



A.A. 2021-2022



INGEGNERIA DEL SOFTWARE

Università degli studi di Napoli

Federico II



Valentino Bocchetti



N86003405



vale.bocchetti@studenti.unina.it

Indice

1	Introduzione al corso	16
1.1	Informazioni sul corso	16
1.2	Comunicazione	16
1.2.1	Docente a Studente	16
1.2.2	Studente a Docente	16
1.3	Esame	17
1.3.1	Progetto	17
1.3.2	Valutazione del progetto	18
1.3.3	Scritto	18
1.3.4	Formazione dei gruppi	18
1.3.5	Cheating Policy	18
1.4	Materiale di studio	18
2	Lezione del 27-09	19
2.1	Ciclo di vita del SW	19
2.2	SW e costi	20
2.3	Dimensioni di un tipico SW	20
2.4	Sviluppo del SW	20
2.5	Ingegneria del SW	20
2.6	Il concetto di Qualità del SW	21
2.7	Qualità esterna e interna	21
2.8	Il concetto di Prodotto SW e Relativi Costi	22
2.8.1	Programmi VS Prodotti	22

2.9	Tipologie di Prodotti SW	22
2.10	Problemi della produzione SW - Costi	22
3	Lezione del 29-09	23
3.1	Costi per fix di problemi	23
3.2	Gli standard ISO per la qualità di un Software	23
3.2.1	ISO 9000	23
3.2.2	ISO 9001	23
3.2.3	ISO 25000:2005	24
3.3	Requirement Engineering	24
3.3.1	Ciclo di Vita del Software	24
3.3.2	Problemi nel processo di sviluppo del software	25
3.4	Requisiti	25
3.4.1	Requisiti Software - The Demons of Ambiguity	26
4	Lezione del 01-10	27
4.1	Raccolta dei requisiti	27
4.2	Tipologie di requisiti	27
4.2.1	Requisiti funzionali nel dettaglio	28
4.2.2	Requisiti non funzionali nel dettaglio	28
4.3	Analisi dei requisiti	31
4.4	Modelli di sistema	31
4.5	UML	31
4.6	Aspetti della modellazione	32
4.7	Considerazioni	32

4.8	Use Case Diagram	33
4.8.1	Gli attori	33
4.8.2	Gli Use Case	34
4.8.3	Identificare gli Use Case	34
5	Lezione del 04-10	34
5.1	Requisiti delle interfacce	34
5.2	Interazione Uomo-Macchina	35
5.3	Interface design process	35
5.4	Stati dell'User Interface Design	36
5.5	Conoscere l'utente	37
5.6	Definizione dei ruoli	37
5.7	Modelli dell'utente	38
5.8	Attenzione	39
5.9	La memoria	39
5.9.1	Memoria a breve termine	40
5.9.2	Memoria a lungo termine	40
5.10	Codifica	40
5.11	Recupero	40
5.12	Visione	41
6	Lezione del 06-10	41
6.1	Diagramma dei Casi d'uso	41
6.2	Descrizione testuale di un UCD	41
6.2.1	Template di A. Cockburn	42

6.3	Guida alla scrittura di Casi d'Uso	43
7	Lezione del 08-10	43
7.1	Modelli di dominio	43
7.2	Euristica three-object-type	43
8	Lezione del 11-10	44
8.1	Sistema motorio	44
8.2	Etonografia	44
8.3	8 golden rule - Shneiderman	44
9	Lezione del 13-10	45
9.1	Usabilità	45
9.2	Affordance e feedback	47
9.3	Nozione di usabilità	47
9.4	Apprendibilità e memorabilità	47
9.4.1	Accessibilità	49
10	Lezione del 15-10	49
10.1	Identificare gli oggetti	49
10.2	Identificare gli Oggetti Entity e l'euristica di Abbott	49
10.2.1	Vantaggi e svantaggi dell'Euristica di Abbot	50
10.2.2	Identificare gli Oggetti Boundary	50
11	Lezione del 18-10	50
11.1	L'ingegneria dell'usabilità	50
11.2	Modello iterativo	51

11.3	Allocazione delle risorse secondo il modello iterativo	51
11.4	Modello ISO 13407	52
11.5	Processo iterativo per la progettazione e sviluppo di un sito web	52
11.6	Ingegneria e Creatività	52
11.6.1	Mimesi	52
11.6.2	Ibridazione	52
11.6.3	Metafora	52
11.6.4	Variazione	53
11.7	Composizione di design pattern	53
12	Lezione del 20-10	53
12.1	UML Statecharts	53
12.2	Modelizzazione con stati e transizioni	53
12.3	Sintassi e Semantica degli Statechart	54
12.3.1	Regioni, vertici e transizioni	54
12.3.2	Transizioni	55
12.3.3	Esempio (una lampada con un singolo bottone)	56
12.3.4	Stati - attività interne	56
12.3.5	Composite States	56
12.3.6	Composite States - parallel regions	57
12.3.7	Shallow History Pseudostates	57
12.3.8	Deep History Pseudostates	58
12.3.9	Fork & Join Pseudostates	58
12.3.10	Gestire gli stati della UI con gli statechart	58

13 Lezione del 22-10	59
13.1 Presentazione progetto	59
14 Lezione del 25-10	59
14.1 Gli Statechart	59
14.2 Transizioni interne	59
14.3 Stati composti	59
14.4 Sottomacchine	59
14.5 Notazioni abbreviate	60
14.6 Sottostati concorrenti	61
14.7 I prototipi	62
14.7.1 Classificazione dei prototipi	62
14.7.2 Classificazione dei prototipi rispetto alla completezza funzionale	63
14.7.3 La tecnica del Mago di Oz	63
14.7.4 Prototipi wire-frame	63
14.7.5 Prototipi ipertestuali	63
14.7.6 Prototipi intermedi	63
14.7.7 Prototipi finali	64
15 Lezione del 27-10	64
15.1 Activity Diagrams	64
15.1.1 Elementi Grafici	65
15.2 Activity Diagram - Elementi	65
15.3 Combinazione di Fork e Join	66
15.4 Swimlanes	66

15.4.1	Esempio Swimlane	67
15.5	Progettazione di un sistema	67
15.6	Scopo del system design	67
15.7	Scopo dell'object design	68
15.8	Output del SW design	68
15.9	System Design	68
15.9.1	Attività di System design	68
15.9.2	Identificare gli obiettivi qualitativi del sistema	69
15.10	Criteri di design	69
15.11	Design Trade-offs	69
16	Lezione del 03-11	70
16.1	Concetti di base di buon design	70
16.1.1	Coesione	70
16.1.2	Accoppiamento	71
16.1.3	Tipi di accoppiamento in OO	71
16.1.4	Esempio	71
16.1.5	Esempio (cont.)	72
17	Lezione del 05-11	72
18	Lezione del 08-11	73
19	Lezione del 10-11	73
19.1	La legge di Demetra	73
19.1.1	Violazione della legge di Demetra	73

19.1.2	Eliminazione della coda della navigazione	74
19.2	SOLID Principles (Software Development is not a Jenga game)	74
19.2.1	Benefici del Single Responsibility Principle	76
19.3	Responsibility-driven design	76
19.3.1	CRC Cards	76
20	Lezione del 12-11	76
20.1	Organizzare sottosistemi in Architetture	76
20.2	Architettura a 3 livelli	77
20.3	Partizionamento del problema	77
21	Lezione del 15-11	77
21.1	L'errore umano	77
21.2	Prevenzione	77
21.3	Diagnosi	78
21.4	Correzione	78
21.4.1	Backward recovery	79
21.4.2	Forward recovery	79
22	Lezione del 17-11	79
22.1	User Navigation	79
22.1.1	Back navigation	79
22.1.2	Modifica del comportamento del tasto indietro del sistema	79
22.2	Hierarchical Navigation	80
22.2.1	Tipi di Hierarchical Navigation	80
22.2.2	Navigation drawer Activity Layout	80

22.2.3	Ancestral navigation (Up button)	80
22.2.4	Lateral Navigation	81
22.2.5	Step per implementare i tabs	81
23	Lezione del 19-11	81
24	Lezione del 22-11	81
24.1	Valutare l'usabilità	81
24.1.1	Valutazioni euristiche	81
24.2	Test di usabilità	82
24.2.1	Laboratorio dell'usabilità	83
24.2.2	Test di usabilità informale	84
24.2.3	Test formativi e sommativi	84
24.2.4	Test di compito e test di scenario	84
24.3	Misure	85
24.4	Come condurre un test di usabilità	85
24.5	Test di usabilità - costi e benefici	85
25	Lezione del 24-11	85
25.1	Cloud Computing - Modelli di servizio	85
25.2	Infrastruttura AWS	86
25.2.1	Computazione e salvataggio dei dati	86
26	Lezione del 26-11	86
26.1	Verifica e Validazione	86
26.1.1	I bug	86

26.1.2	Errori di progettazione	86
26.1.3	Errori di coding	87
26.1.4	Errori di valutazione sull'utilizzo del sistema	87
26.1.5	Qualche errore del passato...	87
26.2	Verifica e Validazione	87
27	Lezione del 29-11	87
27.1	I container - Approccio a micro servizi	88
27.2	Strategie per migliorare l'affidabilità del SW	88
27.3	Terminologia	88
27.3.1	Fault e Failure	88
27.3.2	Test e Casi di test	89
27.3.3	L'oracolo	89
27.3.4	Problemi indecidibile	89
27.3.5	Goal della verifica e della validazione	89
27.4	Verifica statica e dinamica	90
27.4.1	Verifica statica	90
27.4.2	Verifica Statica - Le review	90
27.4.3	Analisi informali	90
27.4.4	Verifica dinamica	90
27.5	Ispezioni	91
27.5.1	Ruoli	91
27.5.2	Fasi	91
27.6	Testing	91

27.6.1	Modello di sviluppo a V	92
28	Lezione del 01-12	92
28.1	Unit testing - junit	92
28.1.1	Vantaggi dell'Unit testing	92
28.2	JUnit	93
28.2.1	Componenti di un test JUnit	93
28.2.2	Asserzioni	93
28.2.3	assertEquals()	94
28.2.4	Test Suite	94
28.3	JUnit 5 - Arrange, Act, Assert	95
28.4	JUnit - Annotazioni	95
28.5	JUnit - Test con eccezioni	95
28.6	JUnit - Test parametrizzati	95
28.7	JUnit - Terminologia	96
29	Lezione del 03-12	96
29.1	Log4j	96
29.2	Logcat	96
29.3	Google Analytics - Firebase	96
29.4	RERAN - Timing and Touch-Sensitive Record and Replay for Android	97
29.5	Seminario con Engineering	97
29.5.1	Scenario - Cliente Interno	97
29.5.2	Censimento dei requisiti - JIRA	97
29.5.3	Lecture interessanti	97

29.5.4	Ingegneria del Software e Pubblica Amministrazione	97
30	Lezione del 06-12	98
30.1	Seminario con RE:Lab	98
31	Lezione del 10-12	98
31.1	Seminario con NTTData	98
31.1.1	Aree di mercato	98
31.1.2	Quality Assurance	99
31.1.3	Attività in ambito	99
31.1.4	Bugs hunting	99
31.1.5	Pianificazione e monitoraggio	99
31.1.6	Un uso pratico	99
31.1.7	Test di automazione	100
31.1.8	Competenze del Tester Automation	100
31.1.9	Linguaggio e framework	100
31.1.10	Test di performance	100
32	Lezione del 13-12	101
32.1	Strategie per il testing	101
32.2	Testing Black Box	101
32.3	Classi di equivalenza (Input Space Partitioning)	101
32.4	Equivalence Partitioning	102
32.4.1	Dominio di partizionamento	102
32.4.2	Identificare le Classi di equivalenza	102
32.4.3	Approcci all'identificazione delle partizioni	102

32.5 Definizione di Test Case	103
33 Lezione del 15-12	103
33.1 Testing dei valori limite (boundary values)	103
33.1.1 Idee di base	104
33.2 Classi di equivalenza dell'input della funzione F e suo Boundary Analysis Test Cases	104
33.3 Caso generale e Limitazioni	104
33.4 Worst Case Testing	104
33.5 Gerarchia	105
33.6 Testing Strutturale - White Box	105
33.6.1 Un modello di rappresentazione dei programmi	105
33.6.2 Costruzione del control flow graph per programmi strutturati	107
33.6.3 Semplificazione di un CFG	107
33.7 Tecniche di Testing Strutturale	107
33.8 Selezione di casi di test	107
33.9 Confronto tra White e Black box Testing	108
33.10 Integration Testing	108
33.10.1 Functional Testing	109
33.10.2 System Testing	109
34 Lezione del 17-12	110
34.1 Modelli di Processo	110
34.2 Modello più semplice di processo	110
34.3 Modello a Cascata	110
34.3.1 Organizzazione sequenziale delle fasi	111

34.3.2	Valutazione del modello a cascata	111
34.3.3	Vantaggi e svantaggi	111
34.4	Modelli Agili	112
34.5	SCRUM	112
34.5.1	Sprints	113
34.5.2	Scrum Framework	113
34.6	User profiling	114
34.7	Le Personas	114
34.7.1	Costruzione delle personas	114
34.7.2	Rappresentazione delle personas	114
35	Lezione del 20-12	115
35.1	Architetture chiuse e aperte	115
35.1.1	Vantaggi e Svantaggi	115
35.2	Architettura MVC - Model View Controller	115
35.2.1	Observer Design Pattern	116
35.2.2	Attori in gioco	116
35.2.3	MVC	116

1 Introduzione al corso

1.1 Informazioni sul corso

Organizzazione

40 lezioni circa:

- ▶ 4-5 lezioni per argomento;
- ▶ 3-4 seminari di aziende;
- ▶ Esercitazioni.

Orario

- ▶ Modulo A → Lunedì e Mercoledì
- ▶ Modulo B → Venerdì

1.2 Comunicazione

1.2.1 Docente a Studente

[Docenti Unina](#)

1.2.2 Studente a Docente

Nessuna chat TEAMS.

Struttura della MAIL:

- ▶ Solo quesiti brevi (per quelli articolati esiste il ricevimento);
- ▶ Mail → sergio.dimartino@unina.it e cutugno@unina.it;
- ▶ Soggetto → [INGSW] e poi l'oggetto;
- ▶ Firmare SEMPRE le MAIL;
- ▶ Non inviare mail per quesiti su aspetti già descritti nel sito istituzionale e/o nel materiale didattico;
- ▶ Le mail che non rispettano queste regole, non riceveranno risposta e saranno fonte di valutazione negativa.

La comunicazione **cliente/committente** è uno degli aspetti chiave dell'Ingegneria del SW.

1.3 Esame

Progetto obbligatorio di gruppo

- ▶ Analisi, Progettazione, Implementazione e Testing di un piccolo SW;
- ▶ Presentazione di gruppo ai docenti
 - ◊ Documentazione + **Demo/PowerPoint** max 20 min.

Scritto

- ▶ Esercizi e domande aperte sull'intero programma;
- ▶ Obbligatorio;
- ▶ Può essere sostenuto nei 12 mesi successivi alla consegna del progetto

Orale obbligatorio se il voto del progetto è superiore di almeno 4 punti a quello dell scritto.

Voto → Media di Progetto e Scritto (ed eventuale orale)

1.3.1 Progetto

Unica traccia, declinata differentemente per i vari gruppi.

Committente → Docenti:

- ▶ Saranno fornite specifiche incomplete e potenzialmente inconsistenti;
- ▶ Le specifiche vanno raffinate in incontri programmati e contingentati (per numero e durata).

Si dovranno produrre 3 documenti, oltre al codice:

- ▶ Analisi dei requisiti;
- ▶ Progettazione di sistema;
- ▶ Progettazione dei casi di test.

Andrà poi concordato con il docente una data per la presentazione.

Questa si articola in 3 fasi:

- ▶ Presentazione tecnica del progetto (Slides)

◇ 15 minuti per convincerci di aver sviluppato un prodotto di alta qualità **interna/esterna**.

- ▶ Demo dell'applicativo dopo la presentazione;
- ▶ Discussione del codice.

Vale il concetto di **Collective Ownership**.

1.3.2 Valutazione del progetto

Qualità della progettazione, presentazione e demo.

Il progetto dell'a. a. 2021-22 può essere consegnato entro il 30 Ottobre 2022.

Traccia facilitata per chi consegna entro la prima sessione.

1.3.3 Scritto

Può essere sostenuto solo dopo aver consegnato il progetto.

1.3.4 Formazione dei gruppi

Consistenza numerica → 2-3 persone:

- ▶ Esperienza di lavoro in team;
- ▶ Saranno ammessi gruppi singoli solo per motivati e documentati impedimenti (ad es lavorativi).

I gruppi:

- ▶ Vengono comunicati nel gruppo del corso;
- ▶ Operate dal docente in base alle disponibilità per studenti che non riescano a stabilire formazioni autonome;

Una variazione dei gruppi deve essere necessariamente comunicata con il docente.

1.3.5 Cheating Policy

Viene usato uno strumento automatico di Cheating Detection per il confronto di tutto ciò che viene consegnato.

In caso di 2 o più progetti ritenuti troppo simili, i progetti saranno annullati ad entrambi i gruppi e sarà data una nuova traccia, più estesa e complessa.

1.4 Materiale di studio

Sommerville. SW della Pearson.

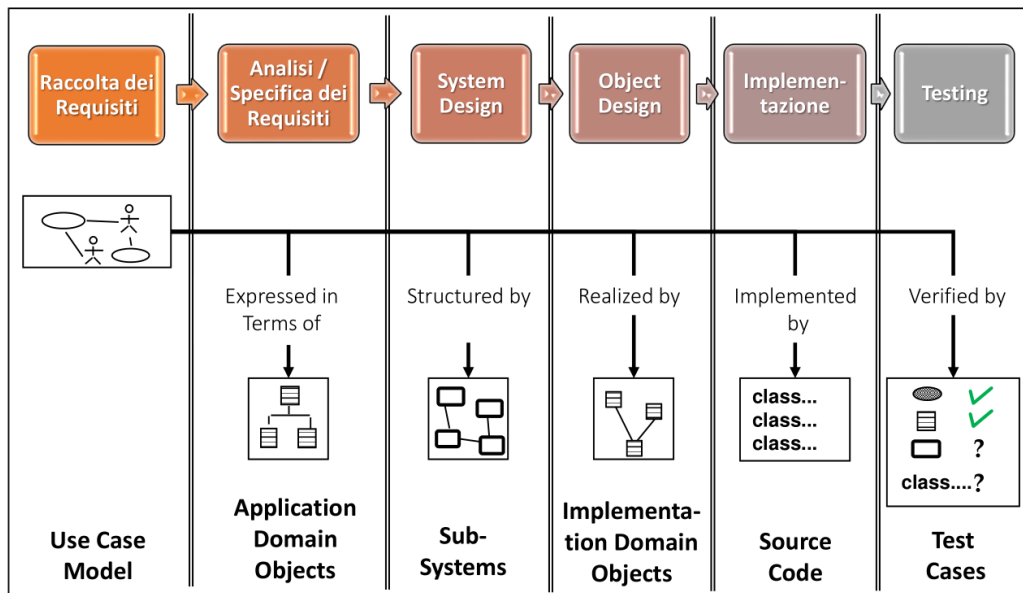
Facile da Usare di A. Polillo.

Consigliato un libro inerente a XML.

2 Lezione del 27-09

2.1 Ciclo di vita del SW

- ▶ Implementazione
 - ◊ Implementato da Source Code;
- ▶ Object Design
 - ◊ Realizzato da Implementation Domain Objects;
- ▶ System Design
 - ◊ Strutturato da Sub-System;
- ▶ Analisi/Specifica dei requisiti
 - ◊ Espressa in termini di Application Domain Objects;
- ▶ Raccolta dei requisiti
 - ◊ Use Case Model.



2.2 SW e costi

- ▶ Manutenzione → 76%;
- ▶ Sviluppo → 24%
 - ◊ Analisi, Progettazione, Coding → 50%;
 - ◊ Testing → 50%.

2.3 Dimensioni di un tipico SW

Dimensioni in termini di linee di codice

Sistema	Linee di codice
App Media per IOS	400.000
Firefox	31.101.855
Windows 10	50.000.000 ca
Vettura alto di gamma	100.000.000 ca

2.4 Sviluppo del SW

- ▶ Approccio Naive → non più gestibile in un contesto aziendale.

Produrre SW non è solo un'arte e neppure una scienza → è un'industria:

- ▶ Si lavora sempre in un contesto di gruppo e di azienda;
- ▶ Vincoli economici e requisiti di qualità.

Per produrre SW sono state sviluppate **metodologie** di progetto, di sviluppo e di verifica.

Tutto questo deve far parte del bagaglio culturale di un laureato in Informatica → Non basta essere i migliori programmatori in informatica.

Si deve sapere come ANALIZZARE, PROGETTARE e VALIDARE un SW nella sua Interezza.

2.5 Ingegneria del SW

Propone metodologie di sviluppo, che riassumono e formalizzano esperienze e conoscenze pregresse.

Molti fattori hanno storicamente limitato l'utilizzo di approcci ingegneristici di produzione.

Definizione

L'ingegneria del SW è la disciplina tecnologica e manageriale che riguarda la produzione sistematica e la manutenzione dei prodotti SW entro tempi e costi preventivati.

2.6 Il concetto di Qualità del SW

Qualità:

- ▶ Grado in cui un sistema, un componente, un processo soddisfa le richieste specificate;
- ▶ Il grado di questo sistema, dei componenti, dei processi soddisfa le richieste del cliente.

INGSW è tutto quello dedicato, direttamente e non, al tema della qualità del SW.

2.7 Qualità esterna e interna

I fattori rispetto a cui si può misurare la qualità del SW vengono classificati in:

- ▶ **Fattori interni** → qualità del SW percepita dagli sviluppatori;
- ▶ **Fattori esterni** → qualità del SW percepita dagli utenti.

Alcune qualità interne:

- ▶ Riparabilità;
- ▶ Manutenibilità;
- ▶ Riusabilità;
- ▶ Verificabilità

Tutte queste qualità fanno parte del Modulo A.

Alcune qualità esterne:

- ▶ Usabilità;
- ▶ Robustezza;
- ▶ Affidabilità

Tutte queste qualità fanno parte del Modulo B.

2.8 Il concetto di Prodotto SW e Relativi Costi

2.8.1 Programmi VS Prodotti

Programma → L'autore è anche l'utente.

Prodotto SW → usato da persone diverse da chi lo ha sviluppato:

- ▶ Molto più di un semplice eseguibile;
- ▶ Una batteria di test automatici;
- ▶ Dati di configurazione, che permettano di installarlo;
- ▶ Manuale utente;

Un prodotto SW è un prodotto industriale:

- ▶ Sviluppato con standard produttivi industriali;
- ▶ Prevede tutta la documentazione che descrive la progettazione e realizzazione del sistema.

2.9 Tipologie di Prodotti SW

- ▶ Prodotti General Purpose → Sistemi prodotti da una organizzazione e venduti a un mercato di massa;
- ▶ Prodotti specifici → sistemi commissionati da uno specifico utente e sviluppati specificatamente per questo da un qualche contraente

La fetta maggiore della spesa è nei prodotti generici ma il maggior sforzo di sviluppo è nei prodotto specifici.

La differenza principale è chi da le specifica del prodotto.

2.10 Problemi della produzione SW - Costi

Il SW ha costi elevati!

Il costo principale è quello degli stipendi degli sviluppatori:

- ▶ Quanto più velocemente i programmatori lavorano, tanto più la software house guadagna.

L'obiettivo di qualsiasi azienda informatica è quindi aumentare la produttività (si cerca quindi di ridurre i costi di programmazione).

Il grosso dei guadagni si fanno attraverso la manutenzione.

3 Lezione del 29-09

3.1 Costi per fix di problemi

I costi maggiori si anno in caso di manutenzione (>10.000).

In maniera minore avremo (dal meno costoso al più costoso):

1. Requirements (~100);
2. Design (~500);
3. Coding (~1000);
4. Testing (~8.0000).

3.2 Gli standard ISO per la qualità di un Software

3.2.1 ISO 9000

Richiede che qualsiasi azienda vada a documentare ciò che fa.

Fare solo ed esclusivamente quello che viene documentato (tenendone traccia e un controllo periodico di ciò che si tiene traccia).

3.2.2 ISO 9001

Uno degli standard più famosi. È il riferimento normativo globale per la certificazione del modello di gestione di qualunque **processo/prodotto** manifatturiero o servizio.

ISO 9126

Caratterizza la qualità del software. Nasce alla fine degli anni '90.

Presenta 3 macro famiglie:

- ▶ Qualità esterna → qualità percepita dagli utenti finali;
- ▶ Qualità interna → qualità del software percepita dagli sviluppatori
 - ◊ Funzionabilità;
 - ◊ Affidabilità;
 - ◊ Efficienza;
 - ◊ Usabilità;
 - ◊ Manutenibilità;
 - ◊ Portabilità;
- ▶ Qualità in uso.

3.2.3 ISO 25000:2005

Nota con l'acronimo **SQuaRE** (da System and Software Quality Requirements and Evaluation).

L'obiettivo è quello di creare un framework più centralizzato e ampio.

Si aggiunge alla **ISO 9126** il campo della sicurezza (che qui deve e ha un ruolo preponderante) e un ampliamento generale (in particolare sull'usability).

È chiaro che, per ovvi motivi, non sarà possibile massimizzare tutti i campi (il SW manager dovrà andare a effettuare una scelta su cosa massimizzare).

3.3 Requirement Engineering

3.3.1 Ciclo di Vita del Software

Processo di sviluppo software

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

Secondo la definizione dello Standard IEEE possiamo immaginare il processo come una serie di **evoluzioni** (modifiche) nel corso del tempo, che pian piano si avvicina al prodotto software finale.

I passaggi chiave sono:

1. La traduzione delle richieste del software da parte del cliente (Software requirement, a carico dell'ingegneria dei requisiti).
2. Trasformare il Software requirement in design (a carico dell'analista informatico);
3. Implementare il design in codice;
4. Testare il codice;
5. Eventualmente installare e verificare il software per uno use case

Fasi

Le prime fasi del ciclo di vita del software prende il nome di fase alta → **raccolta/analisi/specifica dei requisiti**:

- Analisi completa dei bisogni dell'utente e dominio del problema;
- Coinvolgimento di committente e ingegneri del SW;

Obiettivo → Descrivere le caratteristiche di qualità che l'applicazione deve soddisfare.

Output → documento di specifica dei documenti.

La fase successiva è una fase di basso livello → progettazione (**System/Object Design**):

- ▶ Definizione di una struttura opportuna per il SW;
- ▶ Scomposizione del sistema in componenti e moduli
 - ◊ Allocazione delle funzionalità ai vari moduli;
 - ◊ definizione delle relazioni fra i moduli;

Distinzione tra System (Struttura modulare complessiva) e Object Design (dettagli interni)

L'obiettivo è il **come**.

L'output è un documento di specifica di progetto → possibile l'uso di linguaggi per la progettazione (UML)

La fase più bassa invece è quella di **implementazione/testing/installazione/manutenzione**

Filosofia del DevOps → Testing e installazione diventano a carico del programmatore.

3.3.2 Problemi nel processo di sviluppo del software

È qualcosa di altamente intellettuale e creativo, basato su giudizi delle persone:

Non è possibile (almeno con i sistemi attuali) automatizzarlo

I requisiti sono complessi e ambigui:

Il cliente non sa bene, dall'inizio, cosa deve fare il software

I requisiti sono variabili:

Cambiamenti tecnologici, organizzativi, . . .

Modifiche frequenti sono difficili da gestire → difficile stimare i costi ed identificare cosa consegnare al cliente.

3.4 Requisiti

Condizione o capacità necessaria da un utente per risolvere un problema o raggiungere un obiettivo.

Ingegneria dei requisiti → Definire i requisiti del sistema di interesse

3.4.1 Requisiti Software - The Demons of Ambiguity

Insieme di tutte le cose che il software deve fare e dei vincoli operativi (in scenari con risorse HW molto variabili).

Ambiguità dei requisiti

Un requisito è semplicemente una formulazione astratta di un servizio che il sistema dovrebbe fornire, oppure un vincolo di sistema.

The most difficult part of building a software system is to decide, precisely, what must be built.

No other part of the work can undermine so badly the resulting software if not done correctly. No other part is so difficult to fix later.

L'ingegneria dei requisiti comprende tutte le attività che si occupano dei requisiti:

- ▶ Requirement Elicitation;
- ▶ Requirement Analysis;
- ▶ Requirement Specification;
- ▶ Requirement Validation.

Requirement Elicitation

È considerata l'attività più difficile, perché richiede la collaborazione tra più gruppi di partecipanti con differenti background.

Stakeholder → insieme di figure che hanno un interesse nei confronti di un'organizzazione e che con il loro comportamento possono influenzarne l'attività:

La Requirement Elicitation parte da un punto fondamentale → quali sono i **boundary** (confini) del Sistema da Sviluppare (System under Development / SuD)

Vengono specificati:

- ▶ Funzionalità del sistema;
- ▶ Interazione **utente/sistema**;
- ▶ Errori che il sistema deve individuare e gestire;
- ▶ Vincoli e condizioni di utilizzo del sistema.

Non fanno parte dell'attività di raccolta dei requisiti:

- ▶ Selezione delle tecnologie;
- ▶ Progetto del sistema e le metodologie da usare;
- ▶ In generale tutto quello che non è direttamente visibile all'utente.

4 Lezione del 01-10

MEMO Fondamentale la creazione di legami interni ad un team

4.1 Raccolta dei requisiti

Le 3 principali modalità di raccolta sono:

- ▶ Interviste
 - ◇ Chiuse → lo stakeholder risponde ad un insieme predefinito di domande
 - ◇ Aperte → dialogo a ruota libera;
 - ◇ Non sempre sono una tecnica efficace per raccogliere requisiti, poiché non sempre gli stakeholder non sono in **grado/vogliono** descrivere il loro dominio;
- ▶ Scenari
 - ◇ Descrizione narrativa di cosa succede durante un'elaborazione;
 - ◇ Descrizione concreta, focalizzata e informale di una singola caratteristica di un sistema utilizzata da un singolo attore;
 - ◇ Possono essere utilizzati in diversi modi durante il ciclo di vita del software;
 - ◇ Esempio di scenario → suddiviso in scenario principale di successo (nessun errore) e di X scenari alternativi (in caso di fallimento, come gestire il problema).
- ▶ Prototipi/Mock-up
 - ◇ Rappresentazione statica dell'interfaccia, intesa come prototipo per avere feedback dall'utente;
 - ◇ Realizzata con *wire-frame*;
 - ◇ Esempio con [balsamiq](#)

NB → I requisiti cambiano di continuo (si stima un cambiamento pari a circa il 25% dei requisiti iniziali).

4.2 Tipologie di requisiti

I requisiti SW possono essere classificati secondo 2 diversi punti di vista:

- ▶ Livello di dettaglio

- ◊ Requisiti utente → descrivono l'insieme di funzionalità richieste al sistema. Sono scritti in linguaggio naturale; il punto di vista è quello dell'utente;
- ◊ Requisiti di sistema → formulazione dettagliata, strutturata e testabile di servizi e vincoli. Scritti in linguaggio naturale, notazione semi-formali, linguaggio formali; il punto di vista è quello dello sviluppatore.
- ▶ Tipo di requisito rappresentato;
 - ◊ Requisiti funzionali → descrivono le funzionalità offerte dal sistema;
 - ◊ Requisiti non funzionali → descrivono i vincoli sui servizi offerti dal sistema, e sullo stesso processo di sviluppo;
 - ◊ Requisiti di dominio → riflettono vincoli generali del dominio applicativo, non sempre esplicitati.

NB Non esiste nessuna relazione 1 . . . 1 tra requisiti utente e di sistema → solitamente si ha una relazione del tipo 1 . . . * (Un requisito utente → molti requisiti di sistema).

GDPR (General Data Protection Regulation)

GDPR is a long list of regulations for the handling of consumer data.

The goal of this new legislation is to help align existing data protection protocols all while increasing the levels of protection for individuals. It's been in negotiation for over four years, but the actual regulations will come into effect starting May 25th, 2018.

4.2.1 Requisiti funzionali nel dettaglio

Descrivono le interazioni tra il sistema e il suo ambiente indipendentemente dalla sua implementazione (l'ambiente include l'utente e ogni altro sistema interno).

Qualità dei requisiti funzionali

I requisiti funzionali devono essere:

- ▶ Completi → devono indicare tutti i servizi richiesti dagli utenti;
- ▶ Coerenti → i requisiti non devono avere definizioni contraddittorie;

Per sistemi grossi è difficile ottenere requisiti completi e coerenti → i vari stakeholders hanno spesso esigenze molto diverse.

4.2.2 Requisiti non funzionali nel dettaglio

Descrivono gli aspetti del sistema che non sono direttamente legati al comportamento del sistema.

Includono una grande varietà di richieste che si riferiscono a diversi aspetti del sistema, dall'usabilità alle performance → Varie classificazioni (Summerville, FURPS, ...)

Il modello FURPS

Acronimo per la definizione dei requisiti. Classifica i Requisiti in:

- ▶ Funzionalità (unico requisito appartenente a quello funzionale);
- ▶ Usabilità → facilità per l'utente di imparare ad usare il sistema e capirne il funzionamento. Includono
 - ◊ Convenzioni adottate per le interfacce utenti;
 - ◊ Portata dell'Help in linea;
 - ◊ Livello della documentazione utente.
- ▶ Reliability (Affidabilità) → capacità di un sistema o di un componente di fornire la funzione richiesta sotto certe condizioni e per un periodo di tempo. Includono
 - ◊ Un accettabile tempo medio di fallimento;
 - ◊ Abilità di scoprire specifici difetti;
 - ◊ Sostenere specifici attacchi alla sicurezza;
- ▶ Performance → riguardano attributi quantificabili del sistema come
 - ◊ Tempo di risposta;
 - ◊ Throughput;
 - ◊ Disponibilità;
- ▶ Supportabilità → riguardano la semplicità di fare modifiche dopo il deployment. Includono
 - ◊ Adattabilità (abilità di cambiare il sistema per trattare concetti aggiuntivi del dominio di applicazione);
 - ◊ Manutenibilità (abilità di cambiare il sistema per trattare nuove tecnologie e per far fronte a difetti).

I requisiti non funzionali dovrebbero essere sempre verificabili:

- ▶ Goal (non verificabile) → il sistema deve essere facile da usare per controllo espressi, e deve minimizzare gli errori;
- ▶ Requisito non-funzionale (verificabile) → controllori esperti devono imparare a usare tutte le funzioni del sistema in **max.** 2 ore di apprendimento (definisco un'unità di misura in cui deve essere verificabile).

Quantificazione dei requisiti non-funzionali

Propriety	Measure
Speed	Processed transaction/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time
Reliability	Number of help frames Mean time to failure Probability of events causing failure Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Conflitti fra requisiti non funzionali

Molto spesso può capitare di avere richieste non conciliabili → si deve **sacrificare/scegliere** una delle 2 strade.

Validazione dei requisiti

Passo critico nel processo di sviluppo. I requisiti sono continuamente validati da cliente e utenti. La validazione dei requisiti richiede di controllare:

- ▶ **Correttezza** → una specifica è corretta se rappresenta accuratamente il sistema che il cliente richiede e che gli sviluppatori intendono sviluppare;
- ▶ **Completezza** → una specifica è completa se tutti i possibili scenari per il sistema sono descritti, incluso i comportamenti eccezionali;
- ▶ **Chiarezza** → una specifica è chiara se non è possibile interpretare la specifica in 2 modi diversi.
- ▶ **Realismo** → la specifica dei requisiti è realistica se può essere implementata tenendo conto dei vincoli;
- ▶ **Verificabilità** → la specifica dei requisiti è verificabile se, una volta che il sistema è stato costruito, test ripetuti possono essere delineati per dimostrare che il sistema soddisfa i requisiti;
- ▶ **Tracciabilità** → Ogni funzione del sistema può essere individuata e ricondotta al corrispondente requisito funzionale. Include anche l'abilità di tracciare le dipendenze tra i requisiti, le funzioni del sistema, gli artefatti, incluso componenti, classi, metodi e attributi di oggetti (È cruciale per lo sviluppo di test e per valutare i cambiamenti).

4.3 Analisi dei requisiti

Obiettivo → Formalizzare tutto l'insieme dei requisiti utente, trasformandoli in requisiti di sistema.

4.4 Modelli di sistema

L'obiettivo è di descrivere in maniera non ambigua il problema da trattare, in termini di:

- ▶ Categorie di utenti;
- ▶ Insieme di funzionalità;
- ▶ Insieme di informazioni da gestire.

Analizzare i documenti ottenuti dalla Elicitation al fine di:

1. Identificare gli attori → individuare le differenti classi di utenti che il sistema dovrà supportare;
2. Identificare gli Use Case → Derivare dagli scenari un insieme di use case che rappresentano in modo completo il sistema SW da sviluppare. Uno use case è un'astrazione che descrive una classe di scenari;
3. Dettagliare gli Use Case → Assicurarsi che le funzionalità del sistema siano completamente specificate, dettagliando ogni use case e descrivendo il comportamento del sistema in situazioni di errore, e condizione limite;
4. Identificare i requisiti non funzionali → tutti gli stakeholders si accordano sugli aspetti visibili all'utente finale, ma che non sono direttamente relati alle funzionalità del sistema
 - ▶ Vincoli sulle prestazioni;
 - ▶ Utilizzo delle risorse;
 - ▶ Sicurezza;
 - ▶ Qualità.

4.5 UML

Un linguaggio per specificare, visualizzare, e realizzare i prodotti di sistemi software, e anche per il business modeling. UML rappresenta una collezione di "best engineering practices" che si sono dimostrate utili nella progettazione di sistemi complessi e di grandi dimensioni.

- ▶ Insieme di linguaggi (e notazioni) universali, per rappresentare qualunque tipo di sistema (SW, HW, organizzativo, . . .);
- ▶ Serve a specificare le caratteristiche di un nuovo sistema, o a documentarne uno già esistente;
- ▶ È un linguaggio di modellazione e specifica di sistemi, basato sul paradigma OO;
- ▶ È uno strumento di comunicazione tra i diversi ruoli coinvolti nello sviluppo e nell'evoluzione dei sistema
 - ◇ UML svolge un'importantissima funzione di lingua franca nella comunità della progettazione e programmazione a oggetti;

- ◇ Gran parte della letteratura di settore usa UML per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico.

- ▶ È un linguaggio, non un metodo;
- ▶ Può (ed è) utilizzato da persone e gruppi che seguono approcci diversi;
- ▶ È indipendente dal linguaggio di programmazione utilizzato.

4.6 Aspetti della modellazione

- ▶ **Modello funzionale** → rappresenta il sistema dal punto di vista dell'utente. Ne descrive il suo comportamento così come esso è percepito all'esterno, prescindendo dal suo funzionamento interno. Serve nell'analisi dei requisiti;
- ▶ **Modello a oggetti** → rappresenta la struttura e sotto struttura del sistema utilizzando i concetti OO. Può essere utilizzato sia nella fase di analisi che di design, a diversi livelli di dettaglio (class, object e deployment diagram);
- ▶ **Modello dinamico** → rappresenta il comportamento degli oggetti del sistema, cioè le dinamiche delle loro interazioni. È strettamente legato al modello a OO
 - ◇ Sequence diagram, activity diagram e statechart diagram.

4.7 Considerazioni

For 80% of all software, only 20% of UML

Keep the process as simple as possible!

UML permette di descrivere un sistema SW a diversi livelli di astrazione, dal piano più svincolato dalle caratteristiche tecnologiche fino all'allocazione dei componenti SW nei diversi processori in una architettura distribuita.

UML è sufficientemente complesso per rispondere a tutte le necessità di modellazione, ma è opportuno *ritagliarlo* in base alle specifiche esigenze dei progettisti e dei progetti, utilizzando solo ciò che serve nello specifico contesto.

UML Core Conventions

- ▶ I rettangoli sono classi o istanze;
- ▶ Gli ovali sono funzioni o Use Case;
- ▶ Le istanze sono denotate dal nome sottolineato;
- ▶ I tipi sono denotati da nomi non sottolineati;
- ▶ I diagrammi sono dei grafi (i nodi sono entità, gli archi le relazioni).

4.8 Use Case Diagram

Ha lo scopo di rappresentare graficamente l'insieme dei requisiti funzionali di sistema (tipicamente in una pagina). Espressi in termini di:

- ▶ Attori → categorie di utenti (umani e non) del sistema;
- ▶ Use case → funzionalità offerte dal sistema.

Inteso come mezzo di comunicazione Fornitore - Cliente per definire le funzionalità del sistema → Va tenuto quanto più semplice possibile.

Non modella come funziona il sistema

Getta le basi per tutto il processo di sviluppo software.

4.8.1 Gli attori

Un attore modella un'entità esterna, dotata di comportamento, che interagisce con il sistema:

- ▶ Classe di utenti;
- ▶ Sistema esterno;
- ▶ Ambiente fisico.

Presenta un nome univoco ed una descrizione opzionale.

Un attore non corrisponde ad una persona fisica, ma ad un ruolo specifico.

4.8.2 Gli Use Case



Un caso d'uso è una funzionalità (viene espressa con un'ellisse) offerta dal sistema, che porta ad un risultato misurabile per un attore. I casi d'uso modellano i Requisiti Funzionali del SuD.

Un caso d'uso astrae tutti gli scenari per una data funzionalità → Uno scenario è un'istanza di un caso d'uso:

- ▶ Uno Use Case rappresenta, attraverso un flusso di eventi, una classe di funzionalità offerte dal sistema;
- ▶ Uno Use Case specifica le interazioni Attore-Sistema per una data funzionalità.

4.8.3 Identificare gli Use Case

The behaviors of a UseCase can be described by a set of Behaviors such as Interactions, Activities, and StateMachines, as well as by pre-conditions, post-conditions and natural language tex where appropriate.

Un caso d'uso è **SEMPRE** descritto dal punto di vista degli attori (sistema → black-box).

I casi d'uso catturano il comportamento esterno del sistema da sviluppare, senza dover specificare come tale comportamento viene realizzato.

Un caso d'uso è **SEMPRE** iniziato da un attore, quindi può interagire con altri attori. Opzionalmente si aggiungono anche Mock-up.

5 Lezione del 04-10

5.1 Requisiti delle interfacce

La progettazione di una interfaccia è fondamentale quando il cliente è probabilmente un utente non esperto. Il SW deve essere facile da usare per tutto → il sistema deve essere usabile (le curve di apprendimento devono essere quasi lineari).

Pertanto i 3 requisiti delle interfacce sono:

1. Cosa fa il programma? (obiettivo);
2. A chi è rivolto? (target);
3. Come si misura l'usabilità del programma?

Il concetto di facile da usare non è un concetto a posteriori → In tutte le fasi del ciclo di vita dell'interfaccia esistono delle strutture per verificarne l'usabilità generale dell'interfaccia.

5.2 Interazione Uomo-Macchina

È una materia interdisciplinare. Tocca infatti:

- Informatica;
- Ergonomia e Fattori Umani;
- Psicologia cognitiva;
- Psicologia sociale

Tocca prima di tutto il design.

5.3 Interface design process



5 Stages involved in User Interface Design!

Analysis on user needs

- ▶ Purpose of the application.
- ▶ Identify the targeted audience.
- ▶ Analysis on competitors product.
- ▶ Identify the challenges and problems.

- ▶ Create a simple and flexible interface.
- ▶ Use common UI elements.
- ▶ Build a wireframe with the features.
- ▶ Present the final copy of the design.

Interface Design Protocol

Design the Layout

- ▶ Attractive colour themes.
- ▶ Use the right fonts.
- ▶ Optimize images.
- ▶ Use the right function buttons.

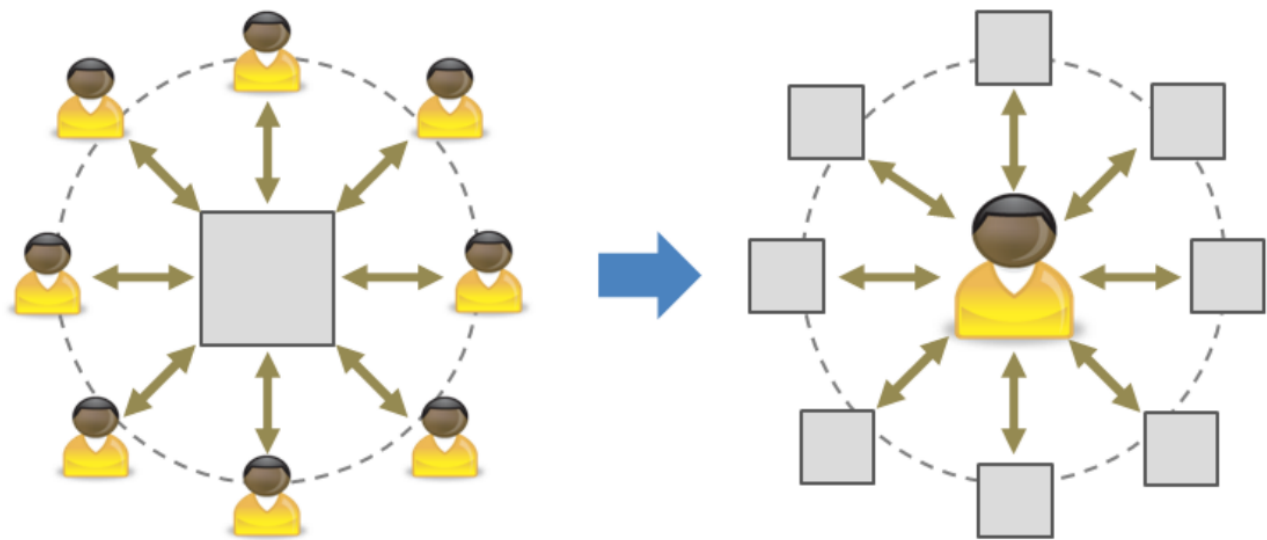
- ▶ Create flexible grid layout.
- ▶ Add responsive images.
- ▶ Organize navigation.
- ▶ Optimize content making it user friendly.

Responsive UI

Evaluate the Design

- ▶ Understand the user.
- ▶ Identify areas of mistakes.
- ▶ Suggest a solution.
- ▶ Encourage feedback.

5.5 Conoscere l'utente



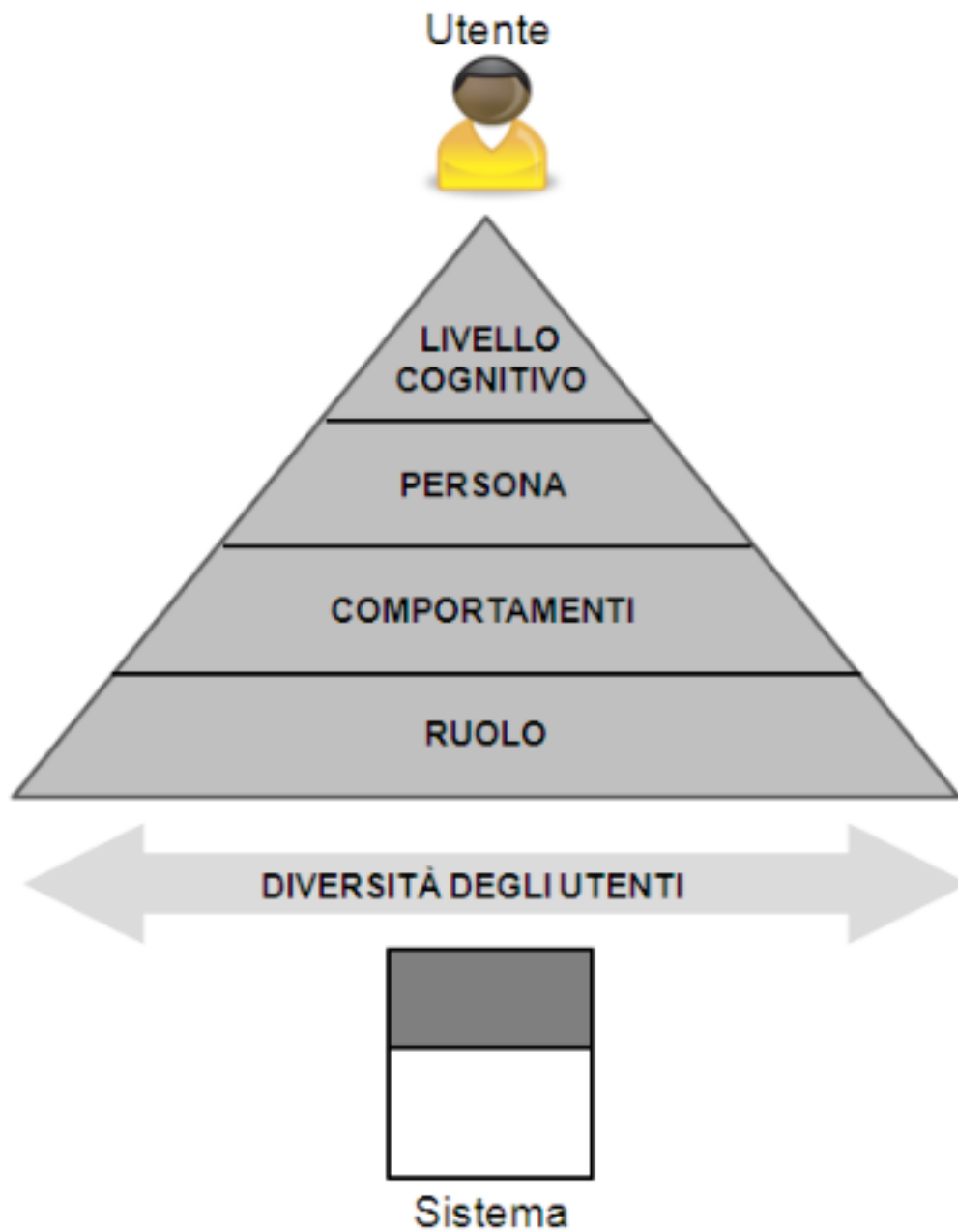
Da una visione centrata sul sistema si passa a una visione centrata sull'utente.

5.6 Definizione dei ruoli

La definizione del ruolo permette di separare le caratteristiche individuali della persona dagli aspetti legati allo scopo dell'interazione con il sistema

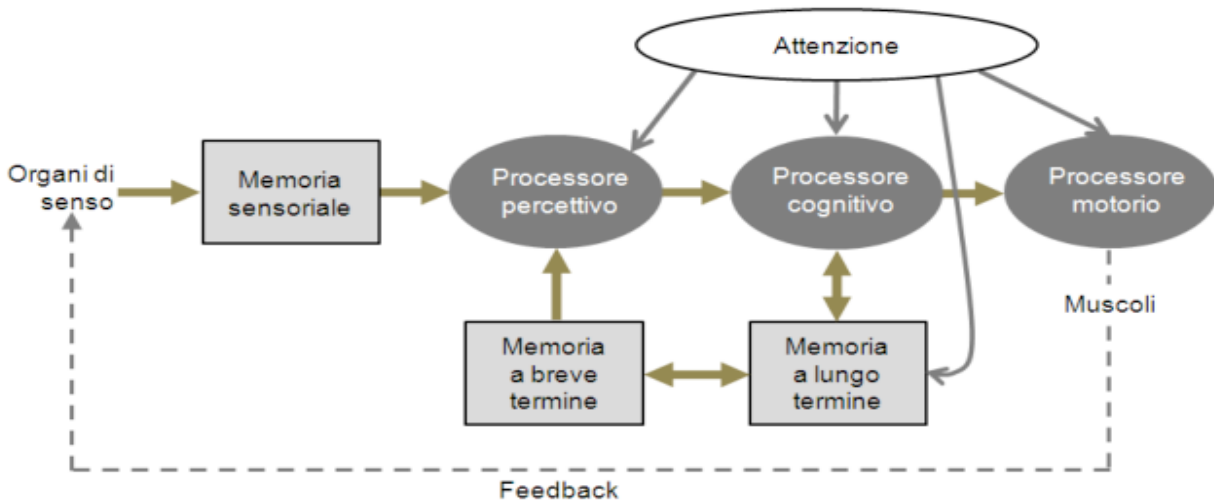


Lo studio dell'utente può essere condotto su piani diversi, in funzione degli aspetti a cui siamo interessati:



5.7 Modelli dell'utente

Le caratteristiche del sistema cognitivo dell'uomo sono studiate dalla psicologia, che fornisce molte informazioni utili per chi si occupa di interazione uomo-macchina.



5.8 Attenzione

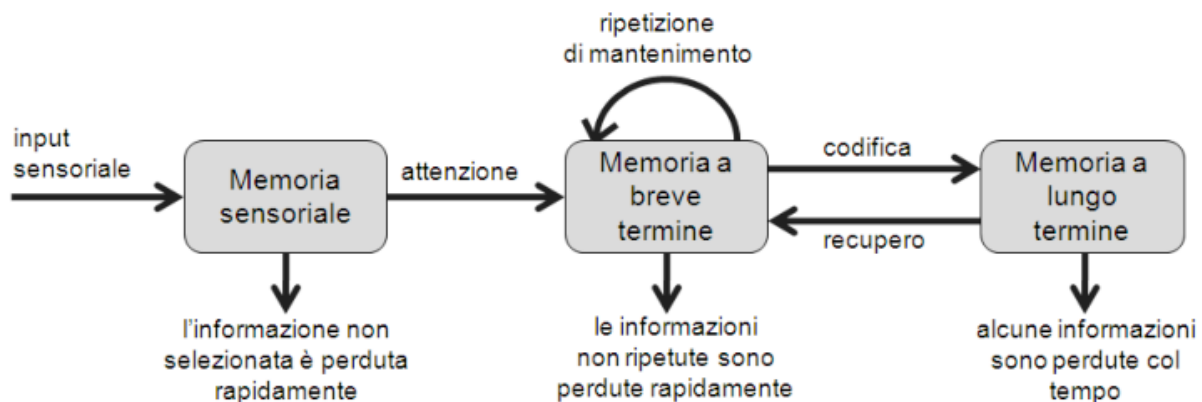
Questo termine non si riferisce a un fenomeno unitario, ma a una serie di processi psicologici fra loro molto differenti.

% Definizione

Definiamo attenzione, l'insieme di processi cognitivi che ci permettano di selezionare, fra tutte le informazioni che arrivano ai nostri sensi, quelle che in qualche modo ci interessano. Selezionare significa ignorare determinati stimoli, a favore di altri.

5.9 La memoria

La memoria può essere rappresentata con il seguente modello:



Questo modello, ispirato all'information processing, ipotizza l'esistenza di una serie di fasi, attraverso cui l'informazione transita, e una serie di *magazzini* destinati a contenerla.

In particolare, i dati sensoriali non ancora elaborati sono inizialmente acquisiti in *memorie sensoriali*. Vengono quindi selezionati attraverso meccanismi in cui gioca un ruolo importante l'attenzione e caricati in una memoria *a breve termine*, che è ritenuta il componente principale dove avvengono le elaborazioni mentali.

5.9.1 Memoria a breve termine

% Definizione

La memoria a breve termine può rivelarsi considerevolmente capace, se effettuiamo un opportuno chunking (se, cioè, creiamo degli opportuni raggruppamenti dotati di significato).

5.9.2 Memoria a lungo termine

% Definizione

La memoria a lungo termine è un sistema molto complesso. Essa ci consente di ricordare le informazioni per molto tempo: potenzialmente, per tutta la vita.

Alcuni la considerano costituita da due grandi sottosistemi, che svolgono compiti diversi: la memoria dichiarativa, e quella procedurale.

La prima riguarda tutte le conoscenze esprimibili a parole che si hanno sul mondo. La seconda conserva le conoscenze su *come fare* le cose.

5.10 Codifica

Le informazioni che ci sono state presentate ripetutamente sono più facili da ricordare.

% Definizione

La tecnica più elementare per memorizzare un'informazione a lungo termine consiste nel *ripeterla mentalmente* una o più volte, prestando attenzione al suo significato.

NB: Non bisogna confondere la semplice ripetizione con la reiterazione (la prima si riferisce al fatto che un elemento viene incontrato più di una volta; la seconda indica che l'elemento viene pensato ripetutamente, in un processo volontario che richiede attenzione).

5.11 Recupero

Per quanto riguarda il recupero delle informazioni dalla memoria a lungo termine, un aspetto importante è la distinzione fra riconoscimento e rievocazione.

% Rievocare (recall)

Significa estrarre dalla memoria un'informazione precedentemente memorizzata.

% Riconoscere (recognition)

Significa individuare, fra diverse alternative, quella (o quelle) che fanno al caso nostro.

5.12 Visione

% Acuità visiva (visual acuity)

Capacità dell'occhio di distinguere due punti vicini.

Essa si misura considerando l'angolo minimo α sotto cui devono essere visti perché l'occhio li percepisca separatamente.

L'acuità visiva è massima in corrispondenza della fovea centrale, e diminuisce verso la periferia. Questo è il motivo per cui, per distinguere bene i particolari di una figura, la dobbiamo fissare direttamente.

6 Lezione del 06-10

6.1 Diagramma dei Casi d'uso

Contiene:

- ▶ Attori;
- ▶ Casi d'uso;
- ▶ Relazioni

6.2 Descrizione testuale di un UCD

È poco dettagliato per capire come partire per l'implementazione del software.

Ogni UCD deve essere accompagnato da una descrizione testuale dettagliata

L'obiettivo è specificare in ogni aspetto l'interazione **attore/sistema** dal punto di vista dell'attore che compie un'azione.

Due rappresentazioni:

- ▶ Casual representation → testo libero;
- ▶ Fully Dressed Use Case → Template da riempire.

Esistono molte rappresentazioni Fully Dressed, concettualmente simili tra loro

In generale una descrizione di un UCD contiene:

- ▶ Una descrizione di ciò che il sistema e gli utenti si aspettano quando inizia lo scenario;
- ▶ Descrizione del flusso normale di eventi;
- ▶ Descrizione di cosa possa causare errori e come possono essere gestiti i problemi risultanti;
- ▶ Una descrizione dello stato del sistema al termine dello Use Case.

6.2.1 Template di A. Cockburn

USE CASE #x	NOME UC			
Goal in Context	Descrizione dell'obiettivo di questo UC			
Preconditions	Tutte le condizioni che devono valere per far partire lo UC			
Success End Condition	Stato in cui si deve trovare il contesto se lo UC è andato a buon fine			
Failed End Condition	Stato in cui si deve trovare il contesto se lo UC non è andato a buon fine, con motivo del problema			
Primary Actor	Attore principale dello UC			
Trigger	Azione dell'attore principale che avvia lo UC			
DESCRIPTION	Step n°	Attore 1	Attore n	Sistema
	1	Azione trigger		
	2			Risposta
	..	Azione 2		

	n			Azione finale

EXTENSIONS	Step	Attore 1	Attore n	Sistema
	m <condition>

	...			Azione finale
SUBVARIATIONS	Step	Attore 1	Attore n	Sistema
	n	Branching Action

	Step a cui ci si ricollega			
OPEN ISSUES -	Elencare tutti gli aspetti ancora da chiarire. Alla consegna del doc non ci devono essere open issues			
Due Date	Data entro cui queste funzionalità devono essere implementate			

6.3 Guida alla scrittura di Casi d'Uso

- ▶ I nomi dei casi d'uso dovrebbero sempre includere verbi;
- ▶ I nomi di Attori dovrebbero essere sostantivi;
- ▶ I casi d'uso devono iniziare con un'azione di un attore (trigger);
- ▶ Le relazioni causali tra passi successivi dovrebbero essere chiare;
- ▶ Un caso d'uso dovrebbe descrivere una transazione utente completa;
- ▶ Le eccezioni dovrebbero essere descritte nelle sezioni apposite;
- ▶ Un caso d'uso non dovrebbe descrivere un'interfaccia del sistema (utilizzare prototipi mock-up);
- ▶ Un caso d'uso non dovrebbe superare 2 o 3 pagine.

7 Lezione del 08-10

7.1 Modelli di dominio

È il più importante modello della fase di Specifica dei Requisiti.

È una rappresentazione visuale di classi concettuali, relative al dominio del problema → si esprime attraverso un class diagram, con classi, attributi e operazioni.

Si focalizza sui concetti che sono manipolati dal sistema, le loro proprietà e le relazioni.

% Definizione

È un **dizionario visuale** dei concetti principali relativi al dominio.

È detto anche **Modelli a Oggetti di Analisi**

È una rappresentazione di oggetti del mondo reale in uno specifico dominio, NON di oggetti SW:

- ▶ NB → sia il modello dinamico che il modello ad oggetti rappresentano concetti a livello utente, non a livello di componenti e classi SW;
- ▶ Le classi di analisi rappresentano astrazioni che saranno dettagliate e/o raffinate successivamente.

7.2 Euristiche three-object-type

Secondo tale euristica, il modello ad oggetti di analisi raggruppa gli oggetti in *Entity*, *Boundary* e *Control*:

- ▶ Gli oggetti **Entity** modellano l'informazione persistente;
- ▶ Gli oggetti **Boundary** modellano le interazioni tra gli attori e il sistema;
- ▶ Gli oggetti **Control** modellano la logica che si occupa di realizzare gli use case;

L'approccio three-object-type porta a modelli che sono più flessibili e facili da modificare:

- ▶ L'interfaccia al sistema (rappresentata da oggetti boundary) è più soggetta a cambiamenti rispetto alle funzionalità (rappresentate da oggetti entity e control).

8 Lezione del 11-10

8.1 Sistema motorio

Per quanto riguarda il processore motorio, ricordiamo:

- ▶ Apprendimento motorio;
- ▶ [Legge di Fitts](#);

% Apprendimento motorio

Apprendimento di particolari sequenze di movimento che coinvolgono l'apparato muscolare. È un apprendimento per gradi.

Il feedback può essere:

- ▶ Qualitativo;
- ▶ Quantitativo.

% La legge di Fitts

Rappresenta il modello matematico di un movimento umano. Calcola il tempo impiegato per muoversi rapidamente da un punto iniziale a un'area con una determinata estensione. Esprime il tempo in funzione della distanza tra punto iniziale e obiettivo finale, correlato all'estensione dell'area considerata.

8.2 Etonografia

Raccoglie e registra informazioni, che verranno gestite dagli altri.

8.3 8 golden rule - Shneiderman

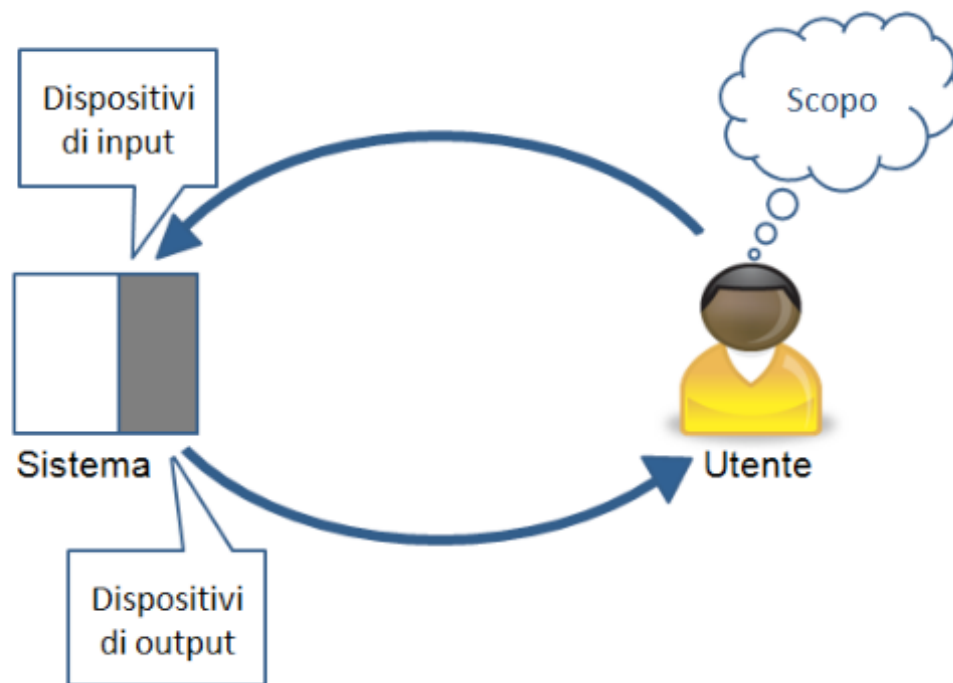
1. Sforzarsi per la coerenza (Strive for Consistency) → la coerenza nel design e nei suoi elementi visivi è uno dei fattori più importanti che vanno tenuti in considerazione;
2. Consentire agli utenti frequenti di utilizzare i collegamenti (Enable Frequent Users to Use shortcuts);
3. Offrire feedback informativo (Offer Informative Feedback);
4. Progettare il dialogo per dare la chiusura (Design Dialog to Yield Closure);
5. Offrire una semplice gestione degli elementi (Offer simple Error Handling);
6. Consentire una facile inversione delle azioni (Permit Easy Reversal of Action);

7. Supportare il focus del controllo interno (Support Internal Locus of Control) → Quando viene progettato un sistema, bisogna progettarlo in modo tale che l'utente senta di avere il controllo del sistema;
8. Ridurre il carico di memoria a breve termine (Reduce Short-Term Memory Load).

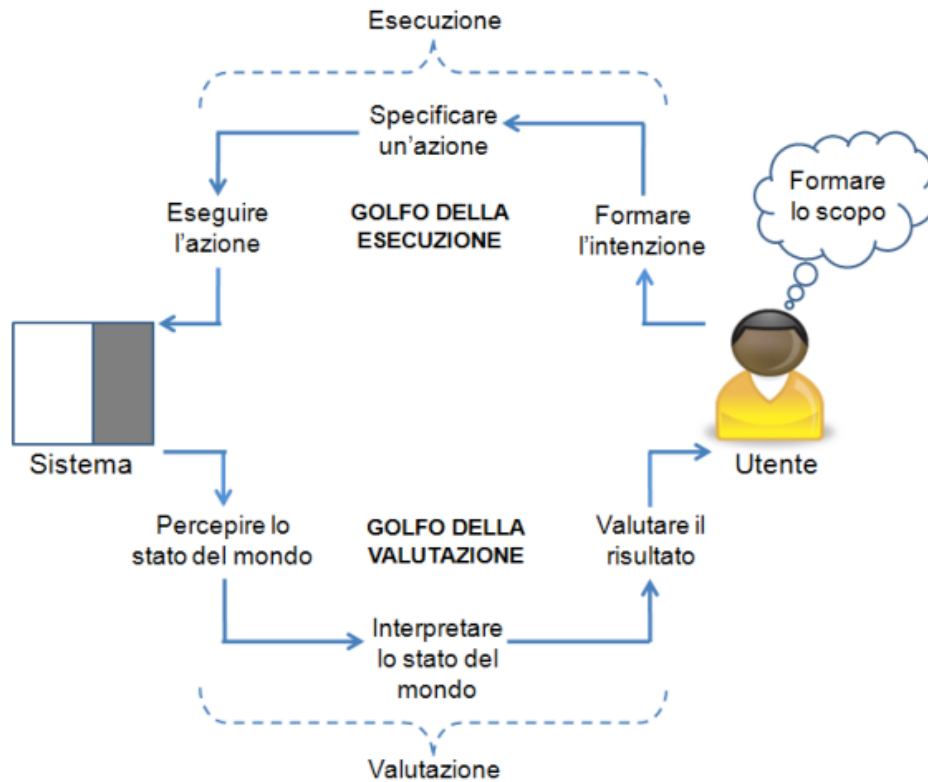
9 Lezione del 13-10

9.1 Usabilità

Il modello più semplice dell'interazione fra un sistema e il suo utilizzatore è rappresentato dal *ciclo di feedback* (*feedback loop*):



L'utente per raggiungere il proprio scopo, fornisce un input al sistema, e riceve da questo una risposta, che viene interpretata e confrontata con lo scopo iniziale. Il risultato di questo confronto porta alla successiva azione dell'utente, innescando così un nuovo ciclo di *stimolo-risposta*.



Questo modello scompone il nostro operare sugli oggetti in sette passi (o stadi) principali:

1. Formare lo scopo → decidiamo quale scopo vogliamo raggiungere.
2. Formare l'intenzione → decidiamo che cosa intendiamo fare per raggiungere lo scopo prefissato;
3. Specificare un'azione → pianifichiamo nel dettaglio le azioni specifiche da compiere;
4. Esegui l'azione → eseguiamo effettivamente le azioni pianificate;
5. Percepire lo stato del mondo → osserviamo come sono cambiati il sistema e il mondo circostante dopo le nostre azioni;
6. Interpretare lo stato del mondo → elaboriamo ciò che abbiamo osservato, per dargli un senso;
7. Valutare il risultato → decidiamo se lo scopo iniziale è stato raggiunto

Il modello permette di individuare con grande chiarezza i momenti in cui possono presentarsi dei problemi. È possibile che si incontrino difficoltà nel passare da uno stadio all'altro (l'attraversamento dei *golfi* che li separano).

In particolare esistono 2 golfi che possono essere particolarmente difficili da superare:

- ▶ *golfo della esecuzione* → separa lo stadio delle intenzioni da quello delle azioni;
- ▶ *golfo della valutazione* → separa lo stadio della percezione dello stato del mondo da quella della valutazione dei risultati.

9.2 Affordance e feedback

% Affordance

Denota la proprietà di un oggetto di influenza, attraverso la sua apparenza visiva, il modo in cui viene usato

Un oggetto che possiede una buona affordance *invita* chi lo guarda a utilizzarlo nel modo corretto, il modo per cui è stato concepito.

Per ridurre l'ampiezza del golfo della valutazione, invece, gli oggetti dovranno fornire un feedback facilmente interpretabile, cioè un segnale che indichi chiaramente all'utente quale modifiche le sue azioni abbiano prodotto sullo stato del sistema.

Il feedback deve essere ben comprensibile e specifico → L'utente deve essere in grado di interpretarlo senza fatica; dovrebbe essere formulato nel modo che l'utente si aspetta.

Fondamentale è la sua tempestività → se la distanza temporale fra azione e feedback è significativa essi possono essere interpretati come eventi tra loro indipendente (in questi casi è opportuno inserire dei feedback intermedi, che segnalino chiaramente all'utente il progredire dello stato del sistema verso lo stato finale desiderato).

9.3 Nozione di usabilità

% Usabilità

L'usabilità di un prodotto è il grado con cui esso può essere usato da specificati utenti per raggiungere specificati obiettivi con efficacia, efficienza e soddisfazione in uno specificato contesto d'uso.

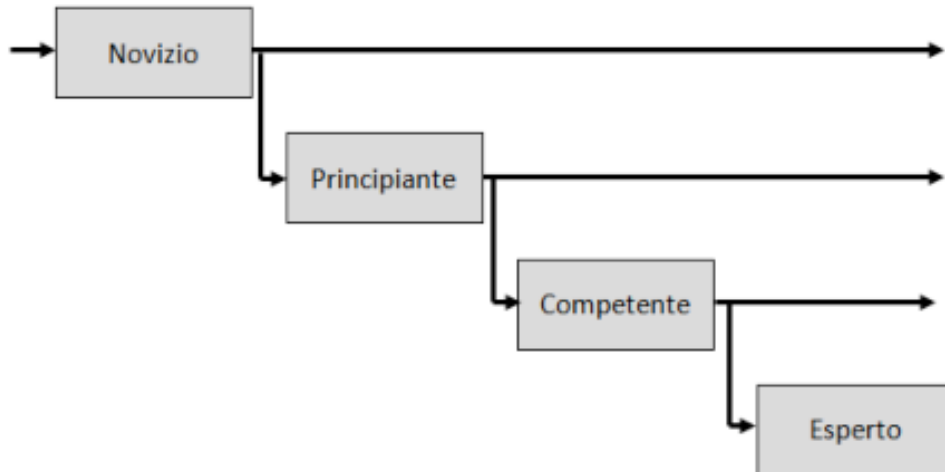
- ▶ Efficacia → accuratezza e completezza con cui gli utenti raggiungono specificati obiettivi (si considera il livello di precisione con cui l'utente riesce a raggiungere i suoi scopi);
- ▶ Efficienza → quantità di risorse spese in relazione all'accuratezza e alla completezza con cui gli utenti raggiungono gli obiettivi;
- ▶ Soddisfazione → libertà del disagio e l'attitudine positiva verso l'uso del prodotto.

L'usabilità non è una proprietà assoluta degli oggetti, ma sempre relativa al compito da svolgere e al contesto d'uso.

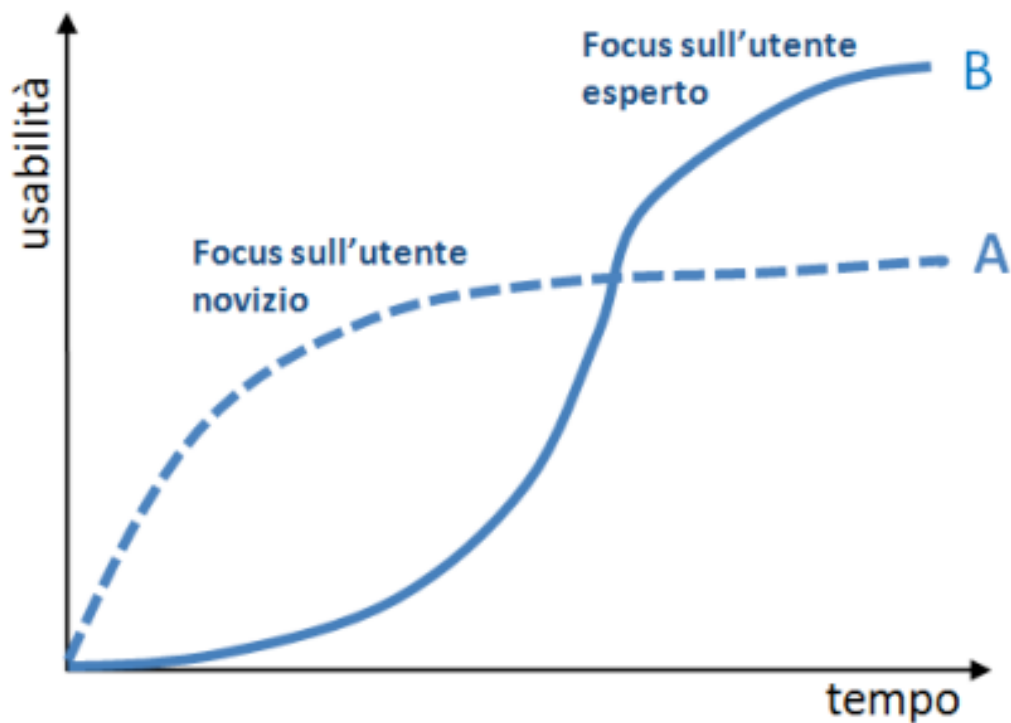
L'usabilità non può essere valutata studiando un prodotto in isolamento.

9.4 Apprendibilità e memorabilità

Evoluzione dell'utente nel rapporto con il sistema



In questo processo di apprendimento, l'utente può riscontrare difficoltà più o meno grandi, a seconda delle caratteristiche del sistema. Prodotti simili possono presentare differenti *profili di apprendimento*:



Un sistema che sia facile da imparare si dice dotato di elevata *apprendibilità*.

Nella progettazione di un sistema, il progettista ha di fronte a sé diverse scelte possibili:

- ▶ Considerare come principali destinatari del prodotto gli *utenti occasionali*;
- ▶ Progettare in primo luogo per gli *utenti continuativi*.

I risultati della progettazione, saranno prodotti molto differenti, destinati a 2 fasce di mercati sostanzialmente diverse.

[J. Nielsen](#) definisce l'usabilità come la somma dei seguenti 5 attributi:

- ▶ Apprendibilità → il sistema dovrebbe essere facile da imparare, in modo che l'utente possa rapidamente iniziare a ottenere qualche risultato dal sistema;
- ▶ Efficienza → il sistema dovrebbe essere efficiente da usare, in modo che, quando l'utente ha imparato a usarlo, sia possibile un alto livello di produttività;
- ▶ Memorabilità → il sistema dovrebbe essere facile da ricordare (un utente occasionale non dovrebbe imparare tutto di nuovo);
- ▶ Errori → Il sistema dovrebbe rendere difficile sbagliare, in modo che gli utenti possano compiere pochi errori durante l'uso;
- ▶ Soddisfazione → il sistema dovrebbe essere piacevole da usare.

9.4.1 Accessibilità

% Definizione

Capacità dei sistemi informatici, nelle forme e nei limiti consentiti dalle conoscenze tecnologiche, di erogare servizi e fornire informazioni fruibili, senza discriminazioni, anche da parte di coloro che a causa di disabilità necessitano di tecnologie assistive o configurazioni particolari.

Si fa uso del termine *accessibilità universali* per enfatizzare un'accessibilità estesa a *tutti i possibili utenti*.

10 Lezione del 15-10

10.1 Identificare gli oggetti

La fase fondamentale dello sviluppo del SW OO è quella dell'analisi e della progettazioni → sono queste le fasi che garantiscono il successo e il raggiungimento degli obiettivi dell'OOP (è un procedimento altamente creativo e soggettivo).

Three-object-type → Classificazione Oggetti in Entity, Boundary e Control.

10.2 Identificare gli Oggetti Entity e l'euristica di Abbott

Gli oggetti Entity rappresentano i concetti del dominio (facilitata dall'uso dell'Euristica di Abbot).

L'euristica di Abbott si basa sull'analisi linguistica per identificare oggetti, attributi, associazioni dai requisiti di sistema → mappano parti delle parole per modellare componenti.

Parti del parlato	Componenti del modello	esempio
Nome proprio	istanza	Alice
Nome comune	Classe	Agente di Polizia
Verbo fare	Operazione	Crea, Submit, Select
Verbo essere	Gerarchia	È un tipo di, è uno di
Verbo avere	Aggregazione	Ha, consiste di, include
Verbo modale	Vincoli	Deve essere
Aggettivo	Attributo	Descrizione dell'incidente

10.2.1 Vantaggi e svantaggi dell'Euristica di Abbot

Vantaggi	Svantaggi
Ci si focalizza sui termini dell'utente	Il linguaggio naturale è impreciso Anche il modello a OO derivato rischia di esserlo
	La qualità del modello dipende fortemente dallo stile di scrittura dell'analista
	Ci possono essere molti più sostantivi delle classi rilevanti, corrispondenti a sinonimi o attributi

10.2.2 Identificare gli Oggetti Boundary

Rappresentano l'interfaccia del sistema con gli attori → modellano l'interfaccia senza descriverne gli aspetti visuali.

11 Lezione del 18-10

11.1 L'ingegneria dell'usabilità

% Ingegneria

Disciplina che si occupa dell'arte o della scienza di applicare la conoscenza scientifica a problemi pratici.

L'Ingegneria dell'usabilità è un processo, che consiste nello specificare *quantitativamente* e in anticipo, quali caratteristiche e in quale misura il prodotto finale da ingegnerizzare dovrà possedere.

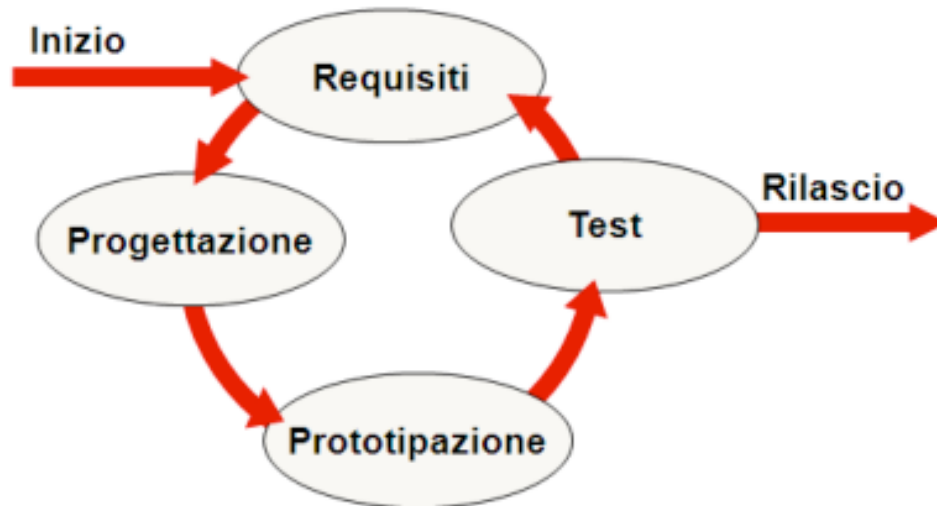
Questo processo è seguito dall'effettiva realizzazione del prodotto, e dalla dimostrazione che esso effettivamente possiede le caratteristiche pianificate.

I principi cardine di questa disciplina possono riassumersi in 3 punti chiave:

1. Focalizzazione sull'utente, all'inizio e durante tutto il processo di progettazione;

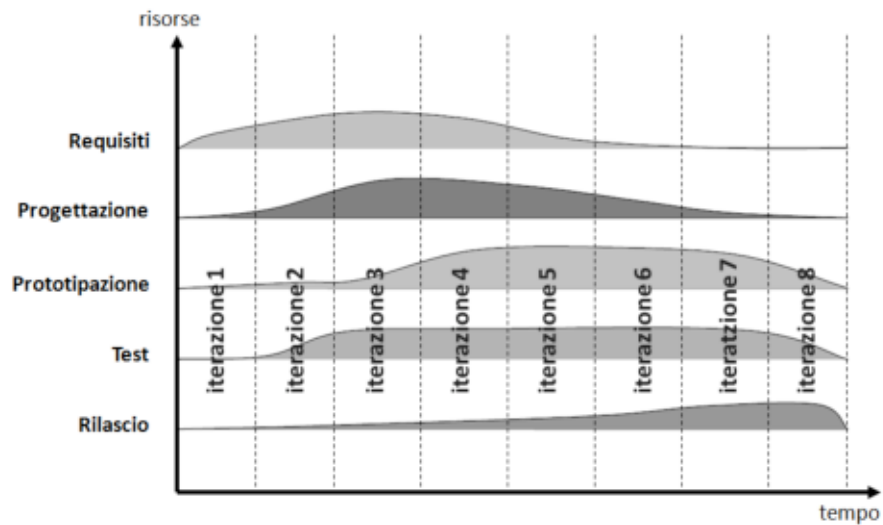
2. Prove con l'utente durante l'intero processo di progettazione, con analisi qualitative e misure quantitative;
3. Modello di progettazione e sviluppo iterativo, per prototipi successivi.

11.2 Modello iterativo



Ovviamente, nelle varie iterazioni, le diverse attività mostrate avranno pesi diversi.

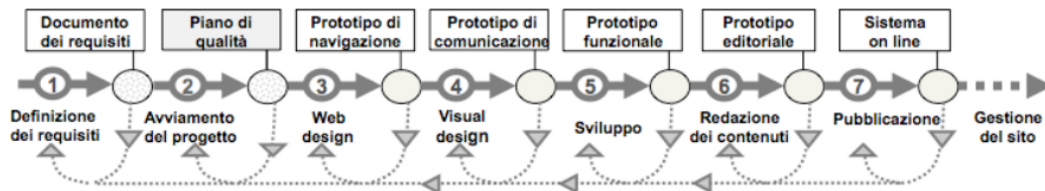
11.3 Allocazione delle risorse secondo il modello iterativo



11.4 Modello ISO 13407



11.5 Processo iterativo per la progettazione e sviluppo di un sito web



11.6 Ingegneria e Creatività

11.6.1 Mimesi

%

Riprodurre un prodotto già esistente, tipicamente realizzandolo con tecnologie differenti.

11.6.2 Ibridazione

%

Concepire un oggetto nuovo mescolando o integrando fra loro aspetti e funzioni di oggetti diversi.

11.6.3 Metafora

%

Descrivere una cosa nei termini di un'altra. Due domini semantici indipendenti vengono messi in contatto.

11.6.4 Variazione

%

Uno dei procedimenti più frequenti nella progettazione. Può produrre innovazioni sostanziali, soprattutto se si considera l'evoluzione nell'arco di più generazioni successive.

11.7 Composizione di design pattern

La conoscenza e l'analisi delle soluzioni di progettazione adottate in altri sistemi costituisce una fonte importante di spunti per l'interaction design.

Con il termine *design pattern* si indica una soluzione generale a un problema di progettazione che si ripropone in molte situazioni, anche diverse fra loro.

Non una soluzione *finita*, ma un modello, un *template* da adattare allo specifico contesto.

Esistono anche delle raccolte di *anti-pattern* a problemi che si ripropongono con una certa frequenza.

Le collezioni di pattern per il design dell'interazione sono molto utili per diversi motivi:

- ▶ Suggestiscono ai progettisti meno esperti le migliori pratiche adottate in ambiti applicativi specifici;
- ▶ Raccolgono, in forma più o meno organica, lo stato della pratica corrente, cioè l'esperienza collettiva delle comunità di progetto nei vari ambiti;
- ▶ Contribuiscono alla formazione di un linguaggio comune, facilitando la comunicazione fra i professionisti della disciplina;
- ▶ Riducono gli sprechi di tempo e risorse;
- ▶ Facilitano l'individuazione delle soluzioni più adatte al problema specifico, contribuendo così a ridurre tempi e costi di progettazione e sviluppo;
- ▶ Contribuiscono alla diffusione di *standard di fatto* ben sperimentati.

12 Lezione del 20-10

12.1 UML Statecharts

Gli statecharts sono largamente usati nell'industria, e non solo per la modellizzazione.

Sistemi reattivi → sistemi che reagiscono a segnali **interni/esterni**.

12.2 Modellizzazione con stati e transizioni

Gli stati rappresentano situazioni in cui alcune invarianti:

- ▶ Condizioni statiche → il sistema aspetta che qualcosa accada;
- ▶ Condizioni dinamiche → il sistema performa su un task specifico.

Le transizioni rappresentano un possibile cambio di stati (da uno all'altro).

12.3 Sintassi e Semantica degli Statechart

12.3.1 Regioni, vertici e transizioni

Una regione è una parte di spazio che contiene vertici e transizione.

Uno statechart presenta una regione principale (top-level).

I vertici rappresentano gli stati → ne esistono di diversi tipi di semantica

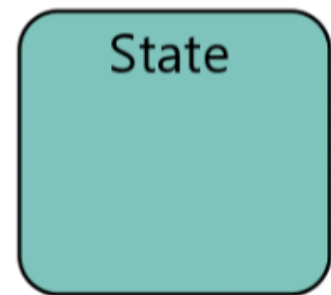
Una transizione è rappresentata da un segmento orientato che connette fra loro 2 stati (collegano i diversi vertici). Indica il passaggio da uno stato all'altro.

In un diagramma per macchine a stati, uno *stato* del sistema modellato è rappresentato da un rettangolo con gli angoli arrotondati, con il nome dello stato al suo interno.

Stati semplici

Rappresentano stati di sistema non strutturati:

- ▶ Rappresentati da un rettangolo con bordi arrotondati;
- ▶ Un nome opzionale;

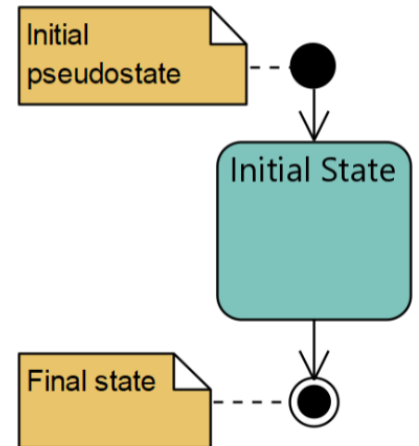


Pseudostato iniziale e Stato finale

Lo pseudostato iniziale è usato per settare lo stato iniziale

Una regione può contenerne al massimo 1.

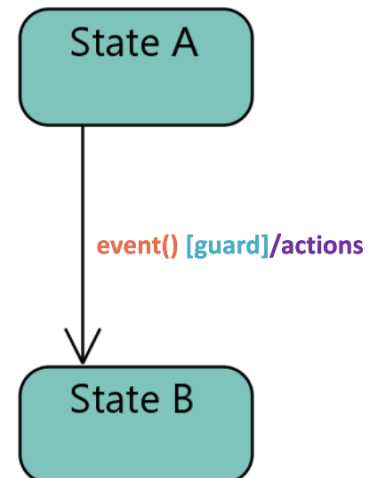
Gli stati finali rappresentano una situazione in cui la computazione è completata.



12.3.2 Transizioni

Indicano cambi di stato. Possono essere decorati con delle etichette del tipo:

- ▶ **triggers** → lista di eventi che possono indurre a un cambio di stato
- ▶ **[guard]** → condizione booleana
- ▶ **\actions** → lista di operazioni per eseguire quando la transizione inizia.



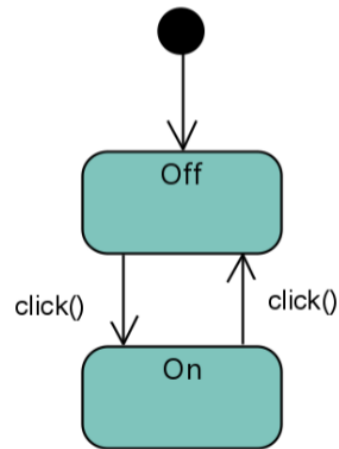
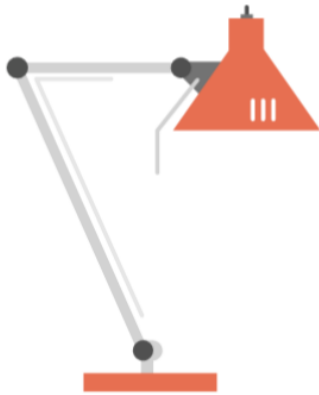
Una transizione può essere eseguita, quando:

- ▶ Sono stati completati tutti gli stati indicati;
- ▶ La condizione della **guard** è vera.

Una transizione si dice spontanea quando non presenta **triggers** e **guard**

Se esistono transizioni eseguibili multiple, solo una di questa viene eseguita (determinazione non deterministica).

12.3.3 Esempio (una lampada con un singolo bottone)



12.3.4 Stati - attività interne

Gli stati possono, opzionalmente contenere una lista di attività interne.

Ogni attività è caratterizzata da una etichetta indicando quando l'attività debba essere invocata.

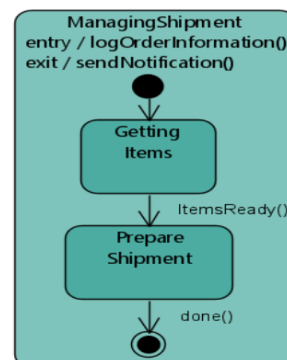
Etichette riservate:

- ▶ **entry** / → attività performate in ingresso;
- ▶ **do** / → attività performate fin quando l'attività è in questo stato;
- ▶ **exit** / → attività performate in uscita.

12.3.5 Composite States

Uno stato contiene:

- ▶ Nome del compartimento;
- ▶ Attività del compartimento interne;
- ▶ Una o più regioni interne.



Uno stato con regioni interne è uno stato composito → Gli stati in una regione interna sono detti sottostati

Permetto di modellare strutture gerarchiche.

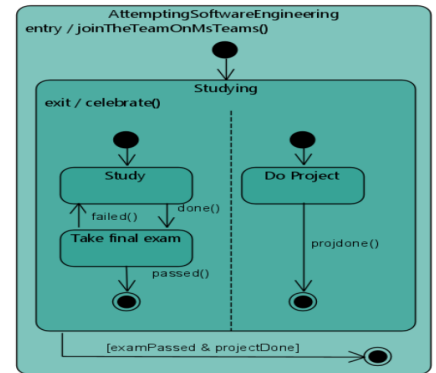
La regione interna gestisce il comportamento dello stato a cui appartiene.

Permette un modo elegante e conciso di rappresentare comportamenti complessi.


12.3.6 Composite States - parallel regions

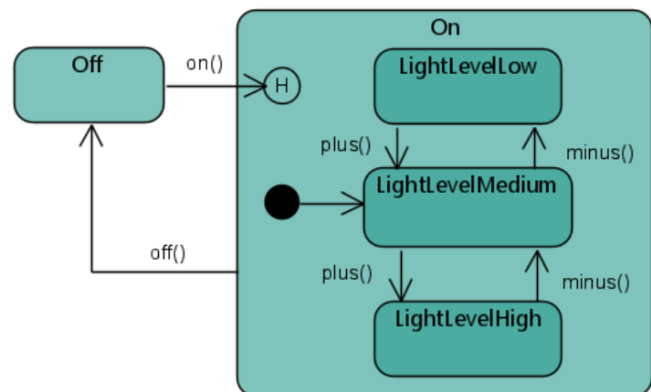
Possono contenere regioni multiple, rappresentano comportamenti che possono accadere in parallelo.

In uscita da uno stato composito, ogni regione è terminata.




12.3.7 Shallow History Pseudostates

- Depicted as a 
- Represents the most recently active state of a composite state, but not substates of that substate!
- Only in composite states, and only one per region

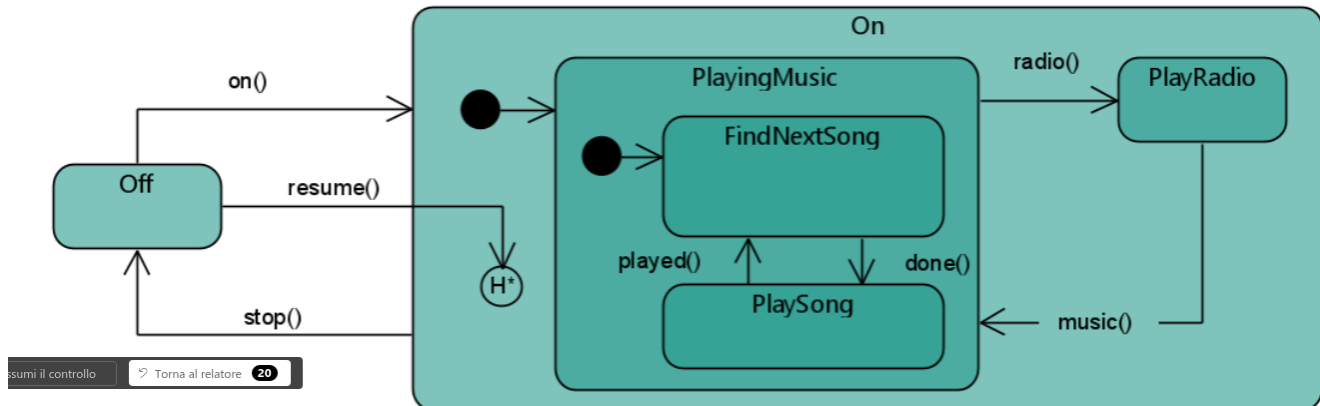


Statechart for a lamp with three different light levels

12.3.8 Deep History Pseudostates

Depicted as a 

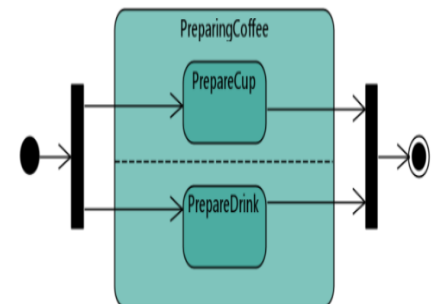
Same as shallow history ones, but restore the entire region configuration (substates of the substates included!)



12.3.9 Fork & Join Pseudostates

Forks suddividono le transizioni in arrivo in molteplici transizioni inserendo i vertici in regioni ortogonali.

Joins fondono i vertici di transizioni esistenti in regioni ortogonali in una singola transizione.



12.3.10 Gestire gli stati della UI con gli statechart

Gli statechart possono essere usati per *guidare* la logica GUI.

Gli statechart sono più semplici da comprendere.

Il comportamento è sdoppiato dai componenti GUI:

- Separare il quando, dal cosa

Gli statechart scalano molto bene al crescere della complessità.

13 Lezione del 22-10

13.1 Presentazione progetto

14 Lezione del 25-10

14.1 Gli Statechart

Sono diagrammi a stati di tipo gerarchico. La loro caratteristica principale è quella di permettere di modellare un sistema descrivendone la sua evoluzione da uno stato all'altro, per livelli di astrazione successivi.

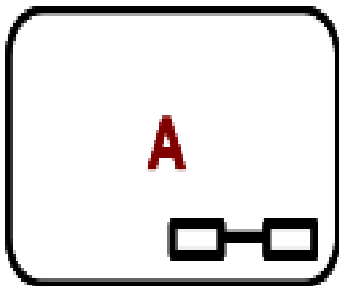
Questo risulta molto utile per descrivere con relativa semplicità situazioni che altrimenti richiederebbero diagrammi molto complessi dal punto di vista grafico.

14.2 Transizioni interne

A volte può essere utile poter indicare che il sistema modellato effettua delle attività (eventualmente al verificarsi di certi eventi e condizioni) senza cambiare stato.

Queste possono essere rappresentate scrivendo eventi, condizioni e attività all'interno dello stato, con la notazione classica per le transizioni $\rightarrow \text{evento}[\text{condizione}]/\text{attività}$.

14.3 Stati composti



Uno *stato composto* (o *superstato*) è uno stato che può essere decomposto in una o più *regioni*, ciascuna delle quali può contenere altri stati (detti *sottostati*).

14.4 Sottomacchine

Può essere necessario richiamare uno stesso diagramma da varie parti di un diagramma di più alto livello. In tal caso, si dice che il diagramma richiamato rappresenta una *sottomacchina* del chiamante.

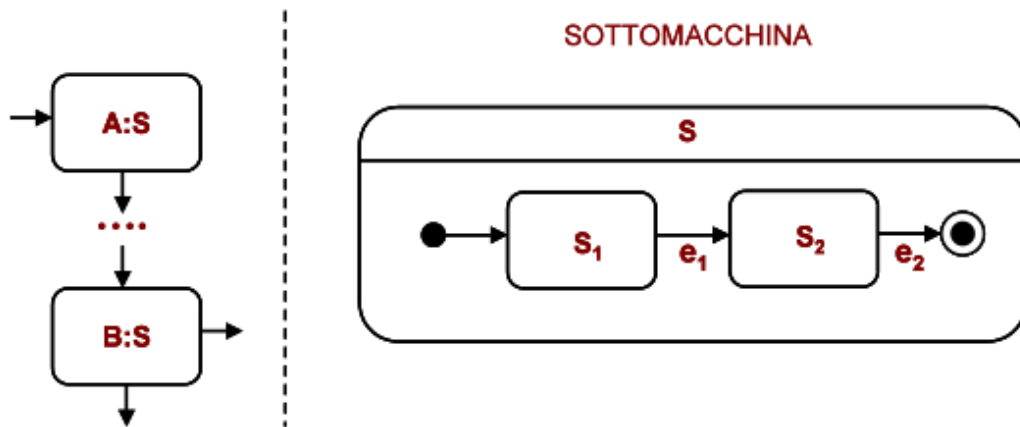
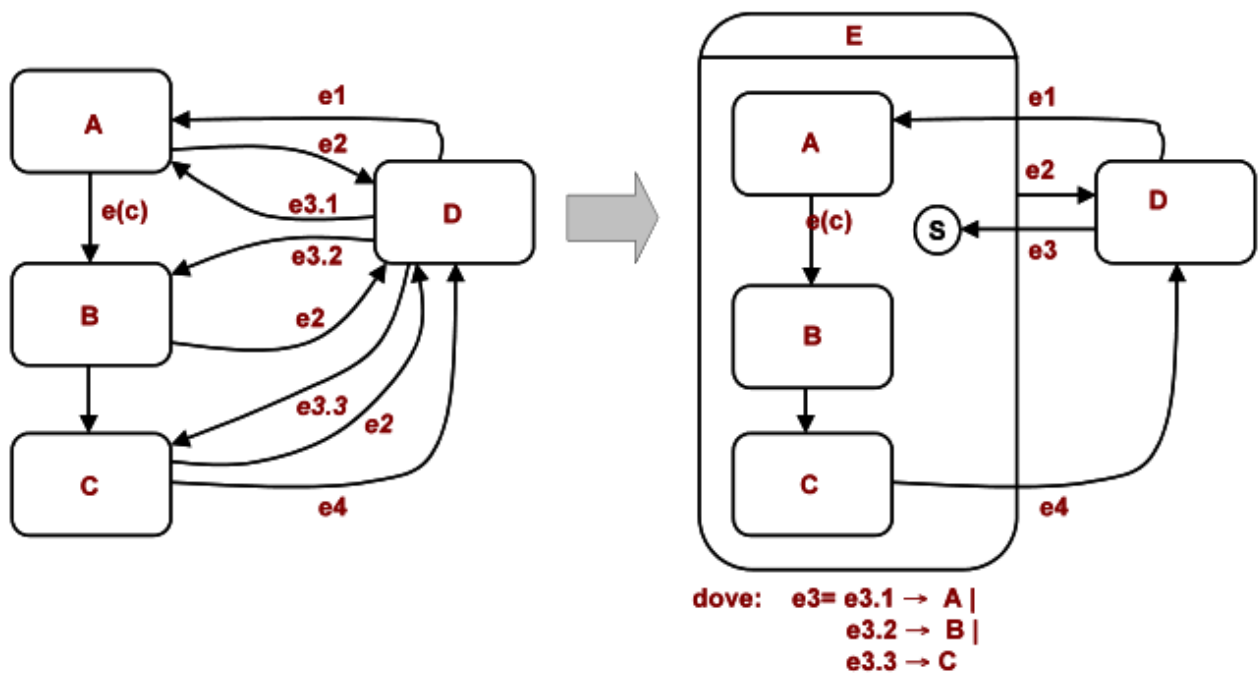


Figura 307. Una sottomacchina S richiamata da più punti di un diagramma a stati di più alto livello

14.5 Notazioni abbreviate

Per permettere di rappresentare in modo semplice anche situazioni complesse, senza dover disegnare nel diagramma troppe transizioni, si possono definire alcune notazioni abbreviate.



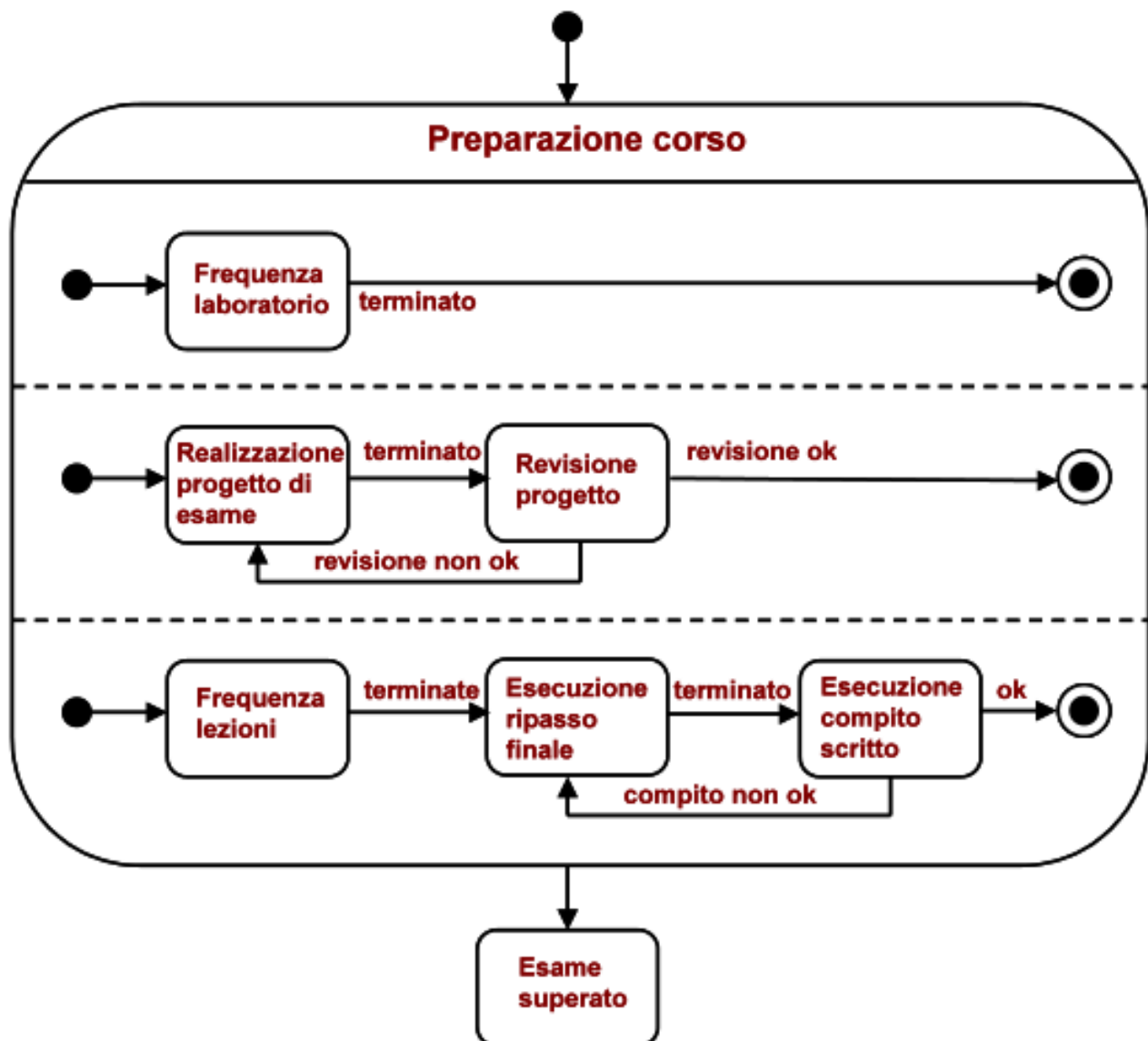
Il diagramma (a) può essere rappresentato, anche come in (b).

14.6 Sottostati concorrenti

Quando le regioni sono più di una, gli stati contenuti in una regione sono *concorrenti* a quelli delle altre. Pertanto, lo stato composto si potrà trovare contemporaneamente in più sottostati, ciascuno appartenente ad una sua diversa regione.

Le regioni di uno stato sono separate da linee tratteggiate, orizzontali o verticali.

Quando tutti i diagrammi raggiungono il loro stato finale, allora termina anche lo stato composto e, nel caso, viene eseguita l'azione associata al suo evento speciale *exit*.



14.7 I prototipi

È una rappresentazione di un prodotto o di un sistema, o di una sua parte, che, anche se in qualche modo limitato, può essere utilizzata a scopo di valutazione.

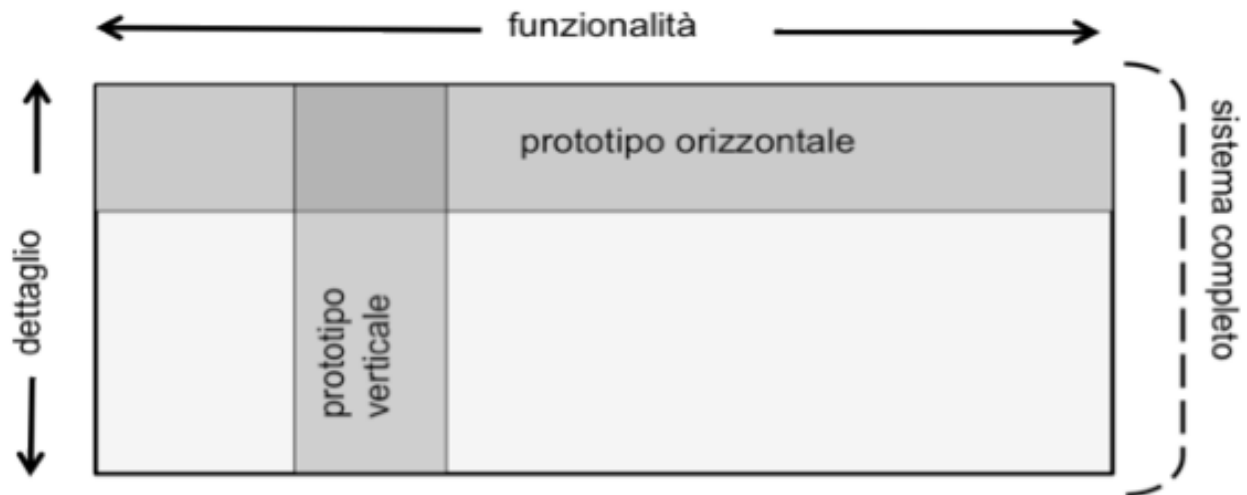
Un prototipo, è, quindi un modello approssimato o parziale del sistema che vogliamo sviluppare, realizzato allo scopo di valutare determinate caratteristiche.

Definire lo scopo di un prototipo è l'arte di identificare i problemi di progettazione più critici.

14.7.1 Classificazione dei prototipi

Scopo	Ruolo	Serve a valutare il ruolo del prodotto nella vita del suo utente
	Interfaccia	Serve a valutare le modalità di interazione fra utente e prodotto
	Implementazione	Serve a valutare aspetti tecnici relativi alla realizzazione tecnica del prodotto
Modo d'uso	Statico	È una rappresentazione statica del prodotto (es storyboard, diagram di vario tipo)
	Dinamico	È una rappresentazione dinamica (ma non interattiva) del prodotto (es video)
	Interattivo	Permette agli utenti di effettuare prove d'uso del prodotto, anche se semplificate e approssimate
Fedeltà	Alta Fedeltà	Assomiglia in tutti gli aspetti al prodotto finale
	Bassa Fedeltà	Assomiglia alla lontana al prodotto finale
Completezza funzionale	Orizzontale	Fornisce tutte le funzioni del prodotto finale, anche se in versione semplificata o limitata
	Verticale	Fornisce solo alcune funzioni, realizzate in dettaglio
Durata	Usa e getta	Non viene conservato dopo l'uso
	Evolutivo	Viene fatto evolvere fino al prodotto finale

14.7.2 Classificazione dei prototipi rispetto alla completezza funzionale



14.7.3 La tecnica del Mago di Oz

Questa tecnica consiste nel realizzare un prototipo interattivo, in cui però le risposte (o parte di esse) siano fornite, se possibile all'insaputa dell'utente, da parte di un essere umano che operi, per così dire, *dietro le quinte* come, appunto, il mago di Oz della favola.

14.7.4 Prototipi wire-frame

Sono prototipi interattivi a bassa fedeltà, di solito usa-e-getta, nei quali la grafica è estremamente semplificata, e mostra solo i contorni degli oggetti.

Permettono di sperimentare le modalità principali di interazione, prima che i dettagli della grafica siano definiti.

14.7.5 Prototipi ipertestuali

Il prototipo, in questo caso, è costituito da una serie di immagini (*snapshot*) che rappresentano l'aspetto del prodotto in corso di progettazione.

Le varie snapshot sono legate fra loro da link ipertestuali, che permettono l'interazione con il prodotto.

14.7.6 Prototipi intermedi

I prototipi intermedi permettono di provare specifici aspetti del prodotto, ma non ancora le sue funzioni complessive, che potranno essere esercitate soltanto alla fine del processo

14.7.7 Prototipi finali

È, in sostanza, il programma completo, con i suoi contenuti definitivi, le sue funzionalità collaudate, e con una base di dati reali o, comunque significativi.

15 Lezione del 27-10

15.1 Activity Diagrams

Forniscono la sequenza di operazioni che definiscono un'attività più complessa.

Permettono di rappresentare processi paralleli e la loro sincronizzazione (*fork*).

Un Activity Diagram può essere associato:

- ▶ Ad uno Use Case;
- ▶ A una classe;
- ▶ All'implementazione di un'operazione

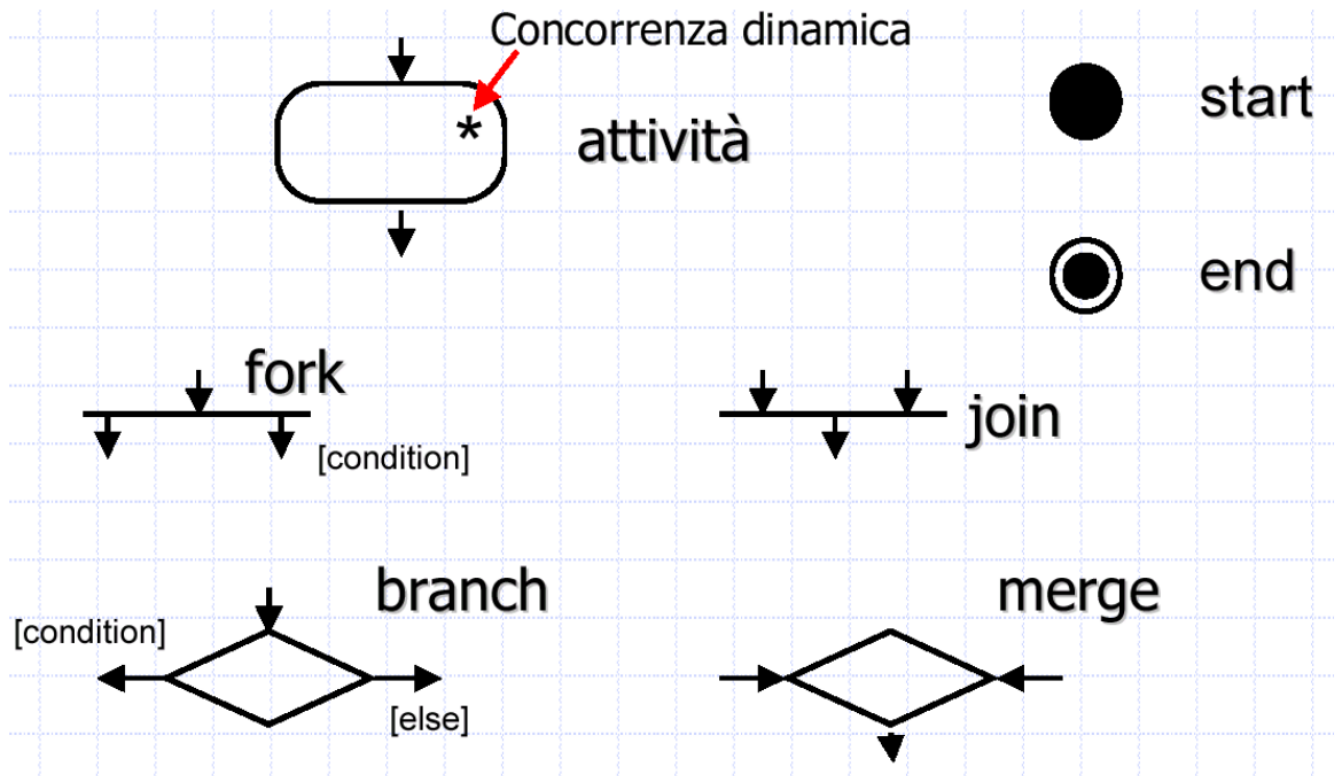
Servono a rappresentare sistemi di workflow, oppure la logica interna di un processo di qualunque livello.

Utili per modellare:

- ▶ Comportamenti sequenziali;
- ▶ Non determinismo;
- ▶ Concorrenza;
- ▶ Sistemi distribuiti;
- ▶ Business workflow;
- ▶ Operazioni.

Le attività possono essere gerarchiche

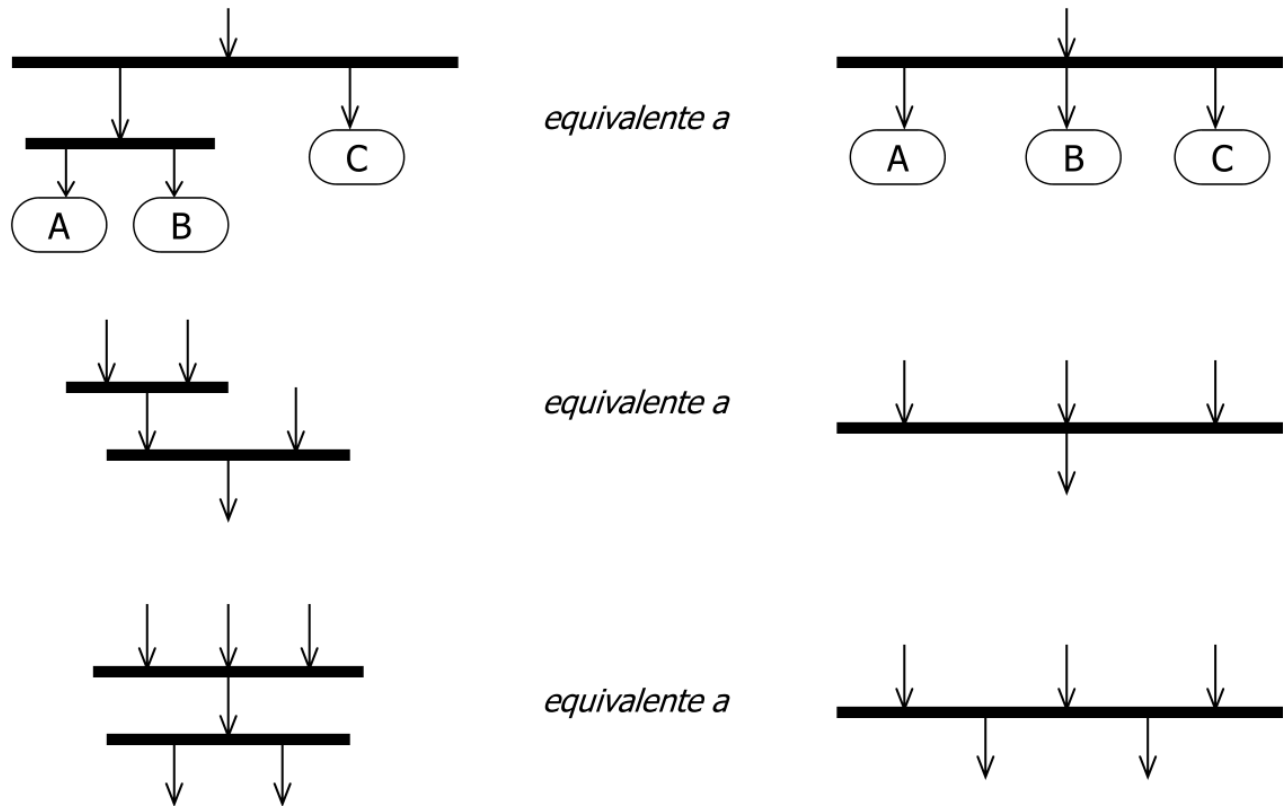
15.1.1 Elementi Grafici



15.2 Activity Diagram - Elementi

- ▶ **Activity** → Una esecuzione non atomica entro uno state Machine
 - ◊ Una activity è composta da action, elaborazioni atomiche comportanti un cambiamento di stato del sistema o il ritorno di un valore;
- ▶ **Transition** → Flusso di controllo tra 2 action successive;
- ▶ **Guard expression** → espressione booleana (condition) che deve essere verificata per attivare una transition;
- ▶ **Branch** → Specifica percorsi alternativi in base a espressioni booleane;
 - ◊ Un branch ha una unica transition in ingresso e 2 o più transition in uscita;
- ▶ **Synchronization bar** → usata per sincronizzare flussi concorrenti
 - ◊ **fork** → per splittare un flusso su più transition verso action state concorrenti;
 - ◊ **join** → per unificare più transition da più action state concorrenti in una sola
 - ◊ Il numero di fork e di join dovrebbero essere bilanciati.

15.3 Combinazione di Fork e Join



15.4 Swimlanes

Costrutto grafico rappresentante un insieme partizionato di **action/activity**.

Identificano le responsabilità relative alle diverse operazioni:

- ▶ Parti di un oggetto;
- ▶ Oggetti diversi.

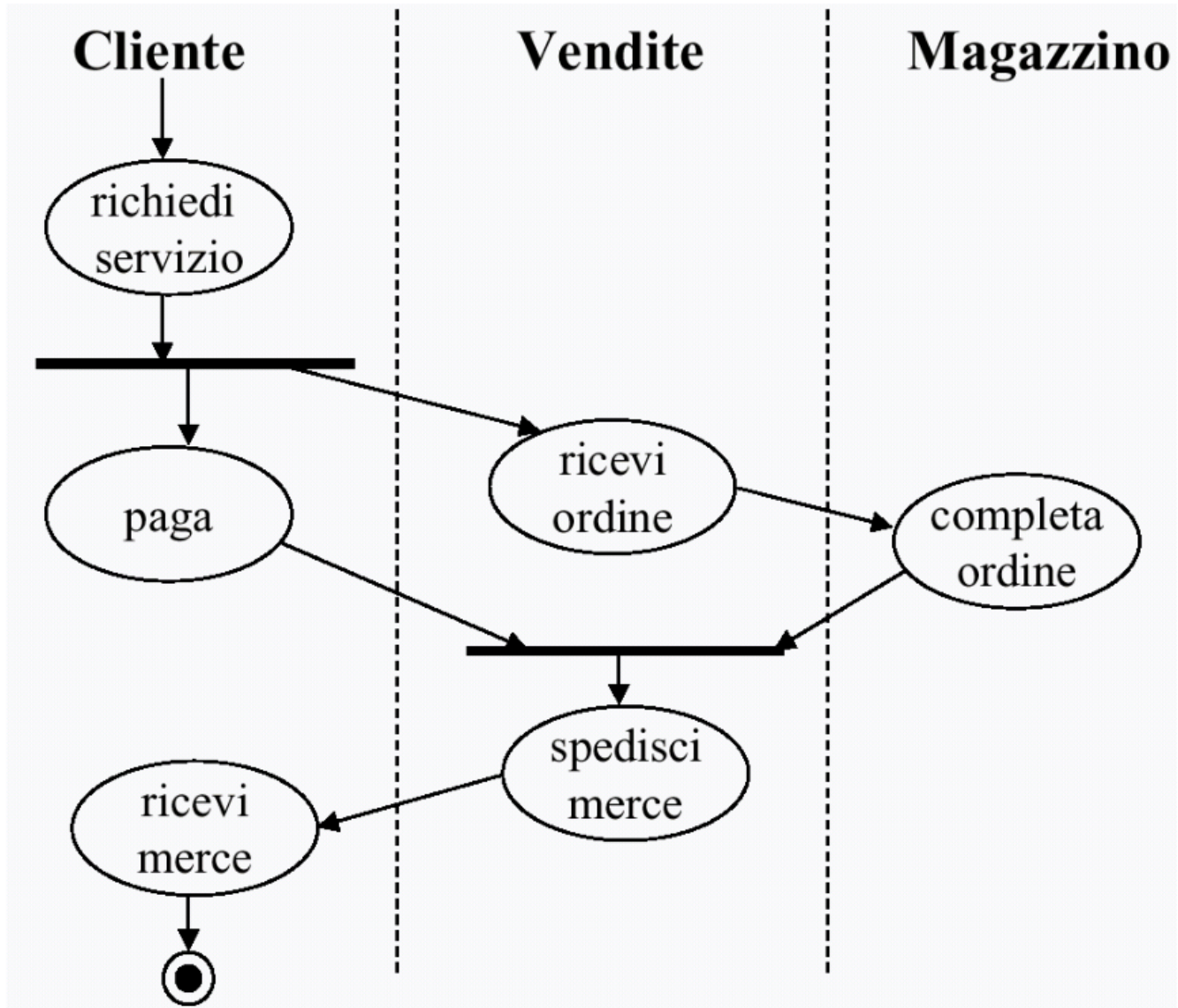
In un Business Model identificano le unità organizzative.

Per ogni oggetto responsabile di **action/activity** nel diagramma è definito un swimlane, identificato da un nome univoco nel diagramma:

- ▶ Le **action/activity** state sono divise in gruppi e ciascun gruppo è assegnato allo swimlane dell'oggetto responsabile per esse;
- ▶ L'ordine con cui gli swimlane si succedono non ha alcuna importanza;

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

15.4.1 Esempio Swimlane



15.5 Progettazione di un sistema

Insieme dei task svolti dall'ingegnere del SW per trasformare il modello di analisi nel modello di design del sistema.

15.6 Scopo del system design

Definire gli obiettivi di design del progetto.

Definire l'architettura del sistema, decomponendolo in sottosistemi più piccoli che possono essere realizzati da team individuale.

Selezionare le strategie per costruire il sistema, quali:

- ▶ Strategie SW/HW;
- ▶ Strategie relative alla gestione dei dati persistenti;
- ▶ Flusso di controllo globale;
- ▶ Politiche di controllo degli accessi.

15.7 Scopo dell'object design

Data l'architettura del Sistema, specificare il dettaglio realizzativo dei sottosistemi più piccoli.

Selezionare le strategie ottimali per l'implementazione:

- ▶ Design Pattern.

15.8 Output del SW design

Documento contenente:

- ▶ Chiara descrizione di ognuna delle strategie elencate in precedenza;
- ▶ Un insieme di modelli **statici/dinamici** del sistema che includano la decomposizione del sistema in sottosistemi
 - ◊ Formalizzano con Class Diagram, Sequence Diagram, StateChart, . . .;
- ▶ Un insieme di modelli **statici/dinamici** che descrivano la progettazione interna dei sottosistemi.

15.9 System Design

Nel system design i nostri interlocutori sono le squadre di programmatori che dovranno implementare il sistema:

- ▶ Chi leggerà i nostri documenti sarà un esperto di informatica;
- ▶ Linguaggio per esperti tecnici.

15.9.1 Attività di System design

Il system design è costituito da 3 macro-attività:

1. Identificare gli obiettivi di design

- ▶ Gli sviluppatori identificano quali caratteristiche di qualità dovrebbero essere ottimizzate e definiscono le priorità di tali caratteristiche;

2. Progettazione della decomposizione del sistema in sottosistemi

- ▶ Basandosi sugli use case ed i modelli di analisi, gli sviluppatori decompongono il sistema in parti più piccole. Utilizzano stili architetturali standard

3. Raffinare la decomposizione in sottosistemi per rispettare gli obiettivi di design

- ▶ La decomposizione iniziale di solito non soddisfa gli obiettivi di design. Gli sviluppatori la raffinano finché gli obiettivi non sono soddisfatti.

15.9.2 Identificare gli obiettivi qualitativi del sistema

È il primo passo del system design.

Identifica le qualità su cui deve essere focalizzato il sistema.

Molti design goal possono essere ricavati dai requisiti non funzionali o dal dominio di applicazione, altri sono forniti dal cliente.

È importante formalizzarli esplicitamente poiché ogni importante decisione di design deve essere fatta seguendo lo stesso insieme di criteri.

15.10 Criteri di design

Possiamo selezionare gli obiettivi obbiettivi di design da una lunga lista di qualità desiderabili.

I criteri sono organizzati in 5 gruppi:

- ▶ Performance → Tempo di risposta, Throughput, Requisiti di Memoria, . . . ;
- ▶ Affidabilità → Robustezza, Disponibilità, Tolleranza ai fault, Sicurezza, . . . ;
- ▶ Costi → Sviluppo, Manutenzione, Gestione, . . . ;
- ▶ Mantenimento → Estendibilità, Modificabilità, Adattabilità, Portabilità, . . . ;
- ▶ Usabilità.

15.11 Design Trade-offs

Quando definiamo gli obiettivi di design, spesso solo un piccolo sottoinsieme di questi criteri può essere tenuto in considerazione.

Gli sviluppatori devono dare delle priorità agli obiettivi di design, tenendo conto anche di aspetti manageriali, quali il rispetto dello schedule e del budget.

16 Lezione del 03-11

16.1 Concetti di base di buon design

Molti criteri di Design sono strettamente dipendenti dal problema trattato → La scelta del miglior bilanciamento tra i vincoli posti è un compito dell'analista, che opera in base alla propria **esperienza/sensibilità**.

Esistono dei Criteri di Design generici, che valgono per qualunque SW OO:

- ▶ Pensare in avanti → *Progettare per anticipare il cambiamento*;
- ▶ Decomposizione → Decomporre il sistema in parti più piccole (i *sottosistemi*)
 - ◊ Modellazione di sottosistemi
 - Subsystem (UML: Package) → Collezioni di classi, associazioni, operazioni e vincoli che sono correlati;
 - Java fornisce i package che sono costruiti per modellare i sottosistemi;
 - C++/C# hanno il costrutto di namespace per indicare lo stesso concetto;
- ▶ Indipendenza funzionale → La suddivisione in sottosistemi può essere guidata da 2 fattori qualitativi fondamentali
 - ◊ Coesione
 - ◊ Accoppiamento.

16.1.1 Coesione

È una misura di quanto siano fortemente relate e mirate le *responsabilità* di un modulo;

- ▶ Se ciascuna unità è responsabile di un singolo compito, diciamo che tale unità ha un'alta coesione (proprietà desiderabile del codice). Questo permette di
 - ◊ Comprendere meglio i ruoli di una classe;
 - ◊ Riusare una classe;
 - ◊ Manutenere una classe;
 - ◊ Limitare e focalizzare i cambiamenti nel codice;
 - ◊ Utilizzare nomi appropriati, efficaci, comunicativi.

Granularità della coesione:

- ▶ Coesione dei metodi → Un metodo dovrebbe essere responsabile di un solo compito ben definito;
- ▶ Coesione delle classi → Ogni classe dovrebbe rappresentare un singolo concetto ben definito.

Indicatori di un'alta coesione:

- ▶ Una classe ha delle responsabilità moderate, limitate ad una singola area funzionale;
- ▶ Collabora con altre classi per completare dei task;
- ▶ Ha un numero limitato di metodi, cioè di funzionalità altamente legate tra loro;
- ▶ Tutti i metodi sembrano appartenere ad uno stesso insieme, con un obiettivo globale simile;
- ▶ La classe è facile da comprendere.

16.1.2 Accoppiamento

È una misura su quanto fortemente una classe è connessa *con/ha conoscenza di/si basa* su altre classi.

2 o più classi si dicono *accoppiate* quando è impossibile riusare una senza dover riusare anche *la/le altra/e*.

Se 2 classi dipendono strettamente e per molti dettagli l'una dall'altra, diciamo che sono *fortemente accoppiate*.

Per un codice di qualità dobbiamo puntare ad un basso accoppiamento.

Un *basso accoppiamento* è una proprietà desiderabile del codice, poiché permette di:

- ▶ Capire il codice di una classe senza leggere i dettagli delle altre;
- ▶ Modificare una classe senza che le modifiche comportino conseguenze sulle altre classi;
- ▶ Riusare facilmente una classe senza dover importare anche altre classi.

Un basso accoppiamento migliora la manutenibilità del SW.

16.1.3 Tipi di accoppiamento in OO

Date 2 classi X e Y:

1. X ha un attributo di tipo Y;
2. X ha un metodo che possiede un elemento di tipo Y;
3. X è una sottoclasse (eventualmente indiretta) di Y;
4. X implementa un'interfaccia di tipo Y.

Lo scenario n° 3 è quello che porta al massimo accoppiamento in assoluto

16.1.4 Esempio

```

1 class Traveler{
2     Car c = new Car();
3     void startJourney(){
4         c.move();
    }
}
```

```

5     }
6 }
7
8 class Car{
9     void move(){
10         // logic ...
11     }
12 }
13
14 // Traveler ha un attributo di tipo Car, e quindi un forte accoppiamento.

```

16.1.5 Esempio (cont.)

```

1 class Traveler{
2     Vehicle v;
3     // Dependency Injection
4     public void setV(Vehicle v){
5         this.v = v;
6     }
7 }
8
9 Interface Vehicle{
10     void move();
11 }

```

```

1 class Car implements Vehicle{
2     public void move(){
3         // logic
4     }
5 }
6
7 class Bike implements Vehicle{
8     public void move(){
9         // logic
10    }
11 }

```

L'introduzione di un'interfaccia **Vehicle** ha spezzato completamente l'accoppiamento tra **Traveler** e **Car**. Conseguenze:

- ▶ Traveler è ora accoppiato con un'interfaccia;
- ▶ Tutte le implementazioni dell'interfaccia funzionano con Traveler, senza richiederne modifiche al codice;
- ▶ Sarà possibile usare in futuro implementazioni di Vehicle non ancora realizzate oggi (*anticipare il cambiamento*);
- ▶ È necessario un'entità esterna che faccia l'inject della reale implementazione dell'interfaccia;
- ▶ È aumentata la complessità del codice.

Il compito di scegliere quale veicolo istanziare viene demandato alla classe **Factory**, (design pattern Factory).

Questo mi permette in un successivo riutilizzo del codice (in un altro progetto) di potermi portar dietro solo 2/3 classi (in questo caso l'interfaccia **Vehicle**, la classe **Traveler** e opzionalmente la classe **Factory**) invece di n classi.

17 Lezione del 05-11

Esercitazione con Cutugno

18 Lezione del 08-11

Esercitazione con Cutugno

19 Lezione del 10-11

19.1 La legge di Demetra

Linea guida chiave per avere Basso accoppiamento.

Il nome deriva dal progetto Demetra (uno dei primi grossi sistemi OO).

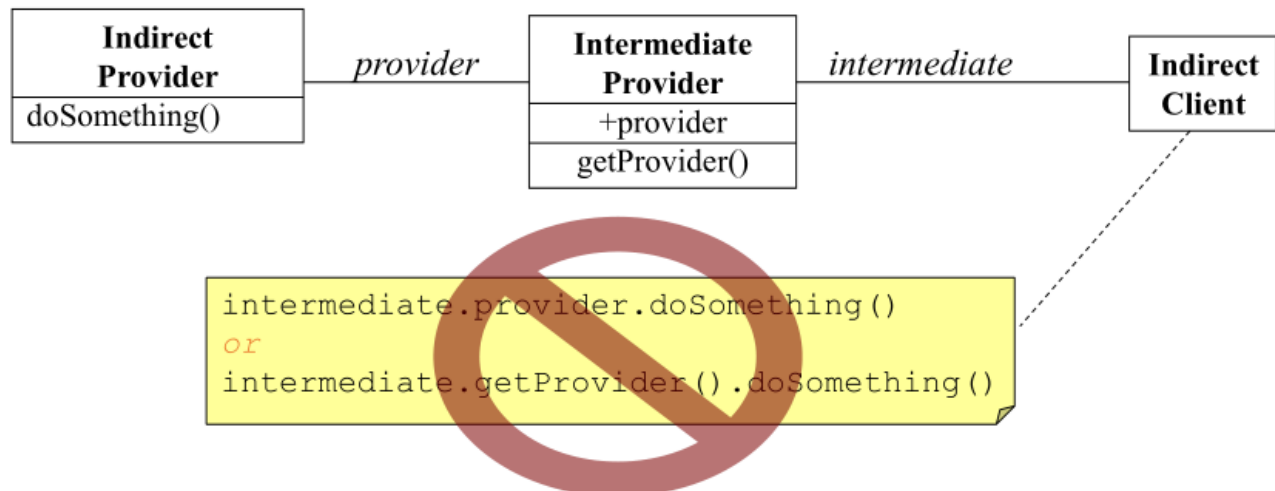
Concetto di base → Spingere al massimo l'information hiding:

- ▶ Evitare catene di . (`Foo.getA().getB().getC()...`);
- ▶ Più è lungo il percorso di chiamate attraversato dal programma, più esso è fragile a cambiamenti.

Un metodo dovrebbe mandare messaggi solo a:

- ▶ L'oggetto contenente (`this`);
- ▶ Una variabile di istanza di `this`;
- ▶ Un parametro del metodo;
- ▶ Un oggetto creato all'interno del metodo.

19.1.1 Violazione della legge di Demetra



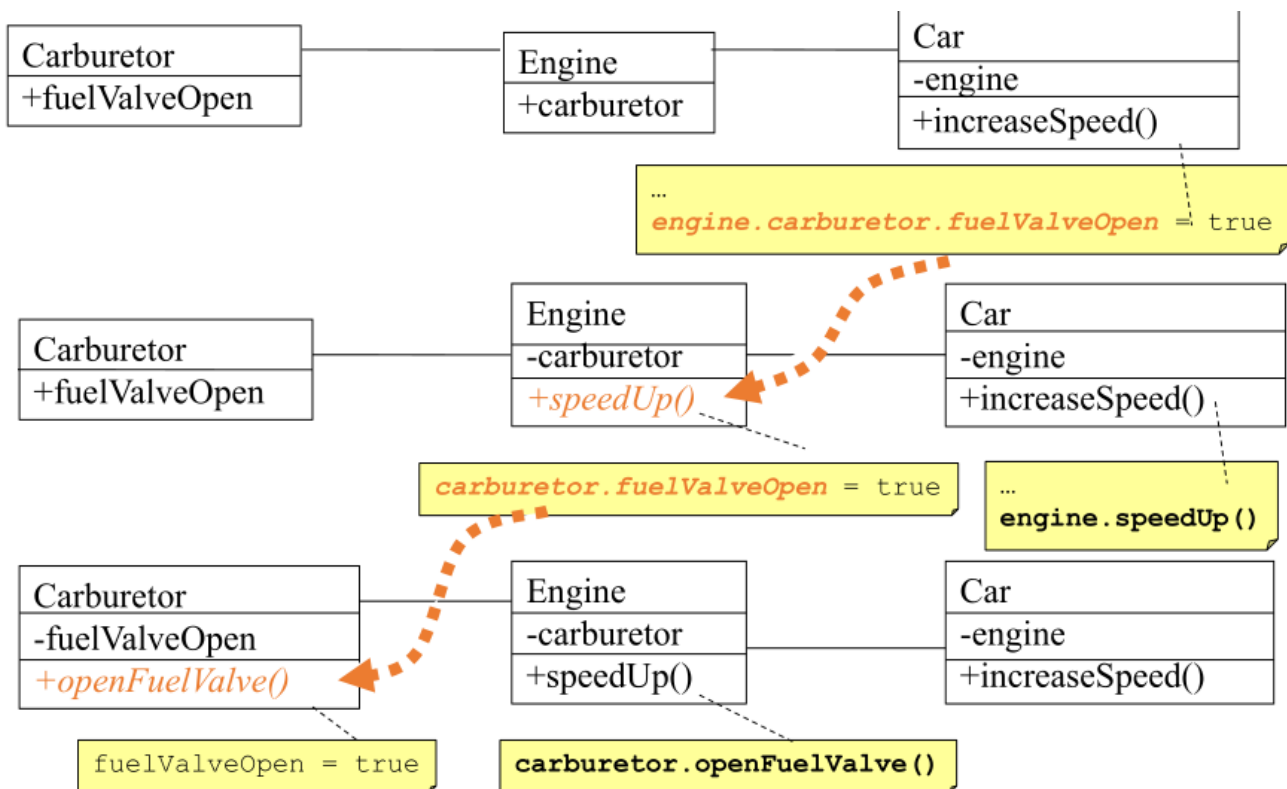
% Law of Demeter

A method *M* of an object *O* should invoke only the methods of the following kinds of objects:

1. *itself*;
2. *its parameters*;
3. *Any object it create or instantiates*;
4. *Its direct component objects*.

19.1.2 Eliminazione della coda della navigazione

Fondamentale risulta l'eliminazione della coda della navigazione:



19.2 SOLID Principles (Software Development is not a Jenga game)

Il concetto di fondo di questi principi è quello di evitare che al minimo cambiamento venga meno il corretto funzionamento di un software

1. Software *smells*

- Il codice risulta rigido → difficile da modificare (blocco monolitico);

2. Il sistema è fragile
 - ▶ Il cambiamento provoca la distruzione del SW;
3. Il sistema è immobile (non riutilizzabile);
4. Il sistema è oscuro;
5. Il sistema è inutilmente complesso;
6. Il sistema permette ripetizioni inutili.

I principi SOLID:

- ▶ Single Responsibility Principle
 - ◊ Una classe dovrebbe avere una sola responsabilità;
 - ◊ Una classe dovrebbe avere un solo motivo per cambiare;
 - ◊ La responsabilità è un motivo di cambiamento.
- ▶ Open/Closed Principle
 - ◊ Le entità SW dovrebbero essere pronte a recepire estensioni, ma chiuse a ricevere modifiche
 - ◊ Le classi dovrebbero essere scritte affinché possano essere estese senza farne modifiche
 - Estendere il comportamento del sistema;
 - ◊ Il design risulta più stabile;
 - ◊ Il codice risulta più semplice da testare;
 - ◊ Il cambiamento è più semplice da mantenere
- ▶ Liskov Substitution Principle
 - ◊ Le sottoclassi dovrebbero essere sostituibili alla classe genitrice;
- ▶ Se un programma utilizza una classe di base, allora posso cambiare la sua istanza con una sua sottoclasse senza generare inconsistenza;
- ▶ Interface Segregation Principle
 - ◊ Le interfacce andrebbero pensate quanto più minimali possibili;
 - ◊ Più interfacce sono preferibili a una singola interfaccia general purpose.
- ▶ Dependency Inversion Principle
 - ◊ Moduli di alto livello non dovrebbero dipendere da moduli di basso livello → entrambi dovrebbero dipendere da astrazioni;
 - ◊ Le astrazioni non dovrebbero dipendere sui dettagli;

19.2.1 Benefici del Single Responsibility Principle

- ▶ Riutilizzo delle classi;
- ▶ Pulizia nel codice;
- ▶ Chiarezza nei nomi scelti;
- ▶ Leggibilità.

19.3 Responsibility-driven design

Pensare alla progettazione di un sistema in termini di:

- ▶ Classe;
- ▶ Responsabilità;
- ▶ Collaborazioni.

Ciascuna classe dovrebbe avere delle responsabilità e dei ruoli ben definiti → Alta Coesione.

La classe che possiede i dati dovrebbe essere responsabile di processarli → Basso accoppiamento.

È facilitata dall'uso delle CRC Cards.

19.3.1 CRC Cards

Scheda da associare ad ogni classe, per rappresentare i seguenti concetti:

- ▶ Nome della classe;
- ▶ Responsabilità della Classe;
- ▶ Collaboratori della Classe.

Responsabilità → Conoscenza che la classe mantiene o servizi che la classe fornisce.

Collaboratore → Un'altra classe di cui è necessario sfruttare la conoscenza o i servizi, per ottemperare alla responsabilità sopra indicate

20 Lezione del 12-11

20.1 Organizzare sottosistemi in Architetture

Un'architettura è una astrazione di come si vuole organizzare un lavoro di un progetto SW.

Descrivo gli elementi chiave astrandomi.

L'architettura Software è la struttura di un sistema, che comprende gli elementi SW, le proprietà visibili esternamente (meccanismo di comunicazione) e la relazione tra loro.

20.2 Architettura a 3 livelli

È un'evoluzione di quella che era in precedenza il concetto di client-server.

20.3 Partizionamento del problema

Suddividere l'architettura in blocchi.

21 Lezione del 15-11

21.1 L'errore umano

% Definizione

Termine generico per comprendere tutti i casi in cui una sequenza pianificata di attività fisiche o mentali fallisce il suo scopo, e quando questo fallimento non possa essere attribuito all'intervento di qualche agente casuale.

Esistono 4 fondamentali tipi di errore:

- ▶ Azione intenzionale ma errata (*mistake*);
- ▶ Azione non intenzionale (*lapsus*);
- ▶ Azione spontanea;
- ▶ Azione involontaria.

21.2 Prevenzione

Prevenire l'errore significa progettare un sistema in modo che la possibilità di errori da parte dei suoi utenti sia minima.

In particolare, un'azione dell'utente non dovrebbe mai causare una caduta del sistema o un suo stato indefinito. Alcune tecniche molto diffuse per prevenire gli errori sono le seguenti:

- ▶ Diversificare le azioni dell'utente;
- ▶ Evitare comportamenti modalici;
- ▶ Usare funzioni obbligatorie;
- ▶ Imporre input vincolati;
- ▶ Non sovraccaricare la memoria a breve termine dell'utente;
- ▶ Richiedere conferme;
- ▶ Usare default inoffensivi.

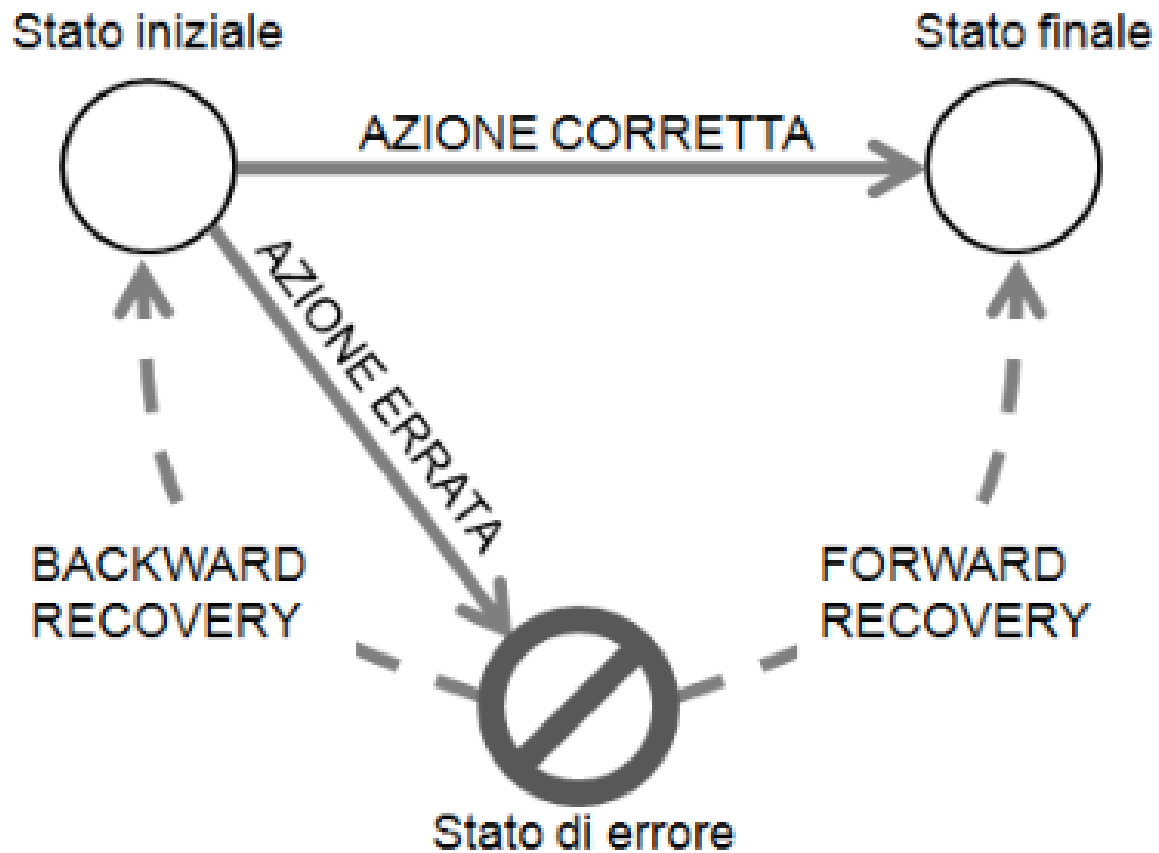
21.3 Diagnosi

Il sistema deve sempre controllare l'input e, nel caso che si riveli scorretto, dovrà fornire all'utente una spiegazione adeguata. Sono 3 le funzioni che un messaggio di errore ben progettato deve svolgere:

1. *Allertare*, cioè segnalare che qualcosa non va;
2. *Identificare*, cioè indicare che cosa non va, e perché;
3. *Dirigere*, cioè spiegare all'utente i passi che deve compiere per ripristinare una situazione corretta.

21.4 Correzione

Quando l'utente commette un errore, deve essere possibile correggerlo. Questo processo può essere attuato, secondo i casi, dall'utente o dal sistema, in modo cooperativo.



A partire da uno stato iniziale del sistema, l'utente compie un'azione.

Se l'azione è corretta, il sistema si porta nello stato desiderato, indicato nella figura come stato finale.

Se invece l'azione non è corretta, il sistema di porta in uno stato di errore. A partire da questo stato, il processo di correzione può avvenire con 2 strategie diverse, a seconda delle situazioni.

21.4.1 Backward recovery

Annulla le conseguenze negative dell'errore commesso, e riportare il sistema nello stato iniziale, dal quale l'utente potrà compiere, in maniera corretta l'azione che aveva sbagliato.

21.4.2 Forward recovery

Invece di tornare allo stato iniziale e ripetere da lì l'operazione, si cerca di raggiungere lo stato finale direttamente dallo stato di errore, senza prima tornare indietro, o richiedere l'intervento dell'utente.

Questo processo può essere effettuato automaticamente dal sistema.

Un sistema che attua sistematicamente strategie di Forward Recovery si dice *error tolerant*.

22 Lezione del 17-11

22.1 User Navigation

22.1.1 Back navigation

Due forme di navigazione:

- ▶ Back Navigation
 - ◊ Gestito dal SO;
 - ◊ Salvato nello back stack.
- ▶ Ancestral navigation
 - ◊ Gestito dall'applicazione (dentro le activity);
 - ◊ Compito del programmatore

22.1.2 Modifica del comportamento del tasto indietro del sistema

```
1 @Override
2 public void onBackPressed(){
3     // Add the back key handler here
4     return ;
5 }
```

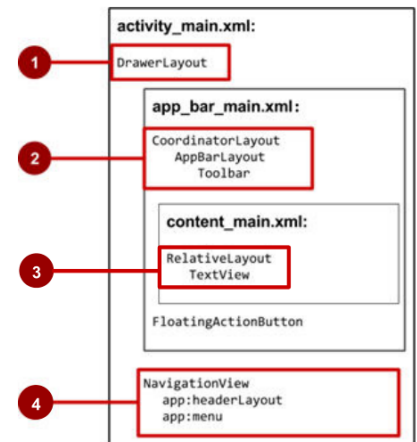
22.2 Hierarchical Navigation

22.2.1 Tipi di Hierarchical Navigation

- ▶ Descendant navigation
 - ◊ *Down from a parent screen to one of its children;*
- ▶ Ancestral navigation
 - ◊ *Up from a child or sibling screen to its parent;*
- ▶ Lateral navigation
 - ◊ *From one sibling to another sibling.*

22.2.2 Navigation drawer Activity Layout

1. `DrawerLayout` è la root view;
2. `CoordinatorLayout` contiene l'app bar layout con `Toolbar`;
3. App content screen layout;
4. `NavigationView` con layout per header e elementi settabili.



22.2.3 Ancestral navigation (Up button)

Permette allo user di passare da un figlio al suo parent

Dichiarazione (XML)

```
<activity android:name=".OrderActivity"
    android:label="@string/title_activity_order"
    android:parentActivityName="com.example.android.optionsmenuorderactivity.MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```


22.2.4 Lateral Navigation

- ▶ Tra fratelli;
- ▶ Da una lista di storie ad una lista di tab;
- ▶ Da una storia a una storia sotto lo stesso tab.

22.2.5 Step per implementare i tabs

1. Definire il tab layout con `TabLayout`;
2. Implementare un fragment e il suo layout per ogni tab;
3. Implementare un `PagerAdapter` da un `FragmentPagerAdapter` o un `FragmentStatePagerAdapter`;
4. Creare un'istanza per il tab layout;
5. Usare il `PagerAdapter` per gestire lo schermo (ogni schermo è un fragment);
6. Settare un listener per determinare quale tab sia stato cliccato.

```
<android.support.design.widget.TabLayout
android:id="@+id/tab_layout"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_below="@id/toolbar"
android:background="?attr/colorPrimary"
android:minHeight="?attr/actionBarSize"
android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">
```

23 Lezione del 19-11

Discussione su virtualizzazione e Cloud computing (per approfondimenti fare riferimento agli appunti di SO2, presente nel repository del 2° anno).

24 Lezione del 22-11

24.1 Valutare l'usabilità

In linea di principio possiamo affermare che esistono 2 tipi di valutazioni:

- ▶ Valutazioni euristiche;
- ▶ Valutazioni effettuate con il coinvolgimento dell'utente.

24.1.1 Valutazioni euristiche

- ▶ Visibilità dello stato del sistema → Il sistema dovrebbe sempre informare gli utenti su ciò che sta accadendo, mediante feedback appropriati in un tempo ragionevole;

- ▶ Corrispondenza tra mondo reale e il sistema;
- ▶ Libertà e controllo da parte degli utenti;
- ▶ Consistenza e standard;
- ▶ Prevenzione degli errori;
- ▶ Riconoscere piuttosto che ricordare;
- ▶ Flessibilità ed efficienza d'uso;
- ▶ Design minimalista ed estetico;
- ▶ Aiutare gli utenti a riconoscere gli errori, diagnosticarli e correggerli;
- ▶ Guida e documentazione.

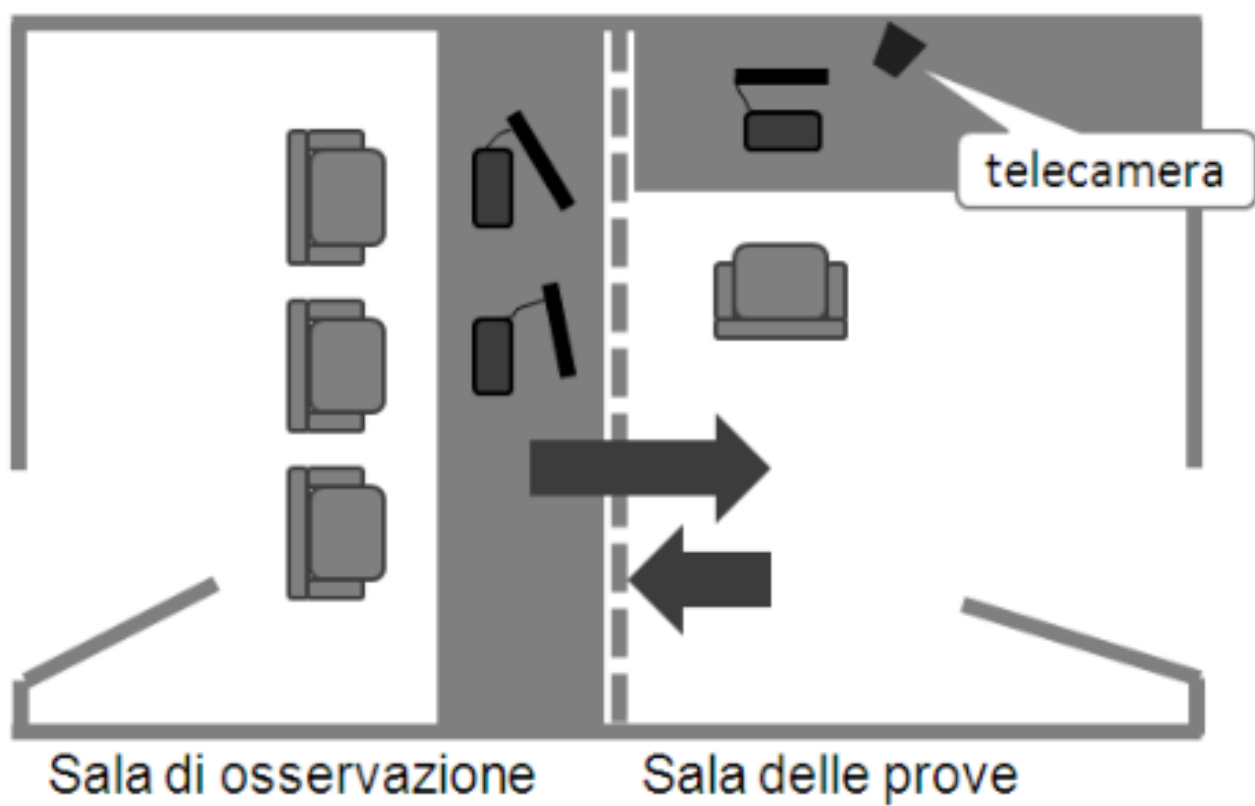
La valutazione euristica ha il vantaggio di essere relativamente poco costosa. Fornisce però risultati piuttosto soggettivi. Quanto più le euristiche sono generali, tanto più il risultato della valutazione dipenderà dall'esperienza, dalla sensibilità e, dalle preferenze personali del valutatore.

24.2 Test di usabilità

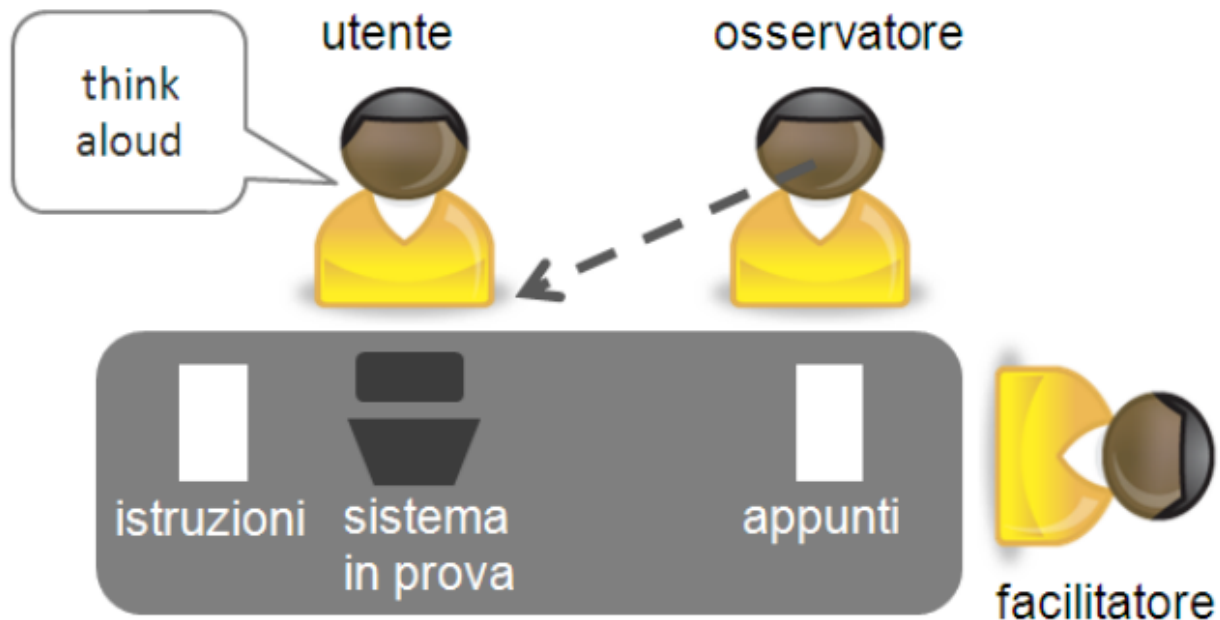
Consiste nel far eseguire a un gruppo di utenti dei compiti tipici di utilizzo del sistema in un ambiente controllato. Si sceglie un campione di utenti che sia rappresentativo della categoria di utenti cui il sistema si rivolge, e si chiede a tutti di svolgere, separatamente, gli stessi compiti.

Il ruolo degli osservatori è critico → Dovrebbero conoscere bene il sistema e aver eseguito correttamente i compiti chiesti agli utenti.

24.2.1 Laboratorio dell'usabilità



24.2.2 Test di usabilità informale



24.2.3 Test formativi e sommativi

I test formativi vengono utilizzati durante il ciclo iterativo di progettazione, per sottoporre i vari prototipi a prove d'uso con gli utenti, allo scopo di identificare i difetti e migliorarne l'usabilità. Questi test permettono di mettere immediatamente in luce i difetti macroscopici, che richiedono una parziale (o totale) riprogettazione dell'interfaccia.

I test sommativi invece sono test più completi dei primi → Si tende pertanto a cercare utenti con caratteristiche diverse, in modo da aumentare la probabilità di ottenere strategie di soluzione differenti.

In ogni caso, tutti gli utenti dovrebbero avere almeno un minimo interesse nelle funzioni del sistema. È sbagliato infatti, per *fare numero*, reclutare le persone più facilmente disponibili, senza altri accertamenti → Vanno selezionati dei potenziali clienti del sistema.

24.2.4 Test di compito e test di scenario

Nei test di compito, gli utenti svolgono singoli compiti, che permettono di esercitare funzioni specifiche del sistema. Questi test possono essere eseguiti anche quando il sistema non è completamente sviluppato.

Nei test di scenario invece, agli utenti viene indicato un obiettivo da raggiungere attraverso una serie di compiti elementari, senza indicarli esplicitamente. L'utente quindi dovrà impostare una propria strategia di azione.

24.3 Misure

Risulta utile inoltre raccogliere delle misure oggettive. Quelle più significative sono il tempo impiegato da ogni utente per l'esecuzione di ciascun compito, e il tasso di successo, cioè la percentuale di compiti che ciascuno riesce a portare a termine.

es.

	Compito1	Compito2	Compito3	Compito4	Compito5	Compito6
Utente1	F	F	S	F	F	S
Utente2	F	F	P	F	P	F
Utente3	S	F	S	S	P	S
Utente4	S	F	S	F	P	S

Dove:

- ▶ $S \rightarrow$ Successo;
- ▶ $F \rightarrow$ Fallimento;
- ▶ $P \rightarrow$ Successo parziale.

24.4 Come condurre un test di usabilità

Un test di usabilità viene condotto in 4 fasi successive:

1. Pianificazione (a inizio progetto);
2. Preparazione (requisiti);
3. Esecuzione (Utenti);
4. Analisi e Proposte (Rapporto di valutazione).

24.5 Test di usabilità - costi e benefici

I test di usabilità possono essere effettuati rapidamente e con costi relativamente contenuti.

I test con pochi utenti non sono significativi \rightarrow le persone sono ovviamente molto diverse l'una dall'altra.

Fondamentale per una buona documentazione è attenersi alla ISO 13407.

25 Lezione del 24-11

25.1 Cloud Computing - Modelli di servizio

- ▶ SaaS (Software as Service);

- ▶ PaaS (Platform as Service);
- ▶ IaaS (Infrastructure as Service).

25.2 Infrastruttura AWS

È una raccolta di servizi di computazione cloud.

25.2.1 Computazione e salvataggio dei dati

- ▶ Servizi (virtuali) a richiesta → EC2
 - ◇ Diversi tipi di istanze per diverse esigenze;
 - ◇ Pagamento a consumo → per EC2 si paga sul consumo in **secondi/ore**;
 - ◇ Il traffico dati non viene incluso;
 - ◇ La persistenza dei dati non è incluso;
 - ◇ Le alternative sono Virtual Machines (*Azure*) e Compute Engine (*Google Cloud*).
- ▶ Servizi di storage (S3, Glacier e simili).

26 Lezione del 26-11

Potenziati scelte in seguito alla laurea triennale (prima parte).

26.1 Verifica e Validazione

The software is done. We are just trying to get it to work...

Finito di scrivere il codice, possiamo dire di aver finito lo sviluppo del software? **Assolutamente no**, gli errori infatti possono succedere in ogni fase del ciclo di vita.

26.1.1 I bug

La storia dell'informatica è piena di casi di sistemi con comportamenti che deviano dalle specifiche.

Il primo esempio è quella della falena che entrando nel Pannello #F della Mark II portò a un cortocircuito, che portava la macchina a fare calcoli aritmetici in maniera errata.

26.1.2 Errori di progettazione

Casi famosi di questo tipo di errore è la NASA. Il primo esempio è la NASA Mars Climate Orbiter → fu persa mentre entrava nell'orbita di Marte (la causa era un errore insito nelle specifiche SW dettate dalla NASA).

26.1.3 Errori di coding

Un esempio di questo errore è **ESA Ariane 5**.

Un grosso errore fu quello di riportare in toto il codice scritto per il suo predecessore (che essendo più piccolo gestiva in maniera diversa la conversione in floating point).

26.1.4 Errori di valutazione sull'utilizzo del sistema

Anche in questo caso ricordiamo il NASA Spirit Rover → il robottino smise di comunicare con la Terra, qualche giorno dopo l'atterraggio (il robot era entrato in Fault mode, e per questo motivo si riavviava in continuazione). Era presente un problema all'intero del FS → era presente una quantità enorme di file di log, che portava a un crash del sistema.

26.1.5 Qualche errore del passato...

Errori di metodologie di sviluppo → Famoso è il caso dei 2 Boeing 737 MAX (Introduzione del MCAS). Per risparmiare:

- ▶ L'MCAS non viene descritta nel manuale di volo dei piloti;
- ▶ Un suo eventuale errore fu classificato come azzardo, invece di catastrofico → Nessuna ridondanza sul sensore su cui si basa il SW.

26.2 Verifica e Validazione

Collaudo del software, in modo tale che sia conforme alle specifiche e combaci con le richieste del cliente.

La fase di verifica e validazione serve ad accertare che il software rispetti i requisiti nella maniera dovuta.

La verifica può essere:

- ▶ statica, se effettuata senza eseguire codice;
- ▶ Dinamica;
- ▶ A run-time.

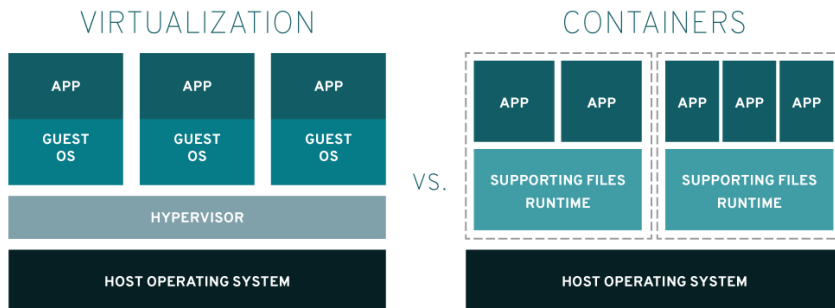
L'obiettivo di fondo è quello di avere software affidabile, che svolga i compiti come previsto.

Si cerca sempre di creare un software **Fault Tolerant**. Introduciamo a questo scopo:

- ▶ Ridondanza;
- ▶ Transazioni.

27 Lezione del 29-11

27.1 I container - Approccio a micro servizi



Un container è un set di 1 o più processi che eseguono isolati dal resto del sistema.

Ricordiamo 2 diverse tecnologie di containerizzazione:

- ▶ **Docker** (lato consumer);
- ▶ **Kubernetes** (lato enterprise);

Docker è un tool per la costruzione automatica di applicazioni in container, in modo tale che possa girare in maniera veloce in una grande varietà di ambienti.

Kubernetes è una piattaforma portabile ed estensibile per la gestione di workload containerizzati e servizi. È molto spesso affiancato a Docker per un migliore controllo e implementazione di applicazioni containerizzate.

27.2 Strategie per migliorare l'affidabilità del SW

Il testing non va inteso come una fase, ma come un lifestyle:

Il testing e l'attività di analisi partono fin dall'ingegneria dei requisiti

27.3 Terminologia

- ▶ **Affidabilità** → La misura di successo con cui il comportamento osservato di un sistema è conforme ad una certa specifica del relativo comportamento;
- ▶ **Fallimento** → Qualsiasi deviazione del comportamento osservato dal comportamento specificato;
- ▶ **Stato di Errore** → Il sistema è in uno stato tale che ogni ulteriore elaborazione da parte del sistema porta ad un fallimento;
- ▶ **Difetto** → La causa di un errore.

27.3.1 Fault e Failure

```
1 // Porta a delle failure nel caso in cui venga passato un numero come 3 (Ottterrò infatti 9 e non 6  
  come ci si aspetta)  
2 int raddoppia(int x){  
3     int y;
```



```
4   y = x*x;  
5   return (y);  
6 }
```

Non tutti i fault portano a failure. Una failure può essere generata da più fault. Una fault può generare diverse failure

27.3.2 Test e Casi di test

Un programma è *stressato* da un caso di test (insieme di dati di input), o *test case*.

Una *batteria di test* è un insieme di test.

L'esecuzione del test consiste nell'esecuzione del programma per tutti i casi di test.

Un test ha successo se rileva uno o più malfunzionamenti del programma.

27.3.3 L'oracolo

Condizione necessaria per effettuare un test è conoscere il comportamento atteso per poterlo confrontare con quello osservato.

L'oracolo conosce il comportamento atteso per ogni caso di prova.

Oracolo umano:

- ▶ Si basa sulle specifiche o sul giudizio.

Oracolo automatico:

- ▶ Generato dalle specifiche (formali);
- ▶ Stesso SW ma sviluppato da altri;
- ▶ Versione precedente (test di regressione).

27.3.4 Problemi indecidibile

Un problema è detto indecidibile se è possibile dimostrare che non esistono algoritmi che lo risolvono.

27.3.5 Goal della verifica e della validazione

Verifica e validazione dovrebbero stabilire un certo livello di fiducia che il SW è adatto ai suoi scopi (fa quello che deve fare).

Questo però non esclude assenza di difetti. Significa invece che deve essere abbastanza buono per l'uso per il quale è stato pensato e proprio il tipo di uso determinerà il livello di fiducia richiesto. Questo dipende dagli scopi e funzioni del sistema, dalle attese degli utenti e dal *time-to-market*.

27.4 Verifica statica e dinamica

27.4.1 Verifica statica

È basata su tecniche di analisi statica del SW senza ricorso alla esecuzione del codice. È un processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto, documentazione.

Tecniche:

- ▶ Review
 - ◊ Ispezione manuale del codice sorgente;
- ▶ Ispezione
 - ◊ Tecnica completamente manuale con lo scopo di trovare e correggere errori;
- ▶ Verifica formale;
- ▶ Esecuzione simbolica;
- ▶ ...

27.4.2 Verifica Statica - Le review

Sono le ispezioni manuali del codice sorgente. Di 2 tipi:

- ▶ **Walkthrough** → Lo sviluppatore presenta informalmente le API, il codice, la documentazione associata delle componenti al team di review;
- ▶ **Inspection** → Simile alla precedente, ma la presentazione delle unità è formale
 - ◊ Lo sviluppatore non può presentare gli artefatti (Questo viene fatto dal team di review che è responsabile del controllo delle interfacce e del codice dei requisiti);
 - ◊ Controlla l'efficienza degli algoritmi con le richieste non funzionali;
 - ◊ Lo sviluppatore interviene solo se si richiedono chiarimenti.

27.4.3 Analisi informali

Analizzare la specifica dei requisiti o il codice attraverso una simulazione manuale.

27.4.4 Verifica dinamica

Controllare che il SW computi correttamente (lanciare quindi in esecuzione il codice).

Eseguo il codice e ne valuto il comportamento:

- ▶ Testing;
- ▶ Debugging → Definisco degli input e ne controllo gli output.

27.5 Ispezioni

Le Ispezioni trovano fault in una componente rivedendo il codice sorgente in meeting formali.

È condotta da un team di sviluppatori, incluso l'autore della componente, un moderatore e uno o più revisori che trovano i bug nella componente

Tecniche di ispezione di codice possono rilevare ed eliminare anomalie e rendere più precisi i risultati. È una tecnica completamente manuale per trovare e correggere errori.

È estendibile ad altri artefatti seguendo principi organizzativi analoghi.

27.5.1 Ruoli

- ▶ Moderatore → Tipicamente proviene da un altro progetto. Presiede le sedute, sceglie i partecipanti, controlla il processo;
- ▶ Lettori, addetti al test → Leggono il codice al gruppo, cercando difetti;
- ▶ Autori → Partecipante passivo. Risponde a domande quando richiesto.

27.5.2 Fasi

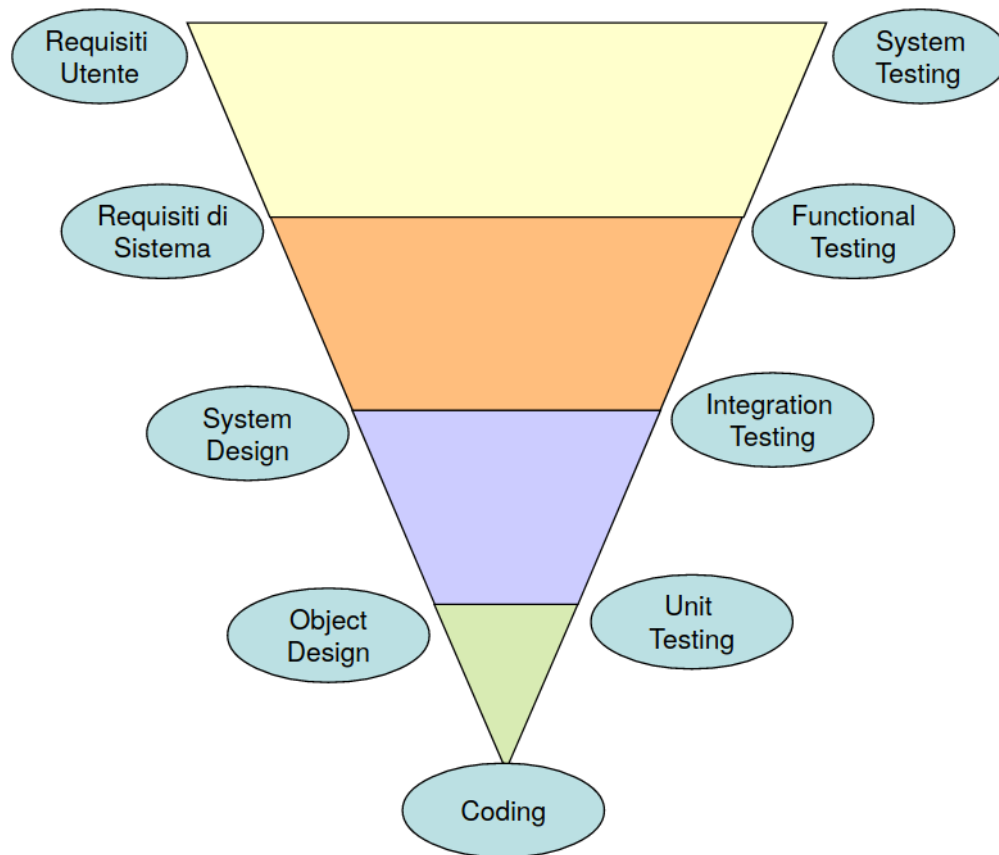
1. Pianificazione;
2. Fasi preliminari;
3. Preparazione;
4. Ispezione;
5. Lavoro a valle;
6. Possibile re-ispezione.

27.6 Testing

Eseguire il blocco di codice e controllarne il comportamento. Avviene a vari livelli:

- ▶ **Unit Testing** → Trovare differenze tra Object Design Model e corrispondente componente;
- ▶ **Integration Testing** → Trovare differenze tra System Design model e sottoinsieme integrato di sottosistemi.
- ▶ **Functional Testing** → Trovare differenze tra Use Case Model e il Sistema.
- ▶ **System Testing** → Trovare differenze tra requisiti non funzionali e il Sistema.

27.6.1 Modello di sviluppo a V



28 Lezione del 01-12

28.1 Unit testing - junit

Unit testing significa scrivere codice che controlla (testa) altro codice.

Eseguire test di metodi è una cosa relativamente veloce.

Eseguire invece test manuali è più oneroso sia per tempo, che per costi (Ice-cream Cone Anti-pattern).

28.1.1 Vantaggi dell'Unit testing

- ▶ Rilevamento in anticipo di errori;
- ▶ Forte manutenibilità;
- ▶ Ne migliora il design;

- ▶ Aiuta la determinazione delle specifiche;
- ▶ Produce documentazione.

28.2 JUnit

Framework open-source per scrivere e far girare test in Java.

Fu scritto inizialmente da **Erich Gamma** e **Kent Beck**.

È una dei framework della famiglia di **xUnit**.

Definisce una famiglia di metodi (gli **assert**).

28.2.1 Componenti di un test JUnit

Classi di test:

- ▶ Una classe di test JUnit deve estendere una classe denominata `TestCase` della libreria.

Metodi di test:

- ▶ Ogni metodo di test deve avere un nome che inizia con la parola *test*;
- ▶ Possono essere realizzati un numero qualsiasi di metodi di test.

Una classe di test, contiene anche:

- ▶ Un metodo **setup()** che viene eseguito prima dell'esecuzione di ogni test
 - ◇ Utile per eseguire operazioni preliminari necessarie per poter soddisfare le precondizioni comuni a più di un caso di test;
- ▶ Un metodo **teardown()** che viene eseguito dopo ogni caso di test
 - ◇ Utile per eseguire operazioni finali necessarie per poter soddisfare le precondizioni comuni a più di un caso di test.

È importante rimettere sempre l'applicazione e le sue risorse nello stato di partenza al termine di ogni test in quanto JUnit non garantisce mai in che ordine verranno eseguiti i test.

28.2.2 Asserzioni

Affermazione che può essere vera o falsa.

I risultati attesi sono documentati con delle *asserzioni* esplicite, non con delle stampe (che richiedono dispendiose ispezioni visuali dei risultati).

Se l'asserzione è:

- ▶ Vera → il test è andato a buon fine;
- ▶ Falsa → il test è fallito ed il codice testato non si comporta come atteso, quindi c'è un errore a tempo dinamico.

Le asserzioni sono utilizzate sia per verificare gli oracoli che le post condizioni. Potrebbero essere utilizzate anche per verificare le precondizioni → Se questa fallisce, il test non viene proprio eseguito (e viene riportato come fallito).

Se un'asserzione non è vera il test-case fallisce:

- ▶ `assertNull()` → afferma che il suo argomento è nullo (fallisce se non lo è);
- ▶ `assertEquals()` → afferma che il suo secondo argomento è `equals()` al primo argomento (il valore atteso);
- ▶ Altre varianti
 - ◇ `assertNotNull()`;
 - ◇ `assertTrue()`;
 - ◇ `assertFalse()`;
 - ◇ `assertSame()`;
 - ◇ `assertIterableEquals()`;
 - ◇ `assertTimeout()`.

28.2.3 assertEquals()

```
1 assertEquals(Object expected, Object actual);
```

Dove:

- ▶ `expected` è il valore atteso;
- ▶ `actual` è il valore effettivamente rilevato.

Va a buon fine se e solo se `expected.equals(actual)` restituisce true.

28.2.4 Test Suite

Meccanismo usato per raggruppare logicamente test ed eseguirli assieme.

L'annotazione `@SuiteClasses` raggruppa una lista di classi, passate come parametro, in una test suite.

L'annotazione `@RunWith` permetta di impostare diversi esecutori di test → Fondamentale ad esempio per indicare l'utilizzo di X versione di JUnit o se si vuole implementare dei propri runner.

28.3 JUnit 5 - Arrange, Act, Assert

```
1 import org.junit.jupiter.Test;
2 import static org.junit.jupiter.api.Assertions.*;
3 class GenericTest{
4     @Test // Test Annotation
5     void shouldPerformOperationExpected(){
6         ClassUnderTest cut = new ClassUnderTest();
7         cut.initialize(list, of, parameters); // Arrange
8         cut.performOperation();
9         assertTrue(cut.someCondition());
10        assertFalse(cut.someOtherCondition());
11    }
12 }
```

28.4 JUnit - Annotazioni

- ▶ @Test → Usato per indicare quali metodi sono testare;
- ▶ @BeforeAll / @AfterAll → Denota metodi che andrebbero eseguiti **prima/dopo** tutti metodi con l'annotazione @Test;
- ▶ @BeforeEach / @AfterEach → Denota metodi ce andrebbero eseguiti **prima/dopo** ogni metodo con l'annotazione @Test;

28.5 JUnit - Test con eccezioni

Facciamo uso di:

```
1 Assertions.assertThrows(<Tipo di Eccezione>, <Runnable>)
2
3 // ESEMPIO
4 Assertions.assertThrows(IllegalArgumentException.class, () -> c.Moltiplica(4,-3));
5
6 var person = new Person("name", "surname", 10);
7
8 assertAll(
9     () -> assertThat(person.name, is("name")),
10    () -> assertThat(person.surname, is("surname")),
11    () -> assertThat(person.age, is(10))
12 );
```

28.6 JUnit - Test parametrizzati

Definisco dei test parametrizzati, in modo da dare un range di valori in cui JUnit testa.

28.7 JUnit - Terminologia

Driver → Classe che invoca i test di una determinata Unit.

Nel caso in cui l'Unit abbia una **dipendenza/associazione** con altre classi → Se sto facendo Unit Testing devo crearne una sua versione simulata (**stub/mock**).

In questo modo posso sganciare l'Unit da una catena di dipendenza.

Driver e Stub insieme formano lo **Scaffolding** (impalcatura) → ambiente per fare Unit Testing.

Un basso accoppiamento porta ad una semplificazione nella fase di testing.

29 Lezione del 03-12

29.1 Log4j

Strumento di audit di logging, in cui il progettista in maniera trasparente registra azioni dell'utente.

[Log4j](#) prevede delle API, per cui è possibile accumulare informazioni.

Per la documentazione completa, vedi [qui](#).

Fondamentale per il logging è il timestamp.

29.2 Logcat

Tool da riga di comando che salva un log dei messaggi di sistema (inclusi stack).

Proprio per l'elevato contenuto di informazioni nel logging è importante far uso di tag e delle priorità dei log.

29.3 Google Analytics - Firebase

Sistema integrato che raccoglie una serie di facility già pronte.

Utilizza un pagamento basato su notifiche.

Fornisce rapporti sugli eventi che ti aiutano a capire come gli utenti interagiscono con la tua app. Gestisce una serie informazioni su:

- ▶ Crash report;
- ▶ Numero di utenti che utilizzano e hanno installato l'app.

29.4 RERAN - Timing and Touch-Sensitive Record and Replay for Android

Tool che permette di intercettare le azioni sul sistema android, gestite dal sistema (Scroll, pinch, swipe), in modo da calcolarne in timing (articolo [completo](#)).

29.5 Seminario con Engineering

Contatti:

- ▶ Giacomo Petillo → Senior Solution Developer
- ▶ Mail → giacomo.petillo@eng.it

29.5.1 Scenario - Cliente Interno

È il caso della sezione `work with us` di [Engineering](#)

29.5.2 Censimento dei requisiti - JIRA

Piattaforma che tiene traccia di problemi e requisiti di un software rilasciato (Strumento molto utile per gestione dei requisiti).

Permette una comunicazione facilitata tra clienti (che segnalano issue) e software house.

29.5.3 Letture interessanti

- ▶ Metodologie di progettazione e sviluppo
 - ◇ Clean Code;
 - ◇ Design Patterns;
- ▶ Spunti per buona programmazione
 - ◇ Seriously Good Software (Del prof Faella).

29.5.4 Ingegneria del Software e Pubblica Amministrazione

Pubblica Amministrazione Centrale:

- ▶ Ministeri;
- ▶ Istituti Nazionali.

Pubblica Amministrazione Locale:

- ▶ Regioni;

- ▶ Comuni.

Principali Attività:

- ▶ Gestionali;
- ▶ Portali per la comunicazione con i cittadini.

Problematiche:

- ▶ Cliente che ha un rapporto complesso con la tecnologia (es. Modifica delle policy)
 - ◇ Nella pubblica istruzione Italiana infatti presenta una grossa carenza (ignoranza del) nel campo informatico.

30 Lezione del 06-12

30.1 Seminario con RE:Lab

Designer, esperti in Human factors, psicologi e ingegneri lavorano per sviluppare sistemi di interazione tra uomo e tecnologia.

Si lavora per step incrementali:

1. Concetto;
2. Prototipo;
3. Sviluppo HW/SW pronto per essere messo in produzione;

31 Lezione del 10-12

31.1 Seminario con NTTData

In NTTData si lavora per Task e servizi → gruppo di persone distribuite in base alle necessità dei progetti (realtà dinamica di un lavoro).

Non si lavora su progetti sviluppati da NTTData, ma si ha la gestione su SW sviluppato da altri.

31.1.1 Aree di mercato

- ▶ Energy & Utilities;
- ▶ Media;
- ▶ Telecomunicazione;
- ▶ Retail;

- ▶ Banking;
- ▶ Insurance;
- ▶ Manifatturiero;
- ▶ Settore Pubblico.

31.1.2 Quality Assurance

È un modo per prevenire errori o difetti nella creazione del prodotto, con lo scopo di evitare problemi nel rilascio del prodotto.
Attori coinvolti:

- ▶ Sviluppo → team che supporta e sviluppa un SW;
- ▶ Quality Assurance → Team che contribuisce alla diagnostica;
- ▶ Utenti → Utilizzatori finali del SW.

31.1.3 Attività in ambito

1. Fase di pianificazione;
2. Studio dei requisiti;
3. Fase di analisi (Verifica);
4. Progettazione;
5. Fase di implementazione;
6. Esecuzione (Convalida);
7. Supporto UAT → Fase in cui viene eseguito supporto agli utenti per lo **user acceptance test**
8. Fase di reportistica;
9. Automazione e performance

31.1.4 Bugs hunting

Defect → **Difetto** + **Failure**. Si intende un malfunzionamento SW. Non si fa distinzione fra causa (difetto) ed effetto (failure).

31.1.5 Pianificazione e monitoraggio

Fondamentale è la pianificazione dei compiti da svolgere e il loro monitoraggio per essere più efficaci (considerando le tempistiche stringenti che un SW richiede).

31.1.6 Un uso pratico

Creazione di un nuovo contratto con particolari caratteristiche. Il processo si compone di:

- ▶ Creazione di un nuovo contratto;

- ▶ Attivazione sui sistemi di back-end del nuovo contratto;
- ▶ Corretta fatturazione.

31.1.7 Test di automazione

Test che vengono sviluppati su SW e contesti più consolidati e maturi.

Vantaggi:

- ▶ Rapidità;
- ▶ Riduzione dei costi;
- ▶ Evitare lavori tediosi;
- ▶ Lavorare con molti input.

31.1.8 Competenze del Tester Automation

DevOps → Il tester automation deve avere competenze di programmazione (necessarie per lo sviluppo degli script).

Il DevOps si pone infatti a metà fra lo sviluppatore ed il tester.

Non si fa in questo caso uno Unit test (che è compito dello sviluppatore), ma un testing end-to-end.

31.1.9 Linguaggio e framework

La scelta del linguaggio da utilizzare per lo sviluppo degli script è legata alle richieste del cliente, le quali tengono conto di varie esigenze.

- ▶ Tecnologie da automatizzare;
- ▶ Utilizzo di tool già conosciuti o consolidate dal cliente;
- ▶ Esigenze economiche;
- ▶ Studio di fattibilità del tool da utilizzare.

Ricordiamo per la parte desktop:

- ▶ [HP UFT](#);
- ▶ [UiPath](#);
- ▶ [Selenium](#).

Ricordiamo per la parte mobile:

- ▶ [Appium](#);
- ▶ [Katalon](#).

31.1.10 Test di performance

Verificano le prestazioni di un sistema.

Serie di stress test, che provano a far crashare il sistema.

Serie di load test, in cui si verificano le performance del sistema con un carico normale.

Serie di endurance test, in cui si carica il sistema con un carico per un lungo periodo.

Le principali attività sono:

- ▶ Sviluppo;
- ▶ Esecuzione;
- ▶ Analisi.

32 Lezione del 13-12

32.1 Strategie per il testing

2 principali macro-strategie di testing:

- ▶ Testing Black-Box → Definizione dei casi di test fondata sui requisiti dell'unità da testare;
- ▶ Testing White-Box α Definizione dei casi di test è fondata sulla struttura del codice dell'unità da testare

32.2 Testing Black Box

La struttura interna del codice non viene tenuta in considerazione (la Unit è una Black Box).

Attività principale di generazione di casi di test:

- ▶ È scalabile (usabile per diversi livelli di testing);
- ▶ Non può rilevare difetti a funzionalità extra rispetto alle specifiche;

La definizione dei casi di test prevede 5 passi:

- ▶ Identificare i metodi da testare;
- ▶ Trovare i parametri sui quali giocare;
- ▶ Suddividere il dominio degli input;
- ▶ Identificare i valori secondo un certo criterio;
- ▶ Eventualmente raffinamento.

32.3 Classi di equivalenza (Input Space Partitioning)

È una strategia di testing SW che divide i parametri di una Unit in partizioni e classi di equivalenza, dalle quali si possano derivare **test cases**.

32.4 Equivalence Partitioning

- ▶ Idea → Se un caso di test non rileva fallimenti per un elemento di una classe di equivalenza, allora probabilmente non rileva fallimento per ogni altro elemento della classe;
- ▶ Non serve quindi scrivere un caso di test per ogni valore della classe ma sono per un suo rappresentante;
- ▶ La validità dell'assunto dipende dalla ragionevole progettazione della partizione del dominio in classi di equivalenza.

32.4.1 Dominio di partizionamento

Dato un dominio D , lo schema di partizioni q di D definisce una serie di blocchi, che devono soddisfare 2 proprietà:

- ▶ I blocchi devono essere ben separati (no overlap);
- ▶ I blocchi devono coprire l'intero dominio D (complete).

32.4.2 Identificare le Classi di equivalenza

Se la variabile di input è un elemento di un insieme discreto (enumerazione)

- ▶ Una classe valida per ogni elemento dell'insieme, una non valida per un elemento non appartenente;
- ▶ Include il caso di valori specifici;
- ▶ Include il caso di valori booleani.

Se la variabile di input assume valori validi in un intervallo di valori di un dominio ordinato:

- ▶ Almeno una classe valida per valori interni all'intervallo, una non valida per valori non inferiori al minimo, e una non valida per valori superiori al massimo.

32.4.3 Approcci all'identificazione delle partizioni

- ▶ Approccio **Interface-based**
 - ◇ Si sviluppano caratteristiche direttamente da parametri di input individuali;
 - ◇ Applicazioni semplici;
 - ◇ Parzialmente automatizzabili in qualche situazione
- ▶ Approccio basato sulle **funzionalità**
 - ◇ Si sviluppano caratteristiche da una vista del comportamento del programma testato;
 - ◇ Difficile da sviluppare (richiede uno sviluppo di design maggiore);
 - ◇ Può risultare in un testing migliore, o un minore testing effettivo.

32.5 Definizione di Test Case

Le classi di equivalenza individuate devono essere utilizzate per identificare casi di test che:

- ▶ Minimizzino il numero complessivo di test;
- ▶ Risultino significativi (affidabili).

Euristiche di base:

- ▶ Si devono individuare tanti casi di test da coprire tutte le classi di equivalenza valide, con il vincolo che ciascun caso di test comprenda il maggior numero possibile di classi valide ancora scoperte;
- ▶ Si devono individuare tanti casi di test da coprire tutte le classi di equivalenza non valide, con il vincolo che ciascun caso di test copra una ed una sola delle classi non valide.

Nel caso di funzioni con più parametri si procede con una combinazione delle classi di equivalenza delle varie caratteristiche seguendo opportuni criteri di copertura delle classi legate alle caratteristiche.

Diverse strategie per il test case:

- ◊ **WECT** (Weak Equivalence Class Testing) → Per ogni classe di equivalenza ci deve essere un Test case che usa un valore da quella classe di equivalenza
 - Se possibile il valore nominale o il valore medio della classe;
 - Prende il nome anche di **Each Choice Coverage** (ECC);
 - Il numero di test è il numero delle classi di equivalenza nella maggiore partizione;
 - È la minima copertura considerabile → Non garantisce un livello adeguato di test per la maggior parte delle situazioni (non considera l'eventuale correlazione tra i parametri di input);
- ▶ **Pair-Wise Coverage** → Un valore da ciascun blocco per ogni caratteristica deve essere combinata con un valore da ogni blocco per ogni altra caratteristica;
 - ◊ Il numero dei test è come minimo il prodotto di 2 larghe caratteristiche;
- ▶ **SECT** (Strong Equivalence Class Testing) → Prodotto cartesiano delle classi di equivalenza
 - ◊ Garantisce un livello di test approfondito.

33 Lezione del 15-12

33.1 Testing dei valori limite (boundary values)

Con le classi di equivalenza si partiziona il dominio di ingresso assumendo che il comportamento del programma su input della stessa classe sia simile.

Con i criteri **WECT** e **SECT** si testano tipicamente i valori medi di ogni classe di equivalenza → Tipici errori di programmazione capitano però spesso al limite di classi diverse. Il testing dei valori limite si focalizza su questo aspetto.

33.1.1 Idee di base

Verificare i valori della variabile di input (per ogni classe di equivalenza) al:

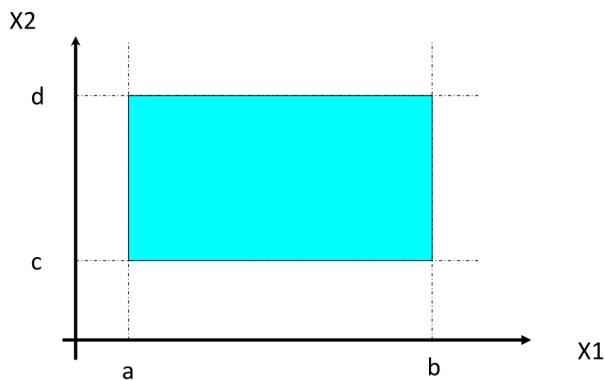
- ▶ Minimo;
- ▶ Immediatamente sopra al minimo;
- ▶ Valore intermedio (nominale);
- ▶ Immediatamente sotto il massimo e al massimo;

Per convenzione: `min`, `min+`, `nom`, `max-`, `max`

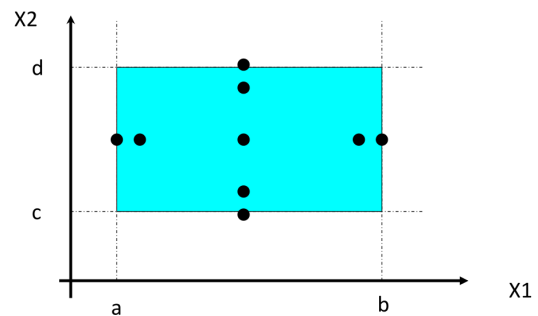
Approccio → Per ogni variabile di input v_i

v_i spazia tra questi 4 valori, mentre tutte le altre sono mantenute al loro valore nominale.

33.2 Classi di equivalenza dell'input della funzione F e suo Boundary Analysis Test Cases



Test set1 = {< x_{1nom} , x_{2min} >, < x_{1nom} , x_{2min+} >, < x_{1nom} , x_{2nom} >, < x_{1nom} , x_{2max-} >, < x_{1nom} , x_{2max} >, < x_{1min} , x_{2nom} >, < x_{1min+} , x_{2nom} >, < x_{1max-} , x_{2nom} >, < x_{1max} , x_{2nom} >}



33.3 Caso generale e Limitazioni

Una funzione con n variabili richiede $4n + 1$ casi di test. Funziona bene con variabili che rappresentano quantità fisiche limitate.

Non considera la natura della funzione e il significato delle variabili. Tecnica rudimentale che tende al test di robustezza (che esegue $6n + 1$ casi di test, date n variabili di input).

33.4 Worst Case Testing

La tecnica dei valori limite estende notevolmente la copertura rispetto al **SECT**, ma non testa mai più variabili contemporaneamente ai valori limite.

Il WCT è più profonda del Boundary Testing, ma molto più costosa → 5^n casi di test.

33.5 Gerarchia

Per n parametri di input:

- ▶ Boundary Value testing $\rightarrow 4n + 1$;
- ▶ Testing robusto $\rightarrow 6n + 1$;
- ▶ Caso peggiore per il boundary Value $\rightarrow 5^n$;
- ▶ Caso peggiore per la robustezza $\rightarrow 7^n$.

33.6 Testing Strutturale - White Box

È una strategia ulteriore, complementare, per definire casi di test.

La definizione dei casi di test e dell'oracolo è basata sulla struttura interna dell'unità. Si selezionano quei test che esercitano tutte le strutture del programma.

Non è scalabile. È un'attività complementare al testing funzionale.

Non può rilevare difetti che dipendono dalla mancata implementazione di alcune parti della specifica.

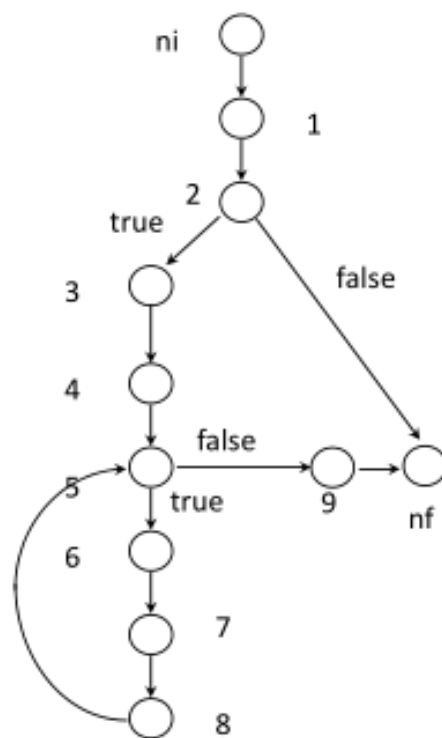
33.6.1 Un modello di rappresentazione dei programmi

Il *Grafo del Flusso di Controllo* (CFG) di un programma P è una rappresentazione, di tutti i sentieri che potrebbero essere percorsi attraverso un programma durante la sua esecuzione.

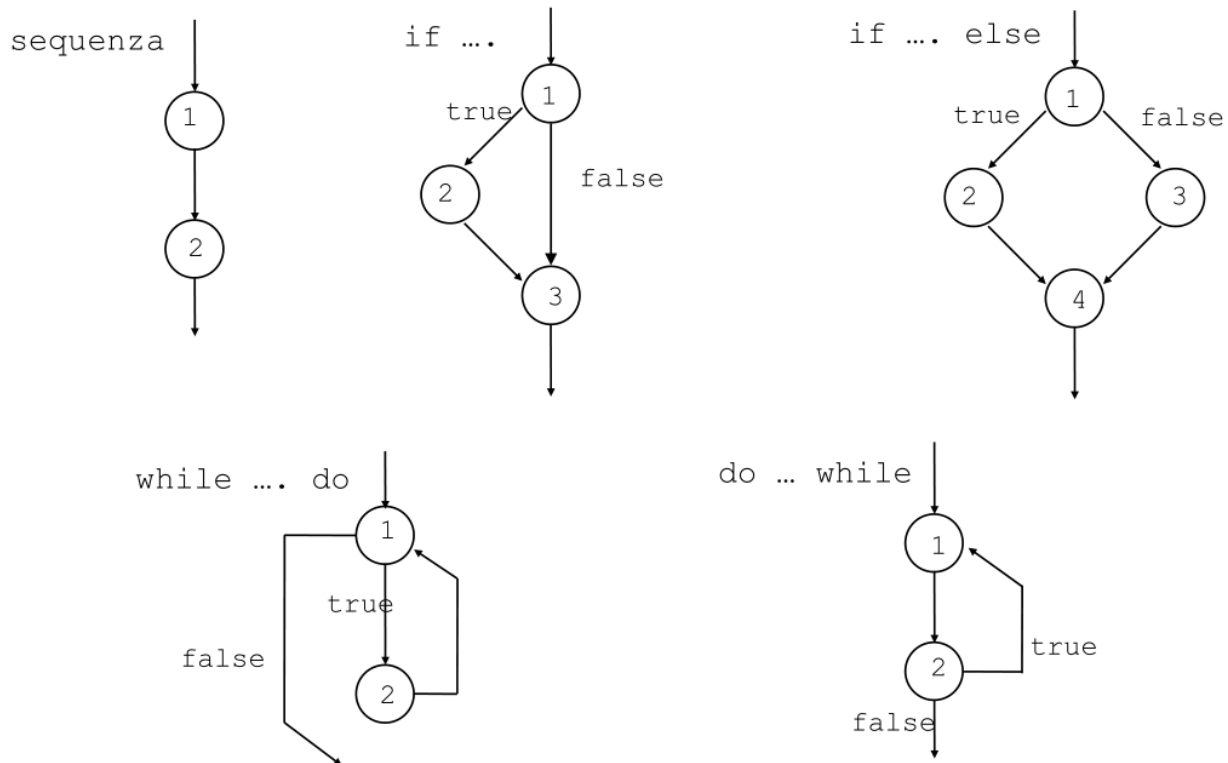
```

1  int Quadrato(int x){
2      int y,n;
3      if (x>0){
4          n = 1;
5          y = 1;
6          while (x>1){
7              n = n + 2;
8              y = y + n;
9              x = x - 1;
10         }
11         return y;
12     }
13 }
14

```



33.6.2 Costruzione del control flow graph per programmi strutturati



33.6.3 Semplificazione di un CFG

Sequenze di nodi possono essere collassate in un solo nodo, purché nel grafo semplificato vengano mantenuti tutti i branch (punti di decisione e biforcazione del flusso di controllo).

Tale nodo collassato può essere etichettato con i numeri dei nodi in esso ridotti.

33.7 Tecniche di Testing Strutturale

In generale fondante sull'adozione di criteri di copertura degli oggetti che compongono la struttura dei programmi.

Copertura → Definizione di un insieme di casi di test in modo tale che gli oggetti di una definita classe siano attivati almeno una volta nell'esecuzione dei casi di test.

Metrica di copertura → **Test Effectiveness Ratio (TER)** = $\frac{\text{\#oggetti coperti}}{\text{\#oggetti totali}}$

33.8 Selezione di casi di test

Pesa molto ciò che voglio testare → Importante è pertanto il **Node Coverage**.

Dato un programma P , viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i nodi di $GFC(P)$, ovvero l'esecuzione di tutte le istruzioni di P (Test Effectiveness Ratio).

$$\text{Test Effectiveness Ratio (TER)} = \frac{\text{n.ro di statement eseguiti}}{\text{n.ro di statement totali}}$$

NB: La copertura delle decisioni implica la copertura dei nodi.

Condition Coverage → Testo tutti i rami del percorso con parametri diversi. Ciò comporta un numero di casi di test esponenziale.

33.9 Confronto tra White e Black box Testing

White-box Testing	Black-Box Testing
Numero potenzialmente infinito da testare	Potenzialmente c'è un'esplosione combinatoria di test case
Testa cosa è stato fatto, non cosa doveva essere fatto	Non permette di scoprire use cases estranei (features)
Non può scoprire Use Case Mancanti	

- ▶ Entrambi sono necessari;
- ▶ Sono agli estremi di un ipotetico **testing continuum**;
- ▶ Qualunque test case viene a cadere tra loro, in base a:
 - ◊ Numero di path logici possibili;
 - ◊ Natura dei dati di input;
 - ◊ Complessità di algoritmi e strutture dati.

33.10 Integration Testing

Quando ogni componente è stata testata in isolamento, possiamo integrarle in sottosistemi più grandi.

Mira a rilevare errori che non sono stati determinati durante lo Unit testing, focalizzandosi su un insieme di componenti che sono integrate.

2 o più componenti sono integrate e analizzate, e quando dei bug sono rilevati, nuove componenti possono essere aggiunte per riparare i bug.

Sviluppare test stub e test driver per un test di integrazione sistematico è time-consuming → L'ordine in cui le componenti sono integrate può influenzare lo sforzo richiesto per l'integrazione.

L'ordine in cui i sottosistemi sono selezionati per il testing e l'integrazione determina la strategia di testing:

- ▶ Big bang integration (non incrementale);
- ▶ Bottom up integration;
- ▶ Top Down integration.

33.10.1 Functional Testing

Verificare che tutti i requisiti funzionali siano stati implementati correttamente.

Impatto dei Requisiti sul testing del sistema:

- ▶ Più espliciti sono i requisiti, più facile sono da testare;
- ▶ La qualità degli use cases influenza il Functional Testing.

L'obiettivo di questo tipo di testing è trovare differenze tra i requisiti funzionali e le funzionalità realmente offerte dal sistema.

I test case sono progettati dal documento dei requisiti e si focalizza sulle richieste e le funzioni chiave.

Il sistema è trattato come una black box.

I test case possono essere riusati (vanno scelti quelli rilevanti per l'utente finale e che hanno una buona probabilità di riscontrare un fallimento).

33.10.2 System Testing

Unit testing e Integration testing si focalizzano sulla ricerca di fallimenti nelle componenti individuali e nelle interfacce tra le componenti.

Questo tipo di testing assicura che il sistema completo è conforme ai requisiti funzionali e non.

Attività:

- ▶ Performance Testing → Testing del sistema sotto stress (sovraccarico con grandi volumi di dati o con ordini di esecuzioni non usuali);
- ▶ Pilot Testing → Primo test pilota sul campo del sistema
 - ◊ Alpha e Beta testing;
- ▶ Acceptance Testing
 - ◊ Benchmark Test → Il cliente prepara un insieme di test case che rappresentano le condizioni tipiche sotto cui il sistema dovrà operare;
 - ◊ Competitor Testing → Il nuovo sistema è testato contro un sistema esistente o un prodotto competitor;
 - ◊ Shadow Testing → Il nuovo sistema e il sistema legacy sono eseguiti in parallelo e i loro output sono confrontati;

- ▶ Installation Testing → Dopo che il sistema è accettato, viene installato nell'ambiente di utilizzo;
 - ◊ Il sistema deve soddisfare a pieno le richieste del cliente, e in seguito (se il cliente è soddisfatto) questo viene rilasciato;
- ▶ Usability Testing → Test della comprensibilità del sistema da parte dell'utente.

34 Lezione del 17-12

34.1 Modelli di Processo

Un processo è un particolare metodo per sviluppare software.

Il processo di sviluppo deve essere modellato esplicitamente, per poter essere gestito e monitorato.

La scelta di un determinato modello di processo pesa sullo sviluppo del SW. L'obiettivo è introdurre stabilità, controllo e organizzazione in una attività tendenzialmente caotica, massimizzando alcune delle caratteristiche del processo.

Esistono dei parametri per valutare la bontà di un modello per sviluppare SW:

- ▶ Comprensibilità → Ovvero quanto è facile da apprendere;
- ▶ Visibilità → Ovvero quanto è semplice capire a che punto si è dello sviluppo SW;
- ▶ Supportabilità;
- ▶ Accettabilità → Capire se le persone coinvolte manifestano il loro consenso;
- ▶ Affidabilità → Se il processo facilita l'individuazione di errori;
- ▶ Robustezza;
- ▶ Manutenibilità;
- ▶ Rapidità.

34.2 Modello più semplice di processo

Edit → Compile → Test.

Molto veloce, non scalabile, e pertanto ingestibile durante la manutenzione.

34.3 Modello a Cascata

Approccio Tayloristico

Introduce un tipo di gestione gerarchica, con passi fissi (non fluidi → separare nettamente i compiti). È infatti più semplice addestrare una singola persona su un singolo task, invece di n task. Abbiamo quindi un processo orientato al prodotto, non al cliente. Otteniamo quindi un processo ripetibile e controllato.

Il modello a cascata applica questo approccio. È un modello sequenziale lineare:

- ▶ Progressione sequenziale (in cascata) di fasi, senza ricicli, al fine di meglio controllare tempi e costi;
- ▶ Definisce e separa le varie fasi e attività del processo → Nullo (o minimo) overlap per le fasi;
- ▶ Consente un controllo dell'evoluzione del processo.

34.3.1 Organizzazione sequenziale delle fasi

Ogni fase raccoglie un insieme di attività omogenee per metodi, tecnologie, skill del personale.

Ogni fase è caratterizzata da:

- ▶ Tasks;
- ▶ Prodotti di tali attività (**deliverables**);
- ▶ Controlli di **qualità/stato** (*quality control measures*).

La fine di ogni fase è un punto rilevante del processo. I semilavorati output di una fase sono input alla fase successiva.

I prodotti di una fase vengono congelati (non sono più modificabili se non innescando un processo formale e sistematico di modifica).

34.3.2 Valutazione del modello a cascata

È facilmente comprensibile, visibile, supportabile, rapido.

Risulta poco accettabile, robusto (questo modello non supporta variazioni in corso d'opera), manutenibile.

Risulta affidabile solo nel caso in cui non ci siano errori → In questo caso avremo la massima qualità raggiungibile. Viceversa risulta impossibile porre rimedio in caso di errori.

34.3.3 Vantaggi e svantaggi

Vantaggi	Svantaggi
Definisce molti concetti utili	Interazione con il committente solo all'inizio e alla fine
Rappresenta un punto di partenza importante per lo studio dei processi SW	I requisiti vengono congelati alla fine della fase di analisi
Facilmente comprensibile e applicabile	I requisiti utente sono spesso imprecisi
Modello molto rigoroso	Errori nei requisiti scoperti solo alla fine del processo
	Il nuovo sistema SW diventa installabile solo quanto è totalmente finito

34.4 Modelli Agili

Il modello a cascata, con la nascita di Internet, risulta poco applicabile (non permette infatti versioni intermedie).

Pertanto nasce l'esigenza di trovarne un nuovo modello, con l'obiettivo di piccole implementazioni incrementali:

- ▶ Ottengo in questo modo feedback immediati;
- ▶ Produco output incrementali (non dovendo pertanto aspettare la fine del progetto);

Le metodologie agili vanno bene se ho la necessità di essere presente immediatamente sul mercato e non abbiamo ancora requisiti cristallizzati.

Alla fine degli anni '90 sono nate una serie di metodologie sotto questa categoria. Tra questi ricordiamo:

- ▶ SCRUM (Che ad oggi è la metodologia più attiva);
- ▶ Crystal;
- ▶ Extreme Programming.

34.5 SCRUM

È un modello agile che permette di focalizzarsi sulla consegna il massimo valore del business nel minor tempo possibile.

Permette di ispezionare rapidamente SW che veramente funziona in breve periodo. Il team si auto-organizza per farlo nel miglior modo possibile.

Chiunque ha modo di visualizzare nel breve periodo (dalle 2 settimane al mese) lo sviluppo del SW.



34.5.1 Sprints

I progetti basati su SCRUM fanno progressi in una serie di sprint.

Hanno una durata media di circa 2/4 settimane o a scadenza mensile.

Una scadenza costante porta a un miglior ritmo.

34.5.2 Scrum Framework

Ruoli:

► Proprietario

- ◇ Definisce le features del prodotto;
- ◇ Definisce la data di rilascio e il suo contenuto;
- ◇ Prioritizza le features in base al suo valore di mercato;
- ◇ Aggiusta il focus delle features e le loro priorità durante ogni iterazione;
- ◇ Accetta o rifiuta il lavoro svolto;

► ScrumMaster

- ◇ Rimuove gli impedimenti;
- ◇ Garantisce che il team è funzionale e produttivo;
- ◇ Organizza il rapporto tra Azienda e Software House;

► Team

- ◇ Tipicamente composta da 5/9;
- ◇ Organizzati in autonomia;
- ◇ Il team dovrebbe cambiare solo tra i vari sprint;

Cerimonie:

► Sprint planning

- ◇ Riunione (Una volta al mese) per definire le features da implementare;
- ◇ Viene creato una Sprint Backlog (i task vengono definiti e ne vengono stimati i tempi per il loro completamento);
- ◇ Si mantiene un design del sistema ad alto livello;

► Review Sprint → Al termine di ciascun sprint, il team presenta i risultati

- ◇ Ha la forma di una demo di presentazione del lavoro svolto;
- ◇ È una review informale;

► Retrospective Sprint → Migliora il processo, dando uno sguardo a ciò che si è svolto e a quello che non funziona

- ◇ Svolto al termine di ogni sprint;

- ▶ **SCRUM meeting giornalieri**

- ◇ Ha lo scopo di evitare meeting non necessari (non è fatta per problem solving).

Artefatti:

- ▶ **Product backlog** → Lista di requisiti, idealmente espressi in modo tale che ogni elemento abbia un valore per l'utente o il cliente del prodotto;

- ◇ Priorità al proprietario.

- ▶ **Sprint Backlog** → Ha lo scopo di porre attenzione su cosa il lavoro debba essere focalizzato durante lo sprint;

- ▶ **Burndown charts** → Un modo veloce (e visivo) per capire lo stato dello sviluppo SW nel tempo.

34.6 User profiling

Sono tecniche che, specialmente negli ultimi anni, fanno largo uso dell'AI.

Si osserva il comportamento di un gruppo di persone e si estraggono delle features.

Importanti risultano per l'user profiling sono la creazione dei cosiddetti focus group.

34.7 Le Personas

Modello inferenziale, basate su previsioni, che generano meta-persone → Raccolte che clusterizzano una parte del nostro target (generalizzazione di un nostro possibile utente).

34.7.1 Costruzione delle personas

1. Identificare le variabili comportamentali;
2. Mappare le interviste sulle variabili comportamentali;
3. Identificare i pattern comportamentali;
4. Sintesi;
5. Controllare che ci sia completezza e ridondanza;
6. Espandere la descrizione degli attributi e comportamenti;

34.7.2 Rappresentazione delle personas

È rappresentato da una pagina di documento.

35 Lezione del 20-12

35.1 Architetture chiuse e aperte

In base a come i componenti comunicano tra di loro posso avere vari tipi di architetture.

Avremo quindi 2 tipi di architetture:

- ▶ Chiusa, in cui ogni livello comunica con il livello direttamente sottostante
 - ◊ Nel caso di sistemi gestionali abbiamo un sistema a 3 livelli;
- ▶ Aperta, in cui non vi è una limitazione sulla comunicazione tra livelli.

35.1.1 Vantaggi e Svantaggi

Aperta	Chiusa
Strati non isolati tra loro	Strati isolanti rispetto agli altri non direttamente sottostanti (minore accoppiamenti e più portatile)
Catena di chiamate corta	Presentano una lunga catena di chiamate (rallentamento possibile del programma)

35.2 Architettura MVC - Model View Controller

Tipo di architettura le connessioni vengono organizzate in maniera diversa (sempre 3 package, ma comunicano in maniera diversa).

Si basa fortemente sul design pattern [Observer](#).

MEMO: Un design pattern ha 4 parti fondamentali:

- ▶ Nome univoco;
- ▶ Problema;
- ▶ Soluzione;
- ▶ Conseguenza e trade-offs delle applicazioni.

35.2.1 Observer Design Pattern

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Viene utilizzato come base architettuale di molti sistemi di gestione di eventi.

Sostanzialmente il pattern si basa su uno o più oggetti, chiamati osservatori o observer, che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto *osservato*, che può essere chiamato soggetto.

Oltre all'observer esiste il **concrete Observer**, che si differenzia dal primo in quanto implementa direttamente le azioni da compiere in risposta ad un messaggio (Il primo è una classe astratta, il secondo no)

35.2.2 Attori in gioco

- ▶ **Soggetto** → Classe che fornisce interfacce per registrare o rimuovere gli observer e che implementa le funzioni di
 - ◊ `Attach(observer);`
 - ◊ `Detach(observer);`
 - ◊ `Notify()` (loop su tutti i ConcreteObserver, ognuno dei quali chiama la funzione `Update()`);
- ▶ **Soggetto Concreto** → Contiene l'attributo `subjectState` che descrive lo stato del soggetto;
- ▶ **Observer** → Definisce un'interfaccia per tutti gli observer, per ricevere le notifiche dal soggetto.
- ▶ **ConcreteObserver** → Mantiene un riferimento `subject` al Soggetto Concreto, per ricevere lo stato quando avviene una notifica.

35.2.3 MVC

Pattern architettuale per la progettazione e strutturazione modulare di applicazioni SW interattive.

Identifica 3 livelli di base:

- ▶ **Model** → Notifica cambiamenti di **stato/dei** dati al view;
- ▶ **View** → Ha riferimento al model e può interrogarlo per ottenere lo stato corrente. Notifica al controller gli eventi dall'interazione con l'utente;
- ▶ **Controller** → Ha riferimento al model e alla View.

Ha particolarmente senso nel caso in cui ho diversi riferimenti allo stesso dato.

Android fa uso di un modello simile a questo (MVP - **Model-view-presenter**) che permette una migliore gestione del passaggio tra finestre di una applicazione.