione ad un driver.

6.1.3 Adapter Pattern

Gli **Adapter Pattern** convertono le interfacce di classe in altre interfacce, che i client si aspettano. Adapter permette di far lavorare insieme classi che altrimenti non potrebbero, a causa di incompatibilità. Viene usato per fornire una nuova interfaccia a componenti legacy ed è anche detto Wrapper. Si distinguono in due tipi: Class Adapter, che sfrutta l'ereditarietà multipla, e l'Object Adapter, che sfrutta ereditarietà singola e delegazioni.

6.1.4 Bridge Pattern

Il **Bridge Pattern** permette a differenti implementazioni di un'interfaccia di essere decise dinamicamente. E' usato per fornire implementazioni multiple sotto la stessa interfaccia. Il Bridge, dunque, separa un'astrazione dalla sua implementazione, in modo che le due possano variare indipendentemente.

6.1.5 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.

6.1.6 Abstract Factory Pattern

In una situazione in cui un unico sistema deve interagire con delle componenti esterne sviluppate da diversi produttori 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 e una implementazione di tale classe per ogni prodotto concreto. 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.

6.1.7 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.

6.1.8 Proxy Pattern

La creazione e l'inizializzazione di oggetti sono due attività molto costose. Il **Proxy Pattern** riduce il costo di accesso agli oggetti, usando un altro oggetto, il Proxy, che fa le veci dell'oggetto reale. Il proxy crea l'oggetto reale solo quando l'utente lo richiede.

Possono essere distinti in *proxy remoti* (i proxy sono rappresentativi di oggetti in differenti spazi locali, utile per il caching), *proxy virtuali* (i proxy fanno le veci di oggetti troppo costosi da scaricare) e *proxy di protezione* (forniscono accesso controllato a oggetti reali, utile quando differenti oggetti hanno differenti diritti di visioni di un documento).

Adapter, Brdige, Facade e Proxy sono variazioni di un singolo tema: riducono l'accoppiamento tra due o più classi, introducono una classe astratta per permettere estensioni future e incapsulano strutture complesse.

6.2 Object Constraint Language

La seconda fase dell'object design è la specifica delle interfacce. L'obiettivo 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

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 - (cancelletto) + 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. 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.

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.

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. 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, ad esempio, *size, includes, select.*

7 Testing

Il **Testing** consiste nel trovare le differenza tra il comportamento atteso e il comportamento osservato dal sistema implementato.

Questa attività, chiaramente, va in contrasto con tutte le attività svolte in precedenza: analisi, design e implementazione sono attività costruttive, mentre il testing tenta di "*rompere il sistema*". Per questo motivo, dovrebbe essere realizzato da sviluppatori che non sono stati coinvolti nelle attività di costruzione del sistema stesso.

I termini maggiormente utilizzati in questo contesto sono **affidabilità** (misura di successo del testing), **failure** (deviazione del comportamento), **stato di errore** (il sistema è in uno stato tale che ogni ulteriore elaborazione porterà ad un failure), e **fault** (causa meccanica/algoritmica di uno stato di errore).

Definiamo il *testing* come la creazione di failure in maniera pianificata, il *debugging* come 'la ricerca di errori quando capitano failure non pianificati, e il *monitoraggio* come la distribuzione di informazioni sullo stato per cercare bug.

In generale, per trattare gli errori si utilizzano i seguenti approcci:

- **Error Prevention**, prima di rilasciare il sistema, usando buone metodologie di programmazione, prevenendo quindi incosistenze e bug.
- **Error Detection**, mentre il sistema è in esecuzione tramite testing, debugging e monitoraggio.
- **Error Recovery**, recuperando da un failure una volta che il sistema è stato rilasciato.

Un **Componente** è una parte del sistema che può essere isolata per essere testata. Su un componente, si usano i Test Driver e i Test Stub.

I **Test Stub** è un'implementazione parziale di una componente da cui dipende la componente testata (in pratica, simula le componenti chiamate dalla componente testata).

Un **Test Driver** è un'implementazione parziale di una componente che dipende dalla componente testata (in pratica, simula il chiamante della componente testata). Il bisogno di driver e stub dipende dalla posizione della componente da testare nel sistema.

L'**oracolo** conosce l'output atteso, per input alla componente, da confrontare con l'output reale per identificare le failure. Una volta che i test sono stati eseguiti e i fallimenti rilevati, gli sviluppano alterano il componente per eliminare il falliumento.

Una **correzione** è un cambiamento apportato a un componente. Chiaramente, una correzione potrebbe introdurre nuovi fault, e per questo si utiliz-

zano delle tecniche per gestire questi difetti, come il **Testing di Regressione**, definito come la riesecuzione di tutti i test effettuati precedentemente.

Il *test dei componenti* può essere effettuato mediante i seguenti approcci:

- **Unit Testing**, con l'obiettivo di confermare che il sottosistema sia codificato correttamente e che esegua le funzionalità attese.
- **Integration Testing**, con l'obiettivo di testare le interfacce tra i sottosistemi.

Il *test del sistema* viene effettuato mediante i seguenti approcci:

- **System Testing**, con l'obiettivo di determinare se il sistema rispecchia i requisiti funzionali e globali.
- **Acceptance Testing**, con l'obiettivo di dimostrare se il sistema rispecchi i requisiti del cliente e che è pronto all'uso.

Per pianificare le attività di testing, si utilizzano i seguenti documenti:

- **Test Plan**, ci si focalizza sugli aspetti manageriali del testing, documentando gli scopi, le risorse e lo schedule delle attività di testing.
- **Test Case Specification**, in cui si documenta ogni test, funzionalmente agli input, driver, stub e output attesi, come i task che devono essere eseguiti.
- **Test Incident Report**, in cui si documenta ogni esecuzione, in particolare i risultati reali dei test e le differenze dagli output attesi.
- **Test Summary Report**, in sui si elencano tutte i failure rilevate durate i test che devono essere investigati.

7.1 Unit Testing

L'**Unit Testing** può essere eseguito in maniera informale, mediante la codifica incrementale oppure mediante analisi statiche o dinamiche.

L'**analisi statica** avviene mediante: esecuzione manuale, leggendo il codice sorgente; walkthrough; ispirazione del codice; tool automatici.

L'**analisi dinamica**, invece, avviene mediante Black-Box Testing (per testare i componenti di input e output) e White-Box Testing (per testare la logica interna del sottosistema o dell'oggetto) e il Testing basato sulle strutture dati (i tipi di dati determinano i casi di test).

Altri tipi di testing non indicati di seguito sono il *Worst Case Testing* e il *Boundary Value Testing*.

7.1.1 Black-Box Testing

Il **Black-Box Testing** si focalizza sul comportamento I/O, senza preoccuparsi della struttura interna del componente. Se per ogni dato input, siamo in grado di prevedere l'output, allora il modulo supera il test.

Tuttavia, è quasi sempre impossibile generare tutti i possibili input, cioè i test case. L'obiettivo diventa, quindi, quello di ridurre il numero di casi di test effettuando un partizionamento: si dividono le condizioni di input in classi di equivalenza e si scelgono i test case per ogni classe di equivalenza.

Per selezionare questi classi, non ci sono regole bensì linee guida: se l'input è valido per un range di valori, si scelgono i test case su classi sotto il range, dentro il range e sopra il range; se l'input è valido solo se appartiene ad un insieme discreto, si scelgono i test case su classi di valori discreti validi e non validi.

Un'altra soluzione per scegliere solo un numero limitato di test case è giungere a conoscenza dei comportamenti interi delle unità da testare: questo è il sunto del **White-Box Testing**.

Sia Black-Box che White-Box sono sistematici, quindi non randomici.

7.1.2 White-Box Testing

Il **White-Box Testing** si focalizza sulla completezza (intesa come copertura). Ogni statement nella componente è eseguita almeno una volta. Indipendentemente dall'input, ogni stato nel modello dinamico dell'oggeto e ogni interazione tra gli oggetti viene testata.

Esistono quattro tipi di White-Box Testing: lo *Statement Testing* (testing algebrico, in cui si testano i singoli statement); il *Loop Testing* (che provoca l'esecuzione del loop che dev'essere saltato completamente); il *Path Testing* (si assicura che tutti i path nel programma siano eseguiti); e il *Branch Testing* (testing condizionale; assicura che ogni possibile uscita da una condizione sia testata almeno una volta).

Potenzialmente, un numero infinito di cammini dovrebbero essere testati. Il White-Box Testing spesso testa ciò che è stato fatto, invece di ciò che dovrebbe essere fatto. Tuttavia, non individua i casi d'uso mancanti. Il Black-Box Testing, invece, ha la peculiarità di essere una potenziale esplosione combinatoria di casi di test. Spesso, però, non è chiaro se i casi di test selezionati scoprono particolari errori o meno, e non casi d'usi estranei.

7.1.3 Equivalence Class Testing

L'**Equivalence Class Testing** nasce dalla motivazione di cercare di avere un testing completo e con la speranza di evitare ridondanze. Le classi di equivalenza sono partizioni dell'input set.

L'intero input set viene coperto, ottenendo la completezza, e le classi sono disgiunte, evitando ridondanza. I test case sono elementi di ogni classe di equivalenza.

Il rpoblema è che le classi di equivalenza devono essere scelte saggiamente, cioè in base ai comportamenti più probabili.

Il **Weak Equivalence Class Testing** sceglie una variabile valore da ogni classe di equivalenza; lo **Strong Equivalance Class Testing**, basato sul prodotto cartesiano degli insieme di partizione e sull'assunzione che le variabili siano indipendenti, testa tutte le interazioni tra le classi.

Se le condizioni di errore hanno alta priorità, dovremmo estendere lo Strong Equivalence Class Testing per includere le classi non valide.

In generale, l'Equivalence Class Testing è appropiato quando i dati in input sono definiti in termini di range e insiemi di valori discreti.

7.2 Integration Testing

Il **Test di Integrazione** rileva bug che non sono stati determinati durante il Test di Unità, focalizzando l'attenzione su un insieme di componenti che vengono integrati.

Due o più componenti vengono integrati e analizzati, e quando dei bug sono rilevati si possono aggiungere nuovi componenti per risolverli.

L'intero sistema viene dunque visto come una collezione di sottosistemi determinati durante il System Design e l'Object Design. L'ordine in cui i sottosistemi vengono selezionati per il testing e per l'integrazione determina le strategie di testing.

L'obiettivo primario del testing di integrazione è, dunque, quello di identificare errori nella configurazione dei componenti.

- *Big-Bang Testing.* Le componenti vengono prima testate individualmente e poi insieme, come un unico sistema. Sebbene sia semplice, è costoso.

- *Bottom-Up Testing.* I sottosistemia l livello più basso della gerarchia sono testati individualmente. I successivi sottosistemi ad essere testati sono quelli che chiamano i sottosistemi testati in precedenza.

  Si ripete quest'ultimo passo finché tutti i sottosistemi non sono testati. L'approccio bottom-up è orientato a metodologie di design object-oriented.

- *Top-Down Testing.* Si testano prima i livelli al top e, successivamente, si combinano tutti i sottosistemi che sono chiamati da quelli già testati e si testa la risultante combinazione di sottosistemi.

  Si ripete il processo fino a quando tutti i sottosistemi non sono incorporati nel test. L'approccio Top-Down prevede impone che i test case siano definiti in funzioni esaminate e richiede di mantenere la correttezza dei test stub, la cui scrittura può essere però difficoltosa.

- *Sandwich Testing.* Si combina l'uso di strategie top-down e bottom-up. Il sistema è visto come se avesse tre strati: un livello di target nel mezzo, uno sopra il target e uno sotto il target.

  Chiaramente, il testing converge al target. Il vantaggio principale è che i test dei livelli in alto e in basso possono essere eseguit in parallelo.

- *Sandwich Testing Rivistato.* Si testano, in parallelo, il livello al top con stub per il target, il livello nel mezzo con driver e stub per i livello al top e in basso rispettivamente, e il livello basso con driver.

7.3 System Testing

Unit Testing e Integration Testing focalizzano l'attenzione sulla ricerca di bug nelle componenti individuali e nelle interfacce fra componenti. Il **System Testing**, invece, assicura che il sistema completo sia conforme ai requisiti funzionali e non. Le attività per questo testing sono le seguenti.

- *Structure Testing.* Simile al White-Box Testing, ha l'obiettivo di coprire tutti i cammini del System Design.

- *Functional Testing.* Simile al Black-Box Testing, ha l'obiettivo di testare le funzionalità del sistema, progettando i test case a partire dal RAD e incentrandosi attorno ai requisiti e alle funzioni chiave (casi d'uso).

- *Performance Testing.* Si spinge il sistema verso i suoi limiti, con l'obiettivo di romperlo. Un esempio è lo Stress Testing.

- *Acceptance Testing.* Ha l'obiettivo di dimostrare che il sistema è pronto per l'uso operativo. A tale scopo, la scelta dei test è fatta dal cliente.

7.4 Object-Oriented Testing

Il testing di sistemi software object-oriented deve sempre tener conto dell'astrazione sui dati, dell'ereditarietà, del polimorfismo, del binding dinamico, delle eccezioni e della concorrenza.

Pertanto, nel caso in cui il software sia object-oriented, l'impatto sul test è considerevole.

Il test del singolo metodo può essere fatto con tecniche tradizionali. L'infrastruttura deve settare opportunamente lo stato per poter eseguire i test ed esaminare lo stato per poter stabilire l'esito dei test (oracoli); tuttavia, dato che lo stato è generalmente privato, si usano approcci intrusivi, come l'aggiunta di metodi testdriver e tecniche di costruzione di macchine a stati finiti, dove gli stati sono l'insieme degli stati della classe e le transizioni sono le invocazioni dei metodi.

Per ciò che concerne i test di integrazione, l'approccio "Big-Bang" è generalmente poco adatto. Per quanto riguarda le strategie top-down e bottom-up, cambia il tipo di dipendenze tra moduli. E' preferibile una strategia bottom-up,

cioè testare prima le classi indipendenti. In generale, le dipendenze tra le classi possono essere espresse un grafo delle dipendenze.

Se il grafo è aciclico, esiste un ordinamento parziale sui suoi elementi: è possibile definire un ordinamento totale topologico e privilegiare le dipendenze di specializzazione. Una volta definito l'ordine di integrazione, si aggiungono le classi incrementalmente, esercitandone le interazioni. La generazione dei casi di test può essere effettuata a partire dai diagrammi di interazione.

Nell'ambito dell'ereditarietà, è necessario identificare le proprietà che devono essere ritestate per ogni classe figlia, al fine di verificare la compatibilità del comportamento tra metodi omonimi in una relazione classe-sottoclasse.

Nell'ambito della genericità, le classi parametriche devono essere istanziate per poter essere testate.

Nell'ambito delle eccezioni, queste ultime modificano il flusso di controllo senza la presenza di un esplicito costrutto di tipo test and branch.

Ci sono problemi nel calcolare l'indice di copertura della parte di codice relativa alle eccezioni: una copertura ottimale solleverebbe tutte le possibili eccezioni in tutti i punti del codice in cui è possibile farlo; una copertura minima, invece, solleverebbe almeno una volta ogni eccezione.

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.

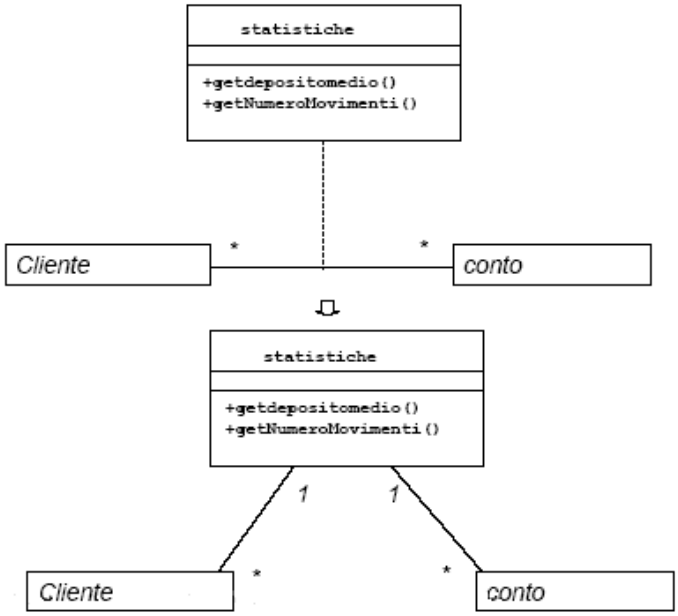

Figura 22 Mapping di una classe di associazione

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.

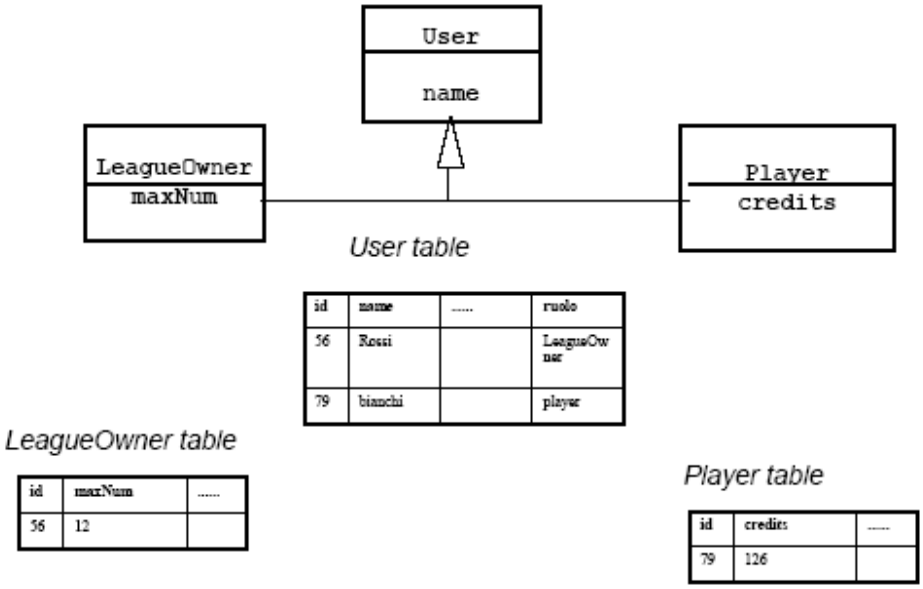

Figura 23 Esempio di mapping di ereditarietà verticale

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.

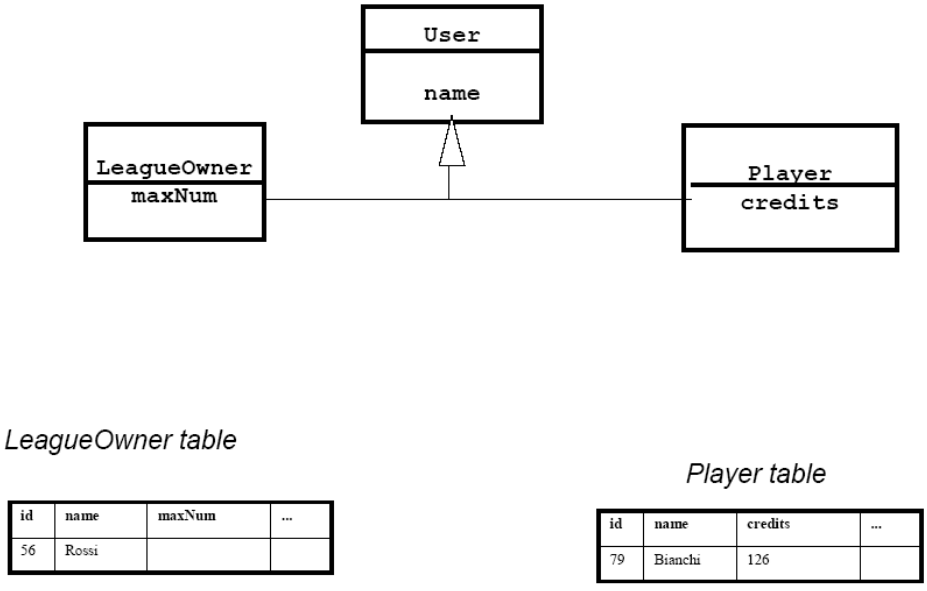

Figura 24 Esempio di mapping di ereditarietà orizzontale

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.