Appunti IS

Nel corso degli anni le metodologie di analisi del corso degli anni sono cambiate: inizialmente con un approccio di programmazione esclusivamente sequenziale il data flow diagram (diagramma di flusso dei dati) era basato su diagrammi ER (usando UML). Con la nascita dell'object oriented, è stato necessario introdurre un diagramma delle classi (che è comunque un diagramma relazionale). La progettazione di un class data non è la semplice traduzione delle classi, questo causa inconsistenza e ridondanza

Quello che in base di dati è definito **schema** (ovvero l'istanza del modello) è definito "**modello**"

Mentre quello che in base di dati è un **modello** in ingegneria del software è **metamodello**

Si definisce **ciclo di vita del software** il lasso di tempo che inizia con la creazione del software e termina con la decaduta del suo utilizzo. Un software nella sua vita presenta varie fasi:

- Concezione
- Requirement phase
- Fase di Design
- Fase di implementazione
- Fase di test
- Installazione
- Fase di checkout
- Fase di manutenzione
- Fase di ritiro (occasionalmente)

Un modello del ciclo di vita di un software rappresenta cosa è stato realizzato e cosa dovrebbe fare

- *"I modelli di processo software sono precise, formalizzate descrizioni di dettaglio delle attività, degli oggetti, delle trasformazioni e degli eventi che includono strategie per realizzare, ottenere l'evoluzione del software" (per **modelli di processo software** si intende **modelli di ciclo di vita del software**)*

Esistono vari modelli di cicli di vita del software:

- Waterfall
- Prototyping
- Incremental delivery
- Spiral model

Questi modelli nascono negli ultimi cinquanta anni per evitare un approccio bruteforce (**code&fix**) del software, dando uno standard per la realizzazione dei software da parte degli sviluppatori dell'epoca regolamentando e indirizzando lo sviluppo dei software verso un approccio più ponderato ed efficiente

Più generalmente le fasi di vita di un software sono:

- Definizione (si occupa di cosa)
    - Specifica i requisiti, le elaborazioni da effettuare, le prestazioni attese, vincoli, interfacce ecc
- Sviluppo (si occupa del come)
    - Definisce la memorizzazione e l'elaborazione dei dati a livello applicativo del software, la strutturazione e la realizzazione delle interfacce ecc
- Manutenzione (si occupa delle modifiche)
    - Eventualmente modifica per migliorare prestazioni, sicurezza, correzioni ecc

1 – Modelli a cascata

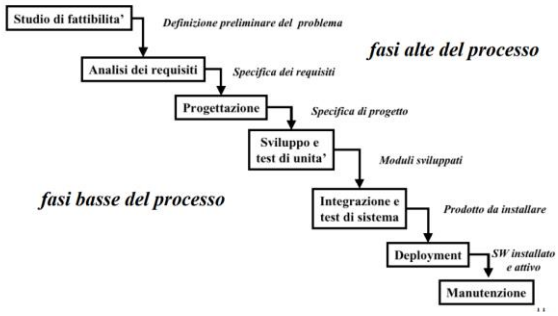
Ogni fase è caratterizzata da un insieme di attività (task) dai prodotti (deliverables) e i relativi controlli (quality control measures). Ogni fase presenta una **milestone** (obiettivo) alla fine di questa.

Ogni fase produce un output che sarà input alla fase successiva, questo implica che un prodotto di una fase è immutabile se non modificato specificatamente attraverso metodi formali

Per esempio, il modello a cascata per la costruzione di una casa:

- **Specifica Requisiti** - definizione di requisiti e vincoli del sistema (vorrei un casa su 2 piani, con autorimessa e cantina…)
- **Progetto di Sistema** - produrre un modello cartaceo (planimetrie, assonometrie, …)
- **Progetto di Dettaglio** - modelli dettagliati della parti (progetto dell' infrastruttura elettrica, idrica, calcoli statici travi...)
- **Costruzione** - realizzare il sistema
- **Test dei Componenti** - verifica le parti separatamente (impianto elettrico, idraulico, portata solai…)
- **Test di Integrazione** - integra le parti (il riscaldamento funziona...)
- **Test di Sistema** - verifica che il sistema rispetti le specifiche richieste
- **Installazione** - avvio e consegna del sistema ai clienti (ci vado a vivere)
- **Manutenzione** - (cambio la guarnizione al lavandino che perde …)

Viene detto a cascata per via della rappresentazione grafica del comportamento delle fasi:



Studio di fattibilità

- Valutazione di costi e benefici (umani, economici, sociali ecc)
    - Valutazione diversa tra produttore e committente del software

L'obiettivo di questo studio è stabilire con certezza se la realizzazione del software è effettivamente utile, gli output allora saranno:

- Definizione del problema
- Scenari – soluzioni alternative
- Costi, tempi e modalità di sviluppo di un'alternativa

Analisi dei requisiti

- Analisi dei bisogni dell'utente e dominio del problema
    - Coinvolgendo committente e sviluppatori del software
    - Discendendo da requisiti necessari e superflui

L'obiettivo è quello di descrivere le funzionalità e caratteristiche di qualità che il software dovrà avere, stabilendo quindi l'usabilità del software

L'output quindi sarà:

- Manuale utente
- Piano di acceptance test del sistema
    - Piano utilizzato che l'utente usa per collaudare il sistema

Progettazione

- Definizione di una struttura opportuna per il software
    - Scomponendo il sistema in moduli e componenti
        - Allocazione delle funzionalità ad ognuno di questi

Si distingue tra:

- Architectural design:
    - Struttura modulare complessiva
- Detailed design:
    - Dettagli interni a ciascuna componente

L'obiettivo è quindi definire il **come** del progetto

L'output è quindi:

- Documento di specifica del progetto

Fasi basse del processo

Test

- Verifica del corretto funzionamento del software
- Si distingue tra:
    - Alfa Testing
        - Test interni al produttore (il controllo è più diretto perché lo sviluppatore non ha bisogno di cercare di riprodurre l'errore)
    - Beta Testing
        - Sistema rilasciato a pochi selezionati
            - Che in teoria dovrebbero riportare gli errori presentati al produttore

Deployment

- Rilascio del software verso l'utenza

PRO

- Pioniere per la definizione di molti concetti
- Storicamente importante perché è il primo metamodello
- Facilmente comprensibile ed applicabile

CONTRO

- L'interazione con l'utente avviene solo all'inizio e alla fine
    - Spesso le richieste dell'utente sono imprecise o incomplete
- Il sistema realizzato è installabile solo quando ultimato

Vengono prodotte varianti del modello waterfall:

V&V(verification & validation) e Retroazione(feedback)

In aggiunta al waterfall troviamo:

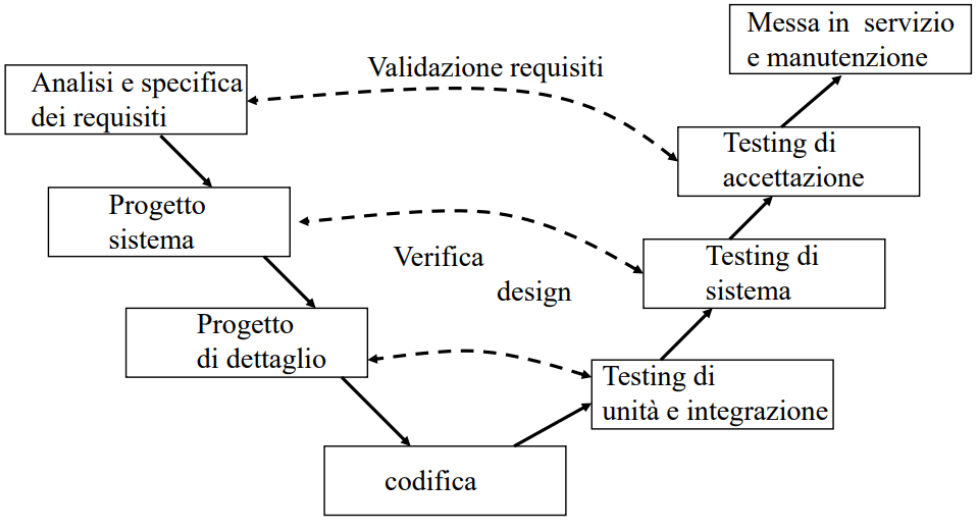
V&V:

- **Verifica**
  - Confrontare la corrispondenza tra la specifica e il prodotto software (e il suo output). Ovvero vengono confrontati la postcondizione della specifica con l'output del prodotto
- **Convalida**
  - Stabilisce quanto il prodotto software è adatto alla missione operativa, ovvero il comportamento se è in linea con quello voluto

Feedback:

- A livello di manutenzione ed operazione, i feedback possono essere inviati a qualsiasi delle fasi precedenti

Modello a V



- Le fasi a sinistra sono collegate a quelle di destra
- Ogni test eseguito a destra viene pianificato dalle rispettive fasi a sinistra, nel caso di errore in uno dei test, si rivedono le attività a partire dalla fase di sinistra a quella corrispondente

Trasformazioni formali

Affidabilità più alta data da una connessione molto più forte alla matematica, creando e sfruttando relazioni forti come quelle matematiche, è utile per "schermarsi" e schivare il maggior numero possibile di issue.

Viene con lo svantaggio di essere molto più complesso e meno user friendly

Modello incrementale

Nel modello incrementale, le implementazioni vengono fatte man mano, così quando un nuovo segmento (o incremento) dev'essere aggiunto non crea un ingorgo con l'architettura già esistente. L'architettura viene costruita in base agli incrementi quindi e non viene già precostruite come negli altri modelli, a livello di

realizzazione dev'essere necessario rendere possibile integrare nuovi incrementi ai sottosistemi già esistenti.

Il rilascio dev'essere necessariamente incrementale così da rilasciare il programma in costante evoluzione in base agli incrementali, a differenza dell'iterativo in cui il programma viene rilasciato per intero una volta finito (tempi di attesa lunghi in quanto *fullfilla* le richieste che arrivano come gli incrementali)

2 – UML

Modellare significa astrarre, ignorando dettagli irrilevanti e rappresentando esclusivamente quelli rilevanti; Anche in ambito software è necessario modellare poiché:

- I sistemi diventano costantemente più complessi,
- La leggibilità del codice è limitata

La modellazione non ha il compito di rappresentare l'**intero** sistema, bensì del sottosistema interessato o anche di singole componenti.

Si distinguono **modello** e **vista**:

- un **modello** è quindi l'astrazione di un sistema
- una **vista** è una rappresentazione di selezionati elementi di un modello (sottoinsieme del modello)

Per la rappresentazione di una vista si utilizza una **notazione**, ovvero un insieme di regole grafiche o testuali per la resa grafica del sistema o del sottoinsieme di questo. Per via della loro definizione, una vista ed un modello di un sistema possono essere **sovrapposti** (overlap). Questa gerarchia è rappresentabile come:

SISTEMA -----descritto da---→ MODELLI -----rappresentati in---→ VISTE

In UML:

- Istanza -> rappresentato sottolineando il nome
- Oggetto -> rappresentato con un rettangolo

Un'istanza non è altro che una rappresentazione di una specifica classe, per esempio: *airplane:System* è un'istanza (airplane) della classe (System)

Nelle rappresentazioni molti a molti (come nel caso tra modelli e viste), implementando la rappresentazione, sarà necessario introdurre una tabella intermedia tra i due oggetti.

L'astrazione formalmente si può definire come la classificazione di **fenomeni** e **concetti**:

- Un fenomeno formalmente è definito come un evento che coinvolge dei membri
- Un concetto invece è una tripla, composta da **nome**, **proprietà** e **membri**

È essenziale, per un'astrazione efficiente ed utile, distinguere tra dominio applicativo(UML) e di soluzione:

- **Dominio applicativo**
  - Analisi astratta, senza preoccuparsi di informazioni tecniche di implementazione (comprensibili anche all'utente)
- **Dominio di soluzione**
  - Nel dominio di soluzione sono contenute tutte le implementazioni, i tecnicismi, ecc. implementando però concetti del dominio applicativo (scendendo quindi di un grado di astrazione)

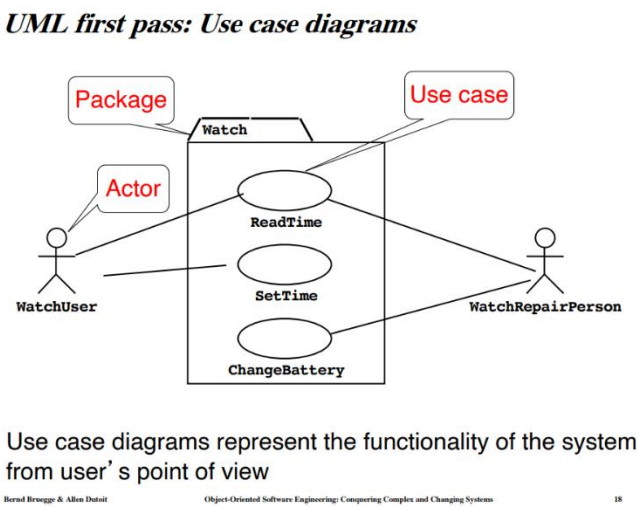
UML è un linguaggio di modellazione ormai diventato standard *de facto* composto dall'unione di 3 notazioni e concetti di metodi object-oriented: **OMT**(James Rumbaugh), **OOSE**(Ivar Jacobson), **Booch**(Grady Booch)

Il primo passo di un grafico UML è :

1. **Definizione dei diagrammi casi d'uso**, ovvero:
    a. la descrizione del **comportamento funzionale** del sistema **visto dall'utente**
2. **Definizione di diagrammi delle classi**, ovvero:
    a. Descrivono la struttura **statica** del sistema:
        i. Oggetti
        ii. Relazioni
        iii. Attributi
3. **Definizione dei diagrammi di sequenza**, ovvero:
    a. Descrivono il comportamento **dinamico** tra attori e sistema e tra oggetti e sistema
4. **Definizione dei diagrammi di stato**, ovvero:
    a. Descrive il comportamento **dinamico** di un **singolo oggetto**
5. **Definizione dei diagrammi di attività**
    a. Modella il comportamento dinamico del sistema, in particolare il workflow

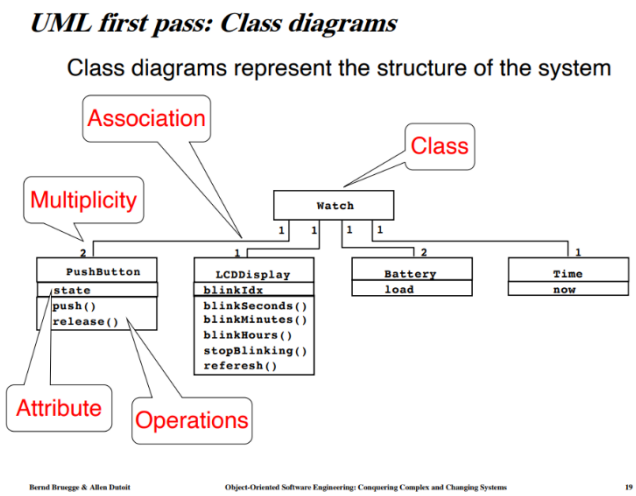
2.1 – Diagrammi casi d'uso

I diagrammi caso d'uso rappresentano le funzionalità del sistema dal punto di vista dell'utente



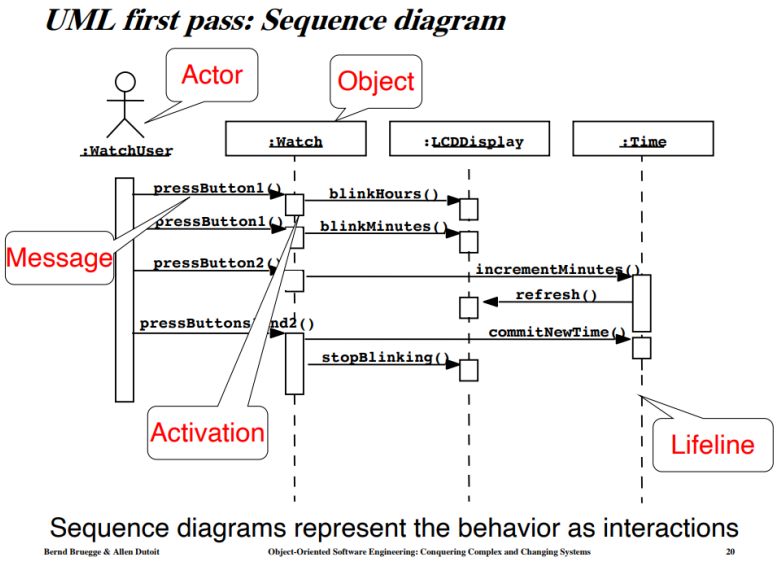
2.2 – Diagrammi delle classi

I diagrammi caso d'uso rappresentano la struttura del sistema, utilizzando classi, associazioni, attributi e oggetti



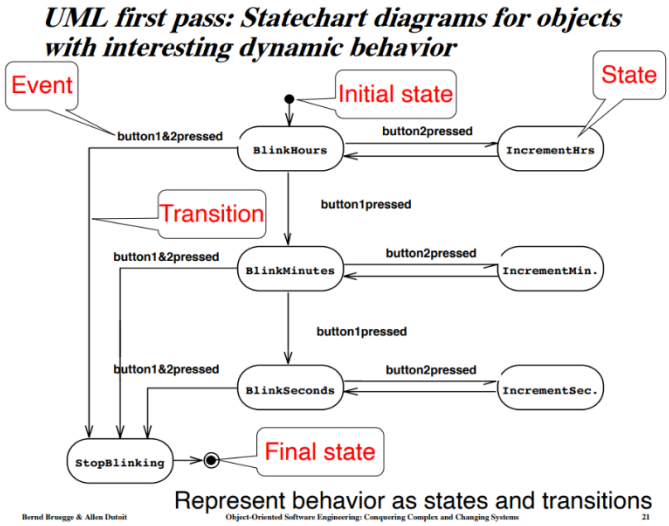
2.3 – Diagrammi delle di sequenza

I diagrammi caso d'uso rappresentano i comportamenti (di istanze dei casi d'uso) sottoforma di interazioni



2.4 – Diagrammi di stato

I diagrammi caso d'uso rappresentano i comportamenti come stati e transizioni



Convenzioni di UML:

- Classi ed istanze sono rappresentati come rettangoli
- Gli ovali sono funzioni o casi d'uso
- Le istanze sono rappresentate con il nome sottolineato
- I tipi sono rappresentati da nomi non sottolineati
- I diagrammi sono grafi

3 – Requirement Elicitation

Nella creazione di un sistema software, è possibile incombere in errori di interpretazione. Per questo il sistema di requirement elicitation serve per formalizzare l'idea del sistema software. Il suo **obiettivo** è la produzione del **Modello dei casi d'uso.** Generalmente le incomprensioni avvengono per via della complessità, questa può essere affrontata con tre tecniche:

- **Astrazione**
- **Decomposizione** (Divide et impera)
    - Dipendentemente dall'obiettivo del sistema si possono avere **due metodi** di decoposizione:
        - **Decomposizione funzionale** (può causare codice non manutenibile)
        - **Orientamento ad oggetti** (In base allo scopo è possibile trovare oggetti differenti, es. boundary objects = oggetti che riguardano l'interfaccia)
- **Gerarchia** (Layering)

1) Il primo passo è quindi la descrizione della funzionalità (Case Model).
2) Poi bisogna procedere trovando gli oggetti (object model), gli oggetti possono essere di vari tipi:
    - Di controllo
        - Favorendo il riuso
    - Bean

Il processo di requirement elicitation è necessario in quanto il case model è usato da tutte le altre fasi dello sviluppo di un software come base o come sostegno per la creazione dei modelli frutto di ogni fase.

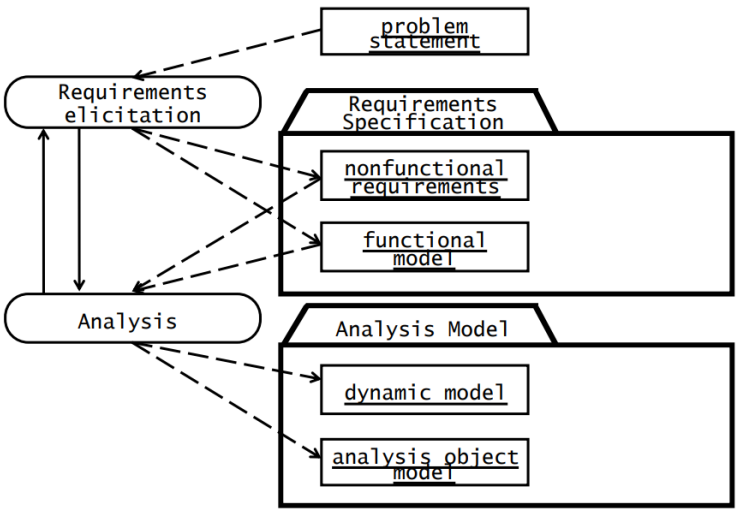
Nell'identificazione dei requisiti, è necessario identificare diversi punti di vista piuttosto che operare su solo uno snapshot, questo avviene ponendosi le seguenti due domande:

- Come possiamo identificare l'obiettivo di un sistema
- Cosa dev'essere incluso e cosa escluso dal progetto

La risposta a queste si può produrre tramite i seguenti processi:

- Requirements Elicitation
    - Raccolta dei requisiti compresi dal cliente
- Requirements Analysis
    - Specifiche tecniche dei requisiti compresi dallo sviluppatore

I prodotti di queste due attività sono:



Specifiche del sistema:

- È il prodotto del requirements elicitation
- è usato in linguaggio naturale

Modello analitico

- è prodotto del requirements analysis,
- questo documento è scritto in maniera tecnica e formale (o semiformale) con l'uso di grafi UML

Problem Statement

- È un documento fatto dal cliente che descrive i problemi a cui è indirizzato il sistema
- È detto pure **Statement of Work**
- Un buon Problem Statement include
    - Situazione attuale
    - Descrizione di uno o più scenari
    - Funzionalità che il sistema dovrebbe supportare
        - Funzionali
        - Non funzionali
        - Vincoli (o pseudo richieste)
    - Obiettivi (Milestones)
        - Milestones che involvono l'interazione con il cliente
        - Contiene la data di consegna del sistema
    - L'ambiente in cui è sviluppato il sistema
        - L'ambiente in cui il sistema consegnato deve superare un set preciso di test del sistema
    - Un set di criteri di test che il sistema deve superare

Validazione dei requisiti

Un altro importante step è la validazione dei requisiti, ovvero, il cliente deve confermare i requisiti proposti, questa valutazione avviene secondo i seguenti criteri:

- Correttezza
    - I requisiti rispecchiano la visione del cliente
- Completezza
    - Include tutti i possibili scenari in cui il sistema può trovarsi
- Consistenza
    - Non ci sono requisiti funzionali o non funzionali che si contraddicono
- Realismo
    - I requisiti possono essere implementati e consegnati
- Tracciabilità
    - Ogni funzione può essere tracciata su un corrispondente set di requisiti funzionali

Evoluzione dei requisiti

Durante la fase di requirements elicitation, i requisiti possono cambiare. Esistono tool per gestire i requisiti:

- RequisitPro from Rational

Tipi di Requirements Elicitation

Esistono 3 tipi:

- Greenfield Engineering
    - Lo sviluppo parte da zero, non esistono progetti preesistenti
    - Basato sulle richieste di un cliente
- Re-Engineering
    - Lo sviluppo fa un re-design o una re-implementazione di un sistema precedentemente esistente usando nuova tecnologia
    - Avviene spesso per passi in avanti nella tecnologia
- Interface Engineering
    - Lo sviluppo fornisce nuovi servizi ad un sistema già esistente in un nuovo ambiente
    - È guidato da esigenze di mercato o da una nuova tecnologia

Difficoltà Requirements Elicitation

Il Requirements Elicitation è un processo complicato, richiede la collaborazione con persone con background diversi, magari clienti con conoscenze riguardo il dominio applicativo, o sviluppatori che hanno conoscenze del dominio delle soluzioni (conoscenza di design o implementazione).

Per interfacciarsi ad un utente che magari non ha conoscenze a livello applicativo, è utile utilizzare:

- Questionari
    - Chiedendo all'utente una lista di domande preselezionate
    - Utilizzati per un'intervista strutturata (piuttosto che inviare il questionario aspettando la risposta)
- Analisi delle mansioni (dell'utente)
    - Analizzare l'ambiente di lavoro dell'utente (etnografia)
- Scenari
    - Esempi di uso del sistema in termini di interazioni tra l'utente e il sistema
    - È una descrizione testuale, di un evento o di una serie di azioni o eventi
- Caso d'uso
    - Astrazione che descrive le classi degli scenari

Scenari e casi d'uso sono molto utili per:

- Rendere comprensibile il sistema all'utente utilizzando esempi di vita comune così che possa riconoscerlo ed immaginarlo in maniera utile
- Utile anche per il progetto favorendo la creazione del manuale utente e di implementazioni comprensibili e riconoscibili all'utente

Gli scenari possono dividersi in:

- As-is scenario:
    - Usato nel descrivere situazioni attuali, spesso utilizzato nel re-engineering. L'utente descrive il sistema
- Visionary scenario:
    - Usato nel descrivere il futuro, permettendo all'utente di immaginare il sistema o feature, spesso usato in greenfield engineering o reengineering
- Evaluation scenario:
    - Compiti usati dall'utente per valutare il sistema
- Training scenario:
    - Istruzioni passo-passo che guidano l'utente all'uso del sistema

Formulare scenari

Nella formulazione degli scenari è utile porre determinate domande al cliente, come:

- *Quali task deve fare il sistema*
- *Quali dati verranno maneggiati da un utente al sistema*
- *Quali cambiamenti esterni possono avvenire al sistema*
- *Quali cambiamenti od eventi un utente dev'essere informato*

Negli scenari, quindi, non è importante parlare di casi ma bensì dimostrare e "raccontare" in che modo il sistema può essere utile o può ritrovarsi.

Questi, sono utili per comprendere esclusivamente le **funzionalità** (**requisiti funzionali**) che il sistema deve avere (*i casi d'uso sono estratti dagli scenari*)

Dagli scenari ai casi d'uso

Un caso d'uso è un flusso di eventi nel sistema che include un'interazione tra un attore e il sistema, questi vengono estratti uno ad uno prendendo una "fetta verticale" del sistema (uno scenario), e una "fetta orizzontale" per definire lo scope con l'utente. Utilizzare poi prototipi illustrativi per mostrare graficamente all'utente il sistema

Formulati gli scenari, quindi, bisogna estrarre (astraendo da uno scenario per volta):

- Nome dei casi d'uso
    - Esempio. "Report Emergency"
- Descrivere per ogni caso d'uso:
    - Attori partecipanti
    - Descrivere entry condition
        - Precondizioni necessarie per ogni caso d'uso
        - Stato del sistema prima del caso d'uso
    - Descrivere il flusso degli eventi
        - Descrizione dei passi di interazione tra utente e sistema
    - Descrivere exit condition
        - Condizione in cui si trova il sistema dopo il caso d'uso
    - Descrivere eccezioni
        - Flussi alternativi
    - Descrivere requisiti *non funzionali*

Flusso di eventi

Il flusso di eventi di un caso d'uso descrive in maniera testuale le interazioni tra gli attori nel caso d'uso, non utilizzando strutture o funzionamenti inutilmente complessi.

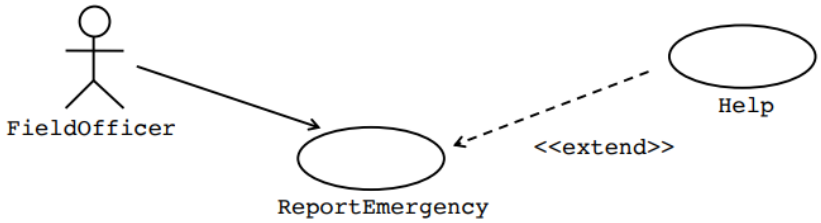
In questa descrizione è necessario nominare quando e chi memorizza informazioni nel database o l'attore che funzioni esegue

Diagrammi casi d'uso

Sono una rappresentazione grafica utile durante la requirement elicitation per mostrare le interazioni tra gli attori e casi d'uso. Gli attori sono rappresentati con uno stickman e i casi d'uso con un'ellisse, i casi d'uso sono caratterizzati dal flusso di eventi (precedentemente definito).

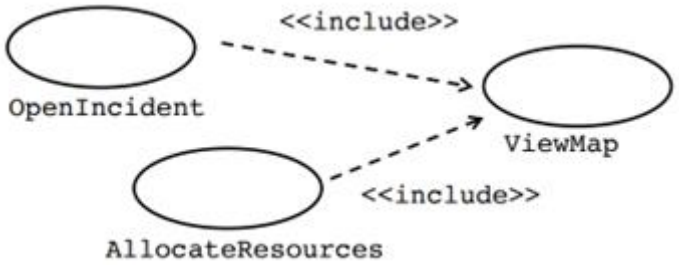
Extends

La relazione `Extends` (**stereotipo**) descrive una dipendenza di un caso d'uso secondario (alternativo/eccezione) ad un caso d'uso principale:



Include

Lo **stereotipo** `Include`, serve per riutilizzare casi d'uso comuni (che compongono altri casi d'uso), es:



Entrambi i casi d'uso *principali* (**`OpenIncident`**, **`AllocateResources`**) vanno verso il caso d'uso *secondario* **`ViewMap`** poiché entrambi utilizzano la suddetta procedura, quindi, **dipendono** da questa

FURPS+

I FURPS sono categorie di requisiti non funzionali:

- **Usability**
  - L'usabilità indica rispettare un requisito specificato dal cliente, secondo le sue convenzioni:
    - Interfaccia utente, il livello della documentazione livello utente, lo scope del supporto online
  - Da questa categoria di requisito dipendono la **learnability**
  - Es. Impedire ad un utente di inserire dati errati in un campo
  - **Si identificano:**
    - Qual è il livello di esperienza degli utenti
    - Quale interfaccia è più familiare agli utenti
    - Che documentazione dovrebbe essere fornita agli utenti
- **Reliability**
  - È l'abilità di un sistema di funzionare correttamente sotto determinate condizioni
    - Abilità di identificare errori specifici
    - Sopportare determinati attacchi alla sicurezza
    - Non crashare con input errati
  - Sostituito con **dependability**, ovvero la proprietà di un sistema di essere affidabile, robusto e sicuro
  - **Si identificano:**
    - Quando dev'essere robusto, disponibile e affidabile il sistema
    - Restartare il sistema nel caso di un fallimento è accettabile?
    - Quanti dati possono essere persi
    - Ci sono requisiti di sicurezza per il sistema?
- **Performance**
  - È uno dei requisiti non funzionali riguardanti l'efficienza intesa come:
    - Tempo di risposta
    - Disponibilità
    - Precisione
  - **Si identificano:**
    - Quando dev'essere responsive il sistema
    - Quali task sono da completare necessariamente in un lasso di tempo (time critical)
    - Quanti utenti concorrenti dovrebbero essere supportati
    - Quanto è grande il volume di dati che il sistema dovrà memorizzare
- **Supportability**
  - Riguarda l'aspetto della manutenibilità del sistema, o come può essere supportato durante l'esercizio
  - Riguarda l'aspetto anche della portabilità
  - **Si identificano:**
    - Quali sono le estensioni previste del sistema
    - Chi manutiene il sistema
    - Ci sono piani per fare porting ad un software od un hardware differente

Il + in FURPS+ indica gli **pseudo requirements:**

- **Implementation requirements**
  - Vincoli sulle tecnologie utilizzate nell'implementazione del sistema
- **Interface requirements**
  - Richieste di interfacce tipo user-friendly ecc
- **Operation requirements**
  - Richieste riguardanti la gestione del sistema in ambito operazionale
- **Packaging requirements**
  - Richieste su come viene consegnato il sistema (tipo file autoestraenti ecc)
- **Legal requirements**
  - Richieste riguardanti le licenze

Documento di analisi dei requisiti

Il documento è formato da un capitolo di introduzione in cui si definiscono:

- Scopo del progetto
- Ampiezza del progetto
- Obiettivi e criteri di successo del progetto
- Definizioni, acronimi e abbreviazioni
- Reference
- Overview

Poi un capitolo in cui viene descritto il sistema corrente

In seguito, un capitolo in cui si descrive il sistema proposto, composto da:

1. Overview
2. Requisiti funzionali
3. Requisiti non funzionali
   3.1. FURPS+
      - È possibile, vicino ad ogni requisito non funzionale, fare riferimento ai casi d'uso che fanno riferimento al suddetto requisito
4. Modello del sistema
   4.1. Scenari
   4.2. Modello dei casi d'uso
      ◊ Con anche diagramma dei casi d'uso
   4.3. Object Model
   4.4. Dynamic Model
   4.5. Modello navigazionale e mock-ups per rappresentare il sistema
5. Glossario

4 – Object Modeling

Data l'analisi dei casi d'uso, il prossimo passo è quello dell'object modeling, dato che i casi d'uso coinvolgono proprio gli oggetti.

Questi sono utili per astrarre dai dati e dalle implementazioni negli specifici casi d'uso. Generalmente un modello si dice "buono" se il modello è in grado di astrarre correttamente la realtà, creando le medesime relazioni nei modelli astratti.

Questi, ovviamente, sono sempre finzione e approssimazione. Infatti, nella formulazione di questi modelli bisogna attenersi alle conoscenze attuali, infatti, le fasi di testing sono necessarie proprio per individuare casi in cui il sistema fallisce (analoga al concetto di conoscenza oggettiva di Popper).

Perciò il compito di un modello è proprio quello di cambiare, affinché non rispecchi il più possibile le relazioni presenti nella realtà (pur utilizzando costrutti astratti)

La descrizione della conoscenza del sistema è data da:

- ◊ Dynamic model
  - State diagram
  - Activity diagram
  - Sequence diagram
- ◊ Object model
  - Class diagram
- ◊ Functional model
  - Scenarios/Use Case

L'object modeling è composto da:

1. Identificare le classi
2. Trovare gli attributi delle classi
   - Individuare gli attributi e distinguerli tra:
     - Specifici dell'applicazione
     - Attributi che sono considerate tali in un sistema, ma classi in un altro sistema
       - Rendere classi gli attributi
3. Trovare i metodi
   - Individuare le operazioni che possono essere:
     - Operazioni generiche (come get/set, ecc)
     - Operazioni di dominio, trovate in:
       - Dynamic model
       - Functional Model
4. Trovare le relazioni tra le classi
   - Le relazioni possono essere:
     - Aggregazioni (parte di una gerarchia)
       - Consiste nell'unire più entità sotto un'altra entità composta da quelle precedentemente nominate
     - Generalizzazioni (tipo di gerarchia)
       - Superclasse e sottoclasse

Inoltre:

- ◊ Obiettivo:
  - Ottenere l'astrazione desiderata

Gli oggetti si dividono in:

- ◊ Entity object
  - Informazione persistente tracciata dal sistema
- ◊ Boundary object (o interface object)
  - Interazione tra utente e sistema
- ◊ Control object
  - Rappresentano i task di controllo fatti dal sistema

Questi tipi sono originati dal design pattern MVC (model, view, controller)

Nella rappresentazione di questi oggetti in UML è possibile utilizzare gli *stereotipi* per specializzare gli oggetti, le relazioni, le classi ecc. in questo caso è possibile assegnare lo stereotipi <<entity>> agli entity object, <<control>> ai control object e <<boundary>> ai boundary object.

È anche possibile omettere gli stereotipi cambiando però il nome degli oggetti con un nome significativo, tipo: NomeOggetto_Boundary che specifica il tipo di oggetto

Tecnica di Abbott

Per identificare gli oggetti nei casi d'uso si utilizza la **tecnica di Abbott**, che segue queste regole:

- ◊ Nome proprio ➔ oggetto
- ◊ Nome comuni ➔ classe
- ◊ Verbo "di fare" ➔ metodo
- ◊ Verbo "di essere" ➔ specializzazione/ereditarietà
- ◊ Verbo "avere" ➔ aggregazione
- ◊ Verbo "di modo" ➔ vincolo
- ◊ Aggettivo ➔ attributo
- ◊ Verbo transitivo ➔ metodo
- ◊ Verbo intransitivo ➔ metodo (legati agli eventi)

Organizzazione basata su team

È una gestione dell'organizzazione gerarchica, organizzata per "progetti". Un progetto è un *sottogruppo* dell'azienda, un progetto a sua volta è diviso in più team.

Team = insieme di partecipanti che lavorano per la stessa attività o task,

Gruppo = insieme di persone che lavorano ad un task simile ma non hanno la necessità di comunicare per completare la loro parte del task

Committee = insieme di persone che fanno revisione del lavoro degli altri, o anche, persone che propone azioni o idee agli insiemi sopra

Le interazioni all'interno di un progetto sono quindi:

- ◊ Reporting
  - Interazione che partendo dallo sviluppatore "riporta" lo stato di avanzamento al capo progetto, o anche il capo progetto che riporta lo stato al manager del progetto
- ◊ Decision
  - Interazione che propaga delle decisioni, per esempio il team leader decide che lo sviluppatore deve pubblicare un API

La comunicazione in una struttura gerarchica può essere molto rallentata e filtrata dai dettagli, quindi rendendo scomodo e svantaggioso per esempio la comunicazione tra due sviluppatori, in quanto le informazioni di uno sviluppatore devono prima passare per il team leader, poi il project manager e poi scendere di nuovo al team leader dello sviluppatore destinatario e infine il destinatario, questo può portare a rimuovere troppi dettagli o a rilasciare informazioni "sensibili" o anche che devono restare a livello basso. La soluzione è permettere la comunicazione tra sviluppatori (peer-based) oppure permettere ai team leader di comunicare tra loro senza passare per il project manager (liason-based)

Ruoli

Il ruolo definisce un insieme di task e funzionalità che ci si aspetta svolgano un partecipante o un team

Esistono tipi diversi in un progetto software:

- Ruolo di management
    - ◊ Organizzazione ed esecuzione del progetto entro dei vincoli
- Ruolo di sviluppo
    - ◊ Specificare, costruzione ed ideazione dei sottosistemi, possono essere
        - Analista
        - Architetto di sistema
        - Object Designer
        - Implementatore
        - Tester
- Ruolo cross-functional (liaison)
    - ◊ Coordinano i vari team tra loro
- Ruolo di consulto
    - ◊ Offrono supporto temporaneo nelle aree dove i partecipanti al sistema con poca esperienza

Work Deconstruct

Task e Work Product

I task sono l'unità atomica (non decomponibile) di un lavoro. Questi definiscono i ruoli. Il prodotto di un task è un **work product**, questi sono soggetti a deadline e saranno usati come input per altri task.

Qualsiasi work product che viene rilasciato all'utente è chiamato **deliverables**

Work package

È la specifica del lavoro che dev'essere completato, include:

- Nome e descrizione del task
- Risorse necessarie per effettuare il task
- Dipendenze di input (work product di altri task) e output (work product prodotto dal task in questione)

Schedule

È il mapping dei task nel tempo, assegnando ad ogni task un tempo di inizio e fine.

Questi mapping possono essere fatti rappresentati tramite:

- **Diagrammi di Gantt**
  - È un grafico a barre che sull'asse orizzontale rappresenta il tempo e su quella verticale vengono disposte disposti tutti i task da fare (la lunghezza di questi task, quindi, indica quanto tempo richiedono)
- **Diagramma PERT**
  - È un diagramma di attività in cui gli archi rappresentano le dipendenze tra i vari task
  - I task vengono rappresentati come tabelle in cui sono specificati data di inizio e deadline
  - Si utilizza soprattutto per identificare i path critici (ovvero il path più lungo possibile attraverso il grafo, ovvero quella che ha più dipendenze)
  - La durata del progetto viene calcolata prendendo la end date dell'ultimo task appartenente al path critico, e sottraendola al primo task di questo path