

Sommario

CONCETTI GENERALI.....	3
UML	3
USE CASE DIAGRAM.....	3
<i>RELAZIONE <<include>></i>	4
<i>RELAZIONE <<extend>></i>	4
<i>EREDITARIETÀ</i>	5
<i>EXTEND vs EREDITARIETÀ</i>	5
<i>SCENARI</i>	5
CLASS DIAGRAM	5
ASSOCIAZIONI E LINK.....	6
CLASSE DI ASSOCIAZIONE	7
RUOLI	7
AGGREGAZIONE.....	7
QUALIFICAZIONE.....	8
<i>EREDITARIETÀ</i>	8
SEQUENCE DIAGRAM	9
EURISTICA PER DISEGNARE UN SEQUENCE DIAGRAM.....	9
STATECHART DIAGRAM	10
ACTIVITY DIAGRAM	10
CONCETTI DI MODELLAZIONE.....	10
MODELLAZIONE OBJECT ORIENTED.....	11
FALSIFICAZIONE E PROTOTIPAZIONE.....	11
REQUIREMENTS ELICITATION	12
REQUISITI FUNZIONALI	12
REQUISITI NON FUNZIONALI.....	12
PSEUDO REQUISITI.....	13
VALIDAZIONE DEI REQUISITI	13
TIPI DI REQUIREMENTS ELICITATION	13
IDENTIFICAZIONE ATTORI	14
IDENTIFICAZIONE SCENARI	14
IDENTIFICAZIONE CASI D'USO.....	14
DOCUMENTAZIONE REQUIREMENTS ELICITATION	14
ANALISI	15
IDENTIFICAZIONE ENTITY OBJECT	15
IDENTIFICAZIONE BOUNDARY OBJECT.....	16
IDENTIFICAZIONE CONTROL OBJECT.....	16
REVISIONE MODELLO DI ANALISI.....	16
GESTIONE DELL'ANALISI	17
ORGANIZZAZIONE E COMUNICAZIONE DEL PROGETTO.....	17
TASK E WORK PRODUCT	18
PIANIFICAZIONE.....	19
CONFIGURATION MANAGEMENT	20
CONCETTI DI CONFIGURATION MANAGEMENT	20
REPOSITORIES E WORKSPACES.....	20
GESTIONE DELLE PROMOZIONI	21
GESTIONE DELLE RELEASE.....	21
GESTIONE DEI BRANCH.....	21
SYSTEM DESIGN: DECOMPOSIZIONE DEL SISTEMA	21
SOTTOSISTEMI E CLASSI.....	22

ACCOPIAMENTO E COESIONE	22
LAYER E PARTIZIONI	23
STILI ARCHITETTURALI	24
<i>REPOSITORY</i>	24
<i>MODEL/VIEW/CONTROLLER</i>	25
<i>CLIENT/SERVER</i>	25
<i>PEER-TO-PEER</i>	26
<i>THREE-TIER</i>	26
<i>FOUR-TIER</i>	27
<i>PIPE AND FILTER</i>	27
IDENTIFICAZIONE DESIGN GOALS	27
IDENTIFICAZIONE DEI SOTTOSISTEMI	27
OBJECT DESIGN	28
APPLICATION OBJECT E SOLUTION OBJECT	29
EREDITARIETÀ DELLE SPECIFICHE E EREDITARIETÀ DELL'IMPLEMENTAZIONE	29
DELEGAZIONE	30
DELEGAZIONE E EREDITARIETÀ IN DESIGN PATTERN	30
ATTIVITA DI RIUTILIZZO: SELEZIONE DI DESIGN PATTERN	31
<i>Design Pattern</i>	31
INCAPSULAMENTO DI ARCHIVI DATI CON BRIDGE PATTERN	32
INCAPSULAMENTO DI COMPONENTI LEGACY CON PATTERN ADAPTER	32
INCAPSULAMENTO DEL CONTESTO CON STRATEGY PATTERN	33
INCAPSULAMENTO DI PIATTAFORME CON ABSTRACT FACTORY PATTERN	33
INCAPSULAMENTO DEL FLUSSO DI CONTROLLO CON IL PATTERN COMMAND	34
INCAPSULAMENTO DI GERARCHIE CON PATTERN COMPOSITE	34
INCAPSULAMENTO DI SOTTOSISTEMI CON IL DESIGN PATTERN FACADE	35
FRAMEWORK APPLICATIVI	35
SPECIFICA DELLE INTERFACCIE	36
CONTRATTI: INVARIANTI, PRECONDIZIONI E POSTCONDIZIONI	38
COLLEZIONI OCL: SETS, BAGS E SEQUENCES	38
DOCUMENTARE L'OBJECT DESIGN	39
MAPPARE IL MODELLO NEL CODICE	39
TRASFORMAZIONE DEL MODELLO	40
TESTING	41
TEST CASE	43
TEST STUB E DRIVER	43
CORREZIONI	44
ATTIVITÀ DI TESTING	44
DOCUMENTARE I TEST	47

Concetti generali

UML

UML ci permette di fornire una notazione standard che possa essere utilizzata da tutti i metodi object-oriented.

Lo sviluppo si concentra su tre tipi di modelli:

- **Modello funzionale:** descrive le funzionalità del sistema dal punto di vista dell'utente, in UML lo rappresentiamo con *use case diagram*
- **Modello a oggetti:** descrive la struttura del sistema in termini di oggetti, attributi, associazioni e operazioni, in UML lo rappresentiamo con il *class diagram*.
 - durante l'analisi dei requisiti, il modello a oggetti viene utilizzato per modellare i concetti del dominio del problema
 - durante il system design, il modello a oggetti viene raffinato e include la descrizione delle interfacce del sottosistema
 - durante l'object design, il modello a oggetti viene utilizzato per modellare le classi
- **Modello dinamico:** descrive il comportamento interno del sistema, in UML lo rappresentiamo con *sequence diagram*, *state-chart diagram* e *activity diagram*

USE CASE DIAGRAM

Lo "use case diagram" viene realizzato durante l'analisi dei requisiti e servono a rappresentare le funzionalità del sistema da un punto di vista dell'utente.

Un caso d'uso descrive una funzionalità fornita dal sistema che produce un risultato visibile per un **attore**.

Un attore descrive qualsiasi entità che interagisce con il sistema, caratterizzato da:

- Nome univoco
- Descrizione univoca

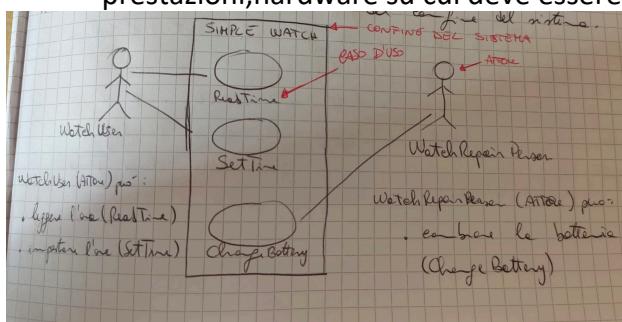
Gli attori avviano un caso d'uso

Un **caso d'uso** descrive il comportamento del sistema dal punto di vista dell'attore.

Un caso d'uso descrive una funzione fornita dal sistema come un insieme di eventi che produce un risultato visibile per gli utenti.

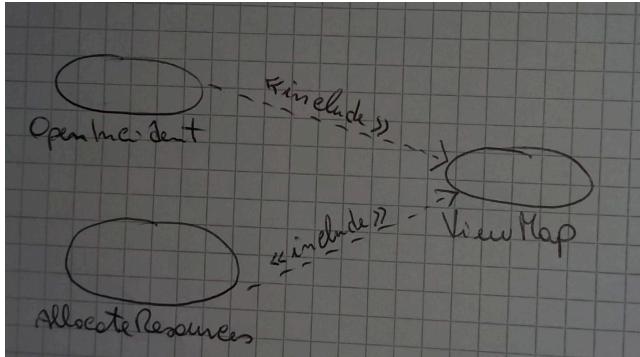
È caratterizzato da:

- Nome univoco
- Attori partecipanti: attori che interagiscono con il caso d'uso
- Condizione d'entrata: condizioni che devono essere soddisfatta prima dell'avvio del caso d'uso
- Flusso di eventi: sequenza di interazioni
- Condizione d'uscita: condizioni che devono essere soddisfatte dopo il completamento
- Requisiti di qualità: requisiti non correlati alle funzionalità del sistema(es. prestazioni, hardware su cui deve essere eseguito)



RELAZIONE <<include>>

Una relazione **include** viene utilizzata quando una funzionalità viene inclusa completamente tra uno o più casi d'uso, in questo modo riduciamo la complessità del modello dei casi d'uso.



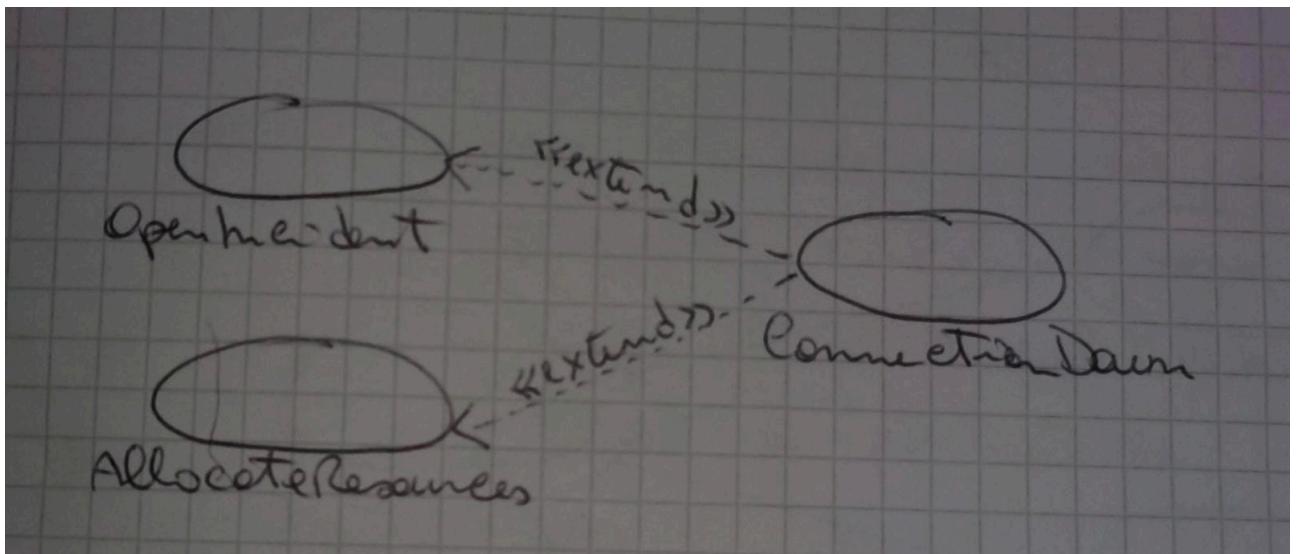
Nella descrizione del caso d'uso che lo include può essere rappresentata in due modi:

- Requisiti di qualità: se il caso d'uso può essere incluso in qualsiasi punto del flusso di eventi
- Flusso di eventi: se il caso d'uso incluso può essere invocato durante un evento

RELAZIONE <<extend>>

Una relazione extend indica che un'istanza di un caso d'uso può includere (in determinate condizioni) il comportamento specifico del caso d'uso esteso.

Separare il comportamento eccezionale dal comportamento comune, ci consente di scrivere casi più brevi e mirati.

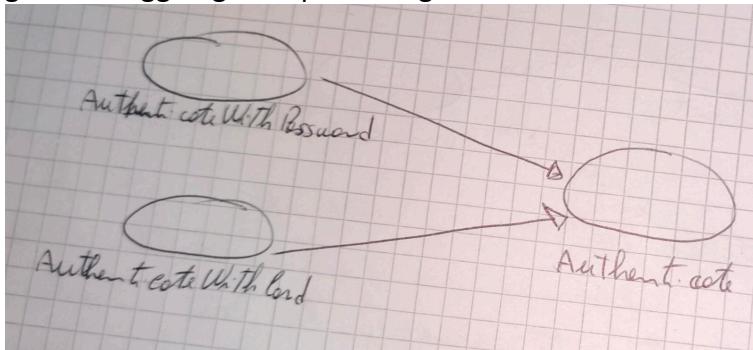


Nella descrizione del caso d'uso che lo estende lo inseriamo nella parte delle Eccezioni.

Nella descrizione del caso d'uso esteso lo rappresentiamo nella condizione d'ingresso.

EREDITARIETÀ

È una relazione dove possiamo specificare che un caso d'uso può specializzarne un altro più generale aggiungendo più dettagli.



Nella descrizione testuale dei casi d'uso specializzati, ereditano dal caso d'uso generale:

- Attore
- Condizione d'ingresso
- Condizione d'uscita

EXTEND vs EREDITARIETÀ

- Extend: ogni caso d'uso descrive un flusso di eventi differente
- Ereditarietà: descrivono entrambi la stessa attività ma a un livello di astrazione differente

SCENARI

Un caso d'uso è un'astrazione che descrive tutti i possibili scenari che coinvolgono la funzionalità descritta.

Uno **scenario** è un'istanza di un caso d'uso e vengono utilizzati per descrivere casi comuni, attraverso:

- Nome univoco
- Attori partecipanti
- Flusso eventi

CLASS DIAGRAM

I "class diagram" sono utilizzati per descrivere la struttura del sistema in termini di **oggetti, classi, attributi, operazioni, e associazioni**.

Le **classi** sono astrazioni che specificano struttura e comportamenti comuni di un insieme di oggetti.

Gli **oggetti** sono istanze di classi che vengono

- Creati
- Modificati
- Distrutti

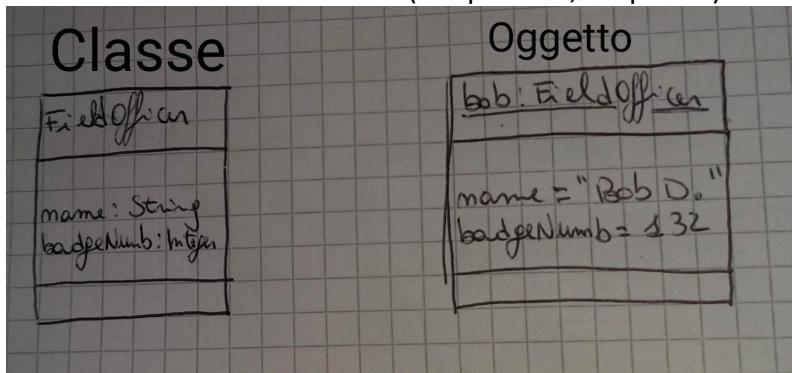
durante l'esecuzione del sistema.

Un oggetto è caratterizzato da:

- Stato: include i valori degli attributi
- Collegamento con altri oggetti

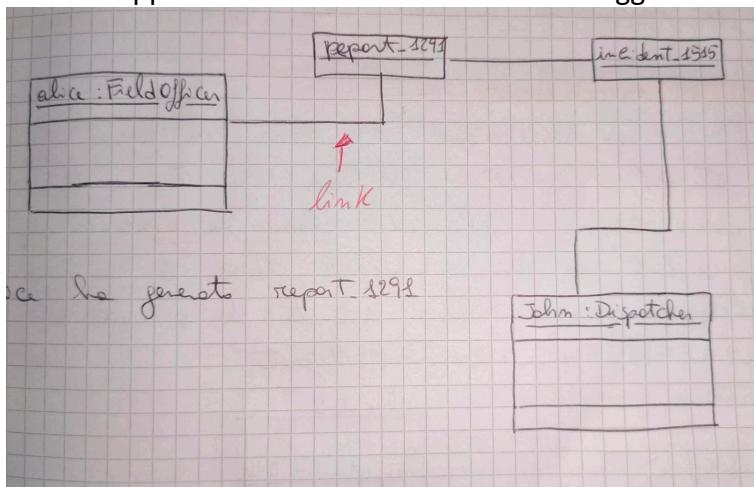
Le classi e gli oggetti in UML si rappresentano come quadrati divisi in 3 parti:

- Parte superiore: nome della classe (o oggetto)
- Parte centrale: attributi ("+" pubblici, "-" privati)
- Parte inferiore: metodi ("+" pubblici, "-" privati)

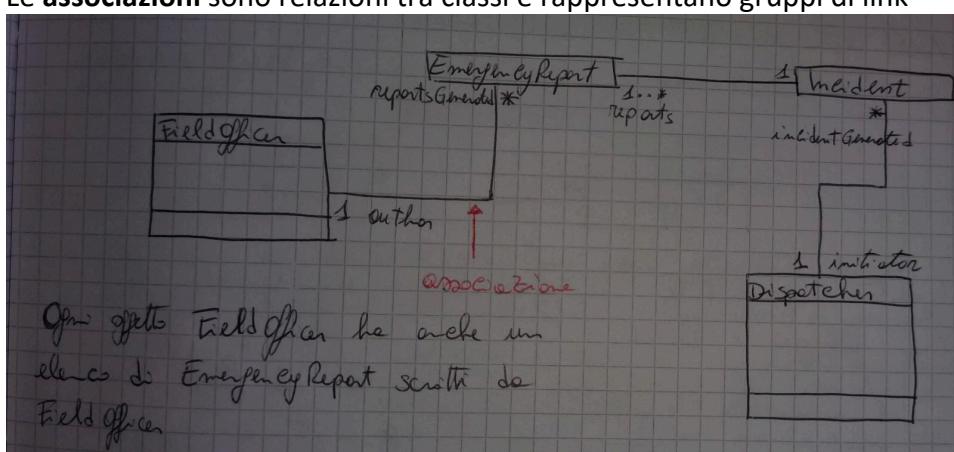


ASSOCIAZIONI E LINK

Un **link** rappresenta una connessione tra due oggetti



Le **associazioni** sono relazioni tra classi e rappresentano gruppi di link

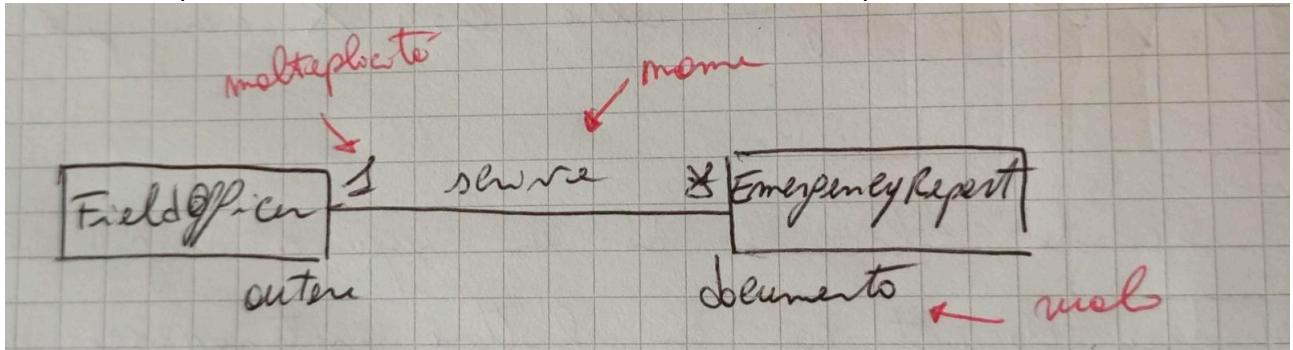


Le associazioni possono essere di due tipi:

- Simmetriche: bidirezionali
- Asimmetriche: unidirezionale

Le associazioni hanno diverse proprietà:

- Nome: descrive l'associazione tra due classi, è facoltativo e non per forza univoco
- Ruolo: si trova all'estremità e identifica la funzione di ciascuna classe
- Molteplicità: si trova all'estremità e identifica il numero di possibili istanze



CLASSE DI ASSOCIAZIONE

Le associazioni sono simili alle classi, in quanto possono avere attributi e operazioni, queste associazioni sono chiamate **classi di associazione**

RUOLI

Ogni estremità di un'associazione può essere etichettata da un ruolo, così da permetterci di distinguere tra le molteplici associazioni originate da una classe.

AGGREGAZIONE

Quando un oggetto può comporre con molteplicità multipla l'oggetto che lo aggredisce.

Es. Cartella aggrega File

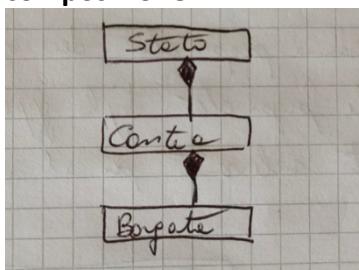
Le aggregazioni non rappresentano associazioni uno a molti, in quanto.

- aggregazione: denotano aspetti gerarchici della relazione
- uno a molti: implica una relazione tra pari

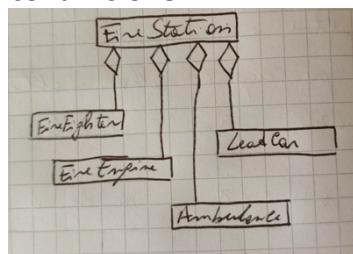
possono essere di due tipi:

- composizione: indica che l'esistenza delle parti dipende da quella generale
- condivisione: indica che il concetto generale e le singole parti esistono in maniera indipendente

composizione

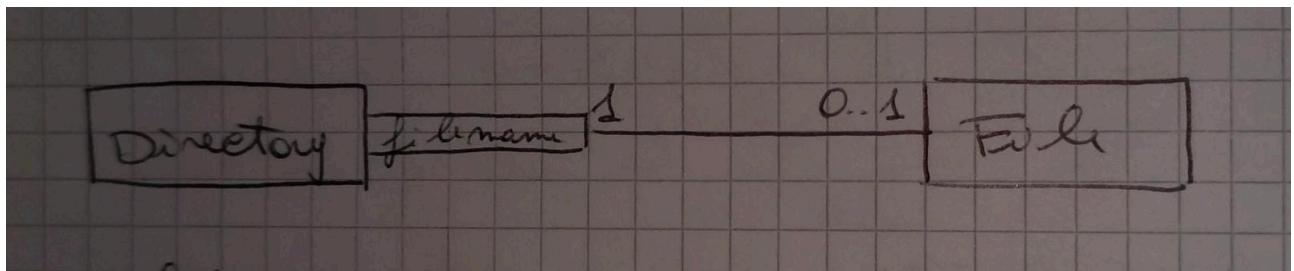


condivisione



QUALIFICAZIONE

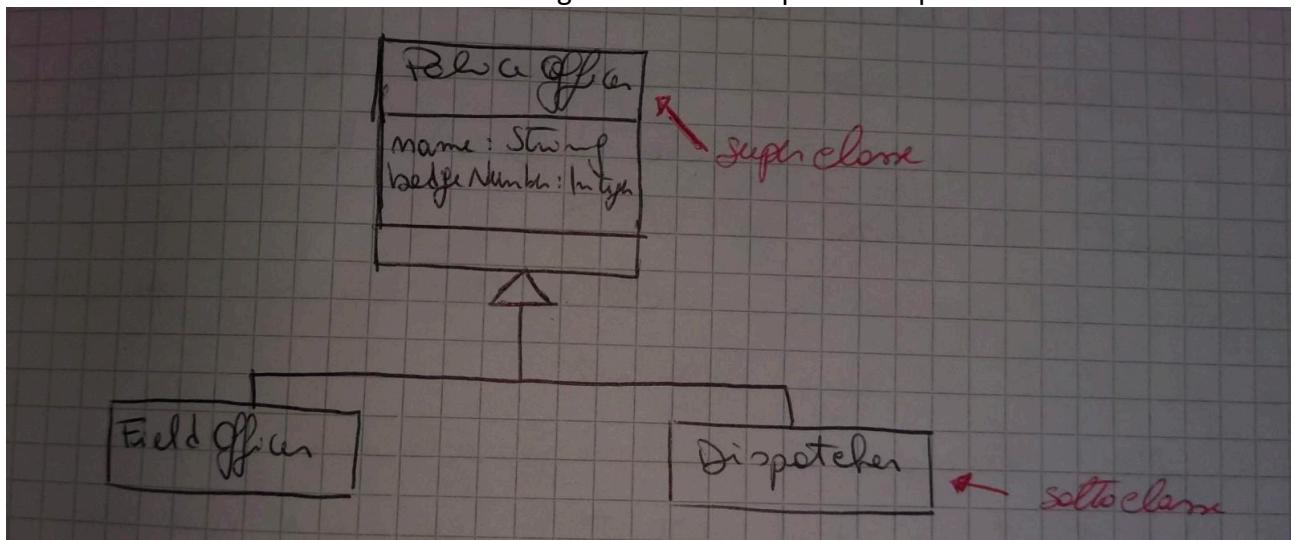
La **qualificazione** è una tecnica per ridurre le associazioni con molteplicità 0...n o 1...n, molte volte in una relazione uno a molti, l'oggetto dal lato "molti" può essere distinto utilizzando un nome



Un file appartiene esattamente a una directory, ogni file è identificato in modo univoco nel contesto della directory.

EREDITARIETÀ

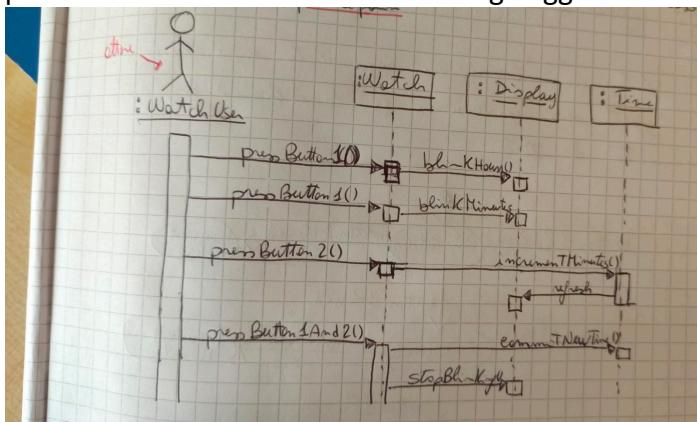
L'**ereditarietà** è la relazione tra una classe generale e una o più classi specializzate.



Le **sottoclassi** ereditano gli attributi e le operazioni dalla **superclasse**.

SEQUENCE DIAGRAM

Il "sequence diagram" viene utilizzato per formalizzare il comportamento dinamico del sistema e per visualizzare la comunicazione tra gli oggetti.



- La colonna più a sinistra rappresenta la sequenza temporale dell'attore, le altre rappresentano la sequenza temporale degli oggetti.
- I nomi degli oggetti sono sottolineati per indicare che sono istanze.
- Le frecce orizzontali rappresentano i messaggi inviati da un'oggetto ad un altro
- Il tempo procede verticalmente
- Il rettangolo rappresenta l'attivazione da cui possono pervenire messaggi
- Gli oggetti già esistenti prima di stimoli sono rappresentati nella parte superiore

EURISTICA PER DISEGNARE UN SEQUENCE DIAGRAM

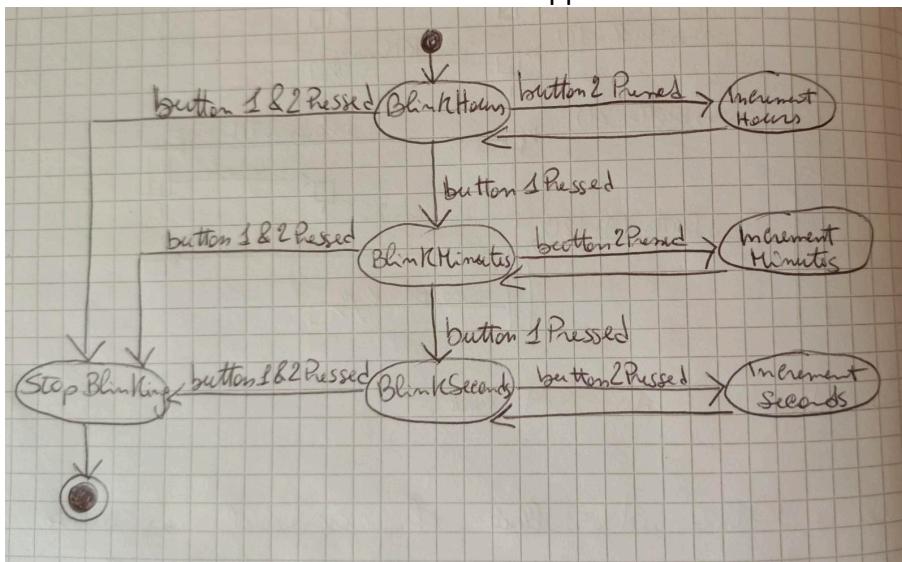
- La prima colonna corrisponde all'attore che ha avviato il caso d'uso
- La seconda colonna dovrebbe essere un boundary (che l'attore ha utilizzato per avviare il caso d'uso)
- La terza colonna dovrebbe essere un control che gestisce il caso d'uso
- I control vengono creati dai boundary
- I boundary vengono creati dai control
- Agli entity si accede tramite boundary o control
- Gli entity non accedono mai a boundary e control

STATECHART DIAGRAM

Gli "statechart diagram" vengono utilizzati per descrivere il comportamento dinamico di un singolo oggetto come numero di stati e transizioni tra questi stati.

Uno **stato** rappresenta un particolare insieme di valori per un oggetto.

Una **transizione** da uno stato a un altro rappresenta uno stato futuro in cui l'oggetto può trovarsi.



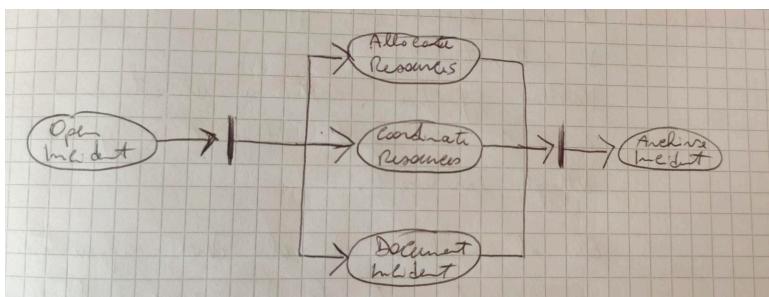
Il cerchio pieno “●” rappresenta lo stato iniziale

Il cerchio pieno circondato da uno vuoto (●) rappresenta lo stato finale

ACTIVITY DIAGRAM

Un "activity diagram" viene utilizzato per descrivere il comportamento di un sistema in termini di **attività**.

Le attività sono elementi di modellazione che rappresentano l'esecuzione di un insieme di operazioni, l'esecuzione di un'attività può essere innescata dal comportamento di altre attività.



CONCETTI DI MODELLAZIONE

- **Sistema:** insieme organizzato di parti comunicanti
- **Sottosistemi:** parti semplici di un sistema più complesso
- **Modellazione:** modellare significa costruire un'astrazione che si concentra su aspetti rilevanti
- **Modello di sistema:** insieme di tutti i modelli creati durante lo sviluppo
- **Vista:** si concentra su un sottosistema di un modello per renderlo comprensibile

- **Data type:** è un'astrazione del contesto di un linguaggio di programmazione, ha un nome univoco, insieme di valori che sono membri del data type e definisce la struttura e le operazioni valide in tutte le istanze del data type
 - Es.: "int" rappresenta il data type di tutti gli interi e ne definisce le operazioni valide per l'aritmetica di interi
- **Abstract data type:** tipo di dato definito da una specifica indipendente dall'implementazione, consentendo di ragionare su un insieme di istanze senza considerare un'implementazione specifica
- **Evento:** istanza di una classe di eventi, è un'occorrenza rilevante nel sistema.
 - Es: invio di messaggi tra due oggetti
- **Messaggio:** l'invio di un messaggio è il meccanismo mediante il quale l'oggetto mittente richiede l'esecuzione di un'operazione all'oggetto ricevente.
È composto da:
 - Nome
 - Serie di argomenti

MODELLAZIONE OBJECT ORIENTED

- **Dominio applicativo:** rappresenta tutti gli aspetti del problema dell'utente, incluso ambiente fisico, altre persone, processi di lavoro, ecc...

Il dominio applicativo cambia nel tempo man mano che i processi di lavoro e le persone cambiano.
- **Dominio delle soluzioni:** è lo spazio di modellazione di tutti i sistemi possibili, è molto più ricco del modello del dominio applicativo perché il sistema è modellato in maniera più dettagliata.
- **Analisi object oriented:** riguarda la modellazione del dominio applicativo
- **Progettazione object oriented:** riguarda la modellazione del dominio delle soluzioni
- **Modello dominio applicativo:** rappresenta entità dell'ambiente che sono rilevanti

FALSIFICAZIONE E PROTOTIPAZIONE

Un modello è una semplificazione della realtà, ovvero vengono messi in evidenza solo i dettagli rilevanti.

La **falsificazione** è il processo per dimostrare che i dettagli rilevanti sono stati rappresentati in modo errato o non rappresentati affatto.

Una tecnica utilizzata nella falsificazione è la **prototipazione**, durante la progettazione dell'interfaccia utente si costruiscono dei prototipi che simulano solo l'interfaccia, il prototipo poi viene presentato ai potenziali acquirenti che la valuteranno e quindi effettueranno una falsificazione.

REQUIREMENTS ELICITATION

La fase di **requirements elicitation** serve a descrivere lo scopo del sistema, gli sviluppatori osservano gli utenti finali per costruire un modello del dominio applicativo e successivamente rappresentarlo attraverso un linguaggio naturale (scenari e casi d'uso).

La fase di requirements elicitation comprende le fasi:

- Identificazione attori: gli sviluppatori identificano i diversi tipi di utenti
- Identificazione scenari: gli sviluppatori osservano gli utenti e sviluppano gli scenari
- Identificazione casi d'uso: una volta concordato con gli utenti una serie di scenari, gli sviluppatori dagli scenari ricavano i casi d'uso
- Raffinamento casi d'uso: durante questa attività gli sviluppatori si assicurano che la specifica dei requisiti sia completa
- Identificazione relazioni casi d'uso: gli sviluppatori identificano le dipendenze tra i casi d'uso, fattorizzando funzionalità comuni.
- Identificazione requisiti non funzionali: durante questa fase, sviluppatori, utente e cliente concordano su aspetti che sono visibili all'utente ma non correlate alle funzionalità (es. Prestazioni, documentazione, ecc...)

Durante questa fase si accede a molte fonti d'informazioni, ma ci concentriamo su due metodi principali per ottenere informazioni o prendere decisioni con utente e clienti:

- Documento JAD: si concentra sullo sviluppare congiuntamente la specifica dei requisiti
- Tracciabilità: si concentra sulle registrazioni, strutturazione, collegamento delle dipendenze tra requisiti e altri work product.

REQUISITI FUNZIONALI

I **requisiti funzionali** descrivono le interazioni tra il sistema e il suo ambiente (utente o sistemi legacy) indipendentemente dalla implementazione.

REQUISITI NON FUNZIONALI

I **requisiti non funzionali** descrivono aspetti del sistema che non sono direttamente correlati al comportamento funzionale del sistema.

Esistono diversi requisiti non funzionali:

- Usabilità: facilità con cui un utente può operare, preparare input e interpretare output di un sistema. (es. Interfaccia utente, documentazione utente)
- Affidabilità: capacità di un sistema di eseguire le funzioni richieste in condizioni stabilite per un periodo di tempo specificato. (es, tempo di attesa dal guasto, capacità di rilevare guasti)
- Performance: riguarda attributi quantificabili dal sistema. (es, tempo di risposta, throughput)
- Supportabilità: la facilità di modifiche al sistema dopo la distribuzione. (es, adattabilità, manutenibilità)

PSEUDO REQUISITI

Il modello FURPS+ definisce ulteriori requisiti, definiti **pseudo requisiti**:

- Implementazione: vincoli all'implementazione. (es, linguaggio di programmazione, piattaforme hw specifiche)
- Interfaccia: vincoli per l'interfacciamento con sistemi legacy e formati d'intercambio
- Operativi: vincoli all'amministrazione e gestione del sistema
- Imballaggio: vincoli alla consegna effettiva del sistema (es, supporto installazione, configurazione)
- Legali: vincoli relative a licenze

VALIDAZIONE DEI REQUISITI

La convalida rei requisiti è una fase fondamentale del processo di sviluppo, i requisiti vengono costantemente validati con il cliente e l'utente.

La convalida rei requisiti implica la verifica che la specifica sia:

- Completa: tutte le funzionalità di interesse sono descritte in base ai requisiti
- Coerente: è coerente se la specifica non contraddice sé stessa, ovvero non esistono requisiti in contraddizione tra loro
- Non ambigua: la specifica non è ambigua se si riesce a definire esattamente un sistema
- Corretta: i requisiti descrivono le caratteristiche del sistema e dell'ambiente d'interesse per il cliente e lo sviluppatore
- Reale: il sistema può essere implementato entro i limiti
- Verificabile: una volta costruito il sistema, è possibile costruire test ripetibili per dimostrare che il sistema rispetti la specifica dei requisiti
- Tracciabile: ogni requisito può essere ricondotto alle funzioni di sistema corrispondenti e ciascuna funzione può essere ricondotta al set di requisiti corrispondenti, è fondamentale per:
 - Sviluppo di test: consente a un tester di valutare la copertura di un caso di test
 - Valutazione delle modifiche: consente all'analista e sviluppatori di identificare i componenti e le funzioni di sistema su cui le modifiche avrebbero un impatto

TIPI DI REQUIREMENTS ELICITATION

Le attività di requirements elicitation possono essere categorizzate in:

- Greenfield engineering: lo sviluppo inizia da zero, non esiste alcun sistema precedente. Si segue tutta la fase di identificazione dei requisiti. Un progetto di questo tipo si rende utile quando l'utente ha particolari esigenze o dalla creazione di un nuovo mercato
- Reengineering: riprogettazione e re-implementazione di un sistema esistente. Un progetto di questo tipo si rende utile con utilizzo di nuove tecnologie o estendere le funzionalità.
- Interface engineering: riprogettazione dell'interfaccia utente di un sistema esistente. La parte legacy rimane la stessa ma si modifica l'interfaccia.

Durante il greenfield enginnering e reengineering gli sviluppatori possono prelevare informazioni da:

- Manuali delle procedure
- Documentazione ai nuovi dipendenti
- Glossari
- Interviste a utenti e clienti

IDENTIFICAZIONE ATTORI

Gli attori rappresentano entità esterne che interagiscono con il sistema, un attore può essere una persona o un sistema esterno.

Una serie di domande per l'identificazione:

- Quali gruppi di utenti sono supportati dal sistema?
- Quali gruppi di utenti eseguono funzioni principali?
- Con quale sistema hw/sw esterno interagisce il sistema?

IDENTIFICAZIONE SCENARI

Uno scenario è una descrizione narrativa di ciò che le persone fanno e sperimentano mentre cercano di utilizzare sistemi o applicazioni.

Gli scenari non possono sostituire i casi d'uso in quanto si concentrano su istanze specifiche ed eventi concreti.

Gli scenari possono essere:

- As-is: descrivono una situazione attuale (es. osservando l'interazione che ha un utente con un sistema esistente)
- Visionary: descrivono un sistema futuro, possono essere visti come prototipi economici
- Evaluation: descrivono le attività dell'utente rispetto alle quali il sistema deve essere valutato
- Training: sono esercitazioni utilizzate per introdurre nuovi utenti nel sistema.

Una serie di domande per identificare gli scenari:

- Quali sono i compiti che l'attore vuole che il sistema svolga?
- A quali informazioni accede l'attore?
- Chi crea le informazioni?

IDENTIFICAZIONE CASI D'USO

Uno scenario è un'istanza di un caso d'uso, ovvero, un caso d'uso specifica tutti i possibili scenari per una determinata funzionalità.

Un caso d'uso rappresenta un flusso completo di eventi.

DOCUMENTAZIONE REQUIREMENTS ELICITATION

Le attività di Requirements elicitation vengono documentate nel RAD, il documento è strutturato:

- Introduzione: fornire una breve panoramica delle funzioni del sistema e delle ragioni del suo sviluppo
- Sistema corrente: descrizione dello stato attuale delle cose. Se il nuovo sistema ne sostituisce uno già esistente, in questa sezione discuteremo le funzionalità e i problemi del sistema corrente. Altrimenti descriviamo come vengono eseguite le attività supportate dal nuovo sistema
- Sistema proposto: suddiviso in 4 sezioni
 - Overview: panoramica funzionale del sistema
 - Requisiti funzionali: funzionalità ad alto livello del sistema
 - Requisiti non funzionali: requisiti a livello di utente che non sono direttamente correlati alle funzionalità
 - System models: descrivono gli scenari, i casi d'uso, modello a oggetti e modello dinamico per il sistema, inoltre comprende mock-ups e navigational path.

Il RAD viene scritto quando il modello dei casi d'uso è stabile, ovvero il numero di modifiche ai requisiti è minimo.

ANALISI

L'analisi si concentra sulla produzione di un modello di analisi che sia:

- Corretto
- Completo
- Coerente
- Verificabile

Nella fase di analisi ci si concentra sulla strutturazione e formalizzazione dei requisiti.

Il modello di analisi è composto da tre modelli individuali:

- Modello funzionale
 - Casi d'uso
 - Scenari
- Modello a oggetto di analisi
 - Class diagram
- Modello dinamico
 - Statechart diagram
 - Sequence diagram

Il **modello a oggetti di analisi** si concentra sui singoli concetti che vengono manipolati dal sistema, le loro proprietà e le loro relazioni, rappresentato in UML questo include Classi, attributi e operazioni.

È costituito da:

- Entity object: rappresentano le informazioni persistenti tracciate dal sistema (es. Year)
- Boundary object: rappresentano le interazioni tra gli attori e il sistema (es. ButtonBoundary)
- Control object: hanno il compito di realizzare il caso d'uso (es. ChangeDateControl)

Il **modello dinamico** si concentra sul comportamento del sistema, rappresentato attraverso i sequence diagram che a loro volta rappresentano interazioni tra insieme di oggetti durante un caso d'uso. Con il modello dinamico assegniamo responsabilità alle singole classi.

IDENTIFICAZIONE ENTITY OBJECT

Gli oggetti partecipanti vengono trovati esaminando ogni caso d'uso e identificando *oggetti candidati*. **Abbott** ideò una tecnica che si basa sull'analisi del linguaggio naturale per identificare oggetti, attributi e associazioni da una specifica dei requisiti.

La tecnica di **Abbott** consiste nel mappare parti del discorso (nomi, verbo avere, verbo essere, aggettivi) su componenti del modello (oggetti, operazioni, ereditarietà, classi).

Il *vantaggio* di questa tecnica è che si concentra sui termini degli utenti, ma soffre di alcune *limitazioni*:

- La qualità del modello a oggetti dipende fortemente dallo stile di scrittura dell'analista, inoltre il linguaggio naturale è uno strumento impreciso e un modello a oggetti derivato da esso rischia di essere impreciso. Si può affrontare tale limitazione riformulando e chiarendo la specifica dei requisiti.

- Esistono molti sostantivi nel linguaggio naturale che classi rilevanti, ordinare tutti i nomi per una specifica dei requisiti di grandi dimensioni richiede tempo.

La tecnica di Abbott funziona bene per generare un elenco di candidati iniziali da descrizioni brevi. Le associazioni possono essere identificate esaminando verbi e frasi verbali.

IDENTIFICAZIONE BOUNDARY OBJECT

I boundary object rappresentano l'interfaccia del sistema con gli attori, modellano l'interfaccia utente in modo grossolano, non descrivono in dettaglio gli aspetti visivi della UI.

IDENTIFICAZIONE CONTROL OBJECT

I control object sono responsabili del coordinamento dei boundary e entity object, di solito hanno una controparte concreta nel mondo reale. È responsabile della raccolta delle informazioni dai boundary e l'invio agli entity.

REVISIONE MODELLO DI ANALISI

Difficilmente un modello di analisi è corretto o completo di prima stesura, sono necessarie diverse iterazioni con il cliente e l'utente prima che il modello di analisi diventi una specifica corretta per poi essere utilizzata in fase di progettazione e implementazione.

L'obiettivo della revisione è assicurarsi che la specifica dei requisiti sia corretta , completa, coerente e non ambigua.

Per assicurarci che il modello sia:

- Corretto: dobbiamo porci delle domande
 - Il glossario degli entity è comprensibile all'utente?
 - Tutte le descrizioni sono conformi alle definizioni degli utenti?
 - Tutti gli entity e boundary object hanno nomi con frasi nominali significativi?
 - Tutti i control object e i casi d'uso hanno nomi con frasi verbali?
 - Vengono descritti e gestiti i casi d'errore?
- Completo: dobbiamo porci delle domande
 - Per ogni oggetto
 - È necessario per qualsiasi caso d'uso?
 - In quale caso d'uso viene creato?
 - Modificato?
 - Distrutto?
 - Per ogni attributo:
 - Quando viene impostato?
 - Qual è il suo tipo?
 - Per ogni associazione:
 - Quando viene attraversata?
 - Perché è stata scelta quella molteplicità specifica?
 - Per ogni control:
 - Dispone delle associazioni necessarie per accedere agli oggetti partecipanti al suo caso d'uso?
- Coerente: dobbiamo porci delle domande
 - Esistono più classi o casi d'uso con lo stesso nome?
 - Sono presenti oggetti con attributi e associazioni simili che non sono nella stessa gerarchia di generalizzazione?

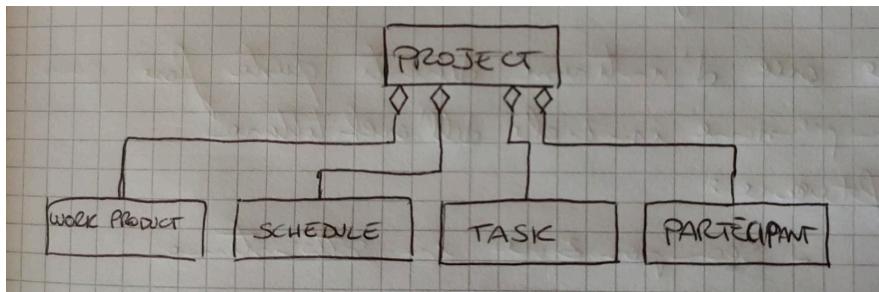
- Realistico: dobbiamo porci delle domande
 - Ci sono nuove funzionalità nel sistema?
 - Sono stati realizzati studi o prototipi per garantire la fattibilità?

GESTIONE DELL'ANALISI

È essenziale che in un progetto multi-team ci sia la coerenza utilizzando tante risorse, il documento di analisi finale deve descrivere un unico sistema coerente e deve essere comprensibile a una singola persona, per fare ciò dobbiamo:

- Documentare l'analisi:
 - il modello di oggetti deve documentare in dettaglio tutti gli oggetti che abbiamo identificato, i loro attributi e quando descriviamo i sequence anche le operazioni.
 - Il modello dinamico deve documentare il comportamento del modello ad oggetti in termini di statechart diagram
- Assegnazione responsabilità: assegnando ruoli e ambiti ben definiti agli individui
- Comunicare sull'analisi
 - Definire territori liberi: forum pubblici e privati
 - Definire obiettivi chiari e criteri di successo: definizione chiare e criteri di successo servono per risolvere eventuali conflitti essendo misurabili e verificabili
- Brainstorming: mettere tutti gli stakeholder nella stessa stanza e generare rapidamente soluzioni e definizioni aiuta a rimuovere barriere sulla comunicazione

ORGANIZZAZIONE E COMUNICAZIONE DEL PROGETTO



- Work product: qualsiasi elemento di progetto (codice, modello o documento), chiamati anche Deliverables
- Schedule: specifica quando deve essere completato il lavoro sul progetto
- Participant: qualsiasi persona che partecipa a un progetto, chiamati anche membri del progetto
- Task: lavoro che deve essere svolto da un participant per creare un work product

Durante la fase di **definizione del progetto** sono coinvolti:

- Project manager
- Possibili clienti
- Architetto software

Le due aree di interesse durante questa fase sono:

- Composizione iniziale dell'architettura software
 - Scomposizione in sottosistemi
- Progetto

- Scheduli
- Lavoro da eseguire
- Risorse necessarie

Tutto ciò è documentato in tre documenti:

- Problem statement
- Documento iniziale architettura software
- Piano di gestione del progetto software iniziale

Durante la fase iniziale il project manager imposta l'infrastruttura del progetto, assume i partecipanti, organizza i team e definisce le tappe principali.

TASK E WORK PRODUCT

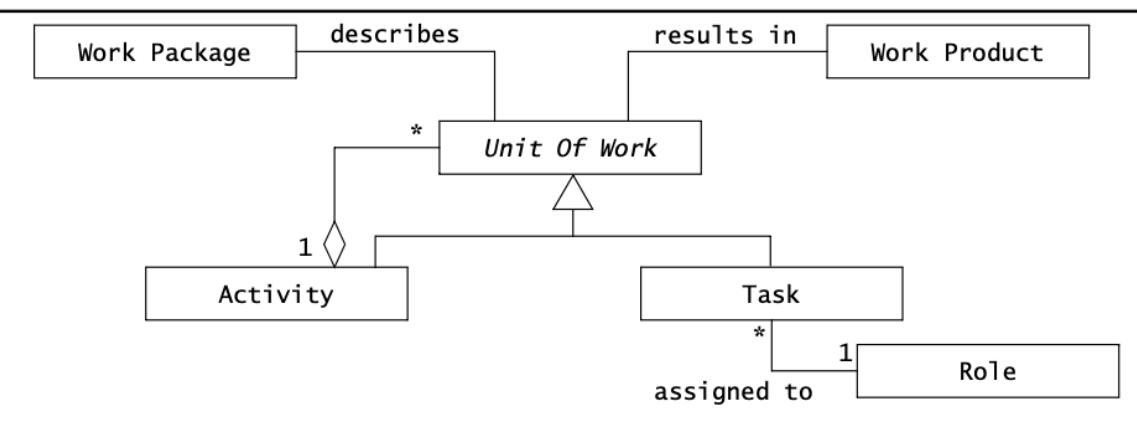
Un **task** è un'assegnazione di lavoro ben definita per un ruolo. I gruppi di task correlate sono chiamati attività.

Un **work product** è un elemento tangibile che risulta da un'attività. Esempi di work product includono un modello a oggetti, un class diagram, una parte di codice sorgente.

La specifica del work da compiere per completare un task o un'attività è descritta in un **work package**.

Un work package include:

- il nome del task
- la descrizione del task
- le risorse necessarie per eseguire il task
- le dipendenze dagli input (work product prodotti da altri task) e gli output (work product prodotti dal task in questione), nonché le dipendenze da altri task.



PIANIFICAZIONE

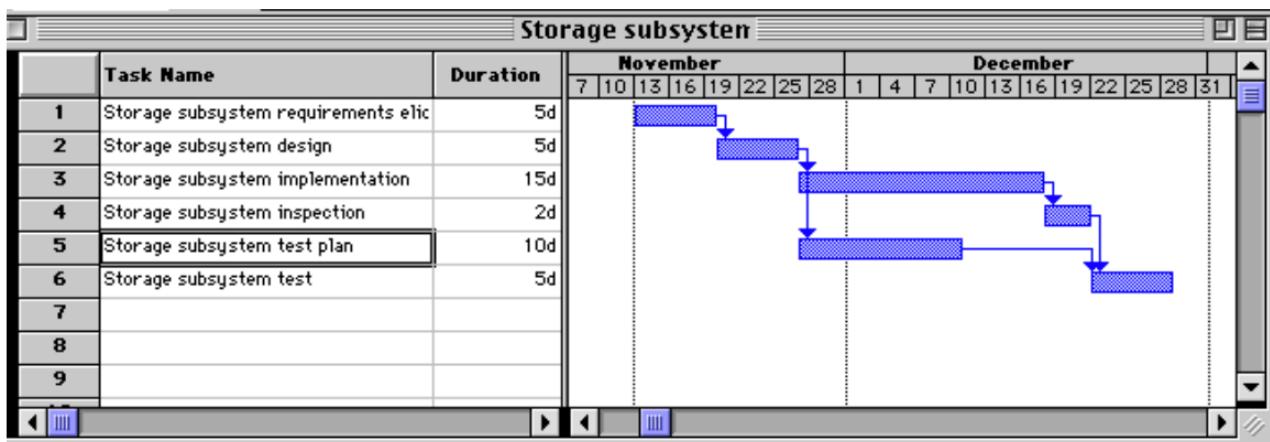
Una pianificazione è la mappatura dei task nel tempo, a ciascun task viene assegnato l'ora di inizio e di fine.

Questo ci consente di pianificare le scadenze per i singoli deliverable.

Le due notazioni diagrammatiche utilizzate più spesso per gli abachi sono **PERT** e **Gantt**

Un diagramma di Gantt è un grafico a barre su cui:

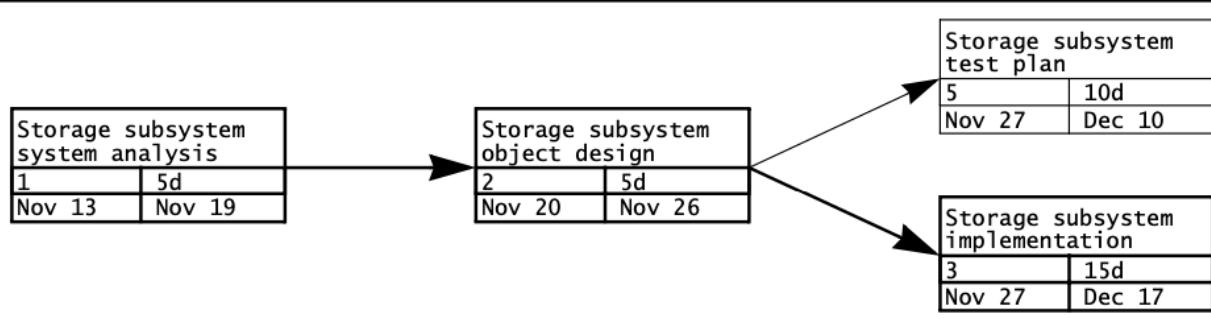
- l'asse orizzontale rappresenta il tempo
- l'asse verticale elenca i diversi task da svolgere



Un grafico PERT rappresenta una pianificazione come un grafico aciclico di attività.

L'inizio e la durata pianificati dei task vengono utilizzati per calcolare il percorso critico, che rappresenta il percorso più breve possibile attraverso il grafico. La lunghezza del percorso critico corrisponde alla pianificazione più breve possibile, assumendo risorse sufficienti per svolgere, in parallelo, compiti indipendenti.

Inoltre, le attività sul percorso critico sono le più importanti, poiché un ritardo in una qualsiasi di queste attività comporterà un ritardo nel progetto complessivo. Le attività e le barre rappresentate con linee più spesse appartengono al percorso critico.



CONFIGURATION MANAGEMENT

Il **configuration management** è la disciplina di gestione e controllo del cambiamento nell'evoluzione dei sistemi software.

I sistemi di configuration management automatizzano l'identificazione delle versioni, la loro memorizzazione e il loro recupero e supportano la contabilità dello stato.

Il configuration management include le seguenti attività:

- **Identificazione degli elementi di configurazione.** I componenti del sistema e dei suoi work product e le loro versioni sono identificati ed etichettati in modo univoco. Gli sviluppatori creano versioni e elementi di configurazione aggiuntivi man mano che il sistema si evolve.
- **Cambia controllo.** Le modifiche al sistema e le versioni per gli utenti sono controllate per garantire la coerenza con gli obiettivi del progetto.
- **Stato contabile.** Lo stato dei singoli componenti, work product e richieste di modifica viene registrato. Ciò consente agli sviluppatori di distinguere le versioni più facilmente e di tenere traccia dei problemi relativi alle modifiche.
- **Auditing.** Le versioni selezionate per il rilascio vengono convalidate per garantire la completezza, la coerenza e la qualità del prodotto. L'audit viene eseguito dal team di controllo qualità.

CONCETTI DI CONFIGURATION MANAGEMENT

- Una **change request** è un rapporto formale emesso da un utente o uno sviluppatore per richiedere una modifica in un elemento di configurazione.
- Una **versione** identifica lo stato di un elemento di configurazione o di un aggregato di configurazione in un momento ben definito.
- Le **versioni** destinate a coesistere sono chiamate *varianti*.
- Una **promozione** è una versione che è stata resa disponibile ad altri sviluppatori del progetto. Una **release** è una versione che è stata messa a disposizione del cliente o degli utenti.
- Un **repository** è una libreria di rilasci.
- Una **master directory** è una libreria di promozioni.

REPOSITORIES E WORKSPACES

Una **libreria software** fornisce funzionalità per memorizzare, etichettare e identificare le versioni degli elementi di configurazione. Una libreria software fornisce anche funzionalità per tenere traccia dello stato delle modifiche agli elementi di configurazione.

Distinguiamo tre tipi di librerie:

- **Workspace developer**, nota anche come *libreria dinamica*, viene utilizzata per lo sviluppo quotidiano dagli sviluppatori. Il cambiamento non è limitato ed è controllato solo dal singolo sviluppatore.
- **Master directory**, nota anche come *libreria controllata*, tiene traccia delle promozioni. La modifica deve essere approvata e le versioni devono soddisfare determinati criteri di progetto prima di essere rese disponibili per il resto del progetto.
- **Software repository**, nota anche come *libreria statica*, tiene traccia delle versioni. Le promozioni devono soddisfare determinati criteri di controllo della qualità prima che una promozione diventi un'uscita.

GESTIONE DELLE PROMOZIONI

La creazione di una nuova promozione per un elemento di configurazione si verifica quando uno sviluppatore desidera condividere l'elemento di configurazione con altri. Gli sviluppatori creano promozioni per rendere l'elemento di configurazione disponibile per la revisione o allo scopo di eseguire il debug di un altro elemento di configurazione.

GESTIONE DELLE RELEASE

La creazione di una nuova versione per un elemento di configurazione o un aggregato CM è una decisione di gestione, generalmente basata sul marketing e sul controllo di qualità. È disponibile una versione per offrire funzionalità aggiuntive (o riviste) o per risolvere bug critici.

GESTIONE DEI BRANCH

Di solito, gli sviluppatori lavorano su più miglioramenti contemporaneamente, queste modifiche vengono assegnate a diversi team a causa della differenza nei rischi associati.

Per supportare entrambe le modifiche contemporaneamente mantenendo i team indipendenti, abbiamo creato un branch, a partire dalle ultime promozioni dei sottosistemi al momento dell'approvazione delle modifiche

SYSTEM DESIGN: DECOMPOSIZIONE DEL SISTEMA

La fase di system design è la fase di trasformazione di un modello di analisi in un modello di system design. Durante il system design, gli sviluppatori definiscono gli obiettivi di progettazione del progetto e scompongono il sistema in sottosistemi più piccoli che possono essere realizzati dai singoli team.

Gli sviluppatori selezionano anche strategie per la costruzione del sistema,

- strategia hardware / software
- gestione dei dati persistenti
- flusso di controllo globale
- politica di controllo degli accessi
- gestione delle boundary condition

Il modello di analisi, tuttavia, non contiene informazioni sulla struttura interna del sistema, la sua configurazione hardware, o più in generale, come dovrebbe essere realizzato il sistema.

Il system design è il primo passo in questa direzione. La fase di system design produce i seguenti prodotti:

- **design goals:** descrivendo le qualità del sistema che gli sviluppatori dovrebbero ottimizzare (derivano dai requisiti non funzionali.)
- **architettura software:** descrive la scomposizione del sottosistema in termini di
 - responsabilità del sottosistema
 - dipendenze tra sottosistemi
 - mappatura del sottosistema sull'hardware
 - flusso di controllo
 - controllo degli accessi
 - archiviazione dei dati
- **casi d'uso limite:** descrivono i problemi di configurazione del sistema, avvio, arresto e gestione delle eccezioni.

Durante la progettazione del sistema, definiamo i sottosistemi in termini di servizi che forniscono. Un **servizio** è un insieme di operazioni correlate che condividono uno scopo comune.

Durante l'object design, definiamo l'interfaccia del sottosistema in termini di operazioni che fornisce.

Successivamente, esaminiamo due proprietà dei sottosistemi:

- **Accoppiamento:** misura le dipendenze tra due sottosistemi
- **Coesione:** misura le dipendenze tra le classi all'interno di un sottosistema

La decomposizione ideale del sottosistema dovrebbe ridurre al minimo l'accoppiamento e massimizzare la coesione.

Esistono due tecniche per mettere in relazione i sottosistemi tra loro

- **Layer:** consente a un sistema di essere organizzato come una gerarchia di sottosistemi, ciascuno dei quali fornisce servizi di livello superiore al sottosistema sopra di esso utilizzando servizi di livello inferiore dai sottosistemi sottostanti.
- **Partizionamento:** organizza i sottosistemi come peer che forniscono reciprocamente servizi diversi.

SOTTOSISTEMI E CLASSI

Per ridurre la complessità del dominio applicativo, abbiamo identificato parti più piccole chiamate "classi", allo stesso modo, per ridurre la complessità del dominio della soluzione, scomponiamo un sistema in parti più semplici, chiamate "**sottosistemi**", che sono costituite da una serie di classi del dominio della soluzione.

Decomponendo il sistema in sottosistemi relativamente indipendenti, i team simultanei possono lavorare su singoli sottosistemi con un sovraccarico di comunicazione minimo.

I componenti sono rappresentati come rettangoli con l'icona del componente nell'angolo in alto a destra.

I componenti possono rappresentare

- **Componenti logici:** corrisponde a un sottosistema che non ha un equivalente di runtime esplicito, ad esempio, singoli componenti aziendali composti insieme in un unico livello di logica dell'applicazione di runtime.
- **Componenti fisici:** corrisponde a un sottosistema che come equivalente di runtime esplicito, ad esempio, un server di database.

ACCOPIAMENTO E COESIONE

L'**accoppiamento** è il numero di dipendenze tra due sottosistemi.

Se due sottosistemi sono

- *Debolmente accoppiati*, sono relativamente indipendenti, quindi le modifiche a uno dei sottosistemi avranno un impatto minimo sull'altro.
- *Fortemente accoppiati*, è probabile che le modifiche a un sottosistema abbiano un impatto sull'altro.

La **coesione** è il numero di dipendenze all'interno di un sottosistema.

Se un sottosistema,

- contiene molti oggetti che sono correlati tra loro e svolgono attività simili, la sua coesione è *alta*.
- contiene un numero di oggetti non correlati, la sua coesione è *bassa*.

LAYER E PARTIZIONI

Una **scomposizione gerarchica** di un sistema produce un insieme ordinato di layer. Un **layer** è un raggruppamento di sottosistemi che forniscono servizi correlati, possibilmente realizzati utilizzando servizi di un altro layer. I layer sono ordinati in quanto ogni layer può dipendere solo da layer di *layer inferiore* e non ha alcuna conoscenza dei *layer superiori*.

In un'**architettura chiusa**, ogni livello può accedere solo al layer immediatamente sottostante. Le architetture chiuse e stratificate hanno proprietà desiderabili:

- basso accoppiamento tra sottosistemi
- i sottosistemi possono essere integrati e testati in modo incrementale

Un problema principale è che l'aggiunta di funzionalità al sistema nelle revisioni successive potrebbe rivelarsi difficile, soprattutto quando le aggiunte non erano previste.

In un'**architettura aperta**, un livello può anche accedere a livelli a layer più profondi. In generale, un'architettura aperta consente agli sviluppatori di aggirare i livelli superiori per affrontare i colli di bottiglia delle prestazioni.

Un altro approccio per affrontare la complessità consiste nel **partizionare** il sistema in sottosistemi peer, ciascuno responsabile di una diversa classe di servizi, ogni sottosistema dipende liberamente dagli altri, ma spesso può funzionare in modo isolato.

STILI ARCHITETTURALI

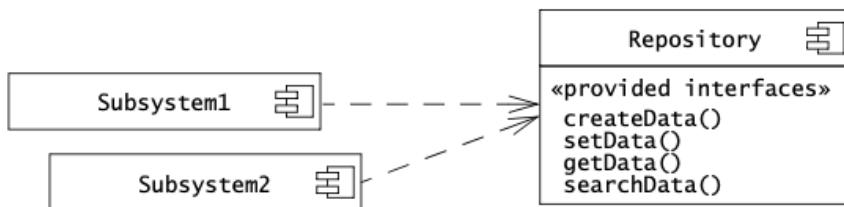
Con l'aumentare della complessità dei sistemi, la specifica della scomposizione del sistema è fondamentale. È difficile modificare o correggere una decomposizione debole una volta avviato lo sviluppo, poiché la maggior parte delle interfacce dei sottosistemi dovrebbe cambiare.

Quindi è emerso il concetto di **architettura software**. Un'architettura software include

- scomposizione del sistema
- flusso di controllo globale
- gestione delle boundary condition
- protocolli di comunicazione tra sottosistemi

REPOSITORY

Nello stile architetturale repository, i sottosistemi accedono e modificano una singola struttura di dati chiamata repository centrale.



I sottosistemi sono relativamente indipendenti e interagiscono solo attraverso il repository. Il flusso di controllo può essere dettato dal repository centrale o dai sottosistemi, utile per sistemi di gestione dei database.

Vantaggi

- La posizione centrale dei dati semplifica la gestione dei problemi di concorrenza e integrità tra i sottosistemi
- adatti per applicazioni con attività di elaborazione dati complesse e in continua evoluzione. Una volta che un repository centrale è ben definito, possiamo facilmente aggiungere nuovi servizi sotto forma di sottosistemi aggiuntivi.

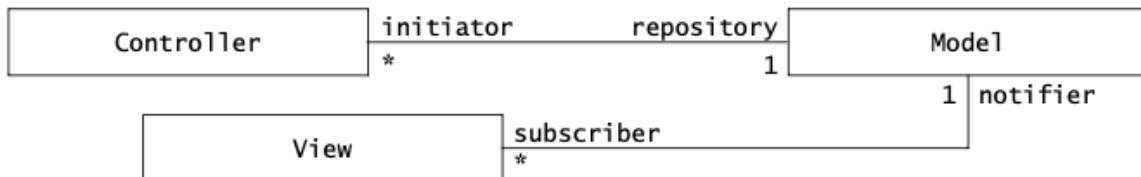
Svantaggi

- il repository centrale può rapidamente diventare un collo di bottiglia, sia dal punto di vista delle prestazioni che da quello della modificabilità.
- L'accoppiamento tra ogni sottosistema e il repository è elevato, rendendo così difficile cambiare il repository senza avere un impatto su tutti i sottosistemi.

MODEL/VIEW/CONTROLLER

Nello stile architettonico Model/View/Controller (MVC), i sottosistemi sono classificati in tre diversi tipi:

- **Modello:** mantengono la conoscenza del dominio, sono sviluppati in modo tale da non dipendere da alcuna vista o sottosistema del controller
- **Visualizzazione:** la mostrano all'utente
- **Controller:** gestiscono la sequenza di interazioni con l'utente



Le modifiche al loro stato vengono propagate al sottosistema di View tramite un protocollo di subscribe/notify, associata a questa sequenza di eventi viene solitamente realizzata con un modello di progettazione **Observer**.

Il design pattern Observer consente di disaccoppiare ulteriormente gli oggetti Model e View rimuovendo le dipendenze dirette dal Model alla View.

Vantaggi

- coerenza tra i dati distribuiti

Svantaggi

- collo di bottiglia delle prestazioni degli altri stili di repository

CLIENT/SERVER

Nello stile architettonico client/server, un sottosistema, il **server**, fornisce servizi alle istanze di altri sottosistemi chiamati **client**, che sono responsabili dell'interazione con l'utente.

La richiesta di un servizio viene solitamente eseguita tramite un meccanismo di chiamata di procedura remota o un broker di oggetti comune.

I client sono responsabili:

- della ricezione degli input dall'utente
- dell'esecuzione dei controlli dell'intervallo
- dell'avvio delle transazioni del database quando vengono raccolti tutti i dati necessari

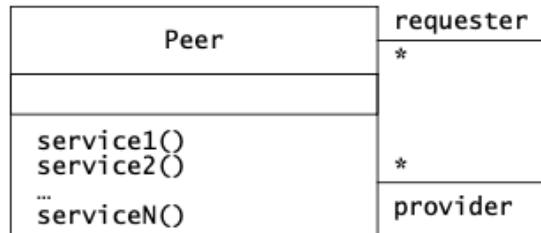
Il server è quindi responsabile

- dell'esecuzione della transazione
- della garanzia dell'integrità dei dati



PEER-TO-PEER

Uno stile architetturale peer-to-peer è una generalizzazione dello stile architetturale client/server in cui i sottosistemi possono agire sia come client che come server, nel senso che ogni sottosistema può richiedere e fornire servizi.

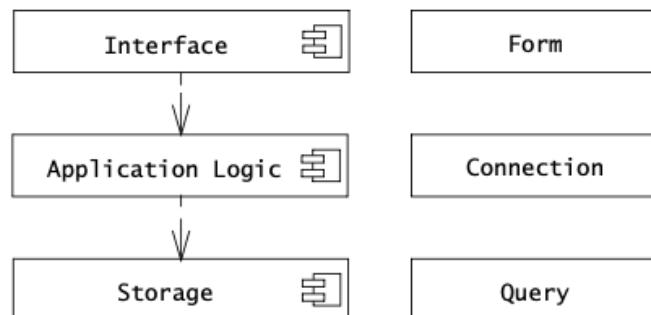


I sistemi peer-to-peer sono più difficili da progettare rispetto ai sistemi client/server perché introducono la possibilità di deadlock e complicano il flusso di controllo.

THREE-TIER

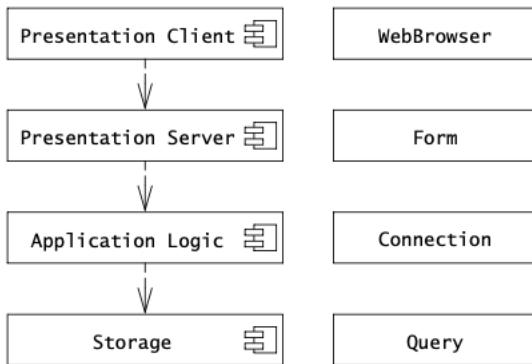
Lo stile architetturale three-tier organizza i sottosistemi in tre livelli:

- **Interfaccia:** include tutti gli oggetti boundary che hanno a che fare con l'utente, comprese finestre, moduli, pagine web e così via.
- **Logica dell'applicazione:** include tutti i control object ed entity, realizzando l'elaborazione, il controllo delle regole e la notifica richiesti dall'applicazione.
- **Archiviazione:** realizza l'archiviazione, il recupero e l'interrogazione di oggetti persistenti.



FOUR-TIER

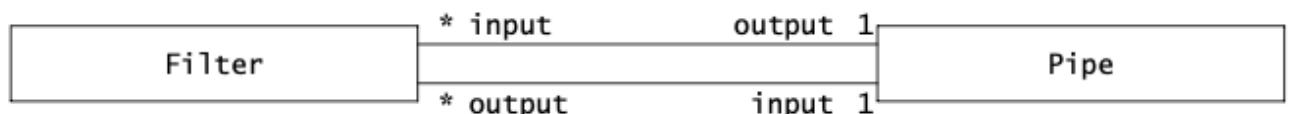
Lo stile architetturale four-tier è un'architettura a tre livelli in cui il livello di interfaccia è scomposto in un livello di *Presentation Client* e in un livello di *Presentation Server*.



Il livello di *Presentation Client* si trova sulle macchine degli utenti, mentre il livello di *Presentation Server* può essere posizionato su uno o più server.

PIPE AND FILTER

Nello stile architetturale pipe and filter, i sottosistemi elaborano i dati ricevuti da un insieme di input e inviano i risultati ad altri sottosistemi tramite un insieme di output. I sottosistemi sono chiamati "filter" e le associazioni tra i sottosistemi sono chiamate "pipe". Ogni filtro conosce solo il contenuto e il formato dei dati ricevuti sulle pipe di input, non i filtri che li hanno prodotti. Ogni filtro viene eseguito contemporaneamente e la sincronizzazione viene eseguita tramite le pipe. Lo stile architetturale del pipe and filter è modificabile: i filtri possono essere sostituiti con altri o riconfigurati per ottenere uno scopo diverso.



IDENTIFICAZIONE DESIGN GOALS

Molti design goals possono essere dedotti dai requisiti non funzionali o dal dominio applicativo. Altri dovranno essere sollecitati dal cliente.

IDENTIFICAZIONE DEI SOTTOSISTEMI

La scomposizione iniziale del sottosistema dovrebbe essere derivata dai requisiti funzionali. Inoltre, la scomposizione del sottosistema viene costantemente rivista ogni volta che vengono affrontati nuovi problemi: diversi sottosistemi vengono uniti in un sottosistema, un sottosistema complesso viene suddiviso in parti e alcuni sottosistemi vengono aggiunti per affrontare nuove funzionalità.

OBJECT DESIGN

Durante la fase di Object Design, colmiamo il divario tra gli oggetti dell'applicazione e i componenti standard identificando oggetti di soluzione aggiuntivi e perfezionando gli oggetti esistenti.

L'Object Design include:

- **Riutilizzo:** durante il quale identifichiamo componenti standard e modelli di progettazione per utilizzare le soluzioni esistenti
- **Specifiche dei servizi:** durante la quale descriviamo con precisione ciascuna interfaccia di classe
- **Ristrutturazione modello a oggetti:** durante la quale trasformiamo il modello Object Design per migliorare la sua comprensibilità ed estensibilità.
- **Ottimizzazione modello a oggetti:** durante la quale trasformiamo il modello Object Design per soddisfare criteri di prestazione come il tempo di risposta o l'utilizzo della memoria.

L'Object Design comprende quattro gruppi di attività

- **Riutilizzare:** i componenti standard identificati durante il System Design vengono utilizzati per aiutare nella realizzazione di ogni sottosistema. Spesso, i componenti e gli schemi di progettazione devono essere adattati prima di poter essere utilizzati. Questo viene fatto avvolgendo attorno ad essi oggetti personalizzati o perfezionandoli usando l'ereditarietà.
- **Specifiche dell'interfaccia:** durante questa attività, i servizi del sottosistema identificati durante il System Design vengono specificati in termini di interfacce di classe, incluse operazioni, argomenti, firme di tipo ed eccezioni. Vengono inoltre identificati ulteriori operazioni e oggetti necessari per trasferire dati tra sottosistemi.
- **Ristrutturazione:** le attività di ristrutturazione manipolano il modello di sistema per aumentare il riutilizzo del codice o soddisfare altri obiettivi di progettazione. Ogni attività di ristrutturazione può essere vista come una trasformazione del grafico su sottoinsiemi di un particolare modello. Le attività tipiche includono
 - la trasformazione di associazioni N-arie in associazioni binarie
 - l'implementazione di associazioni binarie come riferimenti
 - unione di due classi simili da due diversi sottosistemi in una singola classe
 - compressione di classi senza un comportamento significativo in attributi
- **Ottimizzazione:** le attività di ottimizzazione affrontano i requisiti di prestazione del modello di sistema. Ciò include
 - modifica degli algoritmi per rispondere ai requisiti di velocità o di memoria
 - riduzione delle molteplicità nelle associazioni per velocizzare le query
 - aggiunta di associazioni ridondanti per l'efficienza
 - riorganizzazione degli ordini di esecuzione

APPLICATION OBJECT E SOLUTION OBJECT

Gli **application objects**, chiamati anche "oggetti di dominio", rappresentano concetti del dominio rilevanti per il sistema. Durante l'analisi, identifichiamo gli entity objects e le loro relazioni, attributi e operazioni. La maggior parte degli entity objects sono application object indipendenti da qualsiasi sistema specifico.

I **solution objects** rappresentano componenti che non hanno una controparte nel dominio dell'applicazione, come archivi dati persistenti, oggetti dell'interfaccia utente o middleware. Durante l'analisi, identifichiamo anche i solution objects che sono visibili all'utente, come boundary e i control.

EREDITARIETÀ DELLE SPECIFICHE E EREDITARIETÀ DELL'IMPLEMENTAZIONE

Durante l'analisi, usiamo l'ereditarietà per classificare gli oggetti in tassonomie. Questo ci permette di differenziare il comportamento comune del caso generale. L'obiettivo della generalizzazione e della specializzazione è organizzare gli oggetti di analisi in una gerarchia comprensibile.

L'obiettivo dell'ereditarietà durante l'Object Design è ridurre la ridondanza e migliorare l'estensibilità.

Sebbene l'ereditarietà possa rendere un modello di analisi più comprensibile e un modello di progettazione a oggetti più modificabile o estensibile, questi vantaggi non si verificano automaticamente, spesso producono codice più offuscato e più fragile che se non avessero usato l'ereditarietà in primo luogo.

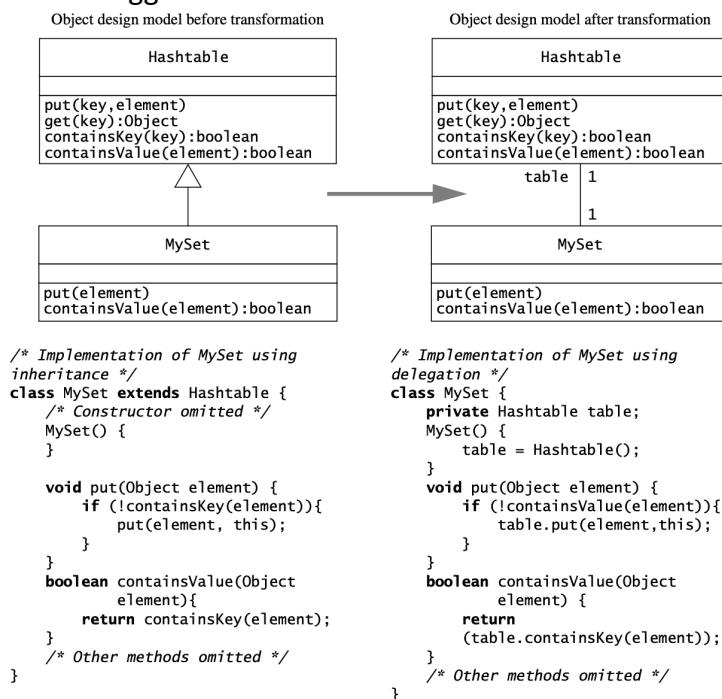
L'uso dell'ereditarietà al solo scopo di riutilizzare il codice è chiamato **ereditarietà dell'implementazione**.

Al contrario, la classificazione dei concetti in gerarchie di tipi è chiamata **ereditarietà delle specifiche**.

DELEGAZIONE

La delegazione è l'alternativa all'ereditarietà dell'implementazione che dovrebbe essere utilizzata quando si desidera il riutilizzo.

Si dice che una classe deleghi a un'altra classe se implementa un'operazione inviando nuovamente un messaggio a un'altra classe.



Questo risolve entrambi i problemi:

- **Estensibilità:** il MySet nella colonna di destra non include il metodo containsKey () nella sua interfaccia e la nuova tabella dei campi è privata. Quindi, possiamo cambiare la rappresentazione interna di MySet in un'altra classe senza influire sui client di MySet.
- **Sottotipazione:** MySet non eredita da Hashtable e, quindi, non può essere sostituito da Hashtable in nessuno dei codici client. Di conseguenza, qualsiasi codice che utilizzava precedentemente Hashtables si comporta ancora allo stesso modo.

DELEGAZIONE E EREDITARIETÀ IN DESIGN PATTERN

L'ereditarietà e la delegazione, utilizzate in diverse combinazioni, possono risolvere un'ampia gamma di problemi:

- disaccoppiare interfacce astratte dalla loro implementazione
- wrappare il codice legacy
- e/o disaccoppiare classi che specificano una politica dalle classi che forniscono il meccanismo

Nello sviluppo orientato agli oggetti, i **design pattern** sono template di soluzioni che gli sviluppatori hanno perfezionato nel tempo per risolvere una serie di problemi ricorrenti.

ATTIVITA DI RIUTILIZZO: SELEZIONE DI DESIGN PATTERN

Il System Design e l'Object Design introducono uno strano paradosso nel processo di sviluppo. Da un lato, durante il System Design, costruiamo solidi muri tra i sottosistemi per gestire la complessità suddividendo il sistema in parti più piccole e per impedire che le modifiche in un sottosistema influiscano su altri sottosistemi. D'altra parte, durante l'Object Design, vogliamo che il software sia modificabile ed estensibile per ridurre al minimo il costo delle modifiche future. Questo conflitto può essere risolto anticipando il cambiamento e progettando per esso, poiché le fonti dei cambiamenti successivi tendono ad essere le stesse per molti sistemi:

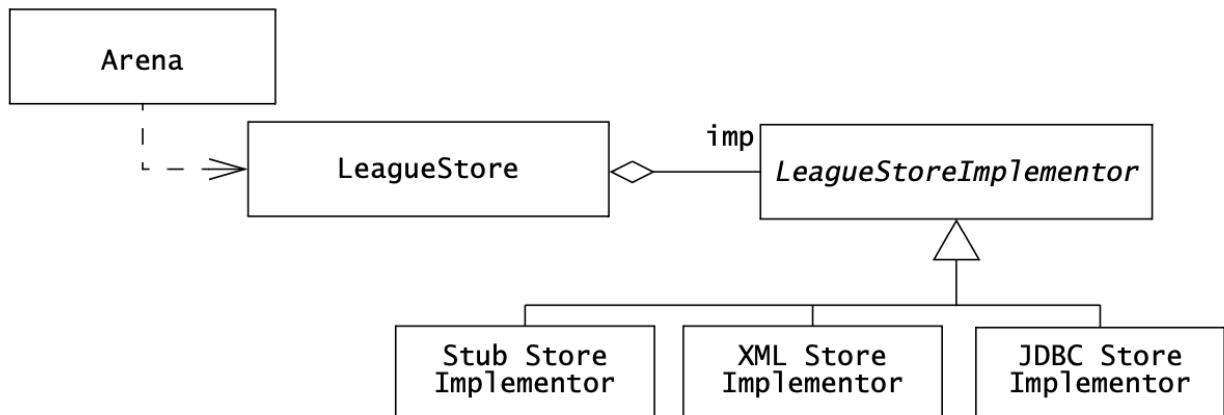
- **Nuovo fornitore o nuova tecnologia:** i componenti commerciali utilizzati per costruire il sistema sono spesso sostituiti da altri equivalenti di un fornitore diverso
- **Nuova implementazione:** quando i sottosistemi vengono integrati e testati insieme, il tempo di risposta complessivo del sistema è, il più delle volte, superiore ai requisiti di prestazione.
- **Nuova vista:** testare il software con utenti reali rivela molti problemi di usabilità. Questi si traducono spesso nella necessità di creare viste aggiuntive sugli stessi dati
- **Nuova complessità del dominio applicativo:** L'implementazione di un sistema innesca
 - idee di nuove generalizzazioni
 - Il dominio applicativo dello stesso potrebbe anche aumentare di complessità
- **Errori:** Molti errori dei requisiti vengono rilevati quando gli utenti reali iniziano a utilizzare il sistema

Design Pattern

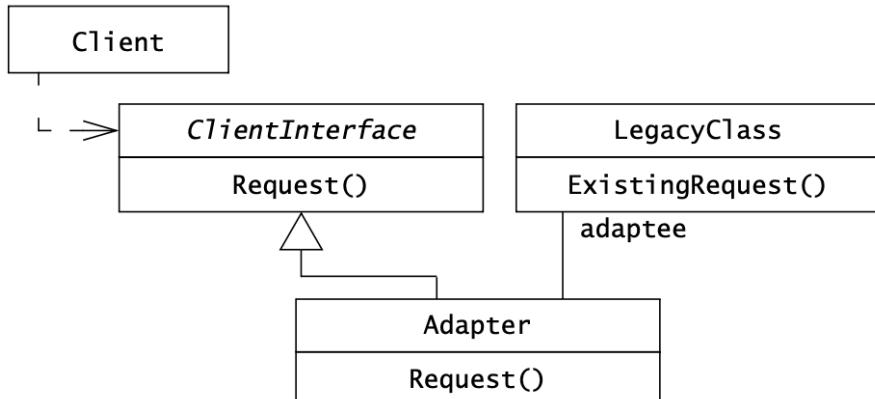
Design Pattern	Cambiamento previsto
Bridge	<i>Nuova tecnologia, nuovo fornitore, nuova implementazione.</i> Questo pattern disaccoppia l'interfaccia di una classe dalla sua implementazione. Ha lo stesso scopo del pattern Adapter tranne per il fatto che lo sviluppatore non è vincolato da un componente esistente.
Adapter	<i>Nuova tecnologia, nuovo fornitore, nuova implementazione.</i> Questo pattern incapsula un pezzo di codice legacy che non è stato progettato per funzionare con il sistema. Limita inoltre l'impatto della sostituzione del pezzo di codice legacy con un componente diverso.
Strategy	<i>Nuova tecnologia, nuovo fornitore, nuova implementazione.</i> Questo pattern disaccoppia un algoritmo dalle sue implementazioni. Ha lo stesso scopo dei pattern Adapter e Bridge, tranne per il fatto che l'unità incapsulata è un comportamento.
Abstract Factory	<i>Nuova tecnologia, nuovo fornitore.</i> Incapsula la creazione di famiglie di oggetti correlati. Ciò protegge il client dal processo di creazione e impedisce l'uso di oggetti di famiglie diverse (incompatibili).
Command	<i>Nuove funzionalità.</i> Questo pattern separa gli oggetti responsabili dell'elaborazione dei comandi dai comandi stessi. Questo pattern protegge questi oggetti dalle modifiche dovute a nuove funzionalità.
Composite	<i>Nuova complessità del dominio applicativo.</i> Questo modello incapsula le gerarchie fornendo una superclasse comune per i nodi aggregati e foglia. È possibile aggiungere nuovi tipi di foglie senza modificare il codice esistente.

INCAPSULAMENTO DI ARCHIVI DATI CON BRIDGE PATTERN

Considera il problema di sviluppare, testare e integrare in modo incrementale sottosistemi realizzati da diversi sviluppatori. I sottosistemi possono essere completati in tempi diversi, ritardando l'integrazione di tutti i sottosistemi fino al completamento dell'ultimo. Per evitare questo ritardo, i progetti utilizzano spesso un'implementazione **stub** al posto di un sottosistema specifico in modo che i test di integrazione possano iniziare anche prima che i sottosistemi siano completati.



INCAPSULAMENTO DI COMPONENTI LEGACY CON PATTERN ADAPTER

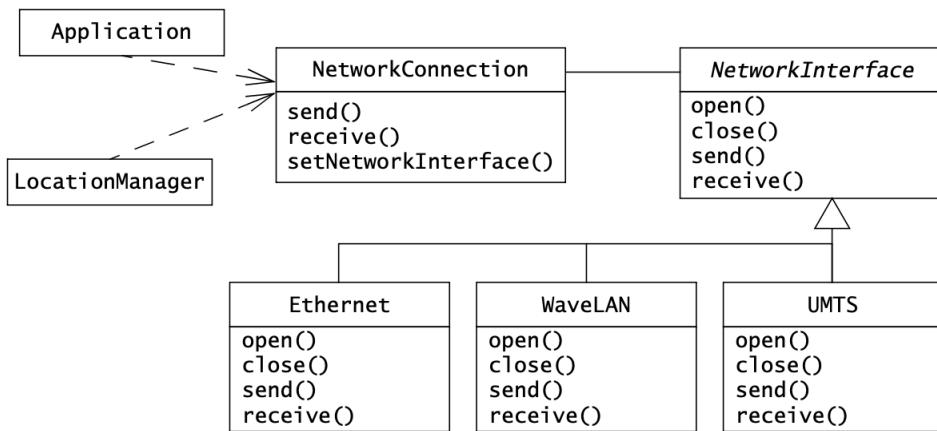


- **Classe client:** le classi client possono essere classi esistenti di una libreria di classi o nuove classi del sistema in fase di sviluppo.
- **Interfaccia pattern:** è la parte del pattern visibile alla classe client. Spesso, l'interfaccia pattern è realizzata da una classe astratta o un'interfaccia. Nel pattern Adapter, questa classe è chiamata **ClientInterface**.
- **Classe implementor:** fornisce il comportamento di livello inferiore del pattern. Nel pattern Adapter, **LegacyClass** e **Adapter** sono classi di implementazione.
- **Classe extender:** specializza una classe implementatore per fornire un'implementazione diversa o un comportamento esteso del pattern.

Il pattern Adapter incoraggia anche l'estensibilità, poiché la stessa classe **Adapter** può essere utilizzata per qualsiasi sottotipo della **LegacyClass**.

INCAPSULAMENTO DEL CONTESTO CON STRATEGY PATTERN

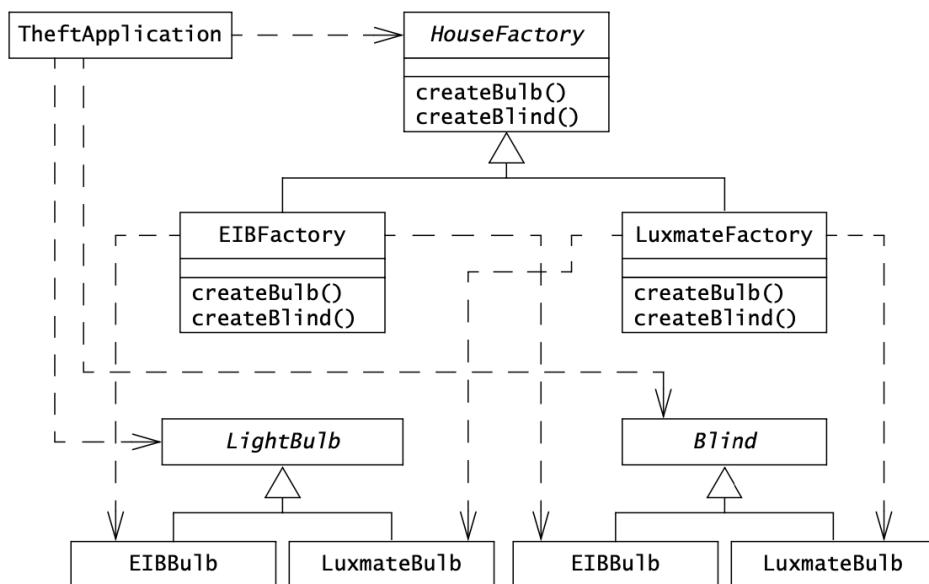
Supponiamo di avere un'applicazione mobile che deve gestire diversi tipi di reti in quanto passa da una rete all'altra in modo dinamico, in base a fattori quali posizione e costi di rete. Inoltre, vogliamo essere in grado di gestire i futuri protocolli di rete senza dover ricompilare l'applicazione. Per raggiungere entrambi questi obiettivi, applichiamo il pattern Strategy.



INCAPSULAMENTO DI PIATTAFORME CON ABSTRACT FACTORY PATTERN

Considera un'applicazione per una casa intelligente: l'applicazione riceve gli eventi dai sensori distribuiti in tutta la casa, sebbene diversi produttori forniscano l'hardware per creare tali applicazioni.

l'interoperabilità in questo dominio è attualmente scarsa, impedendo il mix and match di dispositivi di diversi produttori e, quindi, rendendo difficile lo sviluppo di un'unica soluzione software per tutti i produttori. Usiamo il design pattern Abstract Factory per risolvere questo problema.



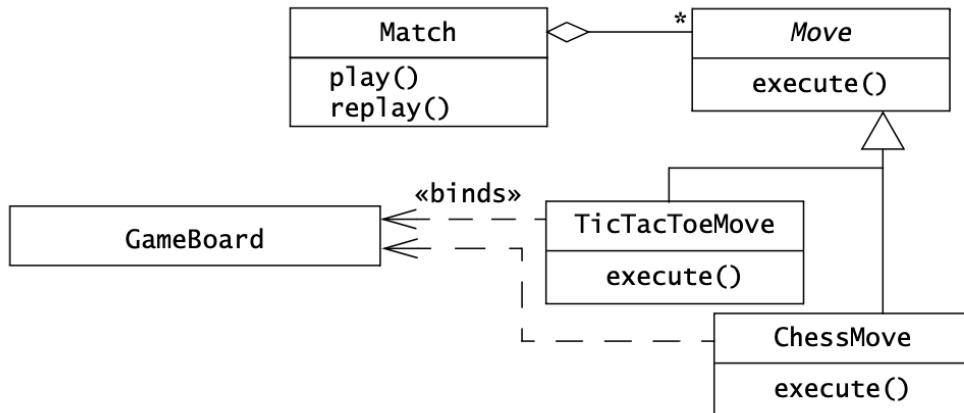
Questi oggetti generici sono chiamati *AbstractProducts* e le loro realizzazioni concrete sono chiamate *ConcreteProduct*.

Una fabbrica per ogni produttore fornisce metodi per la creazione dei *ConcreteProducts*.

Le classi Client accedono solo alle interfacce fornite da AbstractFactory e AbstractProducts, proteggendo così completamente le classi Client dal produttore dei prodotti.

INCAPSULAMENTO DEL FLUSSO DI CONTROLLO CON IL PATTERN COMMAND

Nei sistemi interattivi e nei sistemi di transazione, è spesso desiderabile eseguire, annullare o memorizzare le richieste degli utenti senza conoscere il contenuto della richiesta.

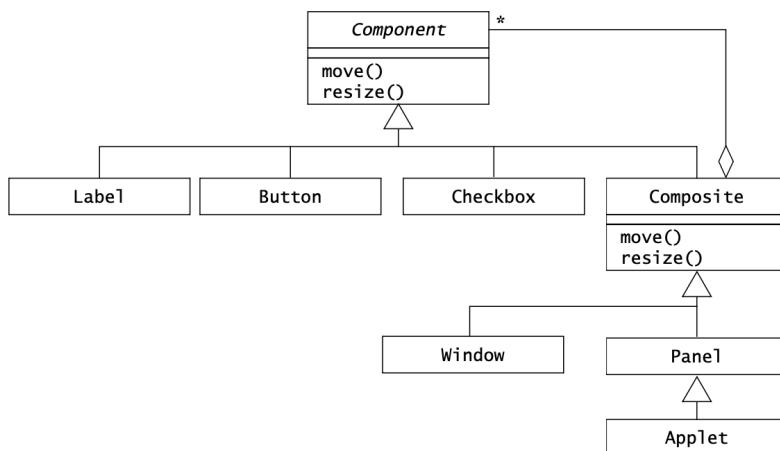


non vogliamo che le classi responsabili della registrazione e della riproduzione delle mosse dipendano da un gioco specifico.

Per risolvere il problema utilizziamo il pattern Command, la chiave per separare le mosse di gioco dalla loro manipolazione è rappresentare le mosse di gioco come oggetti comando che ereditano da una classe astratta chiamata Move.

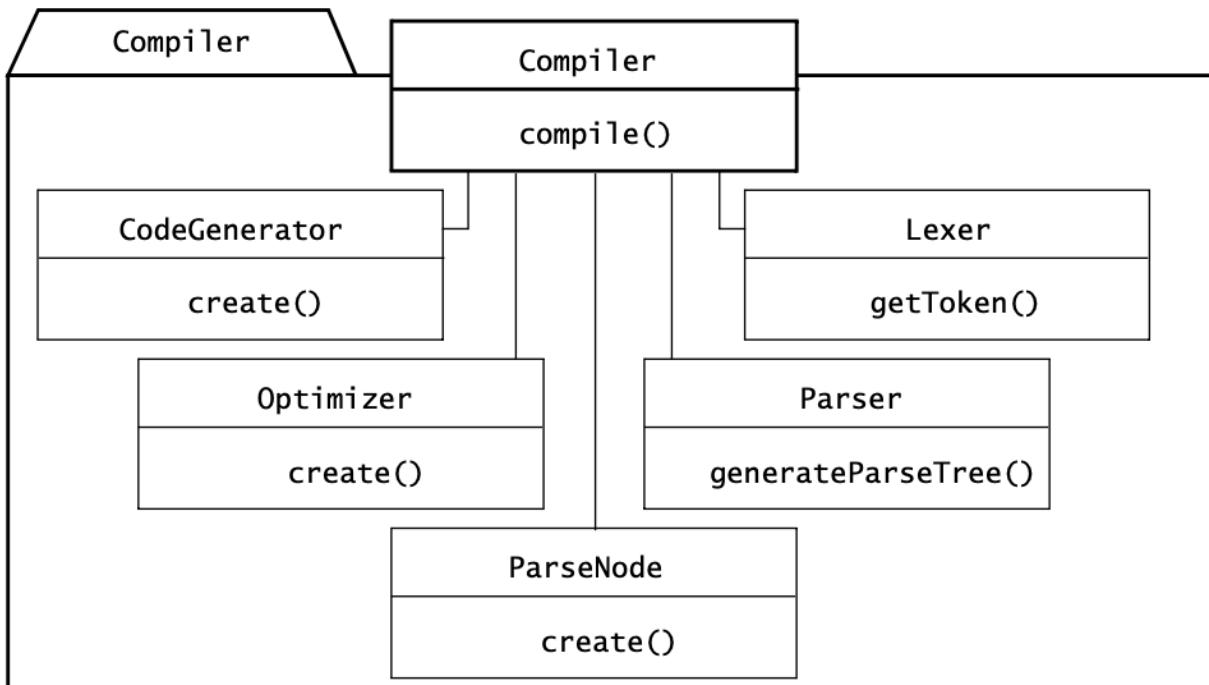
INCAPSULAMENTO DI GERARCHIE CON PATTERN COMPOSITE

I toolkit dell'interfaccia utente, come Swing, forniscono allo sviluppatore dell'applicazione una gamma di classi come elementi costitutivi. Ogni classe implementa un comportamento specializzato. Il design dell'interfaccia utente può aggregare questi componenti in Windows per creare interfacce specifiche dell'applicazione.



INCAPSULAMENTO DI SOTTOSISTEMI CON IL DESIGN PATTERN FACADE

La scomposizione del sottosistema riduce la complessità del dominio della soluzione minimizzando l'accoppiamento tra i sottosistemi. Il design pattern Facade ci consente di ridurre ulteriormente le dipendenze tra le classi incapsulando un sottosistema con un'interfaccia semplice e unificata.



FRAMEWORK APPLICATIVI

Un framework applicativo è un'applicazione parziale riutilizzabile che può essere specializzata per produrre applicazioni personalizzate.

A differenza delle librerie di classi, i framework sono destinati a particolari tecnologie, come l'elaborazione dei dati.

Vantaggi

- Riusabilità: sfrutta la conoscenza del dominio applicativo e il precedente impegno di sviluppatori esperti per evitare la ricreazione e il rinnovo di soluzioni ricorrenti.
- Estensibilità: fornendo **metodi hook**, che vengono sovrascritti dall'applicazione per estendere il framework dell'applicazione. L'estensibilità del framework è essenziale per garantire la personalizzazione tempestiva di nuovi servizi e funzionalità dell'applicazione.

I metodi di hook disaccoppiano sistematicamente le interfacce e i comportamenti di un dominio dell'applicazione dalle variazioni richieste da un'applicazione in un particolare contesto.

I framework possono essere classificati in base alla loro posizione nel processo di sviluppo del software:

- **Framework delle infrastrutture**: mirano a semplificare il processo di sviluppo del software sistema vengono utilizzati internamente all'interno di un progetto software e di solito non vengono forniti a un client.

- **Framework middleware:** vengono utilizzati per integrare applicazioni e componenti distribuiti esistenti
- **Framework applicazioni Enterprise:** sono specifici dell'applicazione

I framework possono anche essere classificati in base alle tecniche utilizzate per estenderli:

- **Framework whitebox:** fare affidamento sull'ereditarietà e sull'associazione dinamica per l'estensibilità. Le funzionalità esistenti vengono estese creando sottoclassi delle classi base del framework e sovrascrivendo i metodi hook.
- **Framework blackbox:** supportare l'estensibilità definendo le interfacce per i componenti che possono essere collegati al framework. La funzionalità esistente viene riutilizzata definendo componenti conformi a una particolare interfaccia e integrando questi componenti con il framework utilizzando la delegazione

Whitebox vs Blackbox

- Whitebox: richiedono una conoscenza approfondita della struttura interna del framework. I framework Whitebox producono sistemi che sono strettamente collegati ai dettagli specifici delle gerarchie di ereditarietà del framework, e quindi i cambiamenti nel framework possono richiedere la ricompilazione dell'applicazione.
- Blackbox: sono più facili da usare rispetto ai framework Whitebox perché si basano sulla delegazione anziché sull'ereditarietà. Tuttavia, i framework blackbox sono più difficili da sviluppare perché richiedono la definizione di interfacce e hook che anticipano un'ampia gamma di potenziali casi d'uso.

Design pattern vs framework. La principale differenza tra framework e pattern è che,

- framework si concentrano sul riutilizzo di progetti, algoritmi e implementazioni concreti in un particolare linguaggio di programmazione
- design pattern si concentrano sul riutilizzo di design astratti e piccole raccolte di classi cooperanti

SPECIFICA DELLE INTERFACCE

Durante l'object design, identifichiamo e perfezioniamo gli oggetti della soluzione per realizzare i sottosistemi definiti durante la progettazione del sistema.

Durante questa attività

- specifichiamo le firme del tipo
- la visibilità di ciascuna delle operazioni
- descriviamo le condizioni in cui un'operazione può essere invocata
- e quelle in cui l'operazione solleva un'eccezione

L'obiettivo della specifica dell'interfaccia è che gli sviluppatori comunichino in modo chiaro e preciso sui dettagli di livello sempre più basso del sistema.

Introduciamo OCL (Object Constraint Language) come linguaggio per specificare invarianti, precondizioni e postcondizioni.

- *Il modello di analisi ad oggetti:* descrive l'entità, il boundary e gli oggetti control visibili all'utente. Il modello di analisi ad oggetti include attributi e operazioni per ogni oggetto.
- *La scomposizione del sottosistema:* descrive il modo in cui questi oggetti vengono partizionati in parti coesive realizzate da diversi team di sviluppatori.

- *La mappatura hardware/software*: identifica i componenti che compongono la macchina virtuale su cui costruiamo gli oggetti della soluzione. Ciò può includere classi e API definite da componenti esistenti
- *I casi d'uso limite*: descrivono, dal punto di vista dell'utente, casi amministrativi ed eccezionali gestiti dal sistema
- *I design pattern*: selezionati durante il riuso del object design descrivono modelli di object design parziali che affrontano problemi di progettazione specifici

L'obiettivo della specifica dell'interfaccia è descrivere l'interfaccia di ogni oggetto, a tal fine, la specifica dell'interfaccia include le seguenti attività:

- **Identificare gli attributi e le operazioni mancanti**: durante questa attività, esaminiamo ogni servizio del sottosistema e ogni oggetto di analisi. Identifichiamo le operazioni
- **Specificare visibilità e firme**: durante questa attività, decidiamo quali operazioni sono disponibili per altri oggetti e sottosistemi e quali vengono utilizzate solo all'interno di un sottosistema. Specifichiamo anche il tipo di ritorno di ciascuna operazione, nonché il numero e il tipo dei suoi parametri.
- **Specificare i contratti**: durante questa attività, descriviamo in termini di vincoli il comportamento delle operazioni fornite da ciascun oggetto. In particolare, per ogni operazione, descriviamo le condizioni che devono essere soddisfatte prima che l'operazione venga invocata e una specifica del risultato dopo che l'operazione ritorna

Ora che stiamo approfondendo i dettagli della progettazione e dell'implementazione degli oggetti, dobbiamo differenziare gli sviluppatori in base al loro punto di vista.

- **Class implementor**: è responsabile della realizzazione della classe in esame. I class implementor progettano le strutture dati interne e implementano il codice per ogni operazione pubblica
- **Class user**: invoca le operazioni fornite dalla classe in esame durante la realizzazione di un'altra classe, chiamata classe client.
- **Class extender**: sviluppa le specializzazioni della classe in esame. Come i class implementor, i class extender possono richiamare operazioni fornite dalla classe di interesse

CONTRATTI: INVARIANTI, PRECONDIZIONI E POSTCONDIZIONI

I contratti sono vincoli su una classe che consentono agli utenti della classe, agli implementatori e agli estensori di condividere le stesse ipotesi sulla classe.

I contratti includono tre tipi di vincoli:

- **Invariante:** è un predicato che è sempre vero per tutte le istanze di una classe. Gli invarianti sono vincoli associati a classi o interfacce
- **Precondizione:** è un predicato che deve essere vero prima che venga richiamata un'operazione. I presupposti sono associati a un'operazione specifica
- **Postcondizione:** è un predicato che deve essere vero dopo che un'operazione è stata invocata. Le postcondizioni sono associate a un'operazione specifica

La parola chiave **context** indica l'entità a cui si applica l'espressione.

Questa è seguita da una delle parole chiave:

- inv
- pre
- post

COLLEZIONI OCL: SETS, BAGS E SEQUENCES

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})

OCL fornisce molte operazioni per accedere alle collezioni.

I più usati sono:

- size: restituisce il numero di elementi nella raccolta
- 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.

Due operazioni aggiuntive sulle collezioni ci consentono di iterare sulle collezioni e testare le espressioni su ogni elemento:

- forAll (variabile | espressione): è True se expression è True per tutti gli elementi della collezione
- exists (variabile | espressione): è True se esiste almeno un elemento nella raccolta per cui espressione è True.

DOCUMENTARE L'OBJECT DESIGN

L'object Design è documentato nell'Object Design Document (ODD). Descrive i compromessi nella fase di object design fatti dagli sviluppatori, le linee guida che hanno seguito per le interfacce dei sottosistemi, la scomposizione dei sottosistemi in package e classi e le interfacce delle classi.

L'ODD viene utilizzato per scambiare informazioni sull'interfaccia tra i team e come riferimento durante i test.

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

MAPPARE IL MODELLO NEL CODICE

Una trasformazione mira a migliorare un aspetto del modello preservando tutte le sue altre proprietà.

Queste trasformazioni si verificano durante numerose attività di progettazione e implementazione di oggetti. Ci concentriamo in dettaglio sulle seguenti attività:

- Ottimizzazione: questa attività affronta i requisiti di prestazione del modello di sistema. Ciò include la riduzione della molteplicità delle associazioni per velocizzare le query
- Realizzare associazioni: durante questa attività, mappiamo le associazioni ai costrutti del codice sorgente, come riferimenti e raccolte di riferimenti
- Mappatura dei contratti alle eccezioni: durante questa attività, descriviamo il comportamento delle operazioni quando i contratti vengono interrotti
- Mappatura dei modelli di classe a uno schema di archiviazione: durante questa attività, mappiamo il class model a uno schema di archiviazione, come uno schema di database relazionale

TRASFORMAZIONE DEL MODELLO

Una **trasformazione del modello** viene applicata a un modello a oggetti e si traduce in un altro modello a oggetti.

Lo scopo della trasformazione del modello a oggetti è quello di semplificare o ottimizzare il modello originale, portandolo in maggiore conformità con tutti i requisiti della specifica. Una trasformazione può aggiungere, rimuovere o rinominare classi, operazioni, associazioni o attributi.

Esistono 4 tipi di trasformazioni:

- Refactoring: un refactoring è una trasformazione del codice sorgente che ne migliora la leggibilità o la modificabilità senza cambiare il comportamento del sistema. Le operazioni di trasformazione, onde evitare di intaccare le funzionalità del sistema ed introdurre errori, vengono fatte eseguendo piccoli passi incrementali intervallati da test.

Before refactoring	After refactoring
<pre>public class User { private String email; } public class Player extends User { public Player(String email) { this.email = email; //... } } public class LeagueOwner extends User { public LeagueOwner(String email) { this.email = email; //... } } public class Advertiser extends User { public Advertiser(String email) { this.email = email; //... } }</pre>	<pre>public class User { public User(String email) { this.email = email; } } public class Player extends User { public Player(String email) { super(email); //... } } public class LeagueOwner extends User { public LeagueOwner(String email) { super(email); //... } } public class Advertiser extends User { public Advertiser(String email) { super(email); //... } }</pre>

- Forward engineering: Il forward engineering viene applicato a una serie di elementi del modello, lo scopo del forward engineering è mantenere una forte corrispondenza tra il modello di object design e il codice e ridurre il numero di errori introdotti durante l'implementazione

Source code after transformation

```
public class User {  
    private String email;  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String value){  
        email = value;  
    }  
    public void notify(String msg) {  
        // ....  
    }  
    /* Other methods omitted */  
}  
  
public class LeagueOwner extends User {  
    private int maxNumLeagues;  
    public int getMaxNumLeagues() {  
        return maxNumLeagues;  
    }  
    public void setMaxNumLeagues  
        (int value) {  
        maxNumLeagues = value;  
    }  
    /* Other methods omitted */  
}
```

- Reverse engineering: il reverse engineering viene applicato a un insieme di elementi del codice sorgente e si traduce in un insieme di elementi del modello. Lo scopo di questo tipo di trasformazione è ricreare il modello per un sistema esistente, o perché il modello è stato

perso o non è mai stato creato, oppure perché non è più sincronizzato con il codice sorgente

TESTING

Il test è il processo di ricerca delle differenze tra il comportamento previsto specificato dai modelli di sistema e il comportamento osservato del sistema implementato.

L'**affidabilità** è una misura del successo con cui il comportamento osservato di un sistema è conforme alla specifica del suo comportamento.

L'**affidabilità del software** è la probabilità che un sistema software non provochi un guasto del sistema per un periodo di tempo specificato in condizioni specificate.

Una **failure** è qualsiasi deviazione del comportamento osservato dal comportamento specificato.

Un **errore** significa che il sistema si trova in uno stato tale che un'ulteriore elaborazione da parte del sistema porterà a un errore, che quindi fa deviare il sistema dal comportamento previsto.

Un **fault**, chiamato anche "bug", è la causa algoritmica di uno stato errato. L'obiettivo del test è massimizzare il numero di errori rilevati, il che consente quindi agli sviluppatori di correggerli e aumentare l'affidabilità del sistema.

Un **test case** è un insieme di input e risultati attesi che esercita un componente di test allo scopo di provocare guasti e rilevare guasti.

Uno **stub di test** è un'implementazione parziale dei componenti da cui dipende il componente testato.

Un **driver di test** è un'implementazione parziale di un componente che dipende dal componente di test.

Gli stub e i driver di test consentono ai componenti di essere isolati dal resto del sistema per il test.

Se nessuno dei test è stato in grado di falsificare il comportamento del sistema software rispetto ai requisiti, è pronto per la consegna. In altre parole, un sistema software viene rilasciato quando i tentativi di falsificazione (test) mostrano un certo livello di fiducia che il sistema software faccia ciò che dovrebbe fare.

Esistono molte tecniche per aumentare l'affidabilità di un sistema software:

- **Prevenzione degli errori:** tentano di rilevare i guasti in modo statico, ovvero senza fare affidamento sull'esecuzione di nessuno dei modelli di sistema, in particolare il modello di codice. Questa tecnica prova a prevenire l'inserimento di errori nel sistema prima che lo stesso venga rilasciato.
- **Rilevamento degli errori:** il debug e il test, sono rispettivamente esperimenti non controllati e controllati, utilizzati durante il processo di sviluppo per identificare stati errati e trovare i guasti prima di rilasciare il sistema. Le tecniche di rilevamento degli errori aiutano a trovare i guasti nei sistemi, ma non tentano di risolverli dai guasti causati.

- **Tolleranza agli errori:** presumono che un sistema possa essere rilasciato con errori e che gli errori di sistema possano essere risolti ripristinandoli in fase di esecuzione. In questi casi è possibile assegnare la stessa attività a più componenti che la eseguono contemporaneamente e alla fine confrontano il risultato ottenuto.

Una **revisione** è l'ispezione manuale di parti o di tutti gli aspetti del sistema senza eseguire effettivamente il sistema. Esistono due tipi di revisione:

- **Walkthrough:** lo sviluppatore presenta in modo informale l'API, il codice e la documentazione associata del componente al team di revisione. Il team di revisione commenta la mappatura dell'analisi e della progettazione degli oggetti al codice utilizzando casi d'uso e scenari della fase di analisi.
- **Inspection:** un'ispezione è simile al walkthrough 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** presuppone che gli errori possano essere trovati partendo da un errore non pianificato. Lo sviluppatore sposta il sistema attraverso una successione di stati, arrivando alla fine e identificando lo stato errato.

Una volta identificato questo stato, è necessario determinare il guasto algoritmico o meccanico che lo causa.

Esistono due tipi di debug:

- **Debug di correttezza:** l'obiettivo è trovare qualsiasi deviazione tra i requisiti funzionali osservati e quelli specificati.
- **Debug di performance:** risolve la deviazione tra i requisiti non funzionali osservati e quelli specificati,

Definiamo il **testing** come il tentativo sistematico di trovare i guasti in modo pianificato nel software implementato, questa definizione di test implica che un test di successo è un test che identifica i guasti.

Un'altra definizione di test spesso utilizzata è che "dimostra che i guasti non sono presenti", se usassimo sempre questa seconda definizione, tenderemmo a selezionare dati di test che hanno una bassa probabilità di causare il fallimento del programma.

La caratteristica di un buon modello di test è che contiene casi di test che identificano i guasti. I test dovrebbero includere un'ampia gamma di valori di input, inclusi input non validi e casi limite, altrimenti i guasti potrebbero non essere rilevati.

Sfortunatamente, un tale approccio richiede tempi di test estremamente lunghi anche per sistemi piccoli.

- **Test planning:** alloca le risorse e pianifica i test. Questa attività dovrebbe svolgersi all'inizio della fase di sviluppo in modo che il tempo e le competenze sufficienti siano dedicati al test
- **Usability testing:** cerca di trovare errori nella progettazione dell'interfaccia utente del sistema. Spesso i sistemi non riescono a realizzare lo scopo previsto semplicemente perché i loro utenti sono confusi dall'interfaccia utente e introducono involontariamente dati errati

- **Unit testing:** cerca di trovare errori negli oggetti partecipanti e / o nei sottosistemi rispetto ai casi d'uso dal modello del caso d'uso
- **Integration testing:** è l'attività di ricerca dei guasti testando i singoli componenti in combinazione
- **Structural testing:** è il culmine del test di integrazione che coinvolge tutti i componenti del sistema

I test di integrazione e i test strutturali sfruttano la conoscenza del SDD, utilizzando una strategia di integrazione descritta nel Test Plan.

- **System testing:** testa tutti i componenti insieme, visti come un unico sistema per identificare i guasti rispetto agli scenari del problem statement e i requisiti e gli obiettivi di progettazione identificati nell'analisi e nella fase di system design, rispettivamente:
 - **Functional testing:** verifica i requisiti del RAD e del manuale utente
 - **Performance testing:** verifica i requisiti non funzionali e gli obiettivi di progettazione aggiuntivi dall'SDD
 - **Acceptance testing e Installation testing:** verificano il sistema rispetto all'accordo di progetto e vengono eseguiti dal cliente, se necessario, con l'aiuto degli sviluppatori

TEST CASE

Un **test case** è un insieme di dati di input e risultati attesi che esercita un componente con lo scopo di causare guasti e rilevare guasti.

Un test case ha cinque attributi:

- Nome: univoco rispetto ai test da effettuare
- Posizione: 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
- Oracolo: 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 casi di test sono classificati in **test blackbox** e **test whitebox**, a seconda di quale aspetto del modello di sistema viene testato.

- **Blackbox testing:** si concentrano sul comportamento di input/output del componente. I test Blackbox non si occupano degli aspetti interni del componente, né del comportamento o della struttura dei componenti. La selezione dei test è basata sulla specifica ed è scalabile a più livelli.
- **Whitebox testing:** si concentrano sulla struttura interna del componente. La selezione dei test è basata sulla logica interna del programma non è scalabile, solo a livello di unità

TEST STUB E DRIVER

L'esecuzione di casi di test su singoli componenti o combinazioni di componenti richiede che il componente testato sia isolato dal resto del sistema.

Un **test driver** simula la parte del sistema che chiama il componente da testare. Un driver di test passa gli input di test identificati nell'analisi del caso di test al componente e visualizza i risultati.

Uno **stub di test** simula un componente chiamato dal componente testato.

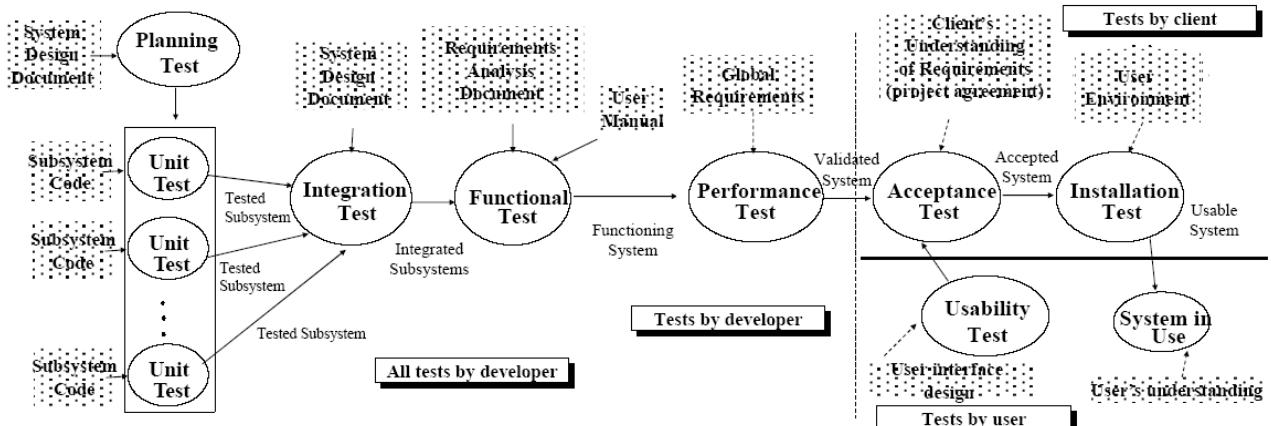
CORREZIONI

Una volta che i test sono stati eseguiti e gli errori sono stati rilevati, gli sviluppatori cambiano il componente per eliminare gli errori sospetti. Una **correzione** è una modifica a un componente il cui scopo è riparare un guasto.

In tutti i casi, la probabilità che lo sviluppatore introduca nuovi difetti nel componente revisionato è alta. Diverse tecniche possono essere utilizzate per ridurre al minimo il verificarsi di tali guasti:

- **Problem tracking:** include la documentazione di ogni guasto, stato errato e guasto rilevato, la sua correzione e le revisioni dei componenti coinvolti nella modifica.
- **Regression testing:** includono la riesecuzione di tutti i test precedenti dopo una modifica. Ciò garantisce che la funzionalità che funzionava prima della correzione non sia stata influenzata.
- **Rationale maintenance:** Include una documentazione che specifica i motivi di un cambiamento o le motivazioni dello sviluppo di una componente.

ATTIVITÀ DI TESTING



- **Component inspection:** le ispezioni rilevano i guasti in un componente esaminando il suo codice sorgente in una riunione formale. Le ispezioni possono essere condotte prima o dopo lo unit test. Un metodo proposto da *Fagan* è quello di suddividere il test in 5 parti:
 - **Overview:** l'autore del componente presenta brevemente lo scopo e l'ambito del componente e gli obiettivi dell'ispezione
 - **Preparation:** i revisori acquisiscono familiarità con l'implementazione del componente
 - **Inspection meeting:** un lettore parafrasa il codice sorgente del componente e il team di ispezione solleva problemi con il componente. Un moderatore tiene traccia della riunione.
 - **Rework:** l'autore rivede il componente
 - **Follow-up:** Il moderatore controlla la qualità del rework e può determinare il componente che deve essere nuovamente ispezionato
- **Usability testing:** Il test di usabilità verifica la comprensione del sistema da parte dell'utente, si concentra sulla ricerca delle differenze tra il sistema e le aspettative degli utenti su ciò che dovrebbe fare. La tecnica per condurre i test di usabilità si basa

sull'approccio classico per condurre un esperimento controllato. Esistono tre tipi di usability test:

- **Scenario test:** Viene presentato uno scenario agli utenti e viene valutato in quanto tempo gli utenti lo comprendono
 - Vantaggio: sono economici da realizzare e da ripetere
 - Svantaggio: l'utente non può interagire direttamente con il sistema e che i dati sono corretti
- **Prototype test:** Viene presentato un prototipo all'utente che rappresenta i punti chiave del sistema
 - Prototipo verticale: implementa completamente un caso d'uso attraverso il sistema, vengono utilizzati per valutare i requisiti fondamentali, ad esempio, il tempo di risposta del sistema o il comportamento dell'utente sotto stress
 - Prototipo orizzontale: implementa un singolo strato nel sistema; un esempio è un prototipo di interfaccia utente, che presenta un'interfaccia per la maggior parte dei casi d'uso
- **Product test:** simile al prototype test, tranne per il fatto che una versione funzionale del sistema viene utilizzata al posto del prototipo. È possibile eseguire un test del prodotto solo dopo che la maggior parte del sistema è stata sviluppata.
Richiede inoltre che il sistema sia facilmente modificabile in modo tale che i risultati del test di usabilità possano essere presi in considerazione
- **Unit testing:** si concentra sugli elementi costitutivi del sistema software, ovvero oggetti e sottosistemi. Ci sono tre motivazioni dietro la focalizzazione su questi elementi costitutivi
 - il test di unità riduce la complessità delle attività di test complessive, permettendoci di concentrarci su unità più piccole del sistema
 - il test di unità rende più facile individuare e correggere i guasti, dato che pochi componenti sono coinvolti nel test
 - il test di unità consente il parallelismo nelle attività di test; ovvero, ogni componente può essere testato indipendentemente dagli altri

Sono state ideate molte tecniche di unit test:

- **Equivalence testing:** questa tecnica di test riduce al minimo il numero di casi di test. I possibili input sono partizionati in classi di equivalenza e per ciascuna classe viene selezionato un test case. Per testare il comportamento associato a una classe di equivalenza, è necessario testare solo un membro della classe. Il test di equivalenza consiste in due fasi:
 - identificazione delle classi di equivalenza
 - selezione degli input di test
- I seguenti criteri vengono utilizzati per determinare le classi di equivalenza:
 - **Copertura:** ogni possibile input appartiene a una delle classi di equivalenza
 - **Disgiunzione:** nessun input appartiene a più di una classe di equivalenza
 - **Rappresentanza:** se l'esecuzione dimostra uno stato errato quando un particolare membro di una classe di equivalenza viene utilizzato come input, lo stesso stato errato può essere rilevato utilizzando qualsiasi altro membro della classe come input
- **Boundary testing:** si concentra sulle condizioni al confine delle classi di equivalenza. Piuttosto che selezionare qualsiasi elemento nella classe di equivalenza, il test di confine richiede che gli elementi siano selezionati al "limite" della classe di equivalenza. Il presupposto alla base del test di confine è che gli sviluppatori spesso trascurano casi speciali al confine delle classi di equivalenza

- **Path testing:** per la tecnica whithebox, viene costruito un diagramma di flusso del programma e si controlla se tutti i blocchi del diagramma vengono attraversati. Vengono individuati dei test case che possano essere attraversati tutti i blocchi del programma
- **Integration testing:** Il test di integrazione rileva i guasti che non sono stati rilevati durante il test di unità concentrandosi su piccoli gruppi di componenti. Due o più componenti vengono integrati e testati e, quando non vengono rilevati nuovi guasti, vengono aggiunti componenti aggiuntivi al gruppo. Questa procedura consente di testare parti del sistema sempre più complesse mantenendo la posizione dei potenziali guasti relativamente piccola.
 - **Horizontali integration testing:** i componenti sono integrati in strati, seguendo la scomposizione del sottosistema
 - **Big bang testing:** presuppone che tutti i componenti vengano prima testati individualmente e poi testati insieme come un unico sistema. Il vantaggio è che non sono necessari stub o driver di prova aggiuntivi
 - **Bottom-up testing:** testa prima ogni componente dello strato inferiore individualmente, quindi li integra con i componenti dello strato successivo. Questo viene ripetuto fino a quando tutti i componenti di tutti gli strati vengono combinati. I driver di test vengono utilizzati per simulare i componenti di livelli superiori che non sono stati ancora integrati. Si noti che non sono necessari stub test
 - **Top-down testing:** testa prima i componenti del livello superiore, quindi integra i componenti del livello inferiore successivo. Quando tutti i componenti del nuovo livello sono stati testati insieme, viene selezionato il livello successivo. Gli stub test vengono utilizzati per simulare i componenti dei livelli inferiori che non sono stati ancora integrati. Si noti che i driver test non sono necessari durante i test top-down
 - **Sandwich testing:** combina le strategie top-down e bottom-up, cercando di utilizzare il meglio di entrambe. Durante il test dei sandwich, il tester deve essere in grado di riformulare o mappare la decomposizione del sottosistema in tre strati
 - uno strato target
 - uno strato sopra lo stato target
 - uno stato sotto lo stato target
 Utilizzando il livello target come centro dell'attenzione, è ora possibile eseguire in parallelo test top-down e bottom-up.
 - **Modified sandwich testing:** testa i tre strati individualmente prima di combinarli in test incrementali l'uno con l'altro.
 - **Vertical integration testing:** si concentrano sull'integrazione precoce. Per un determinato caso d'uso, le parti necessarie di ciascun componente, come l'interfaccia utente, la logica di business, il middleware e lo storage, vengono identificate e sviluppate in parallelo e ne viene testata l'integrazione
- **System testing:** una volta che i componenti sono stati integrati, il test del sistema garantisce che l'intero sistema sia conforme ai requisiti funzionali e non funzionali. Durante il test del sistema, vengono eseguite diverse attività:
 - **Functional testing:** chiamato anche *requirements testing*, trova le differenze tra i requisiti funzionali e il sistema. Il test funzionale è una tecnica blackbox: i casi di test sono derivati dal modello del caso 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

- **Performance testing:** trova le differenze tra gli obiettivi di progettazione selezionati durante la progettazione del sistema e il sistema. Poiché gli obiettivi di progettazione derivano dai requisiti non funzionali, i casi di test possono essere derivati dall'SDD o dal RAD. Vengono eseguiti i seguenti test:
 - **Stress testing:** controlla se il sistema riesce a rispondere a molte richieste simultaneamente
 - **Volume testing:** cerca errori quando il sistema elabora grosse 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
- **Pilot testing:** chiamato anche *field test*, il sistema viene installato e utilizzato da un gruppo selezionato di utenti. Gli utenti esercitano il sistema come se fosse stato installato in modo permanente. Agli utenti non vengono fornite linee guida esplicite o scenari di test.
- **Acceptance testing:** ci sono tre modi in cui il cliente valuta un sistema durante i test di accettazione
 - **Benchmark test:** il cliente prepara una serie di casi di test che rappresentano le condizioni tipiche in cui il sistema dovrebbe funzionare
 - **Competitor testing:** il nuovo sistema viene testato rispetto a un sistema esistente o un prodotto della concorrenza
 - **Shadow testing:** il sistema nuovo e legacy vengono eseguiti in parallelo e i loro risultati vengono confrontati.
- **Installation testing:** Dopo che il sistema è stato accettato, viene installato nell'ambiente di destinazione. Il risultato desiderato del test di installazione è che il sistema installato soddisfi correttamente tutti i requisiti

DOCUMENTARE I TEST

Le attività di test sono documentate in quattro tipi di documenti:

- **Test plan:** si concentra sugli aspetti gestionali del testing. Documenta l'ambito, l'approccio, le risorse e il programma delle attività di test. I requisiti ei componenti da testare sono identificati in questo documento
- **Test case specification:** Questo documento contiene gli input, i driver, gli stub e gli output previsti dei test, nonché le attività da eseguire
- **Test incident report:** Vengono registrati i risultati effettivi dei test e le differenze rispetto all'output previsto
- **Test report summary:** elenca tutti i guasti rilevati durante i test che devono essere esaminati. Dal riepilogo del rapporto di prova, gli sviluppatori analizzano e assegnano la priorità a ogni guasto e pianificano le modifiche nel sistema e nei file