

Ingegneria del Software

Beatrice Chiara Morgillo

15 settembre 2025

Indice

1 Lezione 2: Processi e qualità del software	7
1.1 Processi software	7
1.1.1 I modelli di processo software	7
1.1.2 Il modello a cascata	7
1.2 Ingegnerizzazione dei requisiti	8
1.3 Design del sistema e del software/UI	8
1.4 Implementazione	8
1.5 Verificazione e validazione (V&V)	8
1.6 Operazioni e manutenzione	8
1.7 Qualità del software	8
1.8 Il modello ISO/IEC 25002	9
1.8.1 Qualità del software: idoneità	9
1.8.2 Affidabilità	9
1.8.3 Efficienza	9
1.8.4 Usabilità	9
1.8.5 Sicurezza	10
1.8.6 Compatibilità	10
1.8.7 Manutenibilità	10
1.8.8 Flessibilità	10
1.8.9 Sicurezza	11
1.9 Modello della qualità-in-uso	11
1.9.1 Usabilità	11
1.9.2 Sicurezza	11
1.9.3 Flessibilità	11
2 Lezione 3 - Requisiti d'ingegneria: elicitatione e analisi	12
2.1 Il ciclo di vita del software	12
2.1.1 Requisiti software	12
2.1.2 Requisiti funzionale e non-funzionale	12
2.2 Proprietà dei buoni requisiti	13
2.2.1 Requisiti di testabilità	14
2.2.2 Requisiti non-funzionali testabili	14
2.3 Il processo di ingegnerizzazione del requisito	14
2.4 Il processo RE	14
2.4.1 Elicitazione e analisi dei requisiti	14
2.5 Processo dell'elicitatione e analisi dei requisiti	14
2.6 Importanza della Conoscenza degli Utenti	15
2.6.1 Personas	15
2.6.2 Casi d'Uso delle Personas	15
2.6.3 Storie Utente	15
2.7 Mockup a bassa fedeltà	16
3 Lezione 4: Diagrammi Use Case	17
3.1 Specificazione dei requisiti	17
3.1.1 Requisiti e design	17
3.1.2 Specifica dei requisiti	17
3.2 Specificazioni del linguaggio naturale (NL)	18
3.3 Diagrammi Use Case	18
3.3.1 Attori	18
3.3.2 Euristica per identificare gli attori	18

3.3.3	Associazioni	19
3.3.4	Attori secondari	19
3.3.5	Generalizzazioni degli attori	19
3.3.6	La relazione <<include>>	20
3.3.7	La relazione <<extend>>	20
3.4	Errori da principianti	21
3.5	Esercizi	21
4	Lezione 5: Use Case completi	25
4.1	Specificazioni degli Use Case	25
4.1.1	Descrizione testuale di un Use Case	25
4.1.2	Formati di Use Case	25
4.1.3	Descrizioni degli Use Case completi	26
4.1.4	Scenari principali ed estensioni	26
4.2	Esempio	27
4.3	Validazione dei requisiti	34
4.3.1	Controllo dei requisiti	34
4.3.2	Tecniche di validazione dei requisiti	34
5	Lezione 6 e 7: Statecharts	35
5.1	Statechart	35
5.1.1	Modellazione con stati e transizioni	35
5.1.2	Regioni, vertici e transizioni	35
5.1.3	Stati (semplici)	35
5.1.4	Pseudostati iniziali e stati finali	35
5.1.5	Sintassi delle transizioni	36
5.1.6	Semantica delle transizioni	36
5.1.7	Attività interne degli stati	39
5.1.8	Stati composti	39
5.1.9	Regioni parallele	40
5.1.10	Pseudostati di Shallow History	40
5.1.11	Pseudostati di Deep History	40
5.1.12	Pseudostati Fork e Join	41
5.1.13	Consigli e trucchi per gli Statechart	41
5.2	Statecharts nel mondo reale	42
5.2.1	Model-driven Development	42
5.2.2	Gestione degli stati dell'interfaccia utente con Statecharts	42
5.3	Esempio	43
5.4	Esercizi	46
6	Lezione 8: Usabilità e Progettazione Centrata sull'uso umano	63
6.1	L'ascesa dell'usabilità	63
6.1.1	Usabilità - App per il grande pubblico	63
6.1.2	Usabilità - Software professionale	63
6.1.3	Usabilità - Sistemi critici per la vita	63
6.1.4	Interazione Uomo-macchina (HCI)	63
6.1.5	Usabilità vs User-friendliness	64
6.2	Definizione di usabilità	64
6.2.1	Apprendibilità	64
6.2.2	Efficienza d'uso	64
6.2.3	Memorabilità	64
6.2.4	Errori	65
6.2.5	Soddisfazione soggettiva	65
6.2.6	Compromessi nell'usabilità	65
6.2.7	Ingegneria dell'usabilità	65
6.3	Progettazione Centrata sull'Uomo (HCD)	65
6.3.1	Principi HCD	65
6.3.2	HCD: Comprendere il Contesto e gli Utenti	66
6.3.3	HCD: Progettare per Soddisfare i Requisiti	66
6.3.4	HCD: Progettare per Soddisfare i Requisiti	66
6.3.5	HCI: Valutare la Progettazione	67
6.3.6	Progettazione Centrata sull'Uomo e SDLC	67

6.3.7	Ancora sul processo di design dell'UI	68
6.4	Design come scelta	69
6.4.1	Importanza della critica e dei feedback	71
7	Lezione 9: Software Design: System Design	74
7.1	Progettazione di un sistema	74
7.1.1	Scopo del System Design	75
7.2	System Design	78
7.2.1	Attività di system design	79
7.2.2	Identificare gli obiettivi qualitativi del sistema	81
7.2.3	Criteri di design	82
7.2.4	Compromessi di design	83
7.3	Concetti di base di buon design	87
7.3.1	Anticipare	88
7.3.2	Decomposizione	89
7.3.3	Modellazione di sottosistemi	91
7.3.4	Servizi e interfacce di sottosistemi	92
7.3.5	Esempio	93
7.3.6	Osservazioni	95
7.3.7	Indipendenza funzionale	96
8	Lezione 11: Regole di buon Software Design - La coesione	97
8.1	Coesione	97
8.1.1	Granularità della coesione	98
8.1.2	Single Responsibility Principle	99
8.1.3	Esempio	100
8.1.4	Benefici della coesione	104
8.2	Accoppiamento	107
8.2.1	Basso accoppiamento	108
8.2.2	Tipi di accoppiamento in O-O	110
8.2.3	Esempio	111
8.2.4	Spiegazione dell'esempio	112
8.3	Pensare alle interfacce	113
8.3.1	Esempio di paperboy	114
8.3.2	Problemi col codice	116
8.3.3	Migliorie al codice originale	118
8.3.4	Perché è meglio?	120
8.3.5	Quali sono gli svantaggi?	122
9	Lezione 12: Regole di buon Software Design - La legge di Demetra	123
9.1	La legge di Demetra	123
9.1.1	Violazioni alla legge	125
9.1.2	Eliminare il codice di navigazione	126
9.1.3	Design guidato dalla responsabilità	127
9.2	Le carte CRC	128
9.2.1	Conseguenze	130
9.2.2	Localizzare le modifiche	131
10	Lezione 10: Il design delle cose di tutti i giorni e la natura delle interazioni giornaliere	132
10.1	Le interazioni	132
10.1.1	La Struttura dell'Azione: Il Ciclo di Azione	132
10.1.2	Fasi di Esecuzione	132
10.1.3	Sette Fasi dell'Azione: Esempio	132
10.2	Le Sette Fasi dell'Azione di Don Norman	133
10.2.1	Sette Fasi dell'Azione: Alternative	133
10.3	Principi di Design per aiutare a colmare i golfi	133
10.3.1	Affordances	133
10.3.2	Vincoli	133
10.3.3	Feedback	133
10.3.4	Coerenza	133
10.3.5	Metafore	134
10.3.6	Mappature	134

10.4 Linee Guida di Norman per colmare i golfi	134
10.4.1 Caratteristiche di un Buon Design	134
11 Lezione 13: Modelli e teorie in HCI	135
11.0.1 Cos'è un modello?	135
11.1 Modelli e teorie HCI	135
11.1.1 Il modello a tre stati dell'input grafico	135
11.1.2 Il modello di Guiard dell'abilità bimanuale	135
11.2 Il Modello del Processore Umano (MHP)	136
11.2.1 MPH: Memorie	137
11.2.2 Percezione di stimoli congruenti e incongruenti	137
11.2.3 Perché usare l'MPH?	138
11.2.4 Esempio	139
11.3 Modello GOMS	142
11.3.1 GOMS: Esempio	142
11.3.2 Keyboard Level Model (KLM)	142
11.3.3 Esempio KLM: rimozione di un file	143
11.4 La Legge di Potenza della Pratica	147
11.5 Legge di Hick	147
11.5.1 Applicazione della Legge di Hick	147
11.6 Legge di Fitts	147
11.6.1 Legge di Fitts – Indice di Difficoltà (ID)	147
11.6.2 Legge di Fitts – Tempi di Movimento	148
11.6.3 Legge di Fitts: Applicazioni	148
11.7 Letture e referenze	154
12 Lezione 14: Bad Smell e refactoring	156
12.1 Bad Smells	156
12.1.1 Code Smells e Refactoring	156
12.1.2 Tipi di Bad Smells: Bloaters	156
12.1.3 Tipi di Bad Smells: Abusatori della Programmazione Orientata agli Oggetti	156
12.1.4 Tipi di Bad Smells: Impedimenti al Cambiamento	156
12.1.5 Tipi di Bad Smells: Superflui	157
12.1.6 Tipi di Bad Smells: Accoppiatori	157
12.2 Bad Smells e il concetto di Debito Tecnico	157
12.2.1 Debito Tecnico	157
12.2.2 Debito Strategico, Intenzionale	157
12.2.3 Debito Non-Strategico, Non Intenzionale	158
12.2.4 Conseguenze del Debito	158
12.2.5 Misurare il Debito Tecnico	158
12.3 Correggere i Bad Smells con il Refactoring	158
12.3.1 Refactoring	158
12.3.2 Il Ciclo di Refactoring	158
12.3.3 Refactoring e Test Unitari	159
12.3.4 Perché Dovresti Fare Refactoring?	159
12.4 Refactoring tipici	160
12.4.1 Referenze	166
12.4.2 Codice Duplicato	167
12.4.3 Metodo Lungo	167
12.4.4 Classe Grande	167
12.4.5 Lista Parametri Lunga	167
12.4.6 Cambiamento Divergente	167
12.4.7 Chirurgia con Fucile a Pompa	167
12.4.8 Invidia delle Caratteristiche	167
12.4.9 Grumi di Dati	167
12.4.10 Ossessione Primitiva	168
12.4.11 Test di Tipo	168
12.4.12 Gerarchie di Eredità Parallele	168
12.4.13 Classe Pigra	168
12.4.14 Generalità Speculativa	168
12.4.15 Campo Temporaneo	168
12.4.16 Catene di Messaggi	168

12.4.17 Middle Man	168
12.4.18 Esempio di Middle Man	169
12.5 Esercizi	170
13 Lezione 15: Teoria dei colori, tipografia e psicologia di Gestalt	172
13.1 Percezione del colore tramite i coni	172
13.2 Modelli di colore	172
13.2.1 Modello di colore CMYK	172
13.2.2 Modello di colore RGB	172
13.3 Dimensioni percettive del colore	172
13.3.1 Hue	172
13.3.2 Saturazione	173
13.3.3 Luminosità	173
13.3.4 Rappresentazione HSL	173
13.3.5 Tinte, sfumature e toni	173
13.4 Uso dei colori nel design delle interfacce	173
13.4.1 Teoria del colore	174
13.4.2 Armonie monocromatiche	175
13.4.3 Armonie analoghe	176
13.4.4 Armonie complementari	177
13.4.5 Armonie complementari suddivise	178
13.4.6 Armonie triadiche	179
13.4.7 Riferimenti: Armonie cromatiche	180
13.4.8 Psicologia del colore	181
13.5 Tipografia	181
13.5.1 L'ipotesi dei serif	181
13.5.2 Impatto della tipografia nelle UI	181
13.5.3 Effetto di superiorità della parola	181
13.6 Teoria della percezione Gestalt	181
13.6.1 Psicologia della Gestalt	182
13.7 Principi della Gestalt	182
13.7.1 Somiglianza	182
13.7.2 Somiglianza nelle UI	183
13.7.3 Prossimità	184
13.7.4 Prossimità: moduli	190
13.7.5 Prossimità: posizionamento delle etichette nei moduli	190
13.7.6 Proximity can backfire: example	191
13.7.7 Principle of Connectedness	195
13.7.8 Principle of Common Region	202
13.7.9 Principle of Common Region: examples	205
13.8 Visual Hierarchy in UI Design	207
13.8.1 Creating a Visual Hierarchy	210
13.8.2 Scala	211
13.8.3 Colore/Contrasto	212
13.8.4 Raggruppamento	213
14 Lezione 17: Linee guida e principi nell'Interazione Uomo-Macchina	214
14.1 Linee guida e principi nellHCI	214
14.2 Le otto regole d'oro di Shneiderman	214
14.2.1 Le dieci euristiche di usabilità Nielsen-Molich	214
14.2.2 Dialogo semplice e naturale	215
14.3 Parla la lingua dellutente	215
14.3.1 Minimizza il carico di memoria dellutente	215
14.3.2 Minimizza il carico di memoria dellutente: esempi	215
14.3.3 Coerenza	215
14.3.4 Coerenza interna: esempi	216
14.3.5 Coerenza esterna	220
14.4 Feedback	221
14.4.1 Persistenza del feedback	221
14.4.2 Feedback e tempi di risposta del sistema	221
14.4.3 Labor Perception Bias	221
14.4.4 Uscite chiaramente indicate e reversibilità delle azioni	221

14.4.5 Scorcatoie	222
14.4.6 Linee guida sulle scorcatoie	226
14.5 Messaggi di errore	227
14.5.1 Buoni messaggi di errore	227
14.5.2 Buoni messaggi di errore: sii cortese	231
14.5.3 Buoni messaggi di errore: livelli multipli	231
14.5.4 Prevenire gli errori	231
14.6 Tipi di errori	231
14.6.1 Tipi di errori: Slips	231
14.6.2 Tipi di errori: Mistakes	232
14.6.3 Aiuto e documentazione	232
14.6.4 La verità fondamentale sui manuali utente	232
14.6.5 Puntare all'usabilità universale	232
14.6.6 Progettare dialoghi che portino a una conclusione	232
14.7 Letture e riferimenti	232
15 Lezione 20: Software Design: Architetture Software	234
15.1 Architetture	234
15.1.1 Definizione dell'architettura	236
15.2 Identificazione di sottosistemi	237
15.2.1 Design Principle: Divide and conquer	238
15.2.2 Ways of dividing a software system	239
15.2.3 Layer	241
15.2.4 Macchina Virtuale (Dijkstra)	242
15.2.5 Architettura Chiusa	243
15.2.6 Architettura Aperta	246
15.3 Principali Architetture	248
15.3.1 Architettura Client-server	249
15.3.2 Principali Architetture Software	251
15.3.3 Repository Architecture	252
15.3.4 Esempio di Repository Architecture	254
15.3.5 Vantaggi dell'architettura a repository	255
15.3.6 Svantaggi dell'architettura a repository	256
15.3.7 Layered Architecture	257
15.3.8 Esempio Layered Architecture	258
15.3.9 Architetture three-tier - I	261
15.3.10 Vantaggi di architetture three-tier	263
15.3.11 Vantaggi di architetture three-tier	266
15.3.12 Svantaggi di architetture three-tier	267
16 Lezione 21: MVC	269
16.0.1 Struttura di un'app software	270
16.1 Model-View-Controller	271
16.1.1 Struttura e responsabilità	272
16.1.2 Interazioni fondamentali	274
16.1.3 Considerazioni	276
16.1.4 MVC e qualità del software	277
16.1.5 MVC ed eventi in Java	279
16.1.6 Esempio	281
16.1.7 Applicazioni e livelli	282
16.1.8 MVC per Classi di Dominio e Widget	287

Capitolo 1

Lezione 2: Processi e qualità del software

1.1 Processi software

Un processo software è un insieme di attività correlate che conducono alla produzione di un sistema software. Non c'è un processo universale che funziona sempre, ma ce ne sono di molteplici e tutti includono, in qualche modo, delle attività fondamentali:

1. **Specificazione software:** i vincoli funzionali e operativi vengono definiti.
2. **Implementazione software:** il software incontra i requisiti che devono essere prodotti.
3. **Validazione software:** il software dev'essere validato per verificare che segua i requisiti.
4. **Evoluzione del software:** il software deve evolvere per incontrare i cambiamenti necessari.

1.1.1 I modelli di processo software

Un **modello di processo software** o **ciclo di vita di un sistema software** (sigla in inglese: SDLC) è una rappresentazione semplificata di un processo software.

Un modello di processo software può interessarsi a una prospettiva particolare e ci sono dei modelli di processo molto generali, come il modello a cascata.

1.1.2 Il modello a cascata

I processi software consistono di un numero di passaggi sequenziali, in un processo plan-driven, il risultato di ogni fase è un documento che viene approvato e la fase successiva non può iniziare se la precedente non è completa. I passi da seguire sono:

1. **Ingegnerizzazione dei requisiti.**
2. **Design del sistema.**
3. **Design dell'UI e del software.**
4. **Implementazione.**
5. **Testing.**
6. **Operazioni e manutenzione.**

Questo modello rigido, che segue un approccio che va in base alle previsioni ha senso per l'ingegnerizzazione dell'hardware, dove sono considerati anche gli alti costi di produzione, invece per lo sviluppo software questi passaggi possono sovrapporsi e dare informazioni a vicenda. Durante il design del sistema vengono identificati i problemi coi loro requisiti, durante l'implementazione vengono trovati i problemi col design del software e i requisiti possono cambiare.

1.2 Ingegnerizzazione dei requisiti

L'obiettivo è capire cosa dovrebbe fare il software (e non come dev'essere implementato). Viene fatta un'attenta analisi di cos'ha bisogno l'utente e del problema di dominio.

Vengono inclusi i clienti, gli utenti finali e gli ingegneri del software, in modo da avere come output principale un **documento che specifichi i requisiti software**.

1.3 Desing del sistema e del software/UI

L'obiettivo è avere un design adeguato della struttura del software e si sviluppa su due livelli diversi:

- **Design di sistema:** l'architettura generale del sistema.
 - Decomposizione in moduli e componenti.
 - Allocazione di funzionalità per i moduli e moduli per le componenti hardware.
 - Relazioni e collaborazioni tra i moduli definiti.
- **Design della UI e del software:** dettagli su come implementare ogni modulo.
 - include design di un'architettura software di basso livello.
 - include una prototipazione dell'UI.

Il risultato è un insieme di specificazioni di design, spesso formalizzate usando linguaggi di design come UML.

1.4 Implementazione

Il risultato è "tradurre" le specificazioni di design in un linguaggio/tecnologia di programmazione scelta. Non è una traduzione qualunque, ma una di **alta qualità** e il codice risultante dovrebbe essere "**clean**".

1.5 Verificazione e validazione (V&V)

Si usa per vedere se le implementazioni soddisfano appieno le necessità dell'utente.

- **Verificazione:** sono domande per quanto riguarda la conformità del sistema con le sue specifiche.
- **Validazione:** sono domande per verificare se il sistema incontra le aspettative del cliente.

1.6 Operazioni e manutenzione

L'operazione si occupa di distribuire il sistema e renderlo installabile per il cliente, mettendolo effettivamente in uso. La manutenzione stabilisce che il software potrà cambiare in un certo punto, infatti le necessità del cliente possono cambiare, il contesto d'uso potrebbe e i bug che sono sfuggiti alla V&V potrebbero emergere.

1.7 Qualità del software

Non c'è un principio unico di qualità, ma ci sono approcci e visioni diverse, lo standard che definisce la qualità del modelli software è dato dall'ISO/IEC 25002. Nello standard, il cochetto di qualità del software è modellizzato tramite:

- **Modello della qualità del prodotto**, composto da 9 caratteristiche relative alla qualità delle proprietà del prodotto. Le caratteristiche e le sottocaratteristiche fanno da riferimento del modello per la qualità dei prodotti da specificare, misurare e valutare.
- **Modello della qualità-in-uso**, composto da 3 caratteristiche che influenzano gli stakeholder quando i prodotti o i sistemi sono usati in uno specifico contesto d'uso.

1.8 Il modello ISO/IEC 25002

1.8.1 Qualità del software: idoneità

L'**idoneità funzionale** è un grado per cui una componente o un sistema dà funzioni che incontrano le necessità confermate e implicate quando vengono usate sotto specifiche condizioni.

Queste caratteristiche sono composte da 3 sottocaratteristiche:

- **Completezza funzionale:** un grado per cui le funzioni date coprono tutte le task specifiche e gli obiettivi dell'utente.
- **Correttezza funzionale:** grado per cui il prodotto provvede risultati accurati quando usato degli utenti.
- **Appropriatezza funzionale:** grado per cui le funzioni facilitano il completamento di task e obiettivi specifici.

1.8.2 Affidabilità

L'**affidabilità** valuta come il sistema performa sotto condizioni specifiche per un periodo di tempo specifico. Questa caratteristica è composta dalle seguenti sottocaratteristiche:

- **Impeccabilità:** grado per cui un sistema performa specifiche funzioni senza problemi per una normale operazione.
- **Disponibilità:** grado per cui un sistema è operativo e accessibile quando viene richiesto il suo utilizzo.
- **Tolleranza ai problemi:** grado per cui il sistema opera come deve nonostante la presenza di problemi lato hardware o software.
- **Recuperabilità.**

1.8.3 Efficienza

L'**efficienza** rappresenta il grado con cui un prodotto performa le sue funzioni entro limiti specifici delle risorse, ed è efficiente nell'uso delle risorse.

Questa caratteristica è composta dalle seguenti sottocaratteristiche:

- **Comportamento nel tempo:** il grado per cui il tempo di risposta e i tassi di rendimento di un prodotto o sistema incontrano i requisiti.
- **Utilizzo delle risorse:** il grado per cui la quantità e i tipi delle risorse usate da un prodotto o sistema incontrano i requisiti.
- **Capacità:** grado per cui i limiti massimi di un prodotto o parametro di sistema incontrano i requisiti.

1.8.4 Usabilità

Rappresenta il grado con cui s'interagisce con un prodotto o sistema da parte dell'utente.

Ha le seguenti sottocaratteristiche:

- **Riconoscibilità.**
- **Apprendibilità.**
- **Operatività.**
- **Protezione dell'utente dagli errori.**
- **Inclusività.**

1.8.5 Sicurezza

È il grado per cui un sistema si difende dagli attacchi maliziosi e protegge le informazioni, rinforzando i dati tramite adeguati meccanismi di autorizzazione. Include le seguenti sotto-caratteristiche:

- **Confidenzialità:** grado per cui un sistema assicura che i dati sono accessibili solo a chi ne è autorizzato all'accesso.
- **Integrità:** grado per cui un sistema assicura che il suo stato e i suoi dati sono protetti da modificazioni o rimozioni non autorizzate.
- **Non-ripudio:** grado per cui le azioni o eventi vengono dimostrati di prendere luogo in modo che gli eventi o le azioni possano non essere ripudiate dopo.
- **Responsabilità:** grado per cui le azioni di un'entità possono essere tracciate unicamente da quell'entità.
- **Autenticità:** grado per cui l'identità di un soggetto o risorsa può essere mostrata a chi l'ha rivendicata.

1.8.6 Compatibilità

La **compatibilità** rappresenta il grado in cui un sistema può scambiare informazioni con altri prodotti, sistemi o componenti e/o svolgere le proprie funzioni richieste condividendo lo stesso ambiente e risorse comuni con altri sistemi. Questa caratteristica è composta dalle seguenti sotto-caratteristiche:

- **Coesistenza** - Grado in cui un prodotto può svolgere le proprie funzioni richieste in modo efficiente mentre condivide un ambiente e risorse comuni con altri prodotti, senza impatti negativi su nessun altro prodotto.
- **Interoperabilità** - Grado in cui un sistema, prodotto o componente può scambiare informazioni con altri prodotti e utilizzare reciprocamente le informazioni che sono state scambiate.

1.8.7 Manutenibilità

La **manutenibilità** rappresenta il grado di efficacia ed efficienza con cui un prodotto o sistema può essere modificato per migliorarlo, correggerlo o adattarlo ai cambiamenti dell'ambiente e ai requisiti. Questa caratteristica è composta dalle seguenti sotto-caratteristiche:

- **Modularità** - Grado in cui un software è composto da componenti discreti, in modo che una modifica a un componente abbia un impatto minimo sugli altri.
- **Riutilizzabilità** - Grado in cui un software o un modulo può essere utilizzato come risorsa in più di un sistema.
- **Modificabilità** - Grado in cui un prodotto o sistema può essere modificato in modo efficace ed efficiente senza introdurre difetti o degradare la qualità.
- **Testabilità** - Grado in cui possono essere stabiliti criteri di test per un sistema e possono essere eseguiti test per determinare se tali criteri sono stati soddisfatti.

1.8.8 Flessibilità

La **flessibilità** è il grado in cui un prodotto può essere adattato ai cambiamenti nei suoi requisiti, nei contesti di utilizzo o nell'ambiente operativo. Questa caratteristica è composta dalle seguenti sotto-caratteristiche:

- **Adattabilità** - Grado in cui un sistema può essere adattato in modo efficace ed efficiente a diversi hardware, software o altri ambienti operativi o di utilizzo.
- **Scalabilità** - Grado in cui un sistema può gestire carichi di lavoro in crescita o in diminuzione o adattare la sua capacità per gestire la variabilità.
- **Installabilità** - Grado di efficacia ed efficienza con cui un prodotto o sistema può essere installato e/o disinstallato con successo.
- **Sostituibilità** - Grado in cui un prodotto può sostituire un altro prodotto software specificato per lo stesso scopo nello stesso ambiente.

1.8.9 Sicurezza

La **sicurezza** rappresenta il grado in cui un prodotto evita uno stato in cui la vita umana, la salute, la proprietà o l'ambiente sono messi in pericolo. Questa caratteristica include, tra l'altro, le seguenti sotto-caratteristiche:

- **Sicurezza in caso di guasto** - Grado in cui un prodotto può automaticamente collocarsi in una modalità operativa sicura, o tornare a una condizione sicura in caso di guasto.
- **Identificazione del rischio** - Grado in cui un prodotto può identificare un corso di eventi o operazioni che possono portare a un rischio inaccettabile.
- **Avviso di pericolo** - Grado in cui un sistema fornisce avvisi di rischi inaccettabili per le operazioni o i controlli interni in modo che possano reagire in tempo sufficiente.

1.9 Modello della qualità-in-uso

Modello di qualità in uso, è composto da 3 caratteristiche (ulteriormente suddivise in sotto-caratteristiche) che possono influenzare gli stakeholder quando i prodotti o i sistemi vengono utilizzati in un contesto d'uso specifico. Misura il grado in cui un prodotto o un sistema può essere utilizzato da utenti specifici per soddisfare le loro esigenze al fine di raggiungere obiettivi specifici con efficacia, efficienza, assenza di rischi e soddisfazione in contesti d'uso specifici.

1.9.1 Usabilità

L'**usabilità** misura l'estensione in cui gli utenti possono raggiungere i loro obiettivi in modo efficiente e soddisfacente utilizzando il sistema. Questa caratteristica è composta dalle seguenti sotto-caratteristiche:

- **Efficacia** - Quanto bene gli utenti possono completare i loro compiti previsti utilizzando il sistema.
- **Efficienza** - Le risorse (ad es., tempo, sforzo) necessarie per raggiungere i compiti.
- **Soddisfazione** - Il comfort dell'utente e l'esperienza positiva durante l'utilizzo del sistema.

1.9.2 Sicurezza

La **sicurezza** valuta la capacità del sistema di prevenire danni o pericoli per le persone, l'ambiente e gli interessi commerciali. Questa caratteristica è composta dalle seguenti sotto-caratteristiche:

- **Danno Commerciale**: Valuta quanto bene il sistema previene perdite finanziarie o danni all'attività.
- **Salute e Sicurezza dell'Operatore**: Quanto bene il sistema protegge gli utenti dai rischi per la salute o dai pericoli per la sicurezza durante il suo utilizzo.
- **Salute e Sicurezza Pubblica**: Previene rischi o danni per il pubblico generale attraverso l'uso o il funzionamento del sistema. **Danno Ambientale**: Il sistema dovrebbe evitare o ridurre al minimo gli impatti negativi sull'ambiente.

1.9.3 Flessibilità

La **flessibilità** si riferisce alla capacità del sistema di adattarsi e operare efficacemente in contesti o ambienti diversi. Questa caratteristica è composta dalle seguenti sotto-caratteristiche:

- **Conformità al contesto**: La capacità del sistema di adattarsi ai requisiti e ai vincoli specifici di diversi contesti.
- **Estensibilità del contesto**: Il potenziale del sistema di espandersi o adattarsi a ambienti nuovi o in evoluzione senza modifiche significative.
- **Accessibilità**: Cattura l'efficacia con cui il sistema può essere utilizzato da tutte le persone, comprese quelle con disabilità, in diversi ambienti.

Capitolo 2

Lezione 3 - Requisiti d'ingegneria: elicitazione e analisi

2.1 Il ciclo di vita del software

2.1.1 Requisiti software

I requisiti sono descrizioni di quello che il sistema dovrebbe fare:

- I **servizi** che il sistema dovrebbe dare agli utenti.
- I **vincoli operativi**.

Il termine «**requisito**» è usato in modo incoerente nell'industria del software. In alcuni casi, un requisito è una descrizione astratta e ad alto livello di un servizio che il sistema dovrebbe fornire o di un vincolo sul sistema.

- Questi sono chiamati **Requisiti Utente** (l'attenzione è sulla prospettiva degli utenti finali)

In altri, è una definizione più dettagliata e formale di una funzione del sistema

- Questi sono chiamati **Requisiti di Sistema** (l'attenzione è sul sistema da costruire)

Tale ambiguità è inevitabile, poiché i requisiti possono avere una doppia funzione:

- I **requisiti degli utenti** possono essere la base per un'offerta per un contratto
 - Il cliente può definire requisiti generali degli utenti
 - Diversi appaltatori possono proporre modi diversi per soddisfare i requisiti degli utenti
- I **requisiti di sistema** possono essere la base per il contratto stesso
 - Una volta assegnato un contratto per lo sviluppo del software, l'appaltatore formula un insieme più dettagliato di requisiti di sistema
 - In modo che il cliente comprenda esattamente cosa farà il software e possa validare la proposta
 - Una volta accettati e convalidati, i requisiti di sistema possono essere inseriti nel contratto finale e sono vincolanti!

L'ingegnerizzazione dei requisiti (RE) è una sottoarea dell'Ingegneria del software che si occupa del processo di definire i requisiti di per un quel-che-sarà-software. L'obiettivo è fornire agli ingegneri del software dei metodi, tecniche e strumenti per capire e documentare quel che un sistema software deve fare.

2.1.2 Requisiti funzionale e non-funzionale

I requisiti sono spesso classificati come funzionali o non funzionali:

- **Requisiti Funzionali:**
 - Servizi che il sistema dovrebbe fornire
 - Come il sistema dovrebbe reagire a determinati input o comportarsi in una data situazione
- **Requisiti Non-Funzionali:**

- Vincoli sui servizi o funzioni offerti dal sistema
- Includono vincoli temporali, vincoli di processo o vincoli imposti da standard
- Spesso si applicano all'intero sistema piuttosto che a singole caratteristiche o servizi

In pratica, la distinzione tra questi due tipi di requisiti non è netta. Considera quanto segue: *solo gli utenti autorizzati dovrebbero poter accedere al sistema.*

Sembra un requisito non funzionale, tuttavia, quando sviluppiamo in maggiore dettaglio, genera requisiti aggiuntivi che sono chiaramente funzionali, ad esempio, gli utenti devono essere in grado di effettuare il login e autenticarsi. **I requisiti non sono indipendenti e un requisito spesso genera o vincola altri requisiti**

Requisiti Funzionali

Quando espressi come **requisiti funzionali degli utenti**, possono essere scritti in linguaggio naturale, in modo che possano essere compresi da persone non tecniche (ad es.: utenti e manager).

Quando espressi come **requisiti funzionali del sistema**, dovrebbero essere dettagliati, descrivere accuratamente gli input e gli output del sistema e le eccezioni, in modo che gli ingegneri del software sappiano esattamente cosa implementare.

Requisiti non-funzionali

I requisiti non-funzionali sono non direttamente collegati ai servizi specifici offerti dal sistema. Tipicamente specifica o vincola le caratteristiche del sistema come un insieme:

- I vincoli su come dovrebbero essere implementati.
- Specifica altre proprietà.

Spesso i requisiti non-funzionali sono molto più critici di quelli funzionali.

- Gli utenti possono trovare una strada attorno all'implementazione sub-ottimale di una richiesta funzionale.
- ...ma fallendo a incontrare i requisiti non-funzionali che possono essere indice di un sistema instabile.

Tipi di requisiti non-funzionali

I requisiti non funzionali possono derivare da:

- Caratteristiche richieste del prodotto (**Requisiti di Prodotto**)
 - La velocità del sistema, la quantità di memoria richiesta, requisiti di usabilità, tassi di fallimento accettabili, ...
- Le organizzazioni del cliente e degli sviluppatori (**Requisiti Organizzativi**)
 - Processi operativi che descrivono come il sistema sarà utilizzato, linguaggi di programmazione richiesti, requisiti che specificano l'ambiente operativo, ...
- Fonti esterne (**Requisiti Esterne**)
 - Cosa deve essere fatto affinché il sistema venga approvato da un regolatore di terze parti, requisiti legali, requisiti etici per garantire che il sistema sia accettabile per i suoi utenti e per il pubblico in generale.

2.2 Proprietà dei buoni requisiti

I requisiti dovrebbero essere:

- **Chiari e facili da comprendere** (soprattutto i requisiti dell'utente)
- **Univoci** (l'ambiguità porta a controversie con i clienti)
- **Completi**
- **Coerenti** (cioè, non dovrebbero configgere tra loro)
- **Verificabili** (la mancanza di verificabilità porta a controversie con i clienti). Dato un requisito, dovrebbe essere possibile determinare in modo univoco se il sistema soddisfa quel requisito.

2.2.1 Requisiti di testabilità

Dovrebbe essere possibile determinare in modo univoco se il sistema soddisfa un requisito. Altrimenti, gli sviluppatori potrebbero sostenere che il requisito è soddisfatto, mentre il cliente potrebbe non essere affatto d'accordo! I Requisiti Funzionali del Sistema dovrebbero essere il più dettagliati possibile e non lasciare spazio all'interpretazione. I Requisiti Non Funzionali del Sistema dovrebbero includere indicatori quantitativi ogni volta che è possibile.

2.2.2 Requisiti non-funzionali testabili

Requisiti non molto testabili:

- a Il sistema dovrebbe essere affidabile
- b Il sistema dovrebbe essere semplice da usare
- c Il sistema dovrebbe essere veloce e reattivo

Requisiti più testabili

- a Il sistema dovrebbe avere un tempo di attività al 99.9% al mese.
- b Gli utenti dovrebbero essere capaci di usare tutte le funzioni di sistema dopo al più due ore di allenamento e il numero medio di errori lato utente dovrebbero non superare i 2 all'ora per l'utilizzo del sistema.
- c Il tempo di risposta medio del sistema non dovrebbe superare i 100 millisecondi.

2.3 Il processo di ingegnerizzazione del requisito

I 3 passaggi chiave sono:

- **Elicitazione e analisi dei requisiti:** scopre i requisiti interagendo con gli stakeholder.
- **Specificazione del requisito:** converte i requisiti in una forma standardizzata.
- **Validazione del requisito:** controlla che i requisiti definiscano il sistema che il cliente vuole,

2.4 Il processo RE

In pratica, il processo RE non è lineare, le sue attività sono attualmente interlivellate in un processo iterativo con livelli diversi di granularità, passando per i requisiti: **a livello di business, dell'utente e di sistema**.

2.4.1 Elicitazione e analisi dei requisiti

L'obiettivo di quest'attività è di trovare quel che i cliente vogliono e necessitano ed è considerata la parte più critica, vista la dovuta collaborazione con gli stakeholders.

Gli stakeholder non sanno cosa vogliono dal software, tranne che in termini molto generali. Non sanno cosa sia fattibile e cosa non lo sia, e potrebbero fare richieste irrealistiche. Gli stakeholder sono esperti nel loro dominio. Esprimono naturalmente i requisiti nei propri termini e gergo, con una conoscenza implicita del loro lavoro. Potrebbero dare per scontati alcuni dettagli, quando in realtà non lo sono. Diversi stakeholder, con requisiti diversi, possono esprimere i loro requisiti in modi diversi. Gli ingegneri dei requisiti devono lavorare attorno a comunanze e conflitti. Fattori politici possono influenzare il processo di elicitation. I manager possono richiedere requisiti specifici perché questi consentirebbero loro di aumentare la loro influenza nell'organizzazione. Gli stakeholder hanno opinioni diverse (e talvolta conflittuali) sull'importanza e la priorità dei requisiti. Se alcuni stakeholder sentono che le loro opinioni non sono state adeguatamente considerate, potrebbero tentare deliberatamente di minare il processo di ingegneria dei requisiti. I requisiti cambieranno durante il processo di elicitation. Nuovi requisiti possono emergere da nuovi stakeholder che non erano stati inizialmente considerati.

2.5 Processo dell'elicitazione e analisi dei requisiti

Ci sono de passaggi da seguire:

1. **Scoprire e capire:** interagire con gli stakeholder per scoprire i requisiti.
2. **Classificazione e organizzazione:** i requisiti correlati sono raggruppati e organizzati.

3. **Prioritizzazione e negoziazione:** risolve i conflitti dei requisiti che sorgono dai conflitti delle necessità dei vari stakeholder.
4. **Documentazione:** tiene traccia dei requisiti scoperti per la prossima iterazione del processo di elicitatione.

2.6 Importanza della Conoscenza degli Utenti

La comprensione degli utenti è fondamentale per:

- Scoprire i requisiti
- Progettare interfacce utente efficaci

2.6.1 Personas

- **Definizione:** Archetipi ipotetici che rappresentano utenti reali
- **Scopo:** Promuovere l'empatia e comprendere meglio gli utenti
- **Caratteristiche:**
 - Personalizzazione (nome, età, biografia)
 - Informazioni lavorative
 - Formazione scolastica
 - Obiettivi
 - Punti critici/Frustrazioni
- Non sono persone reali ma vengono definiti con rigore
- Emergono dal processo di elicitatione dei requisiti

2.6.2 Casi d'Uso delle Personas

Utili quando:

- Sviluppiamo software per il pubblico generico
- Gli end-user includono il pubblico generico
- Non abbiamo stakeholder specifici da intervistare

2.6.3 Storie Utente

- Descrizioni narrative di come il sistema può essere utilizzato
- Presentano:
 - Azioni degli utenti
 - Informazioni utilizzate
 - Output richiesti
- Efficaci per comunicare obiettivi generali
- Gli stakeholder spesso le descrivono meglio dei requisiti tecnici

2.7 Mockup a bassa fedeltà

Questi mockup (o wireframe) sono bozze semplificate delle interfacce del sistema lato utente. Sono modi veloci e affetti dal costo per visualizzare la struttura e lo scorrimento base del sistema. L'interesse principale è la funzionalità e il flusso, non l'estetica.

I suoi benefici sono:

- **Comprensione facilitata:** aiuta gli stakeholder a capire le funzionalità base del sistema. Dando un'idea chiara dell'idea iniziale.
- **Aiuta a scoprire i nuovi requisiti:** è più semplice ragionare su interfacce concrete che di proposte astratte.
- **Feedback interessante e interattivo:** promuove l'attenzione da parte degli stakeholder e permette una rapida iterazione e feedback senza un design delineato.
- **Fondamenta per lo sviluppo:** provedde una solida fondamenta per spostarsi verso design ad alta fedeltà e a un'eventuale sviluppo.

Capitolo 3

Lezione 4: Diagrammi Use Case

3.1 Specificazione dei requisiti

3.1.1 Requisiti e design

In linea di principio, i requisiti dovrebbero dichiarare *cosa* il sistema dovrebbe fare e il design dovrebbe descrivere *come* lo fa. In pratica, requisiti e design sono inseparabili:

- I requisiti possono essere strutturati e organizzati sulla base di un'architettura di sistema di alto livello.
- Il sistema può inter-operare con altri sistemi che generano requisiti di design.
- L'uso di un'architettura specifica per soddisfare requisiti non funzionali può essere esso stesso un requisito di dominio.

3.1.2 Specifica dei requisiti

Il processo di scrittura dei requisiti utente e di sistema in un documento di specifica dei requisiti.

- I **requisiti utente** devono essere comprensibili da utenti finali e clienti che non hanno una formazione tecnica.
- I **requisiti di sistema** sono requisiti più dettagliati e possono includere informazioni più tecniche.
- I requisiti possono far parte di un contratto per lo sviluppo del sistema. È quindi importante che questi siano il più completi e dettagliati possibile.

Sono possibili diversi approcci:

- **Linguaggio Naturale:** esprimere i requisiti come frasi numerate in linguaggio naturale. Ogni frase dovrebbe esprimere un singolo requisito.
- **Linguaggio Naturale Strutturato:** utilizzare un modulo o un template standardizzato.
- **Notazioni e Modelli Semi-Formali:** Diagrammi UML dei Casi d'Uso (Use Case Diagrams) e altri modelli di dominio, tipicamente integrati da annotazioni in linguaggio naturale.
- **Specifiche Formali:** Queste notazioni si basano su concetti matematici come macchine a stati finite e infinite, logiche temporali, ecc.

Diversi approcci sono utilizzati in diversi domini:

- Nell'ingegneria di **sistemi critici per la sicurezza** (safety-critical systems), è comune utilizzare specifiche formali e linguaggio naturale strutturato.
- Nell'ingegneria di un'applicazione che listi le cose da fare (to-do list app), si potrebbe usare il linguaggio naturale non strutturato per esprimere i requisiti.
- Nell'ingegneria di un sistema informativo di medie-grandi dimensioni, sfruttare notazioni e modelli semi-formali potrebbe essere un buon compromesso.

3.2 Specificazioni del linguaggio naturale (NL)

I requisiti sono scritti come frasi in linguaggio naturale, eventualmente integrate da diagrammi e tabelle. Questo approccio è utilizzato per scrivere i requisiti perché è espressivo, intuitivo e universale.

- Ciò significa che i requisiti possono essere compresi da utenti e clienti.

Può esprimere sia requisiti funzionali che non funzionali. Definire un formato “standard” e utilizzarlo per tutti i requisiti. Utilizzare il linguaggio in modo coerente. Usare *shall* per i requisiti obbligatori, *should* per i requisiti desiderabili. Utilizzare la formattazione del testo per identificare le parti chiave del requisito. Evitare l’uso di gergo tecnico informatico. Includere una spiegazione (razionale) del motivo per cui un requisito è necessario.

Ci sono dei problemi usando il linguaggio naturale:

- **Mancanza di chiarezza:** È difficile essere precisi senza rendere il documento difficile da leggere.
- **Confusione tra i requisiti:** I requisiti funzionali e non funzionali tendono a essere mescolati.
- **Agglomerazione dei requisiti:** Diversi requisiti distinti possono essere espressi insieme in un’unica affermazione.

3.3 Diagrammi Use Case

Gli Use Case (Casi d’Uso) sono un modo per descrivere le interazioni tra utenti e un sistema utilizzando un modello grafico e testo in linguaggio naturale strutturato. Sono una parte fondamentale del Linguaggio di Modellazione Unificato (UML) e possono essere utilizzati per rappresentare l’insieme dei requisiti funzionali di un sistema.

I casi d’uso identificano:

- **Attori:** Categorie di utenti (non necessariamente umani) del sistema.
- **Casi d’Uso:** Tipi di interazioni (o funzionalità) offerte dal sistema.

Informazioni aggiuntive sulle interazioni possono essere fornite come descrizioni testuali (strutturate) o per mezzo di uno o più modelli semi-formali (ad es.: Diagrammi di Sequenza UML o Diagrammi degli Stati). I casi d’uso hanno le seguenti caratteristiche:

- Sono un modo per supportare la comunicazione con il cliente per definire le funzionalità del sistema. Dovrebbero essere il più semplici possibile.
- Non definiscono *come* il sistema è implementato, ma *cosa* il sistema dovrebbe fare dal punto di vista degli utenti (il sistema è una scatola nera).
- I casi d’uso sono spesso documentati utilizzando un Diagramma dei Casi d’Uso (UCD) di alto livello.

3.3.1 Attori

Gli attori sono rappresentati utilizzando figure stilizzate (stick figures). Rappresentano entità esterne che interagiscono con il **Sistema in Sviluppo** (SUD, System Under Development).

- Classi di utenti (umani).
- Altri sistemi.
- L’ambiente fisico.

Ogni attore ha un nome univoco. Gli attori sono più granulari (coarse-grained) delle Personas: un singolo attore può essere associato a multiple Personas.

3.3.2 Euristica per identificare gli attori

I casi d’uso sono rappresentati come ellissi denominate. Corrispondono a funzionalità offerte dal sistema, fornendo un qualche beneficio o utilità agli attori. I casi d’uso modellano i requisiti funzionali. Un caso d’uso astrae molti possibili scenari (sequenze di azioni) per una determinata funzionalità.

- Uno scenario può essere visto come un’istanza di un caso d’uso.
- Un caso d’uso rappresenta una classe di scenari che mirano a utilizzare la stessa funzionalità.

Per identificare gli attori, ci si può chiedere:

- Quali gruppi di utenti sono supportati dal Sistema in Sviluppo (SUD) nel loro lavoro?
- Quali gruppi di utenti eseguono le principali funzionalità offerte dal SUD?
- Quali gruppi di utenti svolgono le funzioni secondarie del SUD, come l'amministrazione?
- Il SUD interagirà con sistemi o software esterni? Ogni sistema o software esterno con cui il SUD interagisce sarà un attore.

Gli attori non corrispondono a una singola entità, ma rappresentano piuttosto una classe di utenti che può avere lo stesso ruolo: un utente può ricoprire diversi ruoli nello stesso sistema.

3.3.3 Associazioni

Oltre ad attori e casi d'uso, i diagrammi dei casi d'uso includono diversi tipi di relazioni tra di essi.

Un'associazione tra un attore e un caso d'uso indica che l'attore può eseguire il caso d'uso. Graficamente, è rappresentata come una linea che collega un attore a un caso d'uso.

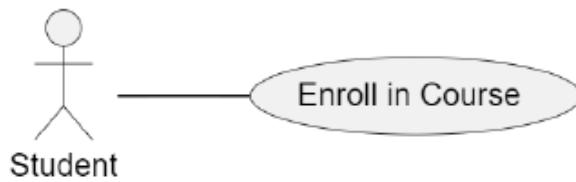


Figura 3.1: Associazioni

3.3.4 Attori secondari

Un caso d'uso può essere associato a più attori. La semantica è che più attori devono collaborare in qualche modo per eseguire quel caso d'uso.

I diagrammi dei casi d'uso UML non includono meccanismi per specificare come diversi attori sono coinvolti in un caso d'uso.

Le modalità di interazione e le diverse responsabilità possono essere specificate con descrizioni aggiuntive.

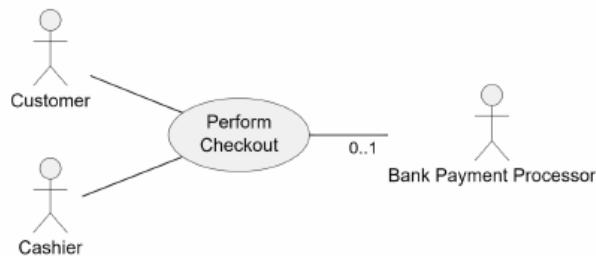


Figura 3.2: Cardinalità tra più attori

3.3.5 Generalizzazioni degli attori

La generalizzazione tra attori può essere applicata quando un attore è un sotto-tipo di un altro attore. Stesso concetto e notazione come nei Diagrammi delle Classi UML. Rappresentata graficamente come una freccia con una testa vuota. L'attore specializzato può eseguire qualsiasi caso d'uso che il genitore può eseguire. La generalizzazione tra casi d'uso è destinata ad essere utilizzata quando un caso d'uso è una specializzazione di un altro. A differenza delle dipendenze <<extend>>, non ci sono punti precisi in cui i casi d'uso specializzati deviano dal genitore. Le specializzazioni possono essere molto diverse rispetto ai casi d'uso parent. La notazione è la notazione UML standard per la specializzazione.

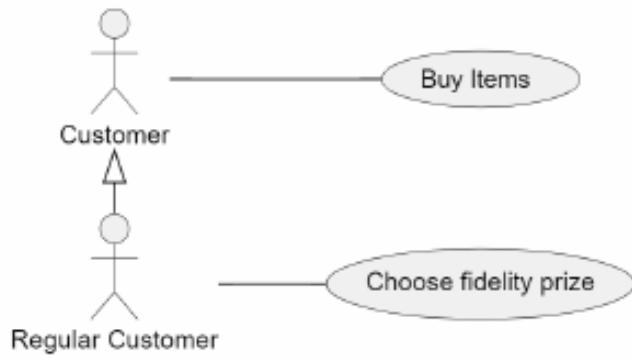


Figura 3.3: Generalizzazione

3.3.6 La relazione <<include>>

La relazione <<include>> è destinata ad essere utilizzata quando ci sono parti comuni del comportamento di due o più Casi d’Uso. Questa parte comune viene quindi estratta in un Caso d’Uso separato, da includere in tutti i Casi d’Uso base che hanno questa parte in comune. Poiché l’uso principale della relazione <<include>> è per il riutilizzo di parti comuni, ciò che rimane in un Caso d’Uso base di solito non è completo di per sé ma dipende dalle parti incluse per essere significativo. Questo si riflette nella direzione della relazione, che indica che il Caso d’Uso base dipende dall’aggiunta ma non viceversa. Questa relazione può essere utile per:

- Scomporre un’interazione complessa in interazioni più piccole e gestibili.
- Fattorizzare sequenze comuni di passaggi tra diversi casi d’uso.

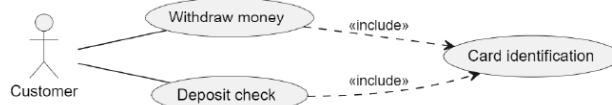


Figura 3.4: include

3.3.7 La relazione <<extend>>

La relazione <<extend>> è destinata ad essere utilizzata quando c’è un comportamento aggiuntivo che può essere aggiunto, possibilmente in modo condizionale, al comportamento definito in uno o più Casi d’Uso. Il Caso d’Uso esteso è definito indipendentemente dal Caso d’Uso che estende ed è significativo indipendentemente da esso. D’altra parte, il Caso d’Uso che estende tipicamente definisce un comportamento che potrebbe non essere necessariamente significativo da solo.

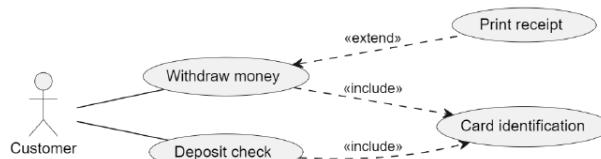


Figura 3.5: extend

Esistono anche dei punti d'estensione per cui si stabilisce nel diagramma Use Case dove può essere inserito il comportamento di un'extend, possono essere utili quando lo Use Case può essere esteso in più punti.

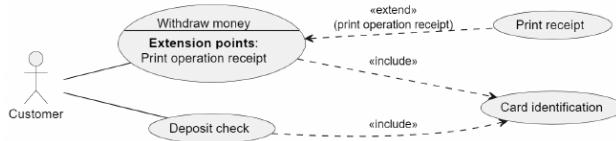


Figura 3.6: Extension point

3.4 Errori da principianti

I casi d'uso dovrebbero fornire qualche beneficio all'attore, aiutare l'attore a completare il suo lavoro o raggiungere qualche obiettivo.

- Tipicamente, i nomi dei Casi d'Uso dovrebbero includere un verbo.
- Tipicamente, i nomi degli Attori dovrebbero essere sostantivi.

Se due attori sono associati allo stesso caso d'uso (con una cardinalità diversa da zero), significa che i due attori sono coinvolti (e necessitano di collaborare) in ogni istanza (scenario) di quel caso d'uso.

Non significa che entrambi gli attori possono eseguire quel caso d'uso in modo indipendente!

Attenzione alle generalizzazioni improprie

- Ogni attore dovrebbe avere i propri casi d'uso.
- Gli attori specializzati possono già eseguire tutti i casi d'uso dei loro antenati.
- Se gli attori specializzati non hanno alcuni casi d'uso propri, la generalizzazione potrebbe essere inutile, oppure potrebbero mancare alcuni casi d'uso.

Note importanti

- La relazione <<include>> non è un buon modo per rappresentare relazioni temporali.
- I diagrammi dei casi d'uso non dovrebbero diventare troppo complessi e confusi.
- Utilizzare le relazioni tra casi d'uso e le generalizzazioni tra attori con moderazione.
- La modellazione dovrebbe essere a un sufficiente grado di astrazione.
- Bisogna essere ordinati (cercare di evitare linee che si intersecano, ecc.).
- Un diagramma complesso è indice di una cattiva analisi.

3.5 Esercizi

Exercise #1

Si vuole realizzare un sistema informativo per la gestione di segnalazioni di guasti informatici all'interno di una rete aziendale. Un impiegato dell'azienda, previa autenticazione, può compilare un form specificando una descrizione del problema, un livello di priorità della riparazione, ed il codice di inventario dell'apparecchio guasto. Un tecnico IT ha la possibilità di visualizzare tutte le segnalazioni pendenti, di prenderne in carico una, specificando una data prevista di soluzione, e di segnalarne la chiusura in seguito ad un intervento. In quest'ultimo caso, inserirà una descrizione dell'intervento effettuato, e una stima del tempo impiegato. Infine, un amministratore può visualizzare diversi report, quali ad esempio il numero di segnalazioni evase nell'ultimo mese o settimana.

Exercise #2

Un sistema gestionale per supportare il processo di raccolta e gestione dei requisiti in un'azienda software permette di gestire più progetti software. Per ciascun progetto, è possibile definire uno o più stakeholder, caratterizzati da un nome (e.g.: "Cassiere", "Cliente") e da una eventuale descrizione. Inoltre, per un certo progetto, il sistema permette di specificare uno o più requisiti. Ciascun requisito è caratterizzato da un titolo, da una descrizione, da una lista non vuota di stakeholder interessati, e da un livello di priorità. I Project Manager possono creare nuovi progetti, caratterizzati da un titolo, un committente (e.g.: azienda, pubblica amministrazione, etc.), da una durata prevista, e da un budget (in Euro). I Project Manager possono inoltre associare a ciascun progetto i Requirement Engineer allocati su quel progetto. I Requirement Engineer possono quindi visualizzare soltanto i progetti cui sono assegnati, e creare/modificare requisiti per quei progetti. Quando un Requirement Engineer crea/modifica un requisito, il sistema controlla automaticamente la presenza di ambiguità o inconsistenze nel titolo e nella descrizione, sfruttando il software esterno "Req-GPT". Se Req-GPT rileva potenziali ambiguità, il sistema non permette di salvare il requisito.

Exercise #3

Si vuole realizzare un sistema per la gestione di prestiti di libri. Gli utenti avranno la possibilità di registrarsi al sito inserendo le proprie informazioni personali come nome, cognome, e-mail e password. Ogni utente avrà un profilo univoco nel sistema. Una volta registrati, gli utenti potranno cercare libri nel catalogo inserendo il titolo, l'autore o il genere desiderato. Il sistema mostrerà quindi i dettagli dei libri trovati, inclusi titolo, autore, anno di pubblicazione e stato attuale (disponibile o in prestito). Gli utenti avranno la possibilità di richiedere il prestito di un libro disponibile. Una volta selezionato il libro di cui richiedere il prestito, l'utente potrà procedere con la richiesta di prestito, che sarà completata una volta effettuato il pagamento del contributo fisso di € 3,00. Per il pagamento, l'utente deve inserire il numero della propria carta di credito, con intestatario della stessa, scadenza, e codice CCV. Le informazioni vengono sul pagamento vengono inviate al servizio esterno UninaPay, che processa il pagamento.

Capitolo 4

Lezione 5: Use Case completi

4.1 Specificazioni degli Use Case

Il Diagramma dei Casi d'Uso (UCD) fornisce una panoramica di alto livello dei requisiti funzionali del sistema. Non è sufficientemente dettagliato per stabilire i requisiti di sistema. Per ogni Casi d'Uso (UC) nell'UCD è necessaria una specifica dettagliata. L'obiettivo è specificare ogni aspetto e dettaglio dell'interazione, dal punto di vista dell'Attore. Ogni possibile scenario e variazione dovrebbe essere descritto.

4.1.1 Descrizione testuale di un Use Case

Una descrizione di un caso d'uso generalmente include:

1. Una descrizione di ciò che il sistema e gli utenti si aspettano quando il caso d'uso inizia.
2. Una descrizione del flusso normale degli eventi nel Casi d'Uso (scenario principale).
3. Una descrizione di ciò che può causare errori e come i problemi risultanti possono essere gestiti.
4. Una descrizione dello stato del sistema dopo che il Casi d'Uso è completato.

4.1.2 Formati di Use Case

I casi d'uso possono essere scritti in diversi formati e livelli di formalità:

- **Breve:** Riepilogo conciso di un paragrafo, solitamente dello scenario di successo principale.
- **Informale:** Formato a paragrafi informali. Paragrafi multipli che coprono vari scenari.
- **Descrizione Completa (Fully-dressed):** Tutti i passaggi e le variazioni sono scritti in dettaglio e ci sono sezioni di supporto, come precondizioni e garanzie di successo.

Le descrizioni Brevi e Informali possono essere utilizzate nelle fasi iniziali della specifica dei requisiti, per avere una rapida idea del soggetto e dell'ambito. Le descrizioni Complete possono essere sviluppate successivamente, per servire come base per un contratto e specificare in maggiore dettaglio il comportamento del sistema da sviluppare.

4.1.3 Descrizioni degli Use Case completi

Sono stati proposti diversi formati per le descrizioni complete dei casi d'uso.

USE CASE #X	Name of the Use Case		
Goal in Context	Description of the objective of this UC		
Preconditions	All the conditions that must apply to start the UC		
Success End Condition	State of the system if the UC was successful		
Failed End Condition	State of the system if the UC failed		
Primary Actor	Primary actor of the UC		
Trigger	Action of the primary actor that initiates the UC		
Main Scenario	Step n.	Actor 1	Actor n
	1	Trigger action	System
	2		Response
	..	Action 2	

	n		Final action

Figura 4.1: Template di Cockburn 1

Extension #1 (short description)	Step	Actor 1	Actor n	System
	x <condition>

	...			Final action (possibly return to a step of the main scenario)
Extension #n (short description)	Step	Actor 1	Actor n	System
	y <condition>

	...			Final action
Open Issues	List all the aspects that still need to be clarified. At the delivery of the doc must be empty			

Figura 4.2: Template di Cockburn 2

4.1.4 Scenari principali ed estensioni

Lo scenario principale è la sequenza di azioni che si verifica quando tutto nel caso d'uso procede senza intoppi come previsto.

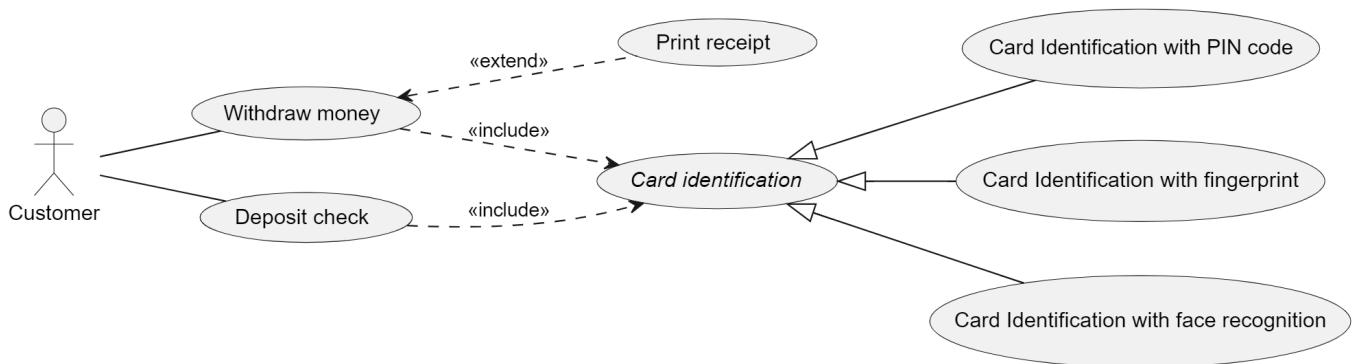
Tuttavia, possono esserci diversi modi per eseguire un caso d'uso:

- Gli utenti possono autenticarsi utilizzando il PIN o uno scanner per impronte digitali.
- Potrebbe verificarsi un errore in qualche punto.

Quando si definisce il comportamento funzionale del sistema, è importante descrivere anche queste sequenze alternative di azioni che possono verificarsi durante l'esecuzione di un caso d'uso.

- Ciò viene fatto utilizzando le **Estensioni**.
- Tipicamente, c'è molto più testo nelle Estensioni che nello Scenario Principale.

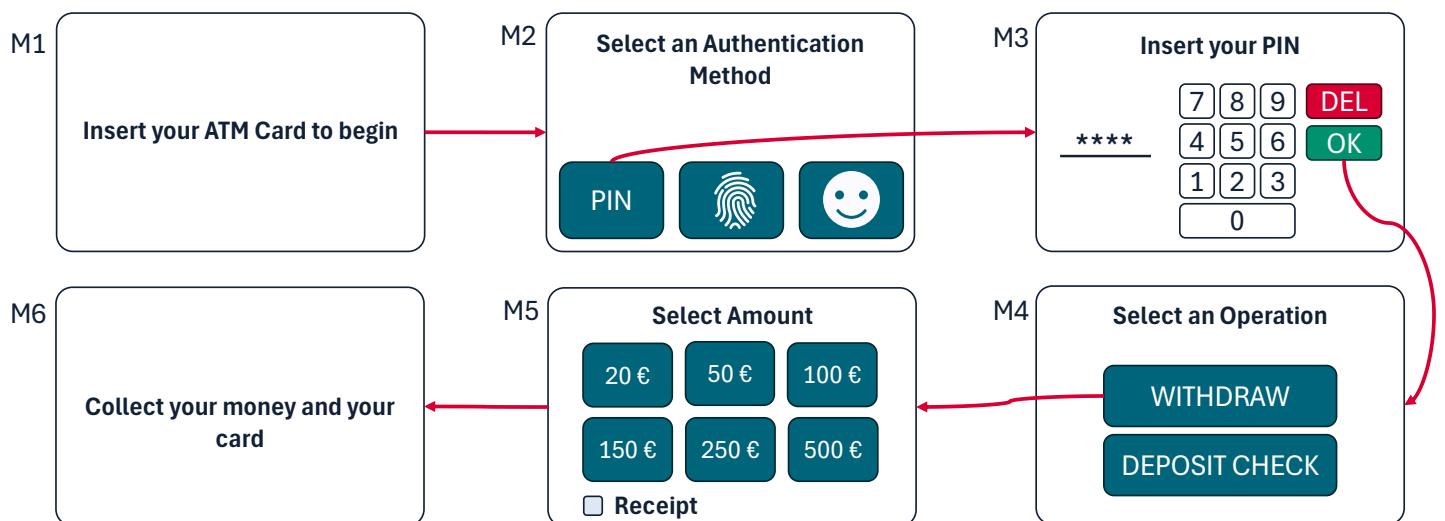
Example: ATM System



- Suppose we want to describe the **Withdraw money** use case, using a fully-dressed format

Example: Mockups

- It may be useful to design some mockups of the system



Example: Fully-dressed Use Case

USE CASE #1	Withdraw money		
Goal in Context	A customer wants to withdraw money from the ATM		
Preconditions	The customer has an account at the Bank and owns a bank card		
Success End Condition	The system keeps track of the withdrawal operation and erogates the requested money		
Failed End Condition	No transaction is made		
Primary Actor	Customer		
Trigger	Customer walks up to the system and touches the screen to activate it		
Main Scenario	Step n.	Customer	System
	1	Touches screen	
	2		Shows M1
	3	Inserts card	
	4		Shows M2

Example: Fully-dressed Use Case

USE CASE #1	Withdraw money		
Main Scenario	Step n.	Customer	System
	5	Touches «PIN» button	
	6		Shows M3
	7	Inserts PIN	
	8		Shows M4
	9	Touches «Withdraw» button	
	10		Shows M5
	11	Touches «50 €» button	
	12		Erogates money, Ejects card, Shows M6

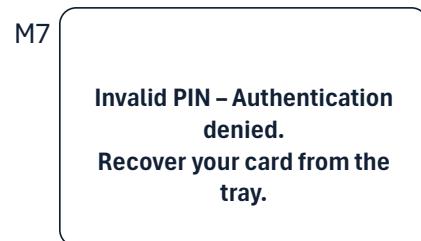
Example: Extensions

- What can go wrong?
 - PIN might not be correct
 - Customer might not have enough money in their account
 - ATM might not have enough cash reserves to erogate the required money
 - Card might be flagged as stolen
 - Card might be unreadable
 - ...
- What could go differently?
 - Customers might authenticate themselves using their fingerprint or face recognition
 - Customers might opt-in to get the printed receipt

Example: Extensions

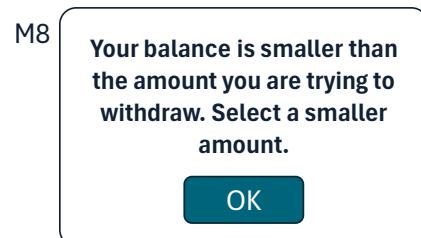
- Each of these scenarios should be detailed using extensions

Extension #1 (customer inserts an invalid PIN)	Step	Customer	System
	7a <wrong PIN is inserted>	Inserts PIN	
	8a		Shows M7 and terminates UC



Example: Extensions

Extension #2 (customer does not have enough money)	Step	Customer	System
	11b	Selects «500€» button	
	12b		Shows M8
	13b	Clicks ok	
	14b		Return to step 10 of the Main Scenario



4.3 Validazione dei requisiti

La validazione punta a dimostrare che i requisiti definiscono il sistema che il cliente desidera veramente. I costi degli errori nei requisiti sono elevati, quindi la validazione è molto importante. Correggere un errore nei requisiti dopo la consegna può costare fino a 100 volte il costo della correzione di un errore di implementazione.

4.3.1 Controllo dei requisiti

Ci sono delle domande da fare per il controllo dei requisiti:

- **Validità:** Il sistema fornisce le funzioni che supportano al meglio le esigenze del cliente?
- **Consistenza:** Ci sono conflitti tra i requisiti?
- **Completezza:** Tutte le funzioni richieste dal cliente sono incluse?
- **Realismo:** I requisiti possono essere implementati considerando il budget e la tecnologia disponibili?
- **Verificabilità:** I requisiti possono essere verificati?

4.3.2 Tecniche di validazione dei requisiti

Esistono delle tecniche per validare i requisiti:

- **Revisioni dei requisiti:** Analisi manuale sistematica dei requisiti.
- **Prototipazione:** Utilizzo di un modello eseguibile semplificato del sistema per verificare i requisiti. Oppure prototipazione visiva (ad esempio, utilizzando mockup/wireframe).
- **Generazione di test case:** Sviluppo di test per i requisiti per verificarne la testabilità.

Dovrebbero essere effettuate revisioni regolari durante la formulazione della definizione dei requisiti. Sia il personale del cliente che quello dell'appaltatore dovrebbero essere coinvolti nelle revisioni. Le revisioni possono essere formali (con documenti completati) o informali. Una buona comunicazione tra sviluppatori, clienti e utenti può risolvere i problemi in una fase iniziale.

Capitolo 5

Lezione 6 e 7: Statecharts

5.1 Statechart

Conosciuti anche come Macchine a Stati (Comportamentali) UML. Ampiamente utilizzati per modellare gli aspetti dinamici dei sistemi (specialmente quelli reattivi). Sistemi che reagiscono a eventi (esterni o interni). Gli Statechart sono ampiamente utilizzati nell'industria, e non solo per la modellazione.

5.1.1 Modellazione con stati e transizioni

Gli stati rappresentano situazioni in cui vale una condizione invariante.

- **Condizioni statiche:** il sistema è in attesa che qualcosa accada.
- **Condizioni dinamiche:** il sistema sta eseguendo un compito specifico.

Le transizioni rappresentano possibili cambiamenti di stato.

5.1.2 Regioni, vertici e transizioni

Uno Statechart UML contiene una regione di primo livello (top-level). Una regione contiene vertici e transizioni. I vertici rappresentano gli stati. Le transizioni sono rappresentate come archi orientati tra due vertici. Esistono diversi tipi di vertici, con semantiche differenti.

5.1.3 Stati (semplici)

Rappresentano stati del sistema non strutturati. Raffigurati come un rettangolo con angoli arrotondati. Un compartimento del nome contiene il nome (opzionale) dello stato, come stringa.

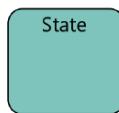


Figura 5.1: Uno stato semplice

5.1.4 Pseudostati iniziali e stati finali

Gli pseudostati iniziali sono utilizzati per segnare lo stato predefinito (iniziale). Una regione può contenere al massimo uno pseudostato iniziale. Gli stati finali modellano una situazione in cui il calcolo è completato (cioè, il sistema non elaborerà ulteriori eventi).

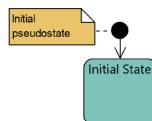


Figura 5.2: Pseudostato e stato iniziale

5.1.5 Sintassi delle transizioni

Le transizioni indicano cambiamenti di stato. Possono essere decorate con un'etichetta della forma:

trigger [guardia] / azioni

- *trigger* (nell'immagine `event()` è un trigger) è un elenco di eventi che possono indurre un cambiamento di stato.
- *guardia* è una condizione booleana.
- *azioni* è un elenco di operazioni da eseguire quando la transizione si attiva.

Tutte le parti sopra indicate dell'etichetta sono opzionali. Sono possibili auto-transizioni (self-transitions).

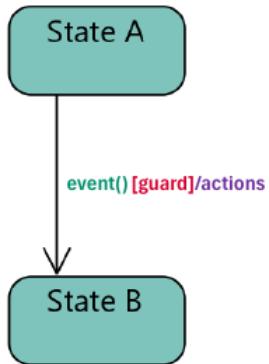


Figura 5.3: trigger [guardia] / azioni

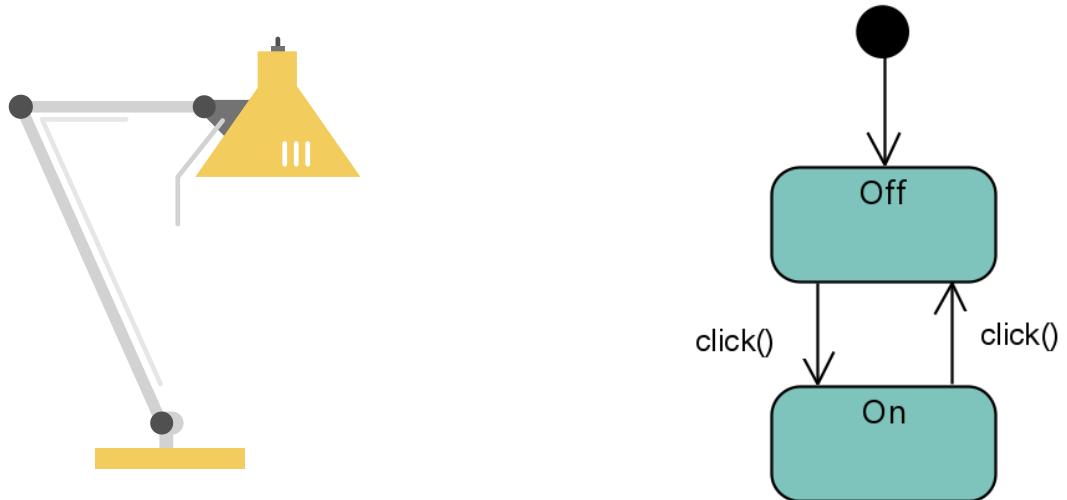
5.1.6 Semantica delle transizioni

Affinché una transizione possa essere attivata:

- Devono essere generati eventi corrispondenti a tutti i trigger;
- La condizione nella guardia deve essere valutata come VERA.

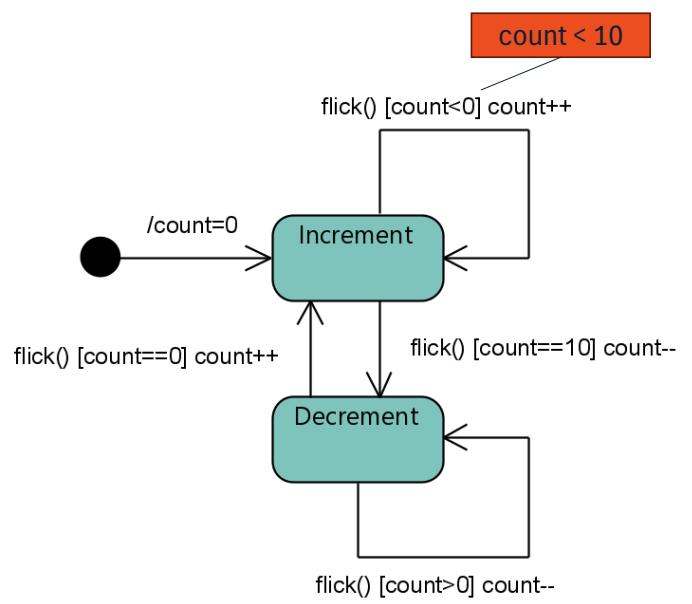
Una transizione spontanea è una transizione senza trigger e senza guardia. Dopo che una transizione viene attivata, la sua lista associata di azioni viene eseguita. Se più transizioni sono attivabili, solo una di esse viene effettivamente attivata (in modo non deterministico). Seguono degli esempi.

Example: a lamp with a single button



Example: a simple counter (Java)

```
public class Counter {  
    private int count = 0;  
    private String mode = "increment";  
    public void flick() {  
  
        if(mode.equals("increment"))  
            count++;  
        else  
            count--;  
  
        if(count==10)  
            mode = "decrement";  
        else if (count==0)  
            mode = "increment";  
    }  
}
```



5.1.7 Attività interne degli stati

Gli stati possono (opzionalmente) contenere una lista di attività interne. Ogni attività è caratterizzata da un’etichetta che indica quando l’attività deve essere invocata. Etichette riservate:

- **entry** / attività eseguita all’ingresso dello stato
- **do** / attività eseguita finché il sistema si trova nello stato (dopo il completamento delle attività di entry)
- **exit** / attività eseguita all’uscita dallo stato

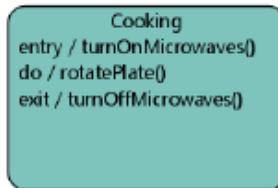


Figura 5.4: Attività interne

5.1.8 Stati composti

Uno stato può contenere:

- un compartimento per il nome
- un compartimento per le attività interne
- una o più regioni interne!

Uno stato con regioni interne è uno stato composto. Gli stati in una regione interna sono chiamati sottostati. Permettono ai modellatori di definire una struttura gerarchica. La regione interna dettaglia il comportamento

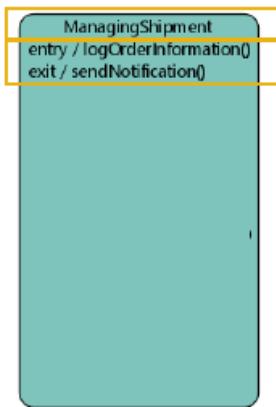


Figura 5.5: Stato composto

dello stato a cui appartiene. Forniscono un modo elegante e conciso per modellare comportamenti complessi (e nascondere la complessità quando non è necessaria).



The ManagingShipment composite state, with the inner region hidden

Figura 5.6: Stato composto 2

5.1.9 Regioni parallele

Gli stati composti possono contenere più regioni, rappresentando comportamenti che possono verificarsi in parallelo. Quando si esce da uno stato composto, tutte le sue regioni vengono terminate.

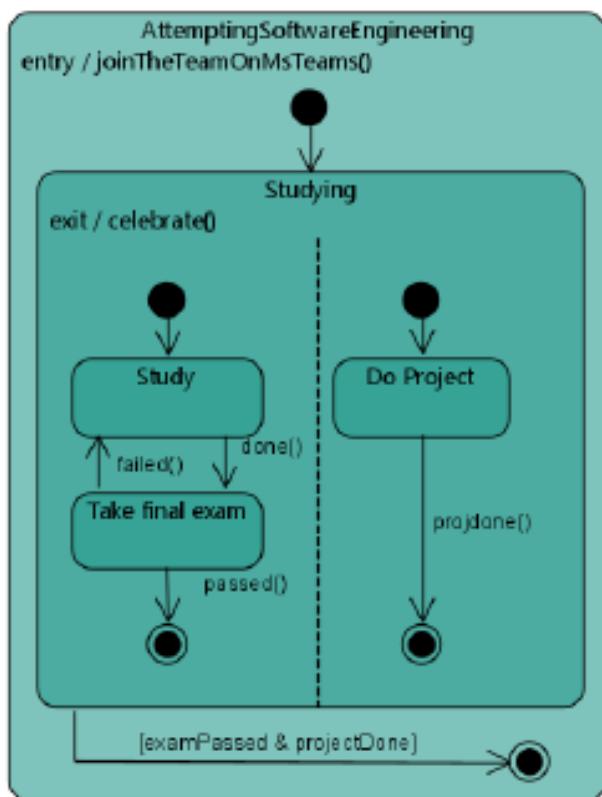


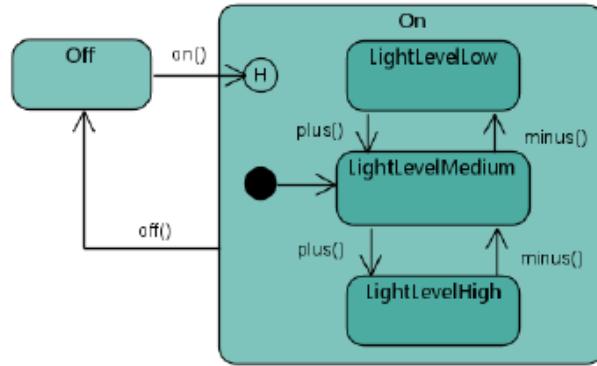
Figura 5.7: Regioni parallele

5.1.10 Pseudostati di Shallow History

Raffigurati come una H (circondata da un cerchio). Rappresenta lo stato più recentemente attivo di uno stato composto, ma non i sottostati di quello stato! Solo negli stati composti, e solo uno per regione.

5.1.11 Pseudostati di Deep History

Raffigurati come una H* (sempre circondata da un cerchio). Stessa funzione di quelli di shallow history, ma ripristina l'intera configurazione della regione (sottostati dei sottostati inclusi!).



Statechart for a lamp with three different light levels

Figura 5.8: Shallow history

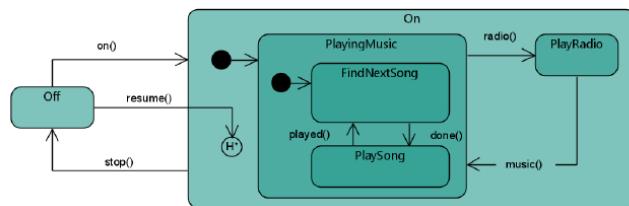


Figura 5.9: Deep History

5.1.12 Pseudostati Fork e Join

- I Fork dividono le transizioni in ingresso in più transizioni che entrano in vertici in regioni ortogonali.
- I Join uniscono le transizioni in uscita da vertici in regioni ortogonali in un'unica transizione.

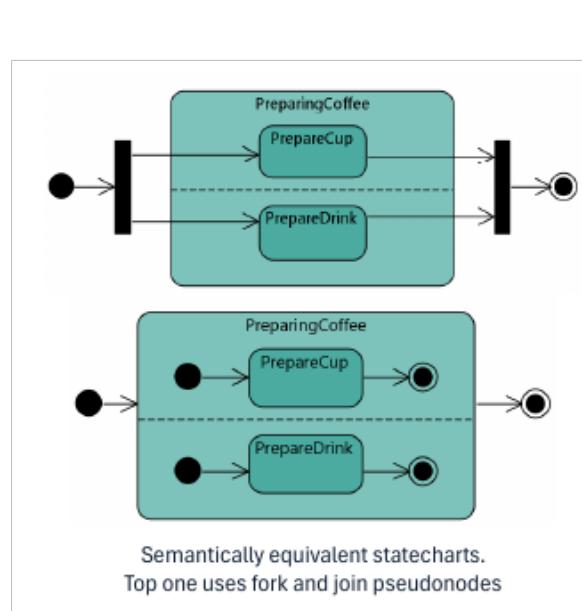


Figura 5.10: Fork e Join

5.1.13 Consigli e trucchi per gli Statechart

Ogni stato dovrebbe tipicamente avere almeno una transizione in entrata e una in uscita. I diagrammi sono tipicamente letti dall'alto a sinistra verso il basso a destra, quindi posizionare gli pseudostati iniziali/finali di conseguenza! Se più stati hanno una condizione di ingresso e/o uscita comune, considerare l'uso di stati composti. Assicurarsi di non modellare il non-determinismo, a meno che non sia ciò di cui si ha veramente bisogno! Al massimo uno stato può essere attivo in una regione in qualsiasi momento!

5.2 Statecharts nel mondo reale

5.2.1 Model-driven Development

Passo successivo nella tendenza all'aumento dell'astrazione Standard de-facto in molti domini del software embedded (es: automotive) Grazie a strumenti come Simulink, è possibile simulare modelli Statechart, generare automaticamente codice e test, e molto altro (es: metodi formali!)

Vantaggi

- In alcuni domini, tipicamente più conveniente, più veloce e porta a una qualità superiore
- Modelli comprensibili da esperti di dominio
- I modelli sono documentazione!
- Minore dipendenza dalla tecnologia
- Minore dipendenza dal personale

Svantaggi

- Gli strumenti sono costosi
- Non abbastanza flessibili per alcune applications
- La generazione di codice è tipicamente supportata per un numero limitato di piattaforme

5.2.2 Gestione degli stati dell'interfaccia utente con Statecharts

Gli Statechart possono anche essere utilizzati per «guidare» la logica della GUI Gli Statechart sono più facili da comprendere (rispetto al codice!) Il comportamento è disaccoppiato dai componenti GUI Separare il QUANDO (codificato nello Statechart) dal COSA (cosa dovrebbe accadere, codificato nel componente UI) Gli Statechart scalano bene con la crescita della complessità

Example: Statechart-based UI with XState

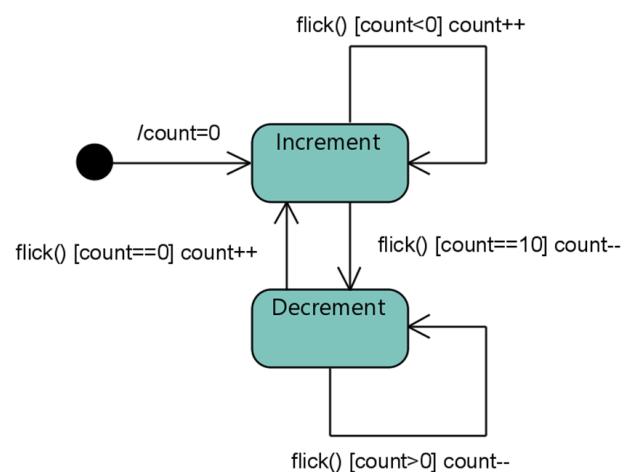
- XState is an open-source Javascript library to create, interpret, and execute statechart models
- Can be integrated with many Javascript UI libraries such as React, Vue, Svelte
- Great at managing UI State through Statecharts
- Also supports testing!
- Available at: <https://xstate.js.org/>



Example: Statechart-based UI with XState

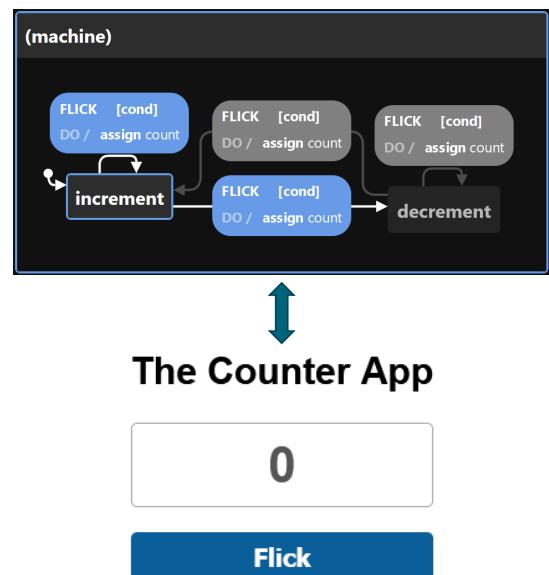
Remember our Counter example?

```
public class Counter {  
    private int count = 0;  
    private String mode = "increment";  
    public void flick() {  
        if(count>10)  
            mode = "decrement";  
        else if (count<0)  
            mode = "increment";  
  
        if(mode.equals("increment"))  
            count++;  
        else  
            count--;  
    }  
}
```



Example: Statechart-based UI with XState

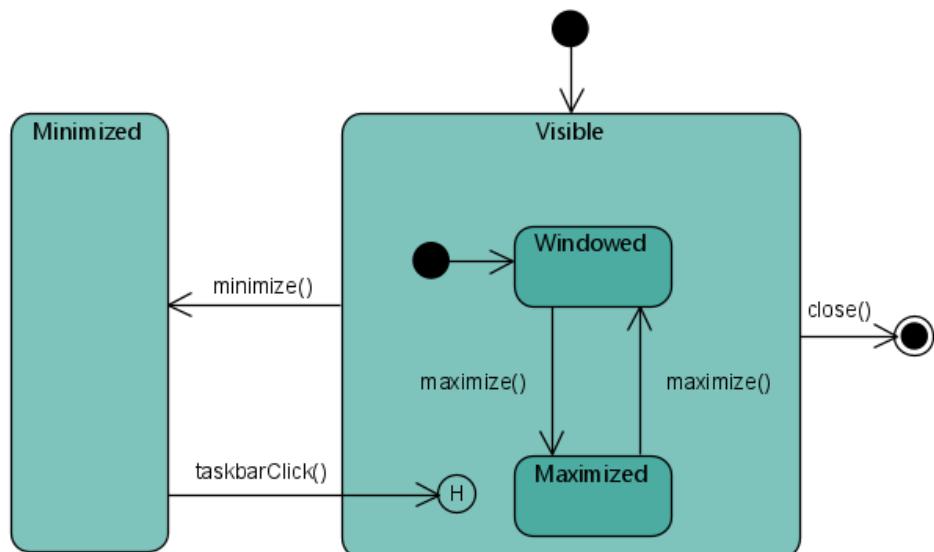
- Let's implement a simple UI for it, and let's do it the Statechart way, with Xstate and React
- Code Sandbox available [here](#):



Exercise #1

Si descriva con uno Statechart il comportamento di una generica finestra (e.g.: minimizzata, massimizzata, modalità finestra, etc.) in un ambiente desktop basato su finestre (come quello di Microsoft Windows).

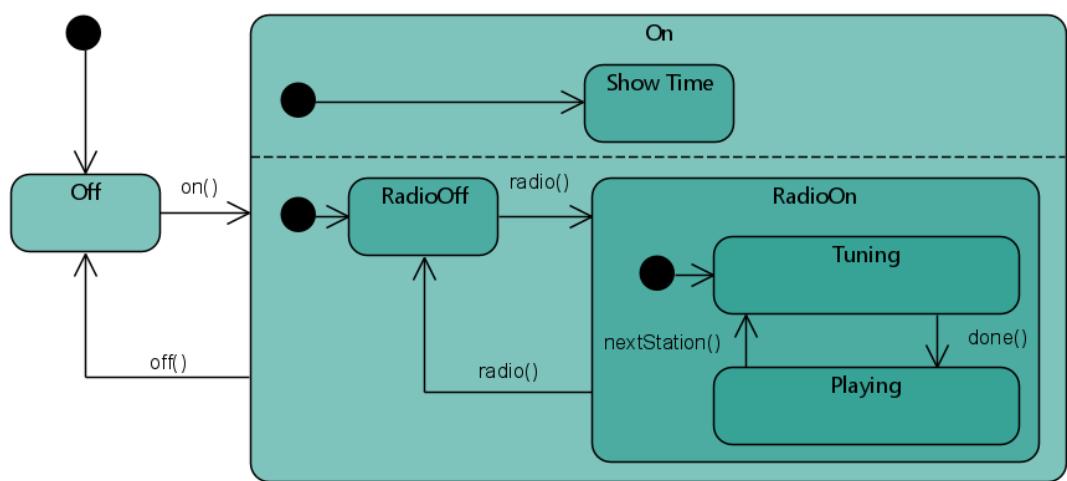
Exercise #1 – Proposed solution



Exercise #2

Un orologio da tavolo, una volta acceso, mostra l'orario corrente sul proprio display LCD e, se l'utente preme un apposito pulsante, può anche sintonizzarsi su stazioni radio e riprodurne le trasmissioni dalle casse integrate. Tramite un pulsante «next station» è possibile passare alla stazione radio successiva, che verrà riprodotta dopo una breve fase di ricerca e sintonizzazione.

Exercise #2 – Proposed solution



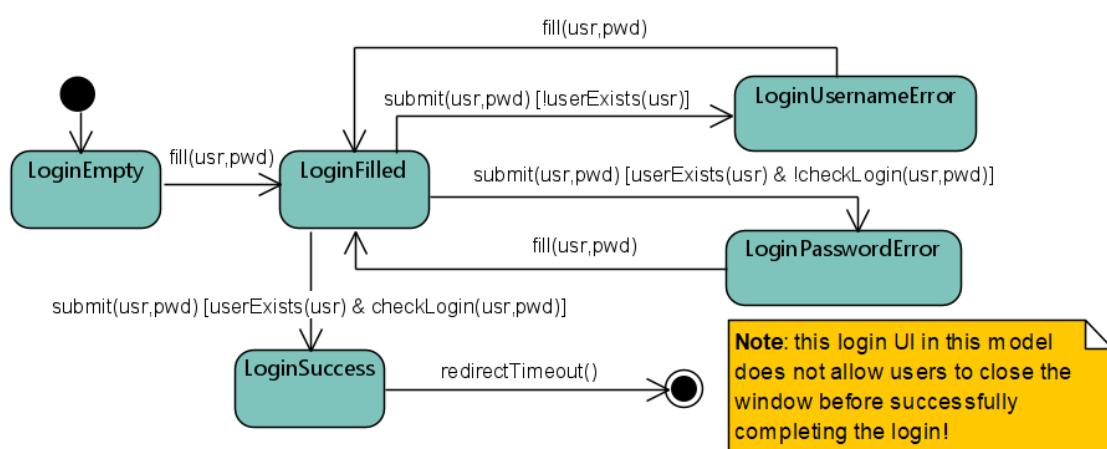
Exercise #2 – Follow up

Come si potrebbe modificare lo statechart precedente per fare in modo che, all'accensione, l'orologio da tavolo riprenda con la riproduzione della radio se la radio era attiva nel momento dello spegnimento?

Exercise #3

La schermata di login di un'applicazione permette agli utenti di inserire le proprie credenziali ed accedere. Se il nome utente inserito non è tra quelli presenti nel sistema, viene mostrato un warning dedicato. Altrimenti, se il nome è presente ma la password errata, viene mostrato un diverso warning e viene abilitato un pulsante per accedere alla funzionalità di reset password. Se le credenziali sono corrette, si accede al sistema.

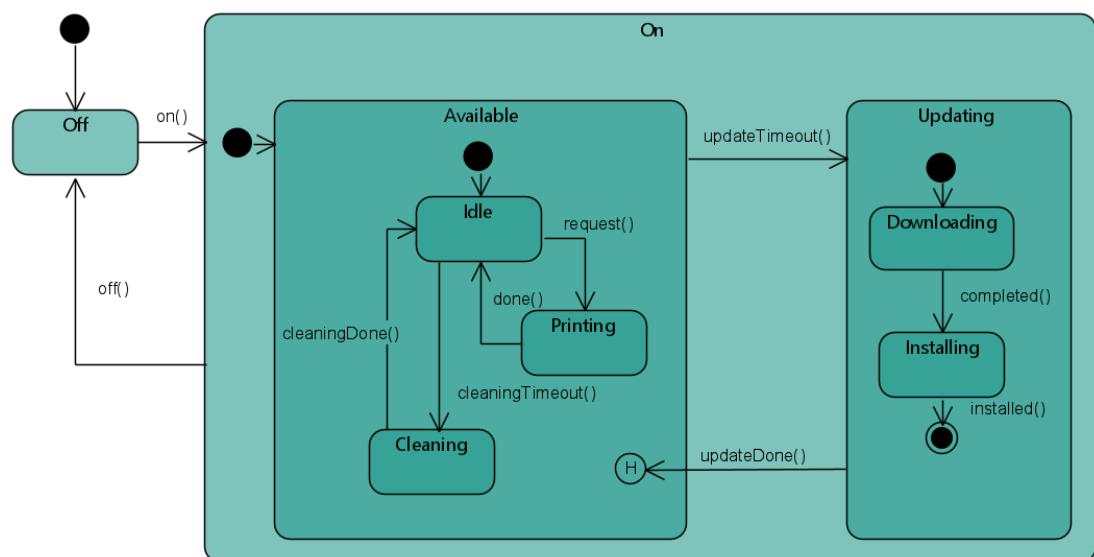
Exercise #3 – Proposed solution



Exercise #4

Una stampante, previa accensione, resta in attesa di ricevere via rete documenti da stampare. In presenza di richieste, la stampante procede alla stampa. Quando è accesa e non è in fase di stampa, la stampante, una volta al giorno, effettua la pulizia delle testine. Inoltre, sempre con cadenza giornaliera, la stampante scarica e installa aggiornamenti dalla casa madre. In questo caso, la stampante interrompe qualsiasi attività in corso per effettuare l'aggiornamento, e le riprende ad aggiornamento effettuato.

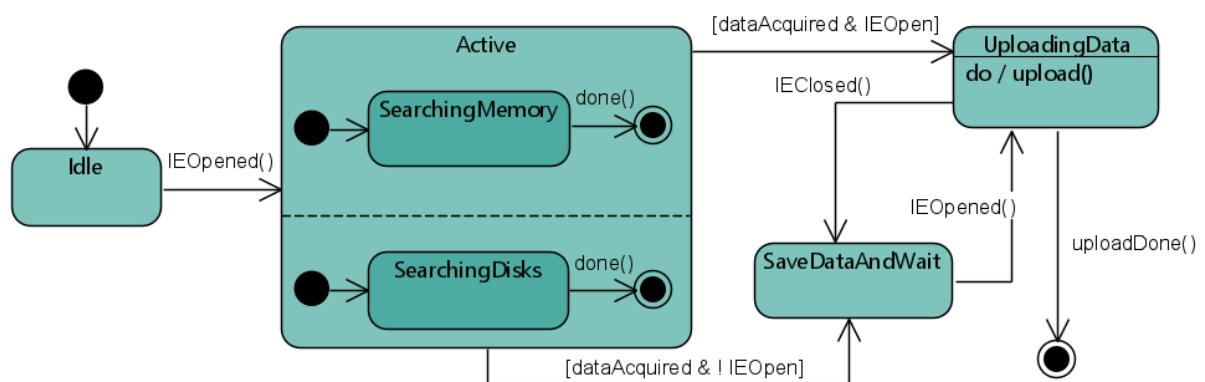
Exercise #4 – Proposed solution



Exercise #5

Un malware, una volta installato su un PC, rimane latente fino a quando l’utente non apre Internet Explorer. A quel punto, il malware si attiva e, in parallelo, ricerca informazioni sensibili nei dischi rigidi e nella memoria del PC. Terminate queste attività, il malware sfrutta una vulnerabilità di Internet Explorer per inviare le informazioni raccolte a un server remoto. Se Internet Explorer viene chiuso prima dell’invio delle informazioni, il malware salva le informazioni trovate e riprova ad inviarle al successivo avvio di Internet Explorer.

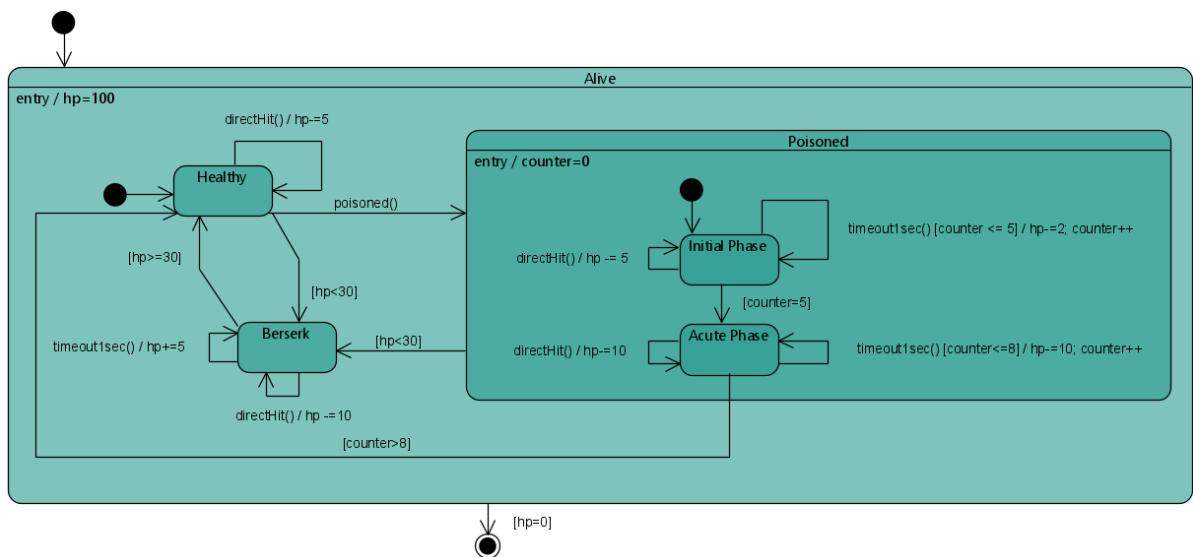
Exercise #5 – Proposed solution



Exercise #6 – Video Game Player

In un videogame, il giocatore inizia la partita con una salute pari a 100 HP. Durante la partita, il giocatore può subire attacchi diretti, che diminuiscono la salute residua di 5 HP. Inoltre, il giocatore può essere avvelenato. L'avvelenamento prevede una fase iniziale che dura 5 secondi, in cui il giocatore subisce 2 HP di danno al secondo, e una fase acuta, che dura 3 secondi e durante la quale il giocatore subisce 10 HP di danno al secondo. Mentre il giocatore è avvelenato in fase acuta, i danni inflitti da attacchi diretti raddoppiano. Quando la salute scende al di sotto della soglia critica di 30 HP, il giocatore passa in modalità “berserk”. Quando è in questa modalità, il giocatore è immune all'avvelenamento e si cura di 5 HP al secondo, ma subisce danni doppi dai colpi diretti. Quando i punti salute scendono a zero, il giocatore muore e la partita termina.

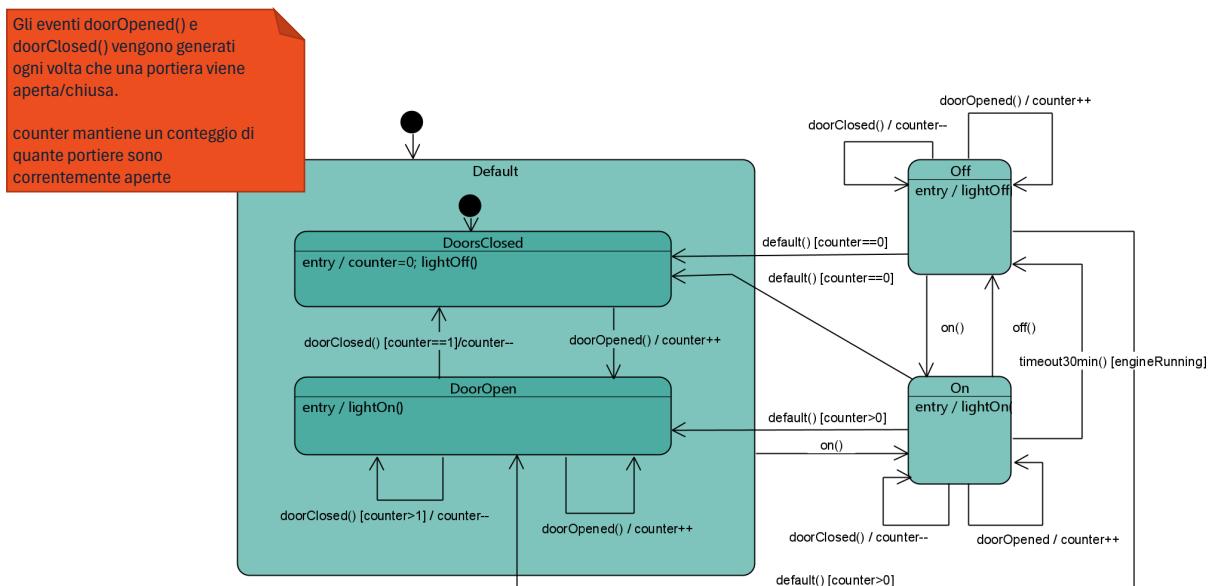
Exercise #6 – Proposed solution



Exercise #7

Le luci di cortesia di un'auto hanno un interruttore che può assumere tre posizioni: ON, OFF, e DEFAULT. Quando l'interruttore è in posizione ON, le luci di cortesia sono sempre accese. Al contrario, quando è in posizione OFF, le luci di cortesia sono sempre spente. Quando l'interruttore è in posizione DEFAULT, le luci si accendono soltanto quando una delle portiere è aperta, e restano spente altrimenti. Inoltre, quando il motore è spento e l'interruttore è in posizione ON, le luci si spengono in ogni caso dopo 30 minuti per evitare di consumare la batteria, e l'interruttore si sposta su OFF.

Exercise #7 – Proposed solution



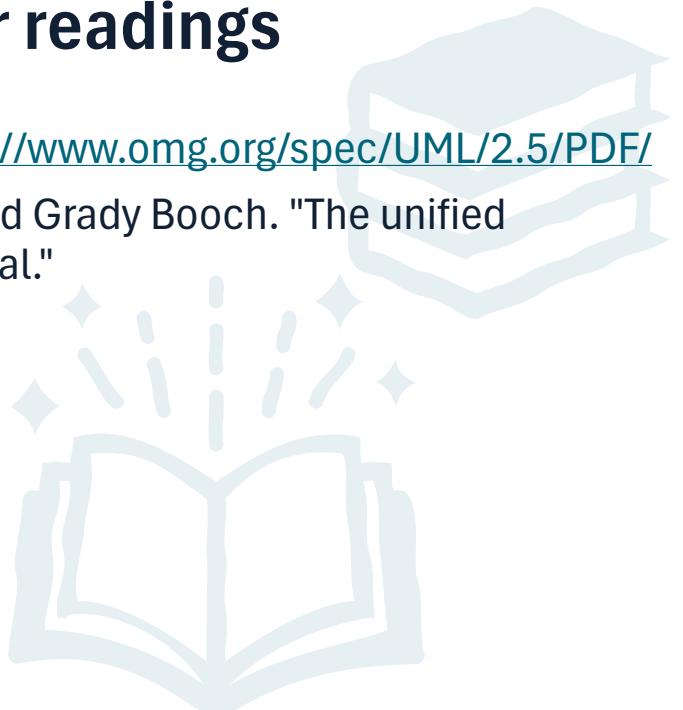
Exercise #7 – Follow up

The proposed solution is quite complex. Is it possible to express the same behaviours with a simpler statechart?

Hint: try introducing some composite states!

References and further readings

- OMG UML Specification (2.5) <https://www.omg.org/spec/UML/2.5/PDF/>
- Ivar Jacobson, James Rumbaugh and Grady Booch. "The unified modeling language reference manual."



Capitolo 6

Lezione 8: Usabilità e Progettazione Centrata sull'uso umano

6.1 L'ascesa dell'usabilità

Oggi dobbiamo preoccuparci di costruire sistemi con una buona usabilità. Usabilità: una misura qualitativa della facilità ed efficienza con cui un essere umano può utilizzare le funzioni e le caratteristiche offerte dal sistema. Indipendentemente dal tipo di software che stiamo costruendo, una buona usabilità può fare la differenza tra successo e fallimento, e persino tra vita e morte!

6.1.1 Usabilità - App per il grande pubblico

Se stiamo costruendo una nuova app di social media, un sito web di e-commerce, o un'altra app per smartphone per fare qualcosa:

- Facilità di apprendimento, bassi tassi di errore e soddisfazione soggettiva sono fondamentali
- L'uso è discrezionale e (generalmente) la concorrenza è agguerrita
- Se gli utenti non riescono a ottenere risultati rapidamente e senza sforzo, rinunceranno e proveranno un fornitore concorrente

6.1.2 Usabilità - Software professionale

Per software utilizzato in ambiente professionale (bancario, assicurativo, gestione della produzione, prenotazioni, fatturazione utilities, ...)

- Il tempo di formazione è un costo, la facilità d'uso è importante
- L'internazionalizzazione può essere necessaria
- La velocità di esecuzione è importante e l'affaticamento, lo stress e l'esaurimento degli operatori sono preoccupazioni
- Ridurre del 10% il tempo medio di transazione potrebbe significare il 10% in meno di operatori, il 10% in meno di postazioni di lavoro, ...

6.1.3 Usabilità - Sistemi critici per la vita

Per sistemi critici per la vita come quelli che controllano il traffico aereo, i reattori nucleari, le utilities energetiche, i servizi di emergenza, le operazioni militari e le cure cliniche)

- I costi elevati dovuti ai lunghi tempi di formazione sono previsti, ma dovrebbero garantire prestazioni rapide e senza errori anche quando gli utenti sono sotto pressione
- Errori o ritardi nell'esecuzione possono causare danni gravi!

6.1.4 Interazione Uomo-macchina (HCI)

L'HCI è la disciplina che studia come gli esseri umani interagiscono con i computer, e come progettare interfacce utente efficaci e usabili.

6.1.5 Usabilità vs User-friendliness

Quando i fornitori di computer e software iniziarono a vedere gli utenti come più di un inconveniente, iniziarono a descrivere i loro sistemi come user friendly Non è un termine molto buono da usare

- **Inutilmente antropomorfico:** gli utenti hanno bisogno di sistemi che li aiutino a svolgere il loro lavoro e non intralcino. Non hanno bisogno che i sistemi siano amichevoli con loro!
- **Implica che i bisogni degli utenti possano essere descritti lungo una singola dimensione** da sistemi che sono più o meno amichevoli
- In pratica, sistemi che sono amichevoli per un utente possono risultare tediosi per altri

6.2 Definizione di usabilità

L'usabilità è definita mediante 5 attributi di qualità:

- **Apprendibilità:** Quanto è facile per gli utenti portare a termine compiti basici la prima volta che incontrano il design?
- **Efficienza:** Dopo che gli utenti hanno appreso il design, quanto rapidamente possono eseguire i compiti?
- **Memorabilità:** Quando gli utenti tornano al design dopo un periodo di inutilizzo, quanto facilmente possono riacquistare la padronanza?
- **Errori:** Quanti errori fanno gli utenti, quanto sono gravi questi errori e quanto facilmente possono riprendersi dagli errori?
- **Soddisfazione:** Quanto è piacevole usare il design?

6.2.1 Apprendibilità

Probabilmente l'attributo di usabilità più fondamentale La maggior parte dei sistemi deve essere facile da apprendere Per alcuni sistemi specializzati, è accettabile che siano difficili da apprendere ma altamente efficienti per utenti esperti I cosiddetti sistemi "walk-up-and-use" (ad es.: sistemi informativi museali) sono pensati per essere utilizzati una sola volta Questi sistemi richiedono essenzialmente un tempo di apprendimento zero: gli utenti dovrebbero avere successo la prima volta che li usano!

6.2.2 Efficienza d'uso

L'efficienza si riferisce al livello di performance stabile dell'utente esperto quando la curva di apprendimento si appiattisce Potrebbero volerci mesi o anni per raggiungere quella fase! Per misurare l'efficienza:

- Decidere una definizione di competenza esperta
- Ottenere un campione rappresentativo di utenti con quel livello di competenza
- Misurare il tempo che impiegano per completare alcuni compiti tipici

6.2.3 Memorabilità

Gli utenti occasionali sono una terza categoria di utenti oltre ai principianti e agli esperti Gli utenti occasionali utilizzano il sistema in modo intermittente A differenza dei principianti, non hanno bisogno di apprenderlo da zero, ma hanno bisogno di ricordare come usarlo in base al loro apprendimento precedente L'uso occasionale tipicamente avviene per:

- Software che non fanno parte del lavoro principale di un utente
- Software che sono intrinsecamente utilizzati a lunghi intervalli (ad es.: per redigere rapporti annuali)
- Software che sono utilizzati solo in circostanze eccezionali

Interfacce memorabili sono utili anche per utenti che tornano a utilizzare il sistema dopo essere stati in vacanza, o hanno temporaneamente smesso di usarlo I miglioramenti nell'apprendibilità rendono un'interfaccia anche facile da ricordare In principio, tuttavia, l'usabilità del ritorno a un sistema è diversa dall'affrontarlo per la prima volta

6.2.4 Errori

Un errore è qualsiasi azione che non raggiunge l'obiettivo desiderato Il tasso di errore può essere misurato contando il numero di errori commessi dagli utenti durante l'esecuzione di un compito (come parte di un esperimento per misurare anche altri attributi di usabilità)

- Il semplice conteggio degli errori potrebbe essere fuorviante: alcuni errori vengono corretti immediatamente dagli utenti e hanno il solo effetto di rallentarli (riducendo in qualche modo la velocità di transazione)
- Altri errori sono più catastrofici per natura: l'utente non se ne accorge, portando a un prodotto di lavoro difettoso; potrebbe essere impossibile riprendersi dall'errore

Gli utenti dovrebbero commettere il minor numero possibile di errori quando utilizzano un software E almeno, dovrebbero commettere pochissimi errori catastrofici, se non nessuno!

6.2.5 Soddisfazione soggettiva

Questo attributo si riferisce a quanto sia piacevole usare il sistema È particolarmente importante per i sistemi utilizzati su base discrezionale: videogiochi, pittura creativa, ... Per alcuni di questi sistemi, il loro valore di intrattenimento è più importante della velocità con cui le cose vengono fatte, poiché si potrebbe voler passare un tempo più lungo divertendosi

6.2.6 Compromessi nell'usabilità

Non è sempre possibile massimizzare tutti gli attributi di usabilità simultaneamente

- Potrebbero essere necessari compromessi: per evitare errori catastrofici, potremmo progettare un'interfaccia utente meno efficiente, che pone domande extra per assicurarsi che l'utente voglia realmente eseguire una certa azione
- In alcuni casi, potremmo ottenere una situazione win-win: l'apprendibilità e l'efficienza d'uso per gli esperti non sono necessariamente in conflitto. Potremmo essere in grado di ottenere il meglio di entrambe le curve di apprendimento, ad esempio includendo acceleratori (es.: scorciatoie da tastiera o hotkeys) nella nostra UI

6.2.7 Ingegneria dell'usabilità

Molti progetti di sviluppo software falliscono nel raggiungere i loro obiettivi Molti di questi fallimenti sono dovuti a scarse comunicazioni tra sviluppatori e clienti e tra sviluppatori e utenti Ciò si traduce in interfacce utente che costringono gli utenti ad adattare e cambiare il loro comportamento piuttosto che soddisfare le esigenze degli utenti

6.3 Progettazione Centrata sull'Uomo (HCD)

Non è un'attività una tantum in cui l'interfaccia utente viene sistemata prima del rilascio Un insieme di attività che idealmente si svolgono durante l'intero Ciclo di Vita del Software ISO 9241-210 definisce la Progettazione Centrata sull'Uomo (HCD)

- Gli sviluppatori devono mantenere una prospettiva centrata sull'uomo
- Gli utenti devono svolgere un ruolo centrale durante l'intero ciclo di vita
- Complementare alle metodologie di progettazione esistenti
- Fornisce una prospettiva centrata sull'uomo che può essere integrata in diversi processi di progettazione e sviluppo

6.3.1 Principi HCD

La progettazione si basa sulla comprensione esplicita di utenti, compiti e ambienti Gli utenti sono coinvolti il più possibile nella progettazione e nello sviluppo La progettazione è guidata e raffinata dalla valutazione centrata sull'utente Il processo può essere iterativo, se necessario La progettazione affronta l'intera esperienza utente Il team di progettazione dovrebbe includere competenze e prospettive multidisciplinari

6.3.2 HCD: Comprendere il Contesto e gli Utenti

Contesto: quali sono i tipi di utilizzo del sistema?

- Sistema critico per la vita?
- Industriale? Commerciale? Militare? Scientifico?
- Intrattenimento?

In quale mercato compete il sistema?

- Progetto di sviluppo software personalizzato?
- Sistema per aziende?
- App per il grande pubblico?

Utenti: conosci i tuoi utenti (come abbiamo fatto nell'Ingegneria dei Requisiti)! Personas e scenari (mantenere sempre presenti i bisogni degli utenti) Ma è necessario considerare anche:

- Attributi fisici (età, genere, dimensioni, portata, angoli visivi, ecc.)
- Abilità percettive (udito, vista, sensibilità al calore...)
- Abilità cognitive (capacità di memoria, livello di lettura, formazione musicale, matematica...)
- Posti di lavoro fisici (altezza tavolo, livelli sonori, illuminazione, versione software...)
- Tratti di personalità e sociali (preferenze, antipatie, pazienza...)
- Diversità culturale e internazionale (lingue, flusso delle finestre di dialogo, simboli...)
- Popolazioni speciali, (dis)abilità

6.3.3 HCD: Progettare per Soddisfare i Requisiti

Una progettazione appropriata del sistema si basa su una chiara comprensione del contesto e degli utenti! La produzione di soluzioni di progettazione dovrebbe includere le seguenti sotto-attività:

- a) progettazione di compiti utente, interazione utente-sistema e interfaccia utente per soddisfare i requisiti utente, tenendo in considerazione l'intera esperienza utente;
- b) rendere le soluzioni di progettazione più concrete (ad es.: prototipi o mock-up);
- c) migliorare le soluzioni di progettazione basandosi su valutazioni centrate sull'utente e feedback;
- d) comunicare le soluzioni di progettazione a coloro che sono responsabili della loro implementazione.

6.3.4 HCD: Progettare per Soddisfare i Requisiti

Progettare l'interazione utente-sistema implica decidere come gli utenti svolgeranno i compiti con il sistema piuttosto che descrivere come appare il sistema. La progettazione dell'interazione dovrebbe includere:

- prendere decisioni di alto livello (ad es. concept di progettazione iniziale, risultati essenziali);
- identificare compiti e sotto-compiti;
- allocare compiti e sotto-compiti all'utente e ad altre parti del sistema;
- Es.: il sistema tiene traccia dell'ID di login e ricorda agli utenti, ma gli utenti ricordano la password;
- identificare gli oggetti di interazione necessari per il completamento dei compiti;
- progettare la sequenza e la tempistica (dinamica) dell'interazione;
- progettare l'interfaccia utente per consentire un accesso efficiente agli oggetti di interazione.

6.3.5 HCI: Valutare la Progettazione

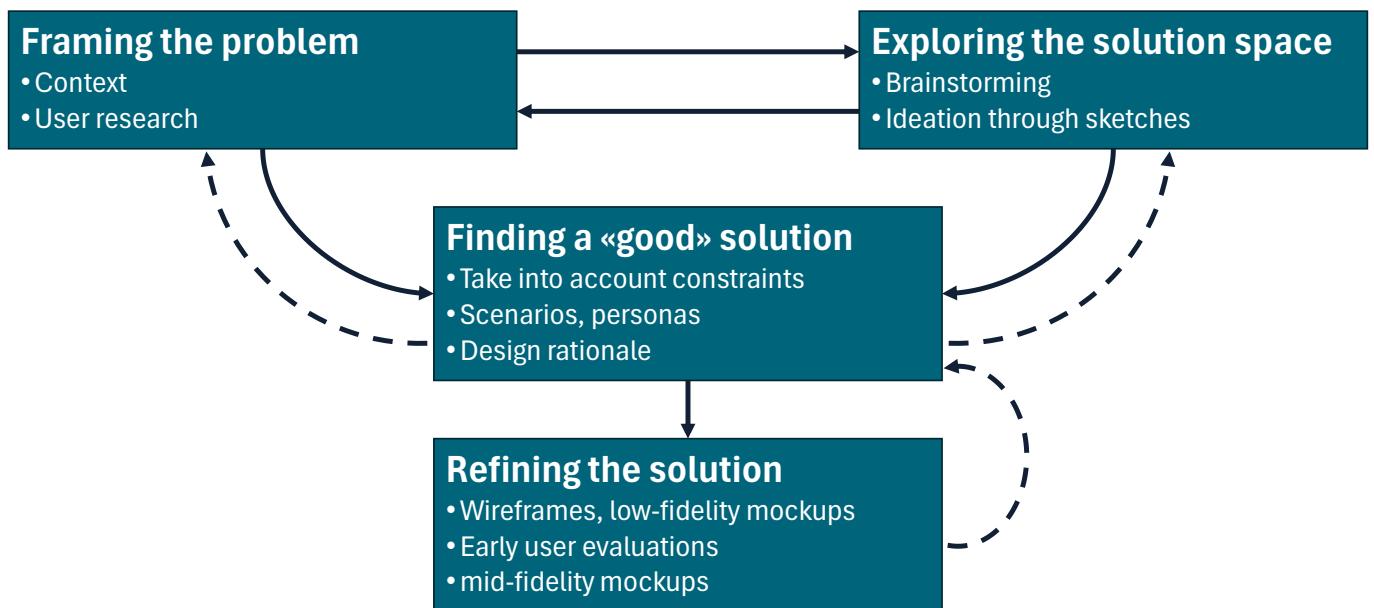
La valutazione centrata sull'utente (cioè, la valutazione basata sulla prospettiva dell'utente) è un'attività richiesta nell'HCD:

- Test basati sull'utente (ad es.: coinvolgendo utenti reali o rappresentativi)
- Approccio basato sull'ispezione (verifica di linee guida o requisiti)

6.3.6 Progettazione Centrata sull'Uomo e SDLC

L'HCD non richiede alcun processo di progettazione particolare È complementare alle metodologie di sviluppo esistenti Ciascuna attività può essere integrata (in misura minore o maggiore) in qualsiasi fase dello sviluppo di un sistema Ad esempio, l'HCD potrebbe essere applicata nella fase di Ingegneria dei Requisiti in un modello di processo a cascata

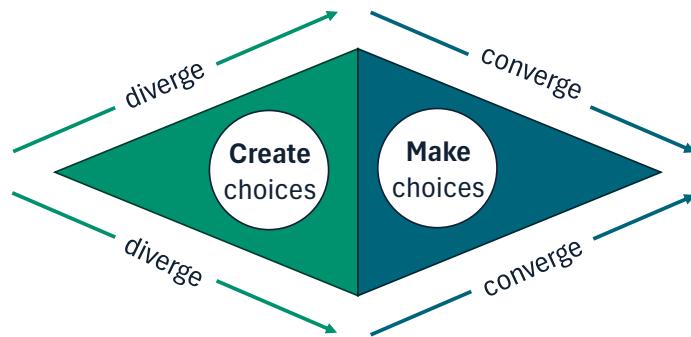
More on the UI Design Process



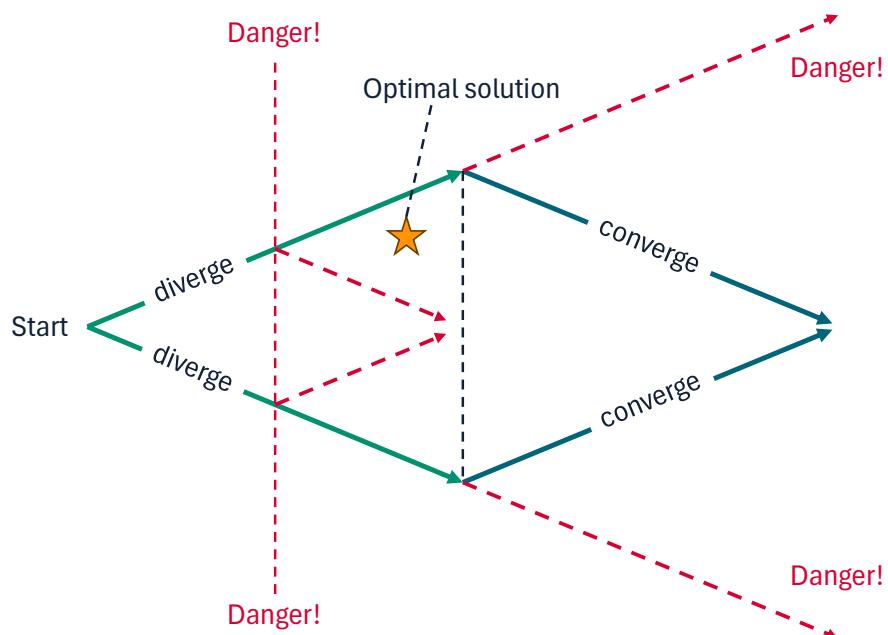
Design as Choices

The design process is an alternance of two phases:

- **Divergence** phase (elaboration)
 - Explore different designs
 - Create choices
- **Convergence** phase (reduction)
 - Choose among different designs



The Design Diamond



- Stopping divergence too early might lead to missing good ideas
- Keep in mind that we are operating within budget and organizational constraints: we need to actually converge!

Importance of Critique and Feedback

Ideas can be **good or bad**

- Both kind of ideas are **useful** in design
- By making clear what is a bad design, we can avoid implementing it
- Bad ideas can also help us justify our good ideas
- Feedback can further improve a good idea!

Tips for Giving Effective Critiques

Hamburger Method

- **Bun**
 - Fluffy and nice
- **Meat**
 - How to improve
- **Bun**
 - Fluffy and nice

I like, I wish, What if?

- **I like...**
 - Lead with something nice
- **I wish...**
 - Something you would improve
- **What if...?**
 - An idea to spark further discussion

Socratic Method

- **Identify one aspect of design and ask «why?»**
 - Forces presenter to give, or develop, motivations for design decisions
 - Not inherently negative, hard to get defensive

Tips for Giving Effective Critiques

- Limit the use of personal pronouns (e.g.: «you»)
 - Critique is about the artifact, not the designer
- A designer deserves honest feedback
 - Be honest, give both positive and negative feedback
 - Be clear and motivate your critiques
- Help with actionable suggestions

Progettazione di un sistema

- La progettazione di un sistema (**Software Design**) è l'insieme dei task svolti dall'ingegnere del software per trasformare il modello di analisi nel modello di design del sistema
- Obiettivo ultimo di System e Object Design è definire un nuovo documento, scritto in linguaggio tecnico/formale, da dare ai programmatore affinchè questi siano in grado di implementare il software descritto nel Documento dei Requisiti Software

Scopo del System Design

1. Definire gli obiettivi di design del progetto
2. Definire l'**Architettura del sistema**, decomponendolo in sottosistemi più piccoli
 - Possono essere realizzati in parallelo da team individuali
3. Selezionare le strategie per costruire il sistema, quali:
 - Strategie hardware/software
 - Strategie relative alla gestione dei dati persistenti
 - Il flusso di controllo globale
 - Le politiche di controllo degli accessi
 - ...

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

Il System Design



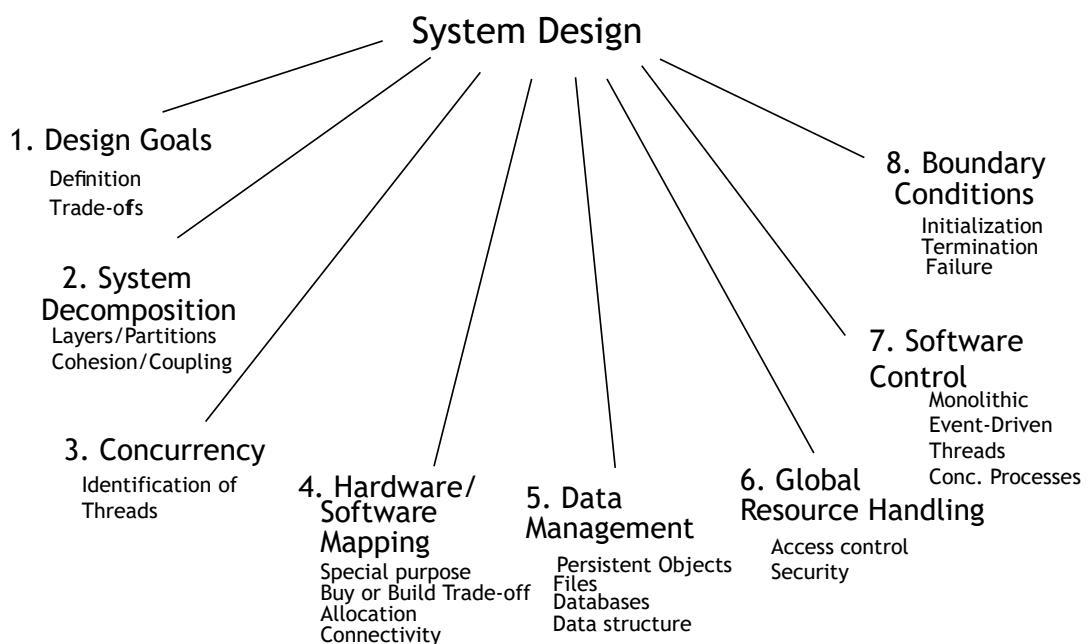
System Design

- Cambia radicalmente punto di vista:
 - Finora abbiamo lavorato per interagire col cliente
 - Linguaggio per profani, poco formale
 - Nel system design i nostri interlocutori sono le squadre di programmatore che dovranno implementare il sistema
 - Chi leggerà i nostri documenti sarà un esperto di informatica
 - Linguaggio per esperti tecnici

Attività di system design

- Il system design è costituito da tre 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.

Passi del System Design



Identificare gli obiettivi qualitativi del sistema

- E' 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
- E' importante formalizzarli esplicitamente poiché ogni importante decisione di design deve essere fatta seguendo lo stesso insieme di criteri

Criteri di design

- Possiamo selezionare gli obiettivi di design da una lunga lista di qualità desiderabili.
- I criteri sono organizzati in cinque 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

Design Trade-offs

- Quando definiamo gli obiettivi di design, spesso solo un piccolo sottoinsieme di questi criteri può essere tenuto in considerazione.
 - Es: non è realistico sviluppare software che sia simultaneamente sicuro e costi poco.
- Gli sviluppatori devono dare delle priorità agli obiettivi di design, tenendo anche conto di aspetti manageriali, quali il rispetto dello schedule e del budget.

Design Trade-offs (cont.)

- Esempi:
 - Spazio vs. velocità. Se il software non rispetta i requisiti di tempo di risposta e di throughput, è necessario utilizzare più memoria per velocizzare il sistema (es. Caching, più ridondanza). Se il software non rispetta i requisiti di memoria, può essere compresso a discapito della velocità.
 - Tempo di rilascio vs. funzionalità. Se i tempi di rilascio sono stringenti, possono essere rilasciate meno funzionalità di quelle richieste, ma nei tempi giusti.
 - Tempo di rilascio vs. qualità. Se i tempi di rilascio sono stretti, il project manager può rilasciare il software nei tempi prefissati con dei bug e, in tempi successivi, correggerli, o rilasciare il software in ritardo, ma con meno bug.
 - Tempo di rilascio vs. staffing. Può essere necessario aggiungere delle risorse al progetto per accrescere la produttività.

Esempio iniziale

Abbozzare gli otto passi del system design per un'applicazione reale



Concetti di base di buon design

Qualità del codice



Concetti di System 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 però dei Criteri di Design generici, che valgono per qualunque software O-O
 - Questi criteri, se ben applicati, migliorano notevolmente la qualità interna del codice, senza introdurre particolari svantaggi

Pensare in avanti

- Quando progettiamo una classe dovremmo sforzarcì di pensare a quali cambiamenti potranno essere richiesti in futuro:
“Progettare per ANTICIPARE IL CAMBIAMENTO”
- Le nostre scelte iniziali devono facilitare l’evoluzione futura
 - E’ una delle richieste fondamentali, nonché fonte di valutazione per il progetto!

La Decomposizione

- Per ridurre la complessità della soluzione, decomponiamo il sistema in parti più piccole, chiamate sottosistemi.
 - Un sottosistema tipicamente corrisponde alla parte di lavoro che può essere svolta autonomamente da un singolo sviluppatore o da un team di sviluppatori.
- Decomponendo il sistema in sottosistemi relativamente indipendenti, i team di progetto possono lavorare sui sottosistemi individuali con un minimo overhead di comunicazione.
- Nel caso di sottosistemi complessi, applichiamo ulteriormente questo principio e li decomponiamo in sottosistemi più semplici.

La Decomposizione

- P= problema,
- C=complessità di P,
- E=sforzo per la risoluzione di P
- Dati 2 problemi P1 e P2, se $C(P1) > C(P2)$, allora $E(P1) > E(P2)$.
- Ma è stato dimostrato che $C(P1+P2) > C(P1) + C(P2)$, e quindi si ha che $E(P1+P2) > E(P1) + E(P2)$

Modellazione di sottosistemi

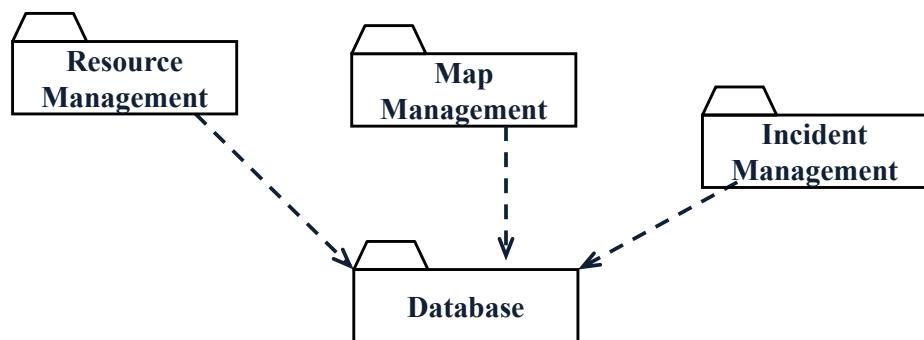
- Subsystem (UML: Package)
 - Collezioni di classi, associazioni, operazioni e vincoli che sono correlati
- JAVA fornisce i package che sono costrutti per modellare i sottosistemi
 - C++ / C# hanno il costrutto di namespace per indicare lo stesso concetto

Servizi e interfacce di sottosistemi

- Un sottosistema è caratterizzato dai servizi che fornisce agli altri sottosistemi
- L'insieme di operazioni di un sottosistema che sono disponibili agli altri sottosistemi forma l'interfaccia del sottosistema
 - Include il nome delle operazioni, i loro parametri, il loro tipo ed i loro valori di ritorno.
- Il system design si focalizza sulla definizione dei servizi forniti da ogni sottosistema in termini di:
 - Operazioni
 - Loro parametri
 - Loro comportamento ad alto livello

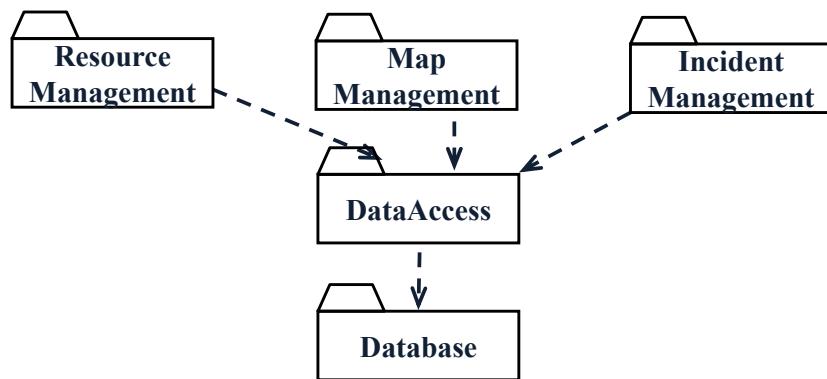
Esempio di scelta di design: accesso diretto al database

- Non riesco a specificare chiaramente le responsabilità di ogni modulo (Gestisce, Salva?)
- Tutti i sottosistemi accedono al database direttamente, rendendoli vulnerabili ai cambiamenti dell’interfaccia del sottosistema Database
 - Se cambia il DBMS, devo modificare il codice in tutte le classi che accedono al DB!



Esempio: accesso al database attraverso un sottosistema di Storage

- Chiariamo le responsabilità:
- Introduco il sottosistema DataAccess che gestisce l'I/O da db
 - Se cambia il DBMS, devo modificare solo il sottosistema Storage!



Osservazioni

- Nell'esempio proposto abbiamo ridotto le relazioni fra i quattro sottosistemi, ma ...
- Abbiamo aumentato la complessità!
- Con l'obiettivo di ridurre le relazioni si rischia di aggiungere dei livelli di astrazione che consumano tempo di sviluppo e tempo di elaborazione.

Indipendenza Funzionale

- La suddivisione in sottosistemi può essere guidata da due fattori qualitativi fondamentali:
 - Coesione (cohesion)
 - Accoppiamento (coupling)
- L'obiettivo è l'indipendenza funzionale dei sottosistemi, che si massimizza con Alta Coesione e Basso Accoppiamento

Coesione

- La Coesione è una misura di quanto siano fortemente relate e mirate le responsabilità (o servizi offerti) di un modulo.
- Se ciascuna unità è responsabile di un singolo compito, diciamo che tale unità ha una alta coesione.
- Un'alta coesione è una proprietà desiderabile del codice, poiché
- permette:
 - Di comprendere meglio i ruoli di una classe
 - Di riusare una classe
 - Di manutenere una classe
 - Di 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

Coesione - Il Single Responsibility Principle

- Una classe dovrebbe avere UNA sola responsabilità
- Una classe dovrebbe avere UN solo motivo per cambiare
- Una responsabilità è "una ragione per cambiare".
- Le responsabilità di una classe sono assi di cambiamento.
- Se ha due responsabilità, queste sono accoppiate nel progetto, e quindi devono cambiare insieme.
- Se una classe ha più di una responsabilità, può diventare fragile quando uno qualsiasi dei requisiti cambia

Coesione – Example

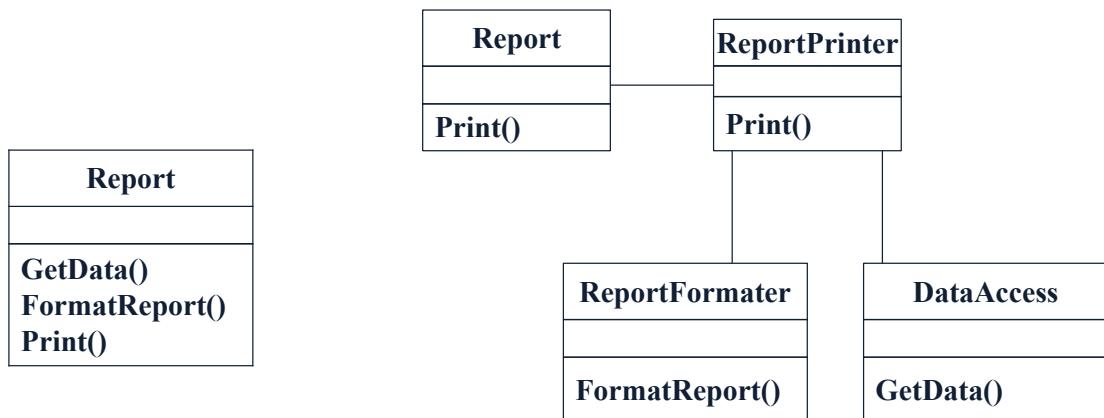
Report
GetData()
FormatReport()
Print()

```
namespace Dimecasts.SOLID
{
    class Report
    {
        private IList<ReportDataElement> GetData()
        {
            Console.WriteLine("\nGetting Data...");
            return new List<ReportDataElement>() { new ReportDataElement("Item 1"), new ReportDataElement("Item 2") };
        }

        private void FormatReport()
        {
            Console.WriteLine("\nFormatting Report...");
        }

        public void Print()
        {
            GetData();
            FormatReport();
            Console.WriteLine("\nPrinting Report...");
        }
    }
}
```

Coesione – Example V2



Coesione – Example V2

```
public class ReportFormatter
{
    public void FormatReport()
    {
        Console.WriteLine("\nFormatting Report...");
    }
}

public class DataAccess
{
    public IList<ReportDataElement> GetData()
    {
        Console.WriteLine("\nGetting Data...");
        return new List<ReportDataElement>() { new ReportDataElement() };
    }
}
```

Coesione – Example V2

```
public class ReportPrinter
{
    public void Print()
    {
        DataAccess dataAccess = new DataAccess();
        ReportFormatter reportFormatter = new ReportFormatter();
        dataAccess.GetData();
        reportFormatter.FormatReport();
        Console.WriteLine("\nPrinting Report...");
    }
}

class Report
{
    public void Print()
    {
        ReportPrinter reportPrinter = new ReportPrinter();
        reportPrinter.Print();
    }
}
```

Benefits of Cohesion

- Riutilizzo - Se tutte le tue classi seguono l'SRP, è più probabile che tu ne riutilizzi alcune
- Chiarezza - Il tuo codice è più pulito poiché le tue lezioni non fanno cose impreviste
- cose
- Denominazione - Poiché tutte le tue classi hanno un'unica responsabilità, scegliere un buon nome è facile
- Leggibilità - Grazie alla chiarezza, ai nomi migliori e ai file più brevi, la leggibilità è notevolmente migliorata.

Coesione

- Indicatori di un'alta coesione (è il nostro obiettivo)
 - Una classe ha delle responsabilità moderate, limitate ad una singola area funzionale
 - Collabora con altre classi per completare dei tasks
 - 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

Accoppiamento

- L'accoppiamento è una misura su quanto fortemente una classe è connessa con /ha conoscenza di/ si basa su altre classi.
- Due o più classi si dicono accoppiate quando è impossibile riusare una senza dover riusare anche la/le altra/e
- Se due classi dipendono strettamente e per molti dettagli l'una dall'altra, diciamo che sono fortemente accoppiate.
 - In caso contrario diciamo che sono debolmente accoppiate
- Per un codice di qualità dobbiamo puntare ad un basso accoppiamento

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 software

Basso Accoppiamento

- Un certo livello di accoppiamento è comunque insito nel concetto di scambio di messaggio del paradigma O-O
 - E' quindi necessario per il funzionamento del sistema
 - Deve essere limitato ove possibile
- Linea Guida: definire le responsabilità delle classi in modo da ottenere un basso accoppiamento
 - Legge di Demetra (vedi slides successive)

Tipi di Accoppiamento in O-O

- Date 2 classi X e Y:
 - X ha un attributo di tipo Y
 - X ha un metodo che possiede un elemento (es: parametro, variabile locale, etc...) di tipo Y
 - X è una sottoclasse (eventualmente indiretta) di Y
 - X implementa un'interfaccia di tipo Y
- Lo scenario 3 è quello che porta al massimo accoppiamento in assoluto.

Esempio

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

class Car
{
void move()
{
// logic...
}
}
```

- Cosa succede se vogliamo riusare la nostra classe Traveler con un altro mezzo di trasporto che non sia Car?
- Cosa succede se vogliamo cambiare i metodi di Car?
- Traveler ha un attributo di tipo Car, e quindi un forte accoppiamento.

Esempio (cont.)

```
•class Traveler
•{
•Vehicle v;
•public void setV(Vehicle v)
•{
•    this.v = v;
•}

•void startJourney()
•{
•    v.move();
•}
•}

•Interface Vehicle
•{
•void move();
•}
```

```
class Car implements Vehicle
{
public void move()
{
// logic
}

class Bike implements Vehicle
{
public void move()
{
// logic
}
```

Spiegazione

- Nell'esempio, 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)
 - E' necessaria un'entità esterna che faccia l'inject della reale implementazione dell'interfaccia
 - E' aumentata la complessità del codice

Thinking about Interfaces

- Un membro del team di sviluppo Java che conosceva le interfacce ha deciso di renderle una parte importante di Java SDK. Ci sono centinaia di esempi nell'SDK J2SE del modo corretto di utilizzare le interfacce.
- Di seguito sono riportati due vantaggi principali derivanti dall'utilizzo delle interfacce:
- Un'interfaccia fornisce un mezzo per stabilire uno standard. Definisce un contratto che promuove il riuso. Se un oggetto implementa un'interfaccia, allora quell'oggetto promette di essere conforme a uno standard. Un oggetto che utilizza un altro oggetto è chiamato consumer. Un'interfaccia è un contratto tra un oggetto e il relativo consumatore.
- Un'interfaccia fornisce anche un livello di astrazione che rende i programmi più facili da capire. Le interfacce consentono agli sviluppatori di iniziare a parlare del modo generale in cui si comporta il codice senza dover entrare in molte specifiche dettagliate.

L'esempio del Paperboy

```
public class Customer {  
    private String firstName;  
    private String lastName;  
    private Wallet myWallet;  
    public String getFirstName(){  
        return firstName;  
    }  
    public String getLastName(){  
        return lastName;  
    }  
    public Wallet getWallet(){  
        return myWallet;  
    }  
}
```

- public class Wallet { private float value;
- public float getTotalMoney() { return value; }
- public void setTotalMoney(float newValue) { value = newValue; }
- public void addMoney(float deposit) { value += deposit; }
- public void subtractMoney(float debit) { value -= debit; }
- }

L'esempio del Paperboy

```
code from some method inside the Paperboy class...
payment = 2.00; // "I want my two dollars!"

Wallet theWallet = myCustomer.getWallet(); if
(theWallet.getTotalMoney() > payment) {
    theWallet.subtractMoney(payment);
} else {
    // come back later and get my money
}
```

Why Is This Bad?

- Traduciamo ciò che il codice sta effettivamente facendo in linguaggio reale:
- A quanto pare, quando il ragazzo dei giornali si ferma e chiede il pagamento, il cliente sta per lasciare che il ragazzo dei giornali tiri fuori il portafoglio dalla tasca posteriore e tiri fuori due dollari.
- Non so voi, ma raramente lascio che qualcuno gestisca il mio portafoglio. Ci sono una serie di problemi del "mondo reale" con questo, per non parlare del fatto che ci fidiamo del ragazzo dei giornali per essere onesto e prendere semplicemente ciò che gli è dovuto.
- Se il nostro futuro oggetto Wallet contiene carte di credito, il paperboy ha accesso anche a quelle... Ma il problema di base è che "il ragazzo dei giornali è esposto a più informazioni di quelle di cui ha bisogno".
- Questo è un concetto importante... La classe del "Paperboy" ora "sa" che il cliente ha un portafoglio e può manipolarlo. Quando compiliamo la classe Paperboy, avrà bisogno della classe Customer e della classe Wallet. Queste tre classi sono ora "strettamente accoppiate". Se cambiamo la classe Wallet, potremmo dover apportare modifiche a entrambe le altre classi.

Why Is This Bad?

C'è un altro problema classico che questo può creare... Cosa succede se il portafoglio del Cliente è stato rubato? Forse la scorsa notte un ladro ha preso la tasca del nostro esempio, e qualcun altro nel nostro team di sviluppo software ha deciso che un buon modo per modellare questo sarebbe stato impostare il portafoglio su null, in questo modo:

```
victim.setWallet(null);
```

- Sembra un'ipotesi ragionevole... Né la documentazione né il codice impongono alcun valore obbligatorio per il portafoglio, e c'è una certa "eleganza" nell'usare null per questa condizione. Ma cosa succede al nostro Paperboy? Tornando al codice...
- Il codice presuppone che ci sarà un portafoglio! Il nostro paperboy otterrà un'eccezione di runtime per la chiamata di un metodo su un puntatore nullo.
- Potremmo risolvere questo problema controllando la presenza di 'null' sul portafoglio prima di chiamare qualsiasi metodo su di esso, ma questo inizia a ingombrare il codice del paperboy ... La nostra descrizione in lingua reale sta diventando ancora peggiore...
- "Se il mio cliente ha un portafoglio, allora guarda quanto ha... se può pagarmi, prendilo"...

Improving The Original Code

```
public class Customer {  
    private String firstName;  
    private String lastName;  
    private Wallet myWallet;  
    public String getFirstName() {  
        return firstName;  
    }  
    public String getLastName() {  
        return lastName;  
    }
```

```
    public float getPayment(float bill) {  
        if (myWallet != null) {  
            if (myWallet.getTotalMoney() > bill)  
                {  
                    theWallet.subtractMoney(payment);  
                    return payment;  
                }  
            }  
        }  
    }
```

The `Customer` no longer has a '`getWallet()`' method,
but it does have a `getPayment()` method.

Improving the original code

```
// code from some method inside the Paperboy class...
    payment = 2.00; // "I want my two dollars!"
    paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
// say thank you and give customer a receipt
} else {
// come back later and get my money
}
```

Why is this better?

- Questa è una domanda lecita... Qualcuno a cui è piaciuto molto il primo pezzo di codice potrebbe obiettare che abbiamo solo reso il Cliente un oggetto più complesso. Questa è un'osservazione giusta, ma si parlerà in seguito degli svantaggi.
- Il primo motivo per cui questo è migliore è perché modella meglio lo scenario del mondo reale... Il codice Paperboy ora "chiede" al cliente un pagamento. Il paperboy non ha accesso diretto al portafoglio.
- Il secondo motivo per cui questo è migliore è perché la classe Wallet ora può cambiare, e il ragazzo dei giornali è completamente isolato da quel cambiamento.
- Se l'interfaccia del Wallet dovesse cambiare, il Cliente dovrebbe essere aggiornato, ma
- Questo è tutto...
- Finché l'interfaccia con il Cliente rimane la stessa, nessuno dei clienti del Cliente si preoccuperà di aver ricevuto un nuovo Portafoglio.
- Il codice sarà più gestibile, perché le modifiche non si "propagheranno" attraverso un progetto di grandi dimensioni.

Why is this better?

- La terza risposta, e probabilmente la più "orientata agli oggetti" è che ora siamo liberi di modificare l'implementazione di 'getPayment()'. Nel primo esempio, abbiamo ipotizzato che il Cliente avrebbe avuto un portafoglio. Ciò ha portato all'eccezione del puntatore nullo di cui abbiamo parlato. Nel mondo reale, però, quando il ragazzo di carta si presenta alla porta, il nostro cliente può effettivamente prendere i due dollari da un barattolo di spiccioli, cercare tra i cuscini del suo divano o prenderli in prestito dal suo coinquilino.
- Tutto questo è "Business Logic", e non interessa al ragazzo di carta... Tutto questo potrebbe essere implementato all'interno del metodo getPayment() e potrebbe cambiare in futuro, senza alcuna modifica al codice del paper boy.

Quali sono gli svantaggi?

- Il cliente sta DAVVERO diventando più complesso? Nel nostro primo esempio, il codice paperboy che utilizzava il Cliente doveva anche conoscere e manipolare il Portafoglio. Non è più così...
- Se il cliente è diventato PIÙ complesso, allora perché i clienti sono ora MENO complessi? Esiste ancora la stessa complessità, è solo meglio contenuta nell'ambito di ogni oggetto ed esposta in modo sano.
- Contenendo questa complessità, otteniamo tutti i vantaggi discussi sopra. Quindi sì, possiamo ammettere che il cliente è diventato più complesso. La verità è che si tratta di una complessità che esisteva prima nel codice, e potrebbe essere esistita in diversi punti. Ora si verifica una sola volta e può essere mantenuto nello stesso punto in cui si verificano le modifiche che possono interromperlo.

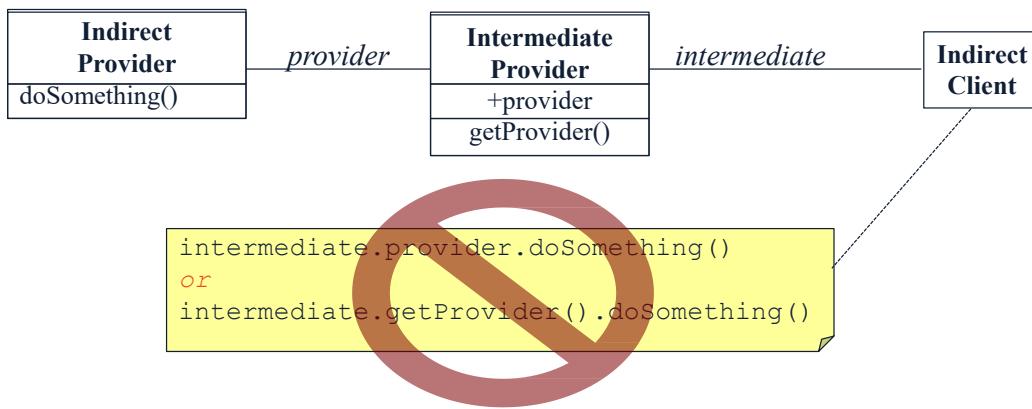
La Legge di Demetra

- E' un'altra linea guida chiave per avere Basso Accoppiamento
- Utilizzata nel progetto Demetra, uno dei primi grossi sistemi O-O, sviluppato all'univ. di Boston.
- Concetto di base: Spingere al massimo l'information hiding
 - No situazioni tipo: Foo.getA().getB().getC().....
 - Più è lungo il percorso di chiamate attraversato dal programma, più esso è fragile a cambiamenti

La Legge di Demetra

- 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

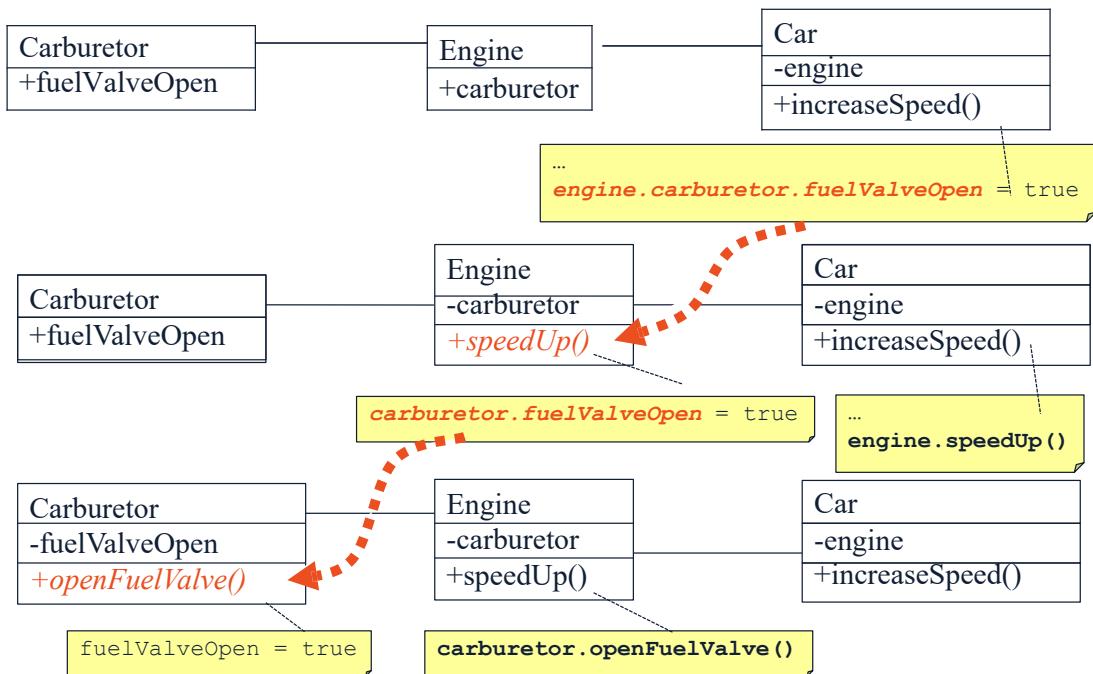
The Law of Demeter, violated



Law of Demeter: A method "M" of an object "O" should invoke only the methods of the following kinds of objects.

- | | |
|--------------------------|---|
| <i>1. itself</i> | <i>3. any object it creates /instantiates</i> |
| <i>2. its parameters</i> | <i>4. its direct component objects</i> |

Eliminate Navigation Code



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 precisi → Alta Coesione
- La classe che possiede i dati dovrebbe essere responsabile di processarli → Basso Accoppiamento
 - Ciascuna classe dovrebbe essere responsabile di gestire i propri dati → Incapsulamento
- E' facilitata dall'uso di CRC Cards

Le CRC Cards

- Introdotte Kent Beck e Ward Cunningham per insegnare progettazione O-O
- Una card CRC è una 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 alle responsabilità sopra indicate

Una CRC Card

- Associata ad ogni classe del class diagram di design

Conseguenze

- Le CRC cards spingono ad ottenere un design con chiare responsabilità e controllo sul numero di classi associate
 - Se non riesco a specificare chiaramente le responsabilità di una classe, c'è un errore di design
 - Se una classe ha bisogno di troppe classi collaboratrici, c'è un potenziale errore di design

Localizzare le modifiche

- La progettazione RDD, con CRC Cards, porta ad una localizzazione delle modifiche
- Quando è necessario operare una modifica, il minor numero possibile di classi dovrebbero essere coinvolte
- La qualità del proprio codice si osserva proprio quando sorge l'esigenza di fare modifiche

Capitolo 10

Lezione 10: Il design delle cose di tutti i giorni e la natura delle interazioni giornaliere

10.1 Le interazioni

Alcune interazioni sono più fluide di altre. Spesso, le difficoltà nell'uso di un sistema non derivano da complessità profonde e sottili. Falliamo nell'usare molti oggetti quotidiani! Quindi, cosa rende difficile un'interazione? Per rispondere, dobbiamo capire cosa accade quando qualcuno fa (o prova a fare) qualcosa. Don Norman ha formulato una teoria per spiegare le fasi dell'azione.

10.1.1 La Struttura dell'Azione: Il Ciclo di Azione

Per realizzare qualcosa, è necessario partire da un'idea di ciò che si vuole ottenere (obiettivo). Poi, bisogna agire sul mondo esterno, muoversi e interagire con qualcuno o qualcosa (esecuzione). Infine, verifichiamo se il nostro obiettivo è stato effettivamente raggiunto (valutazione).

10.1.2 Fasi di Esecuzione

Per portare alle azioni, gli obiettivi devono essere trasformati in una dichiarazione specifica di ciò che deve essere fatto (intenzione). Le intenzioni devono essere tradotte in una sequenza di azioni da eseguire per soddisfare l'intenzione. La sequenza di azioni deve essere fisicamente eseguita, cioè realizzata nel mondo. La valutazione inizia con la nostra percezione del mondo. La percezione deve essere interpretata secondo le nostre aspettative. Quindi, l'interpretazione viene valutata rispetto alle nostre intenzioni e al nostro obiettivo.

10.1.3 Sette Fasi dell'Azione: Esempio

Stiamo leggendo un libro e la stanza sta diventando troppo buia per leggere.

- Stabilire l'obiettivo: Aumentare la luce nella stanza
- Formare l'intenzione: Accendere la lampada
- Specificare la sequenza di azioni: Camminare verso la lampada, raggiungere l'interruttore, azionare l'interruttore
- Eseguire la sequenza di azioni: [camminare, raggiungere, azionare]
- Percepire lo stato del sistema: [sentire il suono “click”, vedere la luce dalla lampada]
- Interpretare lo stato del sistema: L'interruttore ha cambiato posizione. La lampada emette luce. La lampada sembra funzionare
- Valutare lo stato del sistema rispetto agli obiettivi e alle intenzioni: Il livello di luce è aumentato [obiettivo soddisfatto]

10.2 Le Sette Fasi dell’Azione di Don Norman

Le sette fasi dell’azione sono un modello approssimativo. Il processo può iniziare da qualsiasi fase.

- Le persone non sempre si comportano come organismi logici
- Gli obiettivi possono essere confusi, mal definiti e vaghi
- Possiamo rispondere agli eventi del mondo (comportamento guidato dagli eventi)
- Alcune azioni sono opportunistiche piuttosto che pianificate. Le eseguiamo se si presenta l’opportunità

10.2.1 Sette Fasi dell’Azione: Alternative

Notare che un determinato obiettivo può essere soddisfatto utilizzando diverse intenzioni e diverse sequenze di azioni! Se qualcuno entrasse nella stanza e passasse vicino alla lampada, potremmo modificare la nostra intenzione da premere l’interruttore a chiedere all’altra persona di farlo per noi.

Le difficoltà spesso risiedono interamente nel derivare le relazioni tra intenzioni e interpretazioni mentali e azioni e stati fisici. Norman identifica due ”golfi” (divari) che separano gli stati mentali da quelli fisici

- Golfo dell’Esecuzione e Golfo della Valutazione
- Ogni golfo riflette un aspetto della distanza tra la rappresentazione mentale dell’utente e i componenti fisici e gli stati dell’ambiente

10.3 Principi di Design per aiutare a colmare i golfi

10.3.1 Affordances

Le affordances forniscono indizi forti sul funzionamento delle cose. I pulsanti sono fatti per essere premuti. Le manopole sono fatte per essere girate. Le fessure sono fatte per inserirvi oggetti. Le palle sono fatte per essere lanciate o fatte rimbalzare. Quando si sfruttano le affordances, gli utenti sanno cosa fare semplicemente guardando: non serve alcuna immagine, etichetta o istruzione. Le cose complesse potrebbero richiedere spiegazioni, ma le cose semplici non dovrebbero! Le false affordances sembrano offrire una particolare capacità, ma in realtà ne offrono una diversa (o nessuna)! Nelle UI: ad esempio: se qualcosa sembra un pulsante ma non è cliccabile. Le affordances nascoste si verificano quando gli indizi che indicano la funzione di un elemento non sono evidenti e potrebbero non essere visualizzati fino a quando l’azione non viene intrapresa.

10.3.2 Vincoli

Il modo più semplice per assicurarsi che qualcosa sia facile da usare, con pochi errori, è rendere impossibile fare diversamente limitando le scelte dell’utente.

10.3.3 Feedback

Ogni azione con effetti collaterali rilevanti dovrebbe essere esplicitamente confermata dal sistema. Il feedback dovrebbe essere immediato e informativo. Preferibilmente non distrattivo e discreto.

10.3.4 Coerenza

Le interfacce dovrebbero essere coerenti in modi significativi

- All’interno dell’applicazione stessa (coerenza interna)
- Con altre applicazioni esterne (coerenza esterna)

La coerenza aiuta gli utenti a colmare i golfi della valutazione e dell’esecuzione. Ad esempio: se tutte le azioni sono confermate tramite un messaggio toast nell’angolo in alto a destra (come nell’app Lista delle cose da fare che abbiamo visto alcune diapositive fa), è più facile per gli utenti capire in quale stato si trova il sistema. Se i messaggi di conferma fossero diversi per ogni azione...

10.3.5 Metafore

Possono essere utili nelle UI per suggerire un modello mentale esistente e sfruttare conoscenze specifiche che gli utenti già possiedono in domini diversi

- Carrozze senza cavalli, telefoni senza fili...
- Metafora del desktop: Non è un tentativo di simulare una scrivania reale, ma mira a sfruttare la conoscenza che gli utenti hanno di file, documenti, cartelle, cestini, ...

10.3.6 Mappature

Una mappatura è una corrispondenza tra un'interfaccia e l'azione corrispondente nel mondo reale. Mappature efficaci (naturali) possono minimizzare i passaggi cognitivi per trasformare un'azione in effetto, o accelerare il processo di trasformazione della percezione in comprensione. Le mappature naturali possono anche ridurre il carico sulla memoria.

10.4 Linee Guida di Norman per colmare i golfi

- Visibilità. Guardando, l'utente può comprendere lo stato del dispositivo e le alternative per l'azione.
- Un buon modello concettuale. Il designer fornisce un buon modello concettuale per l'utente, con coerenza nella presentazione delle operazioni e dei risultati e un'immagine del sistema coerente e consistente.
- Buone mappature. È possibile determinare le relazioni tra azioni e risultati, tra i controlli e i loro effetti, e tra lo stato del sistema e ciò che è visibile.
- Feedback. L'utente riceve un feedback completo e continuo sui risultati delle azioni.

10.4.1 Caratteristiche di un Buon Design

Ha affordances (rende le operazioni visibili) Offre mappature ovvie (rende evidente la relazione tra l'azione effettiva del dispositivo e l'azione dell'utente) Fornisce feedback sull'azione dell'utente Fornisce un buon modello mentale del comportamento sottostante del sistema

- Un modello mentale è la rappresentazione interna che un utente ha di come un sistema o un'interfaccia funziona.
- Un buon modello mentale consente agli utenti di prevedere come si comporterà l'interfaccia e li aiuta a interagire efficacemente con essa.

Fornisce vincoli (per prevenire errori)

Capitolo 11

Lezione 13: Modelli e teorie in HCI

11.0.1 Cos'è un modello?

Un modello è una semplificazione (astrazione) della realtà

- Una mappatura perfetta della realtà non è un modello (e non è utile!)
- Precisione vs generalità

«Tutti i modelli sono sbagliati, ma alcuni sono utili» I modelli ci permettono di:

- Rappresentare e ragionare su (aspetti di) fenomeni di interesse
- Anticipare (prevedere) risultati

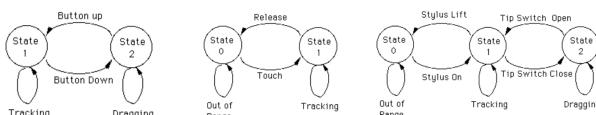
11.1 Modelli e teorie HCI

Nell'HCI, i modelli e le teorie mirano a spiegare come gli esseri umani interagiscono con i computer. I modelli e le teorie HCI possono essere classificati come:

- **Descrittivi:** mirano a sviluppare una terminologia coerente e tassonomie utili
- **Esplicativi:** descrivono sequenze di eventi, possibilmente con relazioni causali. Es.: le sette fasi dell'azione di Norman
- **Predittivi:** mirano a consentire il confronto di alternative di design basate su previsioni numeriche di velocità o errori
- **Prescrittivi:** offrono linee guida ai designer per prendere decisioni

11.1.1 Il modello a tre stati dell'input grafico

Proposto da Will Buxton nel 1990 [1] Descrive l'input grafico con dispositivi di puntamento Diverse tecnologie (mouse, trackpad, tablet con stilo, ecc...)



[1] Buxton, W. (1990). A Three-State Model of Graphical Input. In INTERACT '90.

Figura 11.1: Descrizione di Buxton

11.1.2 Il modello di Guiard dell'abilità bimanuale

Molte interazioni sono asimmetriche rispetto alla mano sinistra/destra Il modello di Guiard descrive ruoli e azioni delle mani preferita/non preferita Non preferita:

- Guida la mano preferita
- Definisce il quadro di riferimento spaziale per la mano preferita

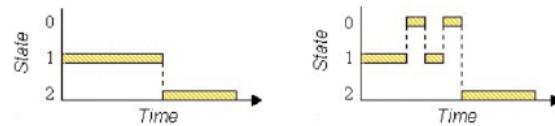


Figura 11.2: Operazioni di trascinamento con un mouse (a sinistra) e con un touchpad lift-and-tap (a destra) (tratto da [1])

- Esegue movimenti grossolani

Preferita:

- Segue la mano non preferita
- Lavora all'interno del quadro di riferimento stabilito dalla mano non preferita
- Esegue movimenti fini

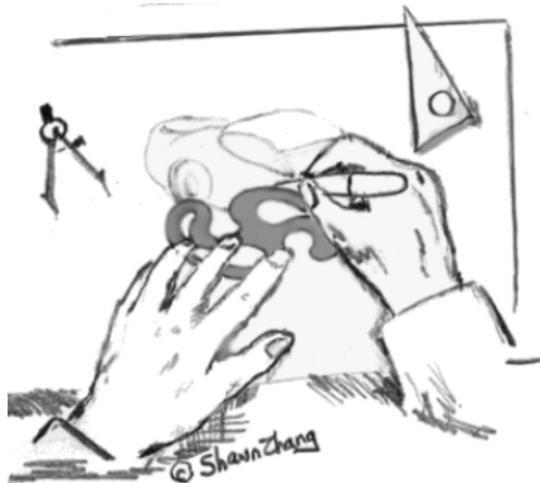


Figura 11.3: Interazione a due mani. (Schizzo di Shawn Zhang)

L'artista acquisisce il modello con la mano sinistra (la mano non preferita guida). Il modello viene manipolato sullo spazio di lavoro (movimento grossolano, definisce il quadro di riferimento). Lo stilo viene impugnato nella mano destra (la mano preferita segue) e portato in prossimità del modello (lavora all'interno del quadro di riferimento stabilito dalla mano non preferita). Ha luogo lo schizzo (la mano preferita esegue movimenti precisi).

11.2 Il Modello del Processore Umano (MHP)

È un modello predittivo a priori, può fornire approssimazioni delle azioni dell'utente prima che utenti reali siano coinvolti nel processo di test (e prima che l'interfaccia utente sia anche solo implementata!) Un essere umano è modellato da un insieme di memorie e processori che funzionano secondo un insieme di principi Modello discreto e sequenziale Processori percettivi, cognitivi e motori Diversi tipi di memoria Parametri del modello:

- Tempi di ciclo del processore: τ
- Tempo di decadimento della memoria: δ
- Capacità della memoria: μ

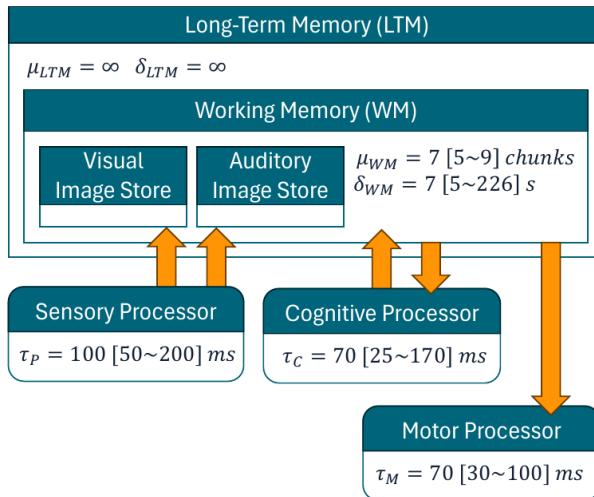


Figura 11.4: Parametri del modello

11.2.1 MPH: Memorie

Memoria di Lavoro (WM) è un sottoinsieme di elementi «attivati» (chunk) dalla **Memoria a Lungo Termine (LTM)**

- I chunk possono essere composti da unità più piccole come lettere in una parola
- Un chunk potrebbe anche consistere in diverse parole, come in una frase ben nota

$\mu_{LTM} = \infty$ e $\delta_{LTM} = \infty$ $\mu_{WM} = 7 \pm 2$ chunk $\delta_{WM} = 7 \pm 2$ s

- Il tempo di decadimento per WM varia ampiamente in base al numero di chunk memorizzati
- δ_{WM} 1 chunk = 73 ± 53 s
- δ_{WM} 3 chunk = 7 ± 2 s

11.2.2 Percezione di stimoli congruenti e incongruenti

I tempi di percezione sono influenzati anche dalla natura degli stimoli. Per esempio, c'è un ritardo significativo nel tempo di reazione (effetto Stroop) tra stimoli congruenti e incongruenti.

MHP: Why?

- You can see an interaction as a «program» for the MHP
- You can then «run» it on the MPH to estimate completion time!
 - Humans differ from each other. MPH allows to tune parameters (e.g.: processor cycle times) to target humans with different abilities

MHP: Example application

- Suppose we are designing the main menu for a command-line utility with 16 different features
- Which alternative would be better?
 - 1x16 (breadth) or 4x4 (nested) menu?

```
*** MAIN MENU ***
A. Feature A
B. Feature B
C. Feature C
...
O. Feature O
P. Feature P
Insert your choice>
```

```
*** MAIN MENU ***
A. Submenu A
B. Submenu B
C. Submenu C
D. Submenu D
Insert your choice>
```

```
*** MAIN MENU ***
A. Feature A
B. Feature B
C. Feature C
D. Feature D
Insert your choice>
*** MAIN MENU ***
E. Feature E
F. Feature F
G. Feature G
H. Feature H
Insert your choice>
```

MHP Example: Breadth (1x16) Menu

```
foreach item in menu:
```

Execute eye movement to item
 Perceive item text, transfer to WM
 Retrieve meaning of item, transfer to WM
 Match code from displayed to needed item
if(Decide on match)
 break

Execute eye movement to menu item letter
 Perceive menu item letter, transfer to WM
 Decide on key
 Press key in response

 τ_M
 τ_P
 τ_C
 τ_C
 τ_C
 τ_M
 τ_P
 τ_C
 τ_M

Average number of iterations
 in a serial search on 16 items

$$(\tau_M + \tau_P + 3\tau_C) \cdot (16 + 1)/2$$

$$2\tau_M + \tau_P + \tau_C$$

$$\begin{aligned} T &= (\tau_M + \tau_P + 3\tau_C) \cdot (16 + 1)/2 + 2\tau_M + \tau_P + \tau_C \\ &= (70 + 100 + 3 \cdot 70) \cdot 8,5 + 2 \cdot 70 + 100 + 70 = \mathbf{3540 \text{ ms}} \end{aligned}$$

MHP Example: Depth (4x4) Menu

- Same procedure and steps as the depth menu
- But this time we do 2 serial searches over 4 items

$$\begin{aligned} T &= 2 \cdot [(\tau_M + \tau_P + 3\tau_C) \cdot (4 + 1)/2 + 2\tau_M + \tau_P + \tau_C] \\ &= 2 \cdot [(70 + 100 + 3 \cdot 70) \cdot 2,5 + 2 \cdot 70 + 100 + 70] = \mathbf{2520 \text{ ms}} \end{aligned}$$

- The 4x4 menu is predicted to be ~30% faster than the 1x16 one!

11.3 Modello GOMS

Goals (Obiettivi), Operators (Operatori), Methods (Metodi), Selection rules (Regole di selezione) Assunzioni:

- L'interazione con un sistema è risoluzione di problemi
 - Scomporre l'interazione in sottoproblemi
 - Determinare gli obiettivi per «affrontare» il problema
 - Specificare la sequenza di operazioni utilizzate per raggiungere gli obiettivi
 - I valori di tempo possono essere assegnati a ciascuna operazione
- a Goals (Obiettivi): Ciò che l'utente vuole ottenere (es., "avviare la funzione di utilità corretta").
- b Methods (Metodi): Possibili sequenze alternative di operatori utilizzate per raggiungere l'obiettivo.
- c Selection rules (Regole di selezione): Criteri per scegliere tra diversi metodi.
- d Operators (Operatori): Azioni di base eseguite dall'utente (es., "spostare il mouse", "fare clic", "controllare l'impostazione").

11.3.1 GOMS: Esempio

Obiettivo: eliminare una parola in un editor di documenti Regola di selezione: se il cursore è alla fine della parola da eliminare, usa il Metodo A, altrimenti usa il Metodo B

1. Metodo A:

- Premere il tasto «backspace».
- Controllare se la parola è stata eliminata e tornare all'operazione precedente se necessario.

2. Metodo B:

- Spostare il cursore del mouse sulla parola.
- Eseguire un doppio clic.
- Premere il tasto «backspace».

11.3.2 Keyboard Level Model (KLM)

KLM è una delle varianti più semplici di GOMS Si concentra sul comportamento osservabile: Tasti, movimenti del mouse, ... Assume prestazioni prive di errori Operatori comuni e i tempi tipici corrispondenti:

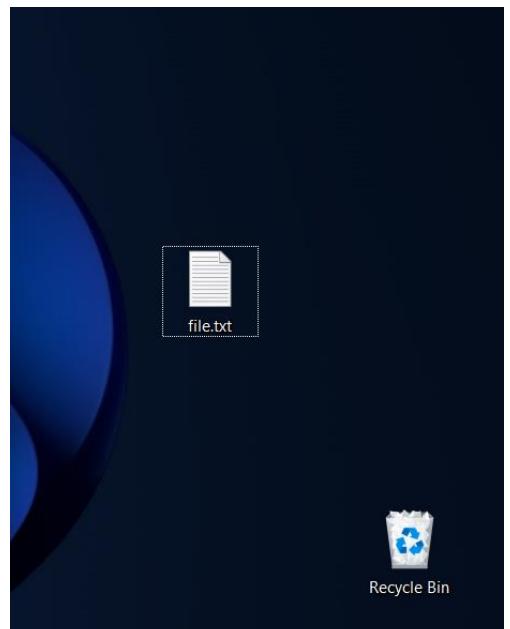
- K (Keystroke - Pressione tasto): 0,2 secondi (200 ms)
- P (Pointing with Mouse - Puntamento con il mouse): 1,1 secondi (1100 ms)
- B (Pressing/holding/releasing mouse button - Premere/tenere/rilasciare il pulsante del mouse): 0,1 secondi (100 ms)
- H (Homing Hands - Posizionamento delle mani): 0,4 secondi (400 ms)
- M (Mental Preparation - Preparazione mentale): 1,2 secondi (1200 ms)
- R (System Response - Risposta del sistema): Variabile; tipicamente intorno a 0,1 secondi (100 ms)

KLM Example: deleting a file

We can delete a file by:

- A. Dragging and dropping it to the trash can icon
- B. Selecting it and pressing the «delete» key
- C. Performing a right click on it, then selecting the «move to trash» option

Which alternative is faster?



KLM Example: deleting a file

Alternative A operators:

- Prepare to perform action (M)
- Point with the mouse over the file icon (P)
- Press and hold mouse button (B)
- Drag file icon to trash can icon (P)
- Release mouse button (B)

Assumptions:

- Hand already on mouse
- Trash icon visible
- File icon visible

$$\text{Total time} = M + 2P + 2B = 1.2 \text{ s} + 2 * 1.1 \text{ s} + 2 * 0.1 \text{ s} = 3.6 \text{ s}$$

KLM Example: deleting a file

Alternative B operators:

- Prepare to perform action (M)
- Point with the mouse over the file icon (P)
- Perform click (B)
- Move hand to keyboard (H)
- Press «delete» key (K)

Assumptions:

- Hand already on mouse
- Trash icon visible
- File icon visible

Total time = M + P + B + H + K = 1.2 s + 1.1 s + 0.1 s + 0.4 s + 0.2 s = 3 s

KLM Example: deleting a file

Alternative C operators:

- Prepare to perform action (M)
- Point with the mouse over the file icon (P)
- Perform right click (B)
- Point with mouse over «delete» option in context menu (P)
- Perform click (B)

Assumptions:

- Hand already on mouse
- Trash icon visible
- File icon visible

$$\text{Total time} = M + 2*P + 2*B = 1.2 \text{ s} + 2*1.1\text{s} + 2*0.1\text{s} = 3.6 \text{ s}$$

11.4 La Legge di Potenza della Pratica

Allen Newell (scienziato cognitivo) negli anni '80 analizzò i tempi di reazione per una varietà di compiti in esperimenti di apprendimento. Notò che le curve di apprendimento ottenute in questi studi hanno una forma molto simile: quella di una **legge di potenza**.

- Il tempo necessario per completare un compito dopo n prove (T_n) è vicino al tempo necessario per completare quel compito la prima volta (T_1) moltiplicato per n^{-a} $T_n \approx T_1 \cdot n^{-a}$
- a è un parametro compreso tra 0,2 e 0,6 (generalmente $\sim 0,4$)

Un utente ha impiegato 5 secondi per eseguire un determinato compito la prima volta che è stato esposto alla nuova interfaccia utente che abbiamo sviluppato. Quante ripetizioni sarebbero necessarie a quell'utente per essere in grado di eseguire il compito in 2 secondi o meno?

- Possiamo calcolare una stima con la legge di potenza della pratica: $T_n \approx T_1 \cdot n^{-a}$
- Risolviamo per n , assumendo $a = 0,4$
- $2s \leq 5s \cdot n^{-0,4}$
- Per $n = 10$, otteniamo che $T_n \approx 1,99s$

11.5 Legge di Hick

La legge di Hick descrive il tempo necessario a una persona per prendere una decisione tra un insieme di possibili scelte. La legge di Hick afferma che il tempo T necessario per raggiungere una decisione aumenta logaritmicamente con il numero di scelte. Nel caso di alternative ugualmente probabili: $T = a + b \cdot \log_2(n + 1)$

- n è il numero di scelte
- a e b sono parametri che dipendono dalle condizioni del contesto (es.: il modo in cui le scelte sono presentate, la familiarità dell'utente,...)

11.5.1 Applicazione della Legge di Hick

Quale modo è più veloce per selezionare tra 64 opzioni?

- Menu a un livello 1x64 $T = a + b \cdot \log_2 64 = a + 6b$
- Menu a due livelli 4x16 $T = a + b \cdot \log_2 4 + a + b \cdot \log_2 16 = 2a + 6b$
- Menu a due livelli 8x8 $T = 2 \cdot (a + b \cdot \log_2 8) = 2a + 6b$
- Menu a tre livelli 4x4x4 $T = 3 \cdot (a + b \cdot \log_2 4) = 3a + 6b$
- Menu a sei livelli 2x2x2x2x2x2 $T = 6 \cdot (a + b \cdot \log_2 2) = 6a + 6b$

11.6 Legge di Fitts

Modella il tempo per acquisire obiettivi nei movimenti mirati

- Raggiungere un controllo in una cabina di pilotaggio
- Muoversi attraverso un cruscotto
- Estrarre un elemento difettoso dal nastro trasportatore
- Fare clic su icone utilizzando un mouse

11.6.1 Legge di Fitts – Indice di Difficoltà (ID)

L'indice di difficoltà di un compito di acquisizione di un obiettivo è definito come $ID = \log_2(A/W + 1)$

- A è l'Aampiezza del movimento (distanza dall'inizio all'obiettivo)
- W è la Larghezza dell'obiettivo (variabilità ammissibile)

11.6.2 Legge di Fitts – Tempi di Movimento

I Tempi di Movimento (MT) dipendono dall'Indice di Difficoltà ID $MT = a + b \cdot ID$ $ID = a + b \cdot \log_2(A/W + 1)$ I tempi di movimento dipendono anche dal sistema, dispositivo di puntamento, utente... Possono essere adattati a casi specifici con parametri non negativi a e b È l'equazione di una linea retta ($y = mx + c$), dove b è il gradiente MT aumenta linearmente con l'ID

11.6.3 Legge di Fitts: Applicazioni

Se dobbiamo ridurre il tempo necessario per eseguire un'azione di ricerca di un obiettivo O riduciamo l'Aampiezza del movimento (avvicinare l'obiettivo) O aumentiamo la Larghezza dell'obiettivo O potremmo lavorare su a e b

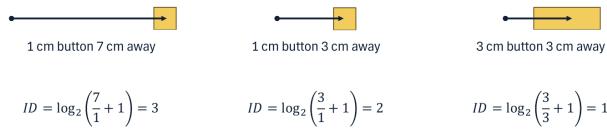
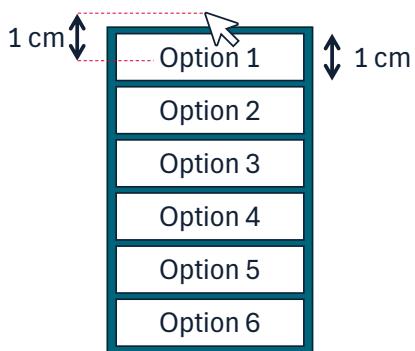


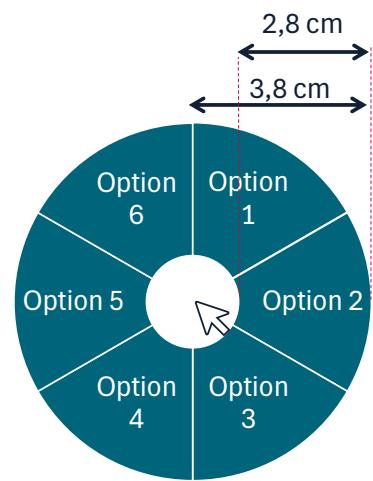
Figura 11.6: Variazioni di ID

Fitt's Law: Applications

- Which will be **faster** on average?



$$Average\ ID = \frac{\sum_{i=1}^6 \log_2 \left(\frac{i}{1} + 1 \right)}{6} \approx 2,04$$



$$Average\ ID = \log_2 \left(\frac{2,4}{2,8} + 1 \right) \approx 0,89$$

Fitt's Law: Applications

The screenshot shows a mobile application interface. At the top, it displays 'Corso di Studi in Informatica'. Below this, there is a list of courses: 'Linguaggi di Programmazione', 'Metodi Statistici per l'Informazione', and 'Esame a Scelta Libera (si veda Tabella Esami a Scelta Libera)'. Under the heading 'III ANNO', there are sections for 'Insegnamento' (with 'Reti e Programmazione Distribuita', 'Ingegneria del Software', and 'Tecniche di Programmazione Avanzata'), 'Copia', 'Cerca', 'Seleziona tutto', and a 'Tecnologie Web' section containing 'Esame a Scelta (si veda Tabella Esami a Scelta Libera)', 'Tirocinio + Altre attività di Orientamento', and 'Prova Finale'. At the bottom, there is a section for 'ESAMI A SCELTA LIBERA' with an 'Insegnamento' section.

The screenshot shows a Microsoft PowerPoint slide titled 'FITT'S LAW - MOVEMENT TIMES'. The slide contains the following text and formula:

- Movement Times (MT) depend on the Difficulty Index ID

$$MT = a + b \cdot ID = a + b \cdot \log_2 \left(\frac{A}{W} + 1 \right)$$

- Movement times all:
 - Can be fitted to specific data
 - It is the equation of a straight line
 - MT increases linearly with the ID

At the bottom of the slide, there is a note: 'Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 01 - Introduction to Web Technologies'.

Fitt's Law: Applications



Shadow of the Tomb Raider
(videogame)

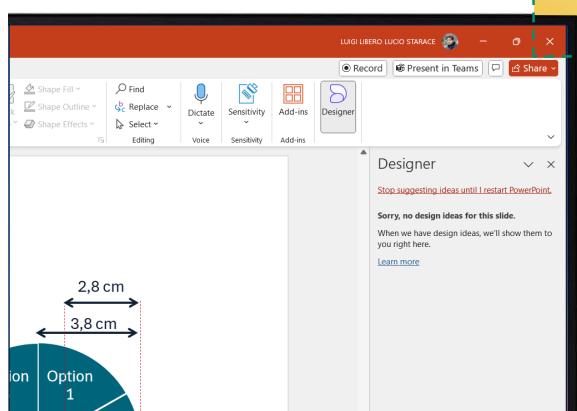


Grand Theft Auto: San Andreas – The Definitive Edition
(videogame)

Fitt's Law: Infinite Widths

- With a pointing device, targets near the edges have an infinite width
- These targets are fairly easy to hit, as $ID = \log_2(\frac{A}{\infty} + 1) = 0!$

Theoretical Effective Target Size



Fitt's Law: Mobile Devices



<https://www.toptal.com/designers/mobile-ui/fitts-law-user-interface-design>



Readings and references

- Buxton, W. (1990). A Three-State Model of Graphical Input. In D. Diaper et al. (Eds), Human-Computer Interaction - INTERACT '90. Amsterdam: Elsevier Science Publishers B.V. (North-Holland), 449-456.
<https://www.dgp.toronto.edu/OTP/papers/bill.buxton/3state.html>
- MacKenzie, I. S. (2003). Motor behaviour models for human-computer interaction. In *HCI models, theories, and frameworks: Toward a multidisciplinary science*
https://www.yorku.ca/mack/mackenzie_chapter.html

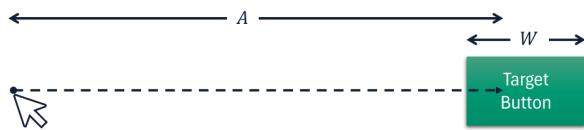


Figura 11.5: Ampiezza e Larghezza

Capitolo 12

Lezione 14: Bad Smell e refactoring

12.1 Bad Smells

12.1.1 Code Smells e Refactoring

I "Code smells" (odori del codice) sono indicazioni che ci sono problemi di design nel sistema. Le tecniche di refactoring correggono i code smells. La maggior parte delle tecniche di refactoring sono abbastanza semplici, e spesso c'è un ottimo supporto negli IDE (attualmente). Sia nel caso dei code smells che delle tecniche di refactoring, ne vengono "scoperti" di nuovi continuamente, quindi la lista dei nomi è piuttosto lunga: esamineremo un piccolo sottosinsieme di ciascuno. Link utili:

- <https://refactoring.com/>
- <https://refactoring.guru/>

12.1.2 Tipi di Bad Smells: Bloaters

I Bloaters sono codice, metodi e classi che sono cresciuti a proporzioni così gigantesche da diventare difficili da gestire. Di solito, questi odori non emergono immediatamente, ma si accumulano nel tempo mentre il programma evolve (e soprattutto quando nessuno si sforza di eliminarli). Esempi

- Metodo Lungo (Long Method)
- Classe Grande (Large Class)
- Ossessione Primitiva (Primitive Obsession)
- Lista Parametri Lunga (Long Parameter List)
- Grumi di Dati (Data Clumps)

12.1.3 Tipi di Bad Smells: Abusatori della Programmazione Orientata agli Oggetti

Tutti questi odori sono applicazioni incomplete o errate dei principi di programmazione orientata agli oggetti.

- Classi Alternative con Interfacce Diverse (Alternative Classes with Different Interfaces)
- Eredità Rifiutata (Refused Bequest)
- Istruzioni Switch (Switch Statements)
- Campo Temporaneo (Temporary Field)

12.1.4 Tipi di Bad Smells: Impedimenti al Cambiamento

Questi odori significano che se è necessario cambiare qualcosa in un punto del codice, bisogna fare molti cambiamenti anche in altri punti. Lo sviluppo del programma diventa molto più complicato e costoso di conseguenza.

- Cambiamento Divergente (Divergent Change)
- Gerarchie di Eredità Parallele (Parallel Inheritance Hierarchies)
- Chirurgia con Fucile a Pompa (Shotgun Surgery)

12.1.5 Tipi di Bad Smells: Superflui

Un superfluo è qualcosa di inutile e non necessario la cui assenza renderebbe il codice più pulito, più efficiente e più facile da capire.

- Commenti (Comments)
- Codice Duplicato (Duplicate Code)
- Classe Dati (Data Class)
- Codice Morto (Dead Code)
- Classe Pigra (Lazy Class)
- Generalità Speculativa (Speculative Generality)

12.1.6 Tipi di Bad Smells: Accoppiatori

Tutti gli odori in questo gruppo contribuiscono all'eccessivo accoppiamento tra classi o mostrano cosa succede se l'accoppiamento è sostituito da un'eccessiva delega.

- Invidia delle Caratteristiche (Feature Envy)
- Classe Libreria Incompleta (Incomplete Library Class)
- Catene di Messaggi (Message Chains)
- Uomo di Mezzo (Middle Man)

12.2 Bad Smells e il concetto di Debito Tecnico

12.2.1 Debito Tecnico

Si verifica ogni volta che il codice soddisfa i requisiti funzionali ma è subottimale o "veloce e sporco". Ad esempio:

- codice "maleodorante"
- algoritmi inefficienti
- design sciatto

Potrebbe essere corretto durante la code-review, potrebbe generare TODOs o nuovi problemi nel sistema di tracciamento dei problemi. Comprendere, misurare e comunicare il debito tecnico è critico nell'industria del software.

12.2.2 Debito Strategico, Intenzionale

Esempi di debito intenzionale o strategico:

- Design non modulare
- Implementazione intenzionalmente troppo semplice/troppo complessa
- Indifferenza alle prestazioni
- Mancanza di generalità o estensibilità
- Mancanza di scalabilità

12.2.3 Debito Non-Strategico, Non Intenzionale

Il debito si accumula anche involontariamente, nessun processo di sviluppo è perfetto. Esempi di debito non intenzionale o non strategico:

- Bad Smells
- Memory leaks
- Copertura dei test insufficiente
- Implementazione involontariamente complessa
- Architettura rigida/fragile
- Colli di bottiglia nelle prestazioni o nella scalabilità
- Codice disordinato o difficile da mantenere

12.2.4 Conseguenze del Debito

Troppo debito = troppo tempo speso a pagare gli interessi. Imprevedibilità nella fase di pianificazione del software, aumento del rischio di investimento. Rallenta il lavoro futuro. Più bug, più costoso correggerli. Sviluppatori frustrati e infelici.

12.2.5 Misurare il Debito Tecnico

Alcuni strumenti relativi alla qualità, come SonarQube, possono fornire una stima accurata del debito tecnico accumulato all'interno di un repository software.

12.3 Correggere i Bad Smells con il Refactoring

12.3.1 Refactoring

Metafora di base:

- Iniziare con una base di codice esistente e migliorarla.
- Cambiare la struttura interna (dal piccolo al medio) preservando la semantica complessiva: cioè, riorganizzare i "fattori" ma ottenere lo stesso "prodotto" finale.

L'idea è che dovresti migliorare il codice in qualche modo significativo. Per esempio:

- Ridurre il codice quasi duplicato
- Migliorare la coesione, ridurre l'accoppiamento
- Migliorare la parametrizzazione, la comprensibilità, la manutenibilità, la flessibilità, l'astrazione, l'efficienza, ecc...

12.3.2 Il Ciclo di Refactoring

Schema di base per il refactoring con un programma funzionante:

1. Scegliere l'odore peggiore
2. Selezionare un refactoring che affronterà l'odore
3. Applicare il refactoring.

Selezionare refactoring per migliorare il codice ad ogni passaggio attraverso il ciclo. Il comportamento del programma non viene modificato. Quindi, il programma rimane in uno stato funzionante. Così il ciclo migliora il codice ma mantiene il comportamento.

La parte più difficile del processo: Identificare l'odore! Quando inizi il refactoring, è meglio iniziare con le cose facili (ad esempio, suddividere grandi routine o rinominare cose per chiarezza). Questo ti permette di vedere e correggere i problemi rimanenti più facilmente.

12.3.3 Refactoring e Test Unitari

Il refactoring dipende fortemente dall'avere una buona suite di test unitari. Con i test unitari, possiamo fare refactoring. Poi eseguire i test automatizzati, per verificare che il comportamento sia effettivamente preservato. Senza buoni test unitari,

- gli sviluppatori potrebbero evitare il refactoring
- A causa della paura di poter rompere qualcosa.

12.3.4 Perché Dovresti Fare Refactoring?

La realtà:

- Estremamente difficile ottenere il design "giusto" la prima volta
- Difficile comprendere pienamente il dominio del problema
- Difficile capire i requisiti dell'utente, anche se l'utente li conosce!
- Difficile sapere come il sistema evolverà in X anni
- Il design originale è spesso inadeguato
- Il sistema diventa fragile nel tempo e più difficile da cambiare

Il refactoring ti aiuta a

- Manipolare il codice in un ambiente sicuro (preservando il comportamento)
- Ricreare una situazione in cui l'evoluzione è possibile
- Comprendere il codice esistente

Il refactoring migliora il design del software

- Senza refactoring il design del programma si deteriorerà
- Il codice mal progettato di solito richiede più codice per fare le stesse cose, spesso perché il codice fa la stessa cosa in luoghi diversi

Il refactoring rende il software più facile da capire. Nella maggior parte degli ambienti di sviluppo software, qualcun altro dovrà eventualmente leggere il tuo codice. Il refactoring ti aiuta a trovare bug. Il refactoring ti aiuta a programmare più velocemente.

Typical Refactorings

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
Extract class	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method	abstract variable
	extract code in new method	
	replace parameter with method	

Extract Method

```
void printOwing() {  
    printBanner();  
    //print details  
    System.out.println ("name:      " + _name);  
    System.out.println ("amount      " + getOutstanding());  
}  
  
  

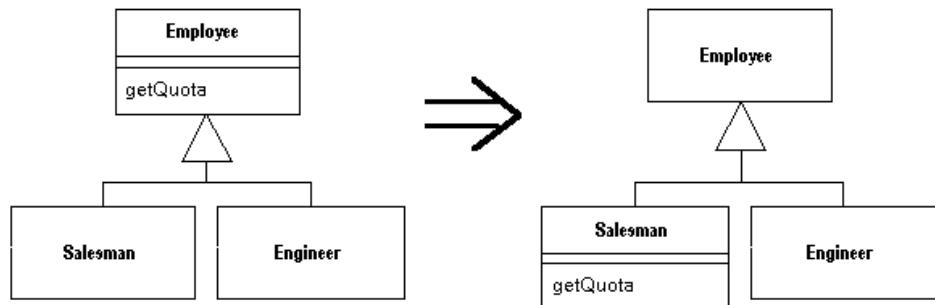

```
void printOwing() {
 printBanner();
 printDetails(getOutstanding());
}

void printDetails (double outstanding) {
 System.out.println ("name: " + _name);
 System.out.println ("amount " + outstanding);
}
```


```

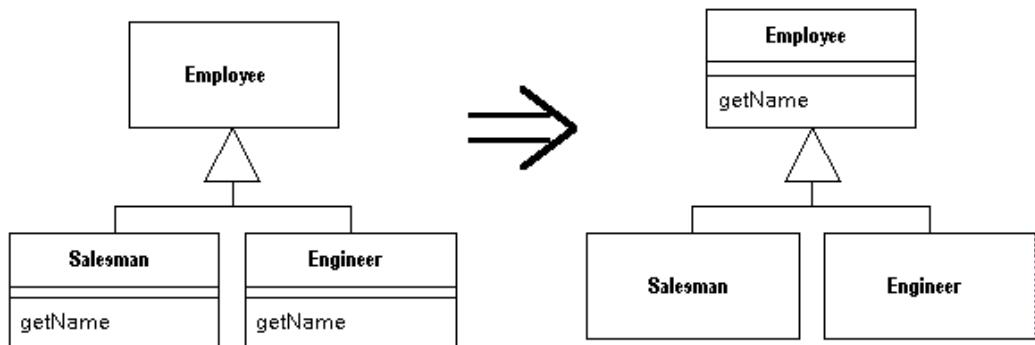
Push method down

- Behavior of a superclass is relevant only for some of its subclasses.
- Move it to those subclasses.



Push method up

- You have methods with identical results on subclasses.
- Move them to the superclass



Add parameter to method

- A method needs more information from its caller.
- Add a parameter for an object that can pass on this information.



Replace Parameter with Method

```
int basePrice = _quantity * _itemPrice;  
  
discountLevel = getDiscountLevel();  
  
double finalPrice = discountedPrice  
(basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;  
double finalPrice =  
    discountedPrice (basePrice);
```

// An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

From Website

- <https://refactoring.guru/>
- <https://github.com/nerdschoolbergen/codesmells/tree/master/assignment/src/main/java/nerdschool/bar>



12.4.2 Codice Duplicato

Se vedi la stessa struttura di codice in più di un posto, puoi essere sicuro che il tuo programma sarà migliore se trovi un modo per unificarli. Il problema di codice duplicato più semplice è quando hai la stessa espressione in due metodi della stessa classe. Esegui Extract Method (Estrai Metodo) e invoca il codice da entrambi i posti. Un altro problema di duplicazione comune è avere la stessa espressione in due sottoclassi sorelle. Esegui Extract Method in entrambe le classi e poi Pull Up Field (Tira Su il Campo). Se hai codice duplicato in due classi non correlate, considera di usare Extract Class (Estrai Classe) in una classe e poi usa il nuovo componente nell'altra.

12.4.3 Metodo Lungo

Più lungo è una procedura, più è difficile da capire. I programmi orientati agli oggetti vivono meglio e più a lungo con metodi brevi. Quasi sempre, tutto ciò che devi fare per abbreviare un metodo è Extract Method. Se provi a usare Extract Method e finisci per passare molti parametri, spesso puoi creare una nuova Classe che incapsuli i parametri.

12.4.4 Classe Grande

Quando una classe sta cercando di fare troppo, spesso si manifesta con troppe variabili di istanza. Quando una classe ha troppe variabili di istanza, il codice duplicato non può essere lontano. Una classe con troppo codice è anche un terreno fertile per la duplicazione. In entrambi i casi Extract Class ed Extract Subclass funzioneranno.

12.4.5 Lista Parametri Lunga

Vecchia scuola: passa tutto come parametri. Era meglio dei dati globali. Con gli oggetti non è necessario passare tutto ciò di cui il metodo ha bisogno, invece passi abbastanza in modo che il metodo possa accedere a tutto ciò di cui ha bisogno. Questo è positivo, perché le lunghe liste di parametri sono difficili da capire, perché sono incoerenti e difficili da usare, perché le stai cambiando continuamente quando hai bisogno di più dati. Usa Replace Parameter with Method (Sostituisci Parametro con Metodo) quando puoi ottenere i dati in un parametro facendo una richiesta a un oggetto che già conosci. Usa Introduce Parameter Object (Introduci Oggetto Parametro).

12.4.6 Cambiamento Divergente

Il cambiamento divergente si verifica quando una classe viene comunemente modificata in modi diversi per ragioni diverse. Se stai facendo questo, stai quasi certamente violando i principi di una chiave astrazione e separazione delle preoccupazioni, e dovrresti fare refactoring del tuo codice. Per sistemare questo, identifica tutto ciò che cambia per una particolare causa e usa Extract Class per metterli tutti insieme.

12.4.7 Chirurgia con Fucile a Pompa

Questa situazione si verifica quando ogni volta che fai un tipo di cambiamento, devi fare molti piccoli cambiamenti a molte classi diverse. Lo stesso tasso di cambiamento in oggetti diversi, particolarmente se sono scollegati. Un cambiamento concettualmente semplice che richiede modifiche al codice in molti posti. Il risultato della Programmazione Copia e Incolla. Quando i cambiamenti sono sparsi ovunque sono difficili da trovare, ed è facile perdere un cambiamento importante. Vuoi usare Move Method (Sposta Metodo) e Move Field (Sposta Campo) per mettere tutti i cambiamenti in una singola classe. Se nessuna classe attuale sembra un buon candidato, creane una.

12.4.8 Invidia delle Caratteristiche

“L’invidia delle caratteristiche” è quando un metodo fa un uso intenso di dati e metodi da un’altra classe. Usa Move Method per metterlo nella classe più desiderata. A volte solo una parte del metodo fa un uso intenso delle caratteristiche di un’altra classe. Usa Extract Method per estrarre quelle parti che appartengono all’altra classe.

12.4.9 Grumi di Dati

Spesso vedrai gli stessi tre o quattro elementi di dati insieme in molti luoghi:

- Campi in un paio di classi
- Parametri in molte firme di metodi

Gruppi di dati che stanno insieme dovrebbero davvero essere trasformati in un proprio oggetto, Es.: $x,y \rightarrow Point$. Il primo passo è cercare dove i grumi appaiono come campi e usare Extract Class per trasformare i grumi in un oggetto. Per i parametri dei metodi usa Introduce Parameter Object o Preserve Whole Object (Preserva Oggetto Interio) per snellirli.

12.4.10 Ossezione Primitiva

Le persone nuove agli oggetti sono talvolta riluttanti a usare piccoli oggetti per piccoli compiti, come classi per soldi che combinano numeri e valuta, intervalli con un limite superiore e inferiore, e stringhe speciali come numeri di telefono e codici postali. Molti programmati sono riluttanti a introdurre classi "piccole" che rappresentano cose facilmente rappresentate da primitivi—numeri di telefono, codici postali, importi di denaro, intervalli (variabili con limiti superiori e inferiori). Se il tuo primitivo ha bisogno di dati o comportamenti aggiuntivi, considera di trasformarlo in una classe. Per esempio, potresti voler formattare il tuo primitivo in un modo speciale, come (215)898-0587 o 19104-6389.

12.4.11 Test di Tipo

La maggior parte delle volte quando vedi un'istruzione switch su un tipo dovresti considerare il polimorfismo. Usa Extract Method per estrarre l'istruzione switch e poi Move Method per portarla nella classe dove è necessario il polimorfismo.

12.4.12 Gerarchie di Eredità Parallelle

È davvero un caso speciale di chirurgia con fucile a pompa. In questo caso ogni volta che fai una sottoclasse di una classe, devi fare una sottoclasse di un'altra. Ci sono due modi per procedere. Per rimuovere il parallelo: fare refactoring di una o entrambe le gerarchie fino a quando i loro membri sono congruenti, quindi collassarle a coppie. Per rimuovere la duplicazione tra i paralleli: definire responsabilità distinte raffinate da ciascuna gerarchia e riposizionare i metodi come appropriato. — Ward Cunningham. Se usi Move Method e Move Field, la gerarchia sulla classe di riferimento scompare.

12.4.13 Classe Pigra

Ogni classe che crei costa denaro e tempo da mantenere e capire. Una classe che non sta portando il suo peso dovrebbe essere eliminata. Se hai sottoclassi che non stanno facendo abbastanza prova a usare Collapse Hierarchy (Collassa Gerarchia) e i componenti quasi inutili dovrebbero essere sottoposti a Inline Class (Inserisci Classe).

12.4.14 Generalità Speculativa

Ottieni questo odore quando qualcuno dice "Penso che avremo bisogno di fare questo un giorno" e hai bisogno di ogni tipo di ganci e casi speciali per gestire cose che non sono richieste. Questo odore è facilmente rilevabile quando gli unici utenti di una classe o metodo sono casi di test. Se hai classi astratte che non stanno facendo abbastanza, usa Collapse Hierarchy. La delega non necessaria può essere rimossa con Inline Class. I metodi con parametri inutilizzati dovrebbero essere soggetti a Remove Parameter (Rimuovi Parametro). I metodi nominati con strani nomi astratti dovrebbero essere riparati con Rename Method (Rinomina Metodo).

12.4.15 Campo Temporaneo

A volte vedrai un oggetto in cui una variabile di istanza è impostata solo in determinate circostanze. Questo può rendere il codice difficile da capire perché di solito ci aspettiamo che un oggetto usi tutte le sue variabili. Usa Extract Class per creare una casa per queste variabili orfane mettendo tutto il codice che usa la variabile nel componente.

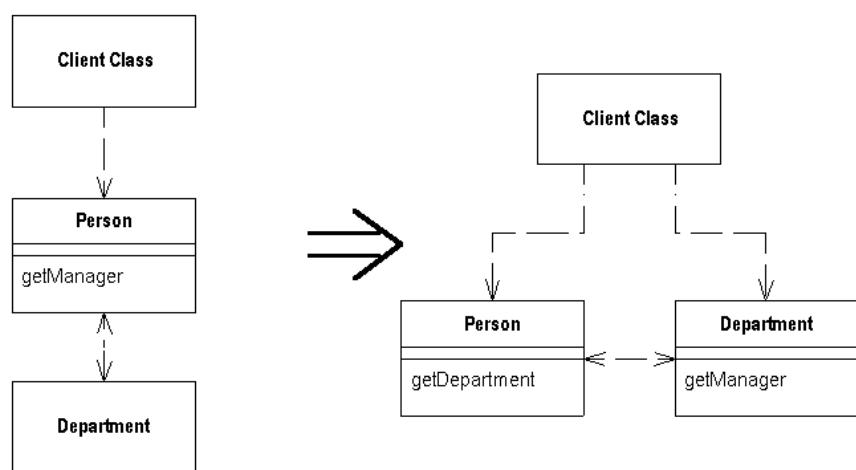
12.4.16 Catene di Messaggi

Le catene di messaggi si verificano quando vedi un client che chiede a un oggetto un altro oggetto, che il client poi chiede per un altro oggetto ancora, che il client poi chiede per un altro oggetto ancora, ecc.: intermediate.getProvider().doSomething() Altrimenti: ch = vehicle->getChassis(); body = ch->getBody(); shell = body->getShell(); material = shell->getMaterial(); props = material->getProperties(); color = props->getColor(); Navigare in questo modo significa che il client è strutturato sulla struttura della gerarchia.

12.4.17 Middle Man

Una delle principali caratteristiche degli Oggetti è l'incapsulamento. L'incapsulamento spesso viene con la delega. A volte la delega può andare troppo oltre. Per esempio, se trovi che metà dei metodi sono delegati a un'altra classe, potrebbe essere il momento di usare Remove Middle Man (Rimuovi Uomo di Mezzo) e parlare con l'oggetto che realmente sa cosa sta succedendo. Se solo pochi metodi non stanno facendo molto, usa Inline Method per incorporarli nel chiamante. Se c'è un comportamento aggiuntivo, puoi usare Replace Delegation with Inheritance (Sostituisci Delega con Eredità) per trasformare l'uomo di mezzo in una sottoclasse dell'oggetto reale.

Remove Middle Man



Rifactorizzare il seguente codice

```
void printProperties(IList users)
{
    for (int i = 0; i < users.size(); i++)
    {
        string result = "";
        result += users.get(i).getName();
        result += " ";
        result += users.get(i).getAge();
        Console.WriteLine(result);

        // ...
    }
}
```

Rifactorizzare il seguente codice

```
public bool Login(string username, string password) {
    var user = userRepo.GetUserByUsername(username);

    if(user == null)
        return false;

    if (loginValidator.IsValid(user, password))
        return true;

    user.FailedLoginAttempts++;

    if (user.FailedLoginAttempts >= 3)
        user.LockedOut = true;

    return false;
}
```

Capitolo 13

Lezione 15: Teoria dei colori, tipografia e psicologia di Gestalt

13.1 Percezione del colore tramite i coni

Non tutti i coni sono uguali. Esistono 3 tipi diversi, con fotopigmenti specializzati per percepire il colore: blu, verde e rosso. Ogni tipo di cono è sensibile a una diversa banda dello spettro luminoso. Il rapporto di stimolazione neurale tra i tre tipi ci dà una percezione continua dei colori.

I tipi di coni non sono distribuiti uniformemente al centro della retina: principalmente rossi, pochissimi blu. Sensibilità limitata alle lunghezze d'onda corte, alta sensibilità a quelle lunghe. Pochi coni blu al centro della retina, è più difficile mettere a fuoco piccoli oggetti blu. Con l'età, il cristallino tende ad assorbire più lunghezze d'onda corte, riducendo ulteriormente la sensibilità ai blu.

13.2 Modelli di colore

I modelli di colore sono modelli matematici astratti che descrivono come i colori possono essere rappresentati tramite tuple di numeri. Due schemi di colore ampiamente utilizzati sono:

- CMYK
- RGB

13.2.1 Modello di colore CMYK

Il CMYK è un modello di colore sottrattivo. Utilizza: Ciano, Magenta, Giallo, Key (nero). È generalmente usato per materiali stampati. Utilizza pigmenti d'inchiostro per mostrare il colore. I colori risultano dalla luce riflessa.

13.2.2 Modello di colore RGB

L'RGB è un modello di colore additivo. Utilizza Rosso, Verde, Blu. È pensato per i display dei computer. Utilizza la luce per mostrare il colore. Il colore risulta dalla luce emessa. I display non possono produrre diversi canali di colore nella stessa posizione: la griglia dei pixel è tipicamente divisa in regioni di colore singolo (subpixel), che contribuiscono al colore visualizzato quando osservate a distanza.

13.3 Dimensioni percettive del colore

HSL (Hue, Saturation, Lightness) è una delle rappresentazioni più comuni dei colori nel modello RGB.

13.3.1 Hue

La tonalità (*Hue*) è la lunghezza d'onda dominante in un colore. In HSL, è un valore tra 0 e 360:

- 0 (e 360) è rosso
- 60 è giallo
- 120 è verde
- 180 è ciano

- 240 è blu
- 300 è magenta

13.3.2 Saturazione

Indica quanto il colore è «intenso».

- Tipicamente è una percentuale tra 0 e 100
- 100% significa che il colore è brillante e puro
- 80% significa che il colore è miscelato con il 20% di grigio
- 0% significa che si ottiene solo il grigio

13.3.3 Luminosità

La luminosità (*Lightness*) indica quanta luce ha il colore.

- 0% è molto scuro (nero)
- 50% è a metà, né troppo scuro né troppo brillante
- 100% è molto brillante (bianco)

13.3.4 Rappresentazione HSL

HSL è un modello a coordinate cilindriche. Definendo Hue, Saturation e Lightness, si identifica un punto nello spazio cilindrico.

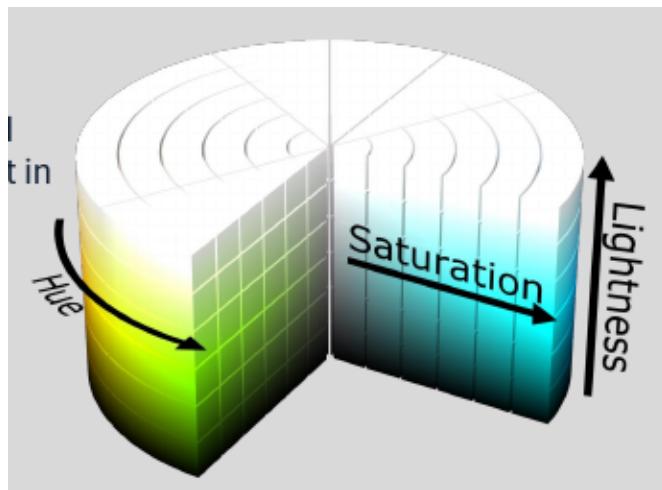


Figura 13.1: Rappresentazione cilindrica di HSL

13.3.5 Tinte, sfumature e toni

- Una tinta (*tint*) è una miscela di un colore con il bianco, aumentando la luminosità
- Una sfumatura (*shade*) è una miscela con il nero, diminuendo la luminosità
- Un tono (*tone*) è una miscela con il grigio

13.4 Uso dei colori nel design delle interfacce

Il colore è uno strumento potente nell'arsenale di un designer di UI.

- Può far risaltare alcuni elementi (vedremo di più su questo alla fine della lezione)
- Può trasmettere significato (ma attenzione alle differenze regionali!)

Quindi, più colori ci sono, meglio è?

- Non proprio. Una UI dovrebbe generalmente includere non più di 6 colori diversi.
- Corollario: scegli una palette e sii coerente!

Una buona regola pratica è la regola del 60-30-10:

- 60% colore primario
- 30% colore secondario
- 10% colore d'accento (per le parti che vogliamo far risaltare)

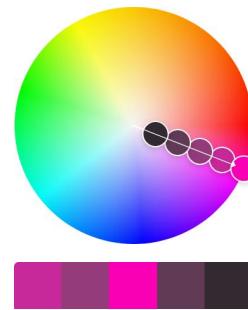
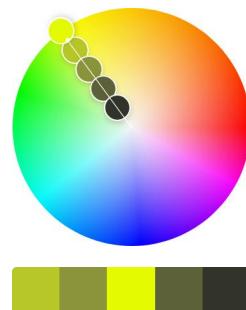
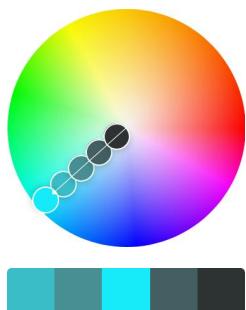
13.4.1 Teoria del colore

Non tutti i colori «stanno bene» insieme. Possiamo usare le armonie cromatiche per costruire palette (insiemi di colori) per la nostra UI che non siano in contrasto tra loro.

- Monocromatica
- Analoghi
- Complementari
- Complementari suddivisi
- Triadica

Monochromatic Color Armonies

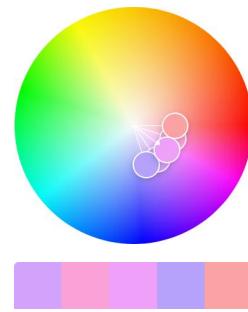
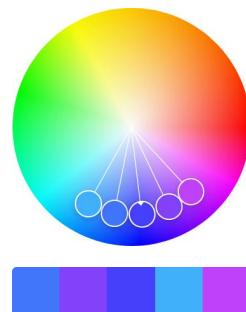
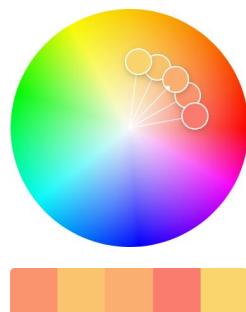
- Select one main color and generate others as different tints, tones, or shades of that single hue



Palettes generated at <https://color.adobe.com/create/color-wheel>

Analogous Color Armonies

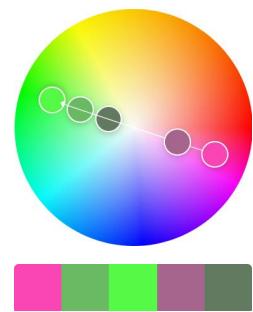
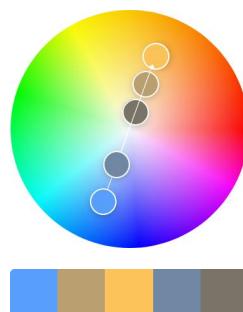
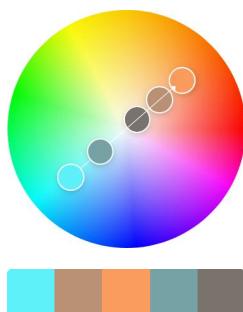
- Pick adjacent colors on the color wheel
- Leads to color schemes with reduced contrast



Palettes generated at <https://color.adobe.com/create/color-wheel>

Complementary Color Armonies

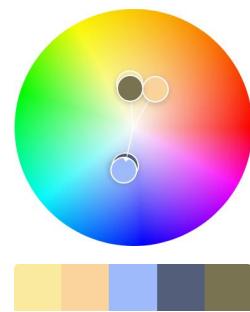
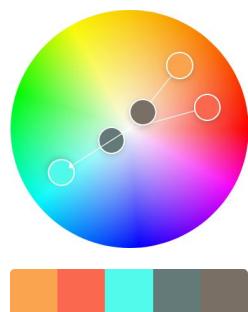
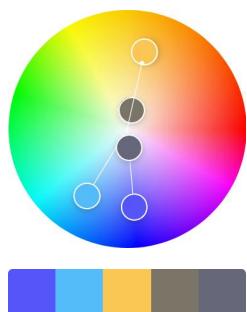
- Pick colors that are opposite on the color wheel
- Produces color scheme with higher contrast



Palettes generated at <https://color.adobe.com/create/color-wheel>

Split-complementary Color Armonies

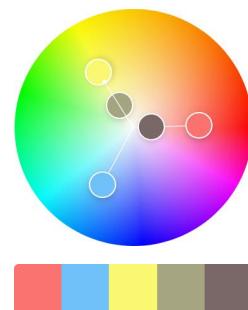
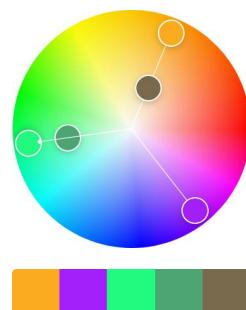
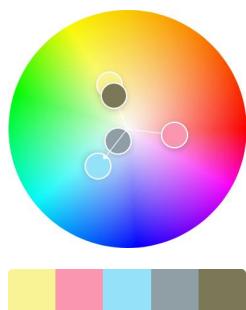
- Pick one color and combine with colors from either side of its complementary color
- Softens contrast w.r.t. complementary armonies



Palettes generated at <https://color.adobe.com/create/color-wheel>

Triadic Color Armonies

- Pick three colors equidistant on the color wheel (120° apart)
- Softens contrast w.r.t. complementary armonies

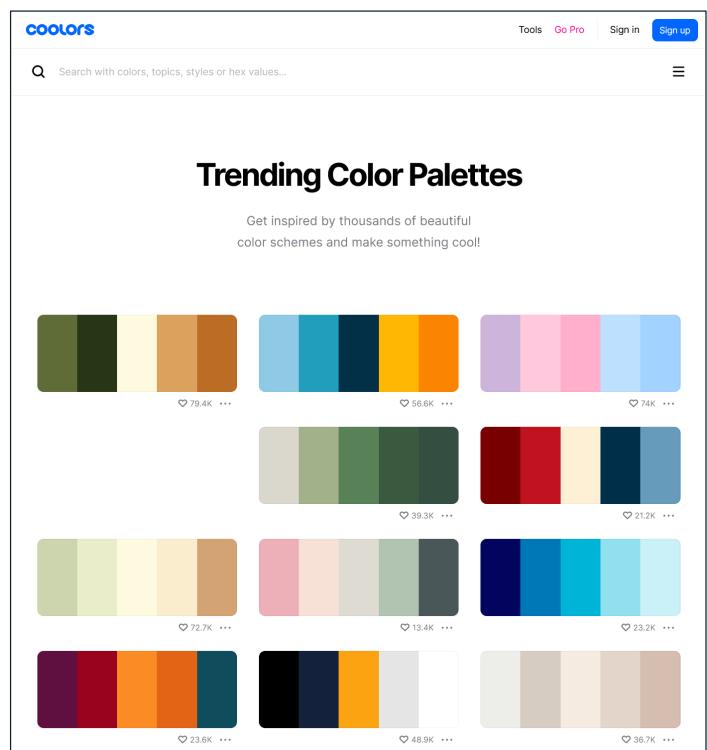


Palettes generated at <https://color.adobe.com/create/color-wheel>

Color Armonies

You can explore palettes made by others at:

- <https://coolors.co/palettes/trending>
- <https://color.adobe.com/explore>



13.4.8 Psicologia del colore

La mente umana può reagire inconsciamente ai colori (psicologia del colore).

- Il nero è associato a eleganza, potere e autorità
- Il blu è percepito come autorevole, affidabile, degno di fiducia
- Il rosso può essere associato a passione, desiderio, amore, energia, pericolo
- Il verde può essere associato a natura, freschezza, serenità, salute, denaro

Non ci sono molte ricerche che dimostrano l'effetto reale di un colore sulle emozioni. Inoltre, tieni presente che esistono differenze regionali! In Cina, il rosso è il colore associato al denaro. Negli Stati Uniti, è il verde.

13.5 Tipografia

13.5.1 L'ipotesi dei serif

I caratteri con grazie (*serif*) sono più facili da leggere – e quindi preferibili per lunghi testi – perché le grazie forniscono ancoraggi che guidano l'occhio del lettore. I font senza grazie (*sans serif*) mancano di questi ancoraggi e sono quindi meno adatti per lunghi testi. In pratica, le differenze individuali superano qualsiasi effetto della presenza/assenza di grazie, cioè alcune persone leggono più velocemente di altre.

13.5.2 Impatto della tipografia nelle UI

La tipografia può essere un buon modo per veicolare messaggi nelle interfacce.

- Leggere non significa solo riconoscere sequenze di lettere
- La tipografia può trasmettere messaggi aggiuntivi:
 - «Questo è pensato per essere facile da leggere»
 - «Questo è un messaggio giocoso»
 - «Questo può essere percepito come codice sorgente»

La leggibilità è un aspetto importante dell'usabilità.

- Gli utenti devono leggere etichette e informazioni nelle nostre UI
- La tipografia gioca un ruolo importante nella facilità di lettura. Anche l'aspetto delle parole può essere importante.

13.5.3 Effetto di superiorità della parola

A volte, i lettori riconoscono una parola dalla sua forma, prima ancora di riconoscere le lettere che la compongono. Questo è chiamato «Effetto di superiorità della parola» in psicologia cognitiva. Le persone generalmente leggono il testo latino minuscolo più velocemente di quello maiuscolo. Non usare il maiuscolo per lunghi testi!

13.6 Teoria della percezione Gestalt

La teoria della percezione Gestalt (nota anche come psicologia della Gestalt) si concentra su come la mente umana elabora le informazioni visive. La psicologia della Gestalt si riferisce all'idea di insieme unificato.

- Generalmente percepiamo qualcosa di diverso dalla semplice somma degli elementi che vediamo
- Diamo significato alla somma delle parti piuttosto che ai singoli elementi

Quindi, cosa vedi nell'immagine qui sotto?

- Un triangolo bianco sopra
- Un triangolo con bordi neri sotto
- Tre cerchi neri parzialmente coperti

Perché non vediamo solo un insieme di linee e macchie? La nostra mente tende a completare oggetti incompleti e a vedere connessioni tra elementi in base alla loro apparenza o posizione relativa.

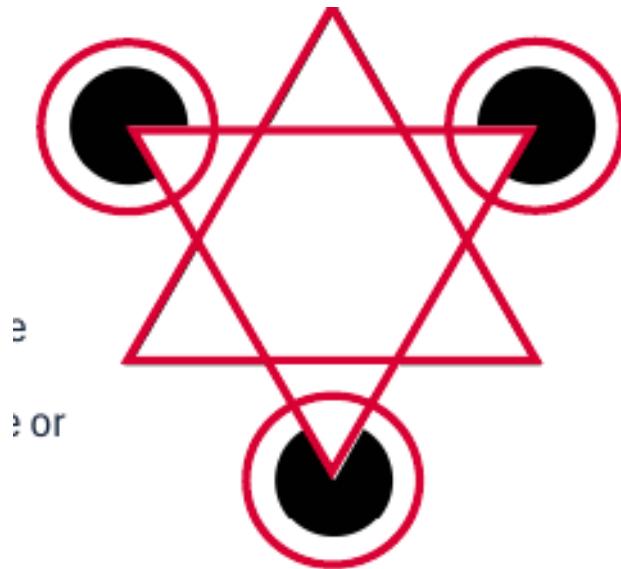


Figura 13.2: Gestalt in azione

13.6.1 Psicologia della Gestalt

La psicologia della Gestalt studia queste tendenze della nostra mente e come si manifestano. Queste tendenze sono talvolta chiamate «**leggi della percezione**».

- Non sono vere leggi, ma principi o euristiche importanti
- Nelle prossime slide vedremo alcuni di questi principi
- Capendoli, possiamo usarli per rendere le nostre UI più intuitive
- Vogliamo che le nostre UI lavorino con, e non contro, il modo in cui il cervello elabora gli stimoli visivi

13.7 Principi della Gestalt

13.7.1 Somiglianza

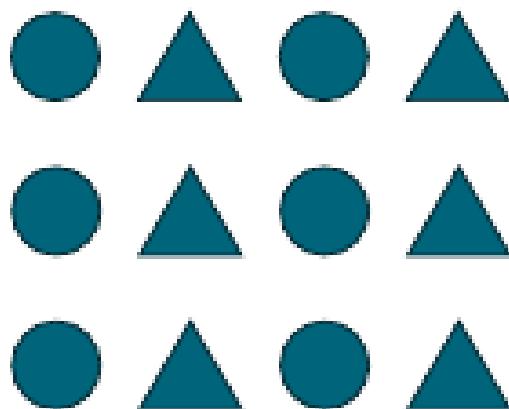


Figura 13.3: Forme righe-colonne

Scommetto che hai interpretato l'immagine sopra come quattro colonne e non tre righe. Gli elementi che condividono una caratteristica visiva sono percepiti come più correlati rispetto agli elementi dissimili. Le caratteristiche visive possono essere forme, dimensioni, colori, font, movimento, orientamento...

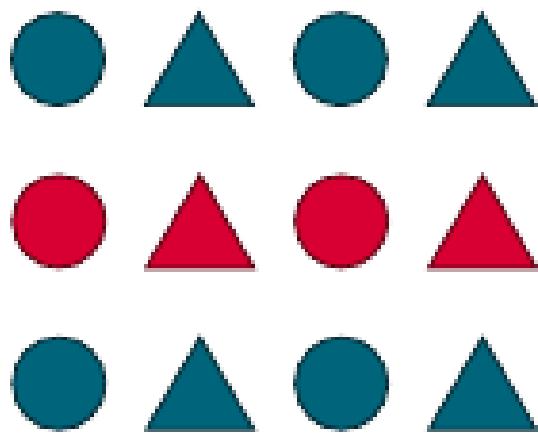


Figura 13.4: Aggiungendo un colore...

Aggiungendo un'altra caratteristica visiva (colore), la percezione può cambiare. Probabilmente ora vedi tre righe. La somiglianza cromatica spesso prevale su altre caratteristiche visive.

13.7.2 Somiglianza nelle UI

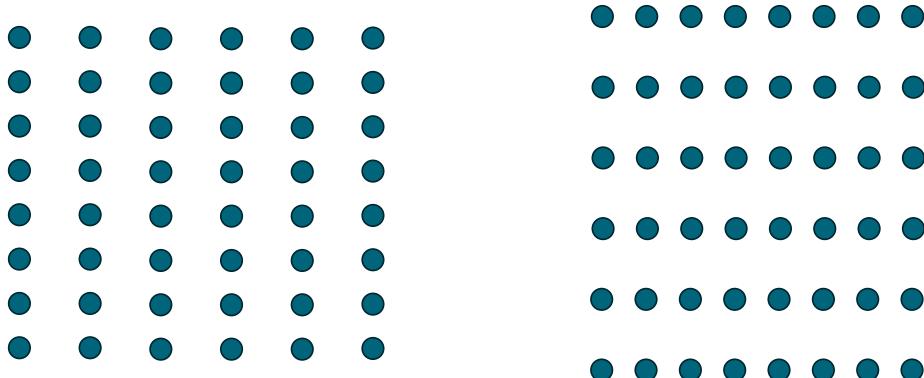
La somiglianza può essere usata per raggruppare elementi correlati.

- Se vuoi che elementi diversi siano percepiti come raggruppati e correlati, puoi farli condividere una o più caratteristiche visive
- Può essere usata per comunicare funzionalità comuni (ad esempio: pensa ai colori per segnalare i link nelle pagine web)

La somiglianza può anche essere usata per enfatizzare le differenze.

Principle of Proximity

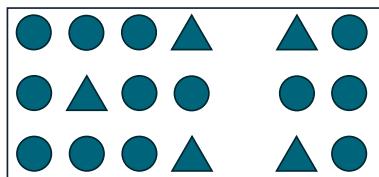
What pattern are you seeing in this image?



- Chances are you saw **six columns** on the left, and **six rows** on the right!

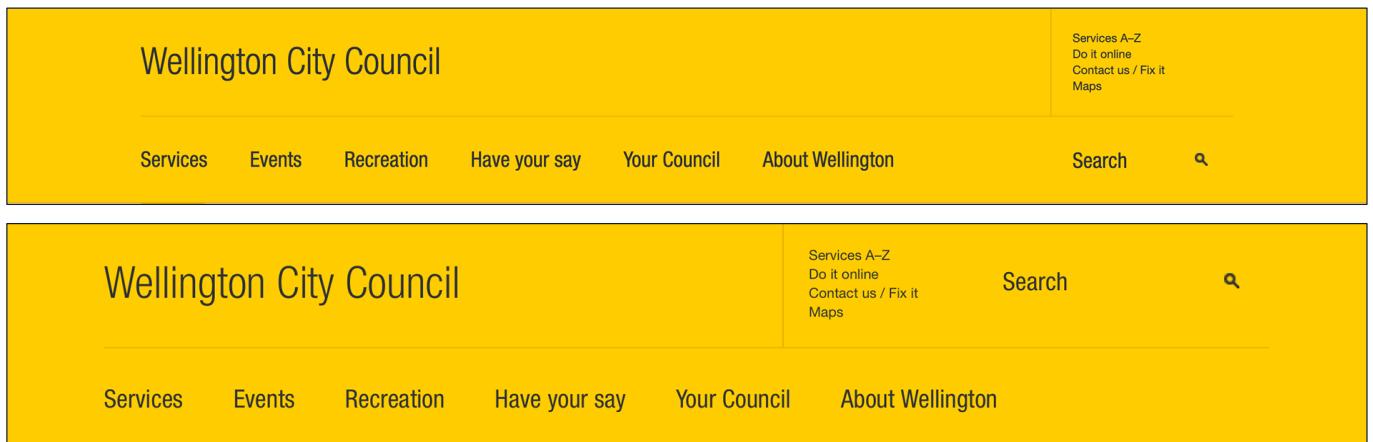
Proximity

What about in this picture?



- You probably see **two groups** of shapes (even though we included different shapes in them!)
- Whitespace is a powerful tool for making some elements appear to be related or unrelated
- When designing a UI, the spacing between elements is used in a way that helps users achieve their goals, and not misguides them

Proximity: example



Wellington City Council website, as seen on a desktop monitor (top) and tablet (bottom): The “Search control” is separated from the rest of the main navigation, which suggest it’s a different type of functionality.

Pics from <https://www.nngroup.com/>

Proximity: text

Clingy Cat Solutions

All cats young and old can exhibit clingy behavior. The good news is that we can change needy behavior through the use of positive reinforcement training. You will find that, in very simple ways, you can change your cat's mind and redirect unwanted neediness. Plus, training can be both mentally and physically stimulating for your cat. You may be hesitant to start because you're worried about the time involved. While it's true that some training goals can take some time to achieve, training sessions with a cat should always be short and sweet. Why not get started? All you need is 10 minutes a day. Here are three common needy behaviors and methods for resolving them.

1. Laptop Lover

My 17-year-old cat, Bob, has his own subtle (yet hugely irritating) way of bothering. One minute I'm typing away and Bob is nowhere to be seen. I'm deep in concentration, writing about cat behavior, when I notice a letter on the screen that I didn't put there. It came from a sneaky paw attached to a striped body that just seemed to materialize in the form of a giant cat blob lying flat next to the computer. Does this sound familiar?

If you want things to change, you'll need to be a bit more observant and consistent than I've been with Bob. You'll need to catch him in the act of sneakiness.

As you're typing away, watch him out of the corner of your eye. As he approaches the table and looks up to jump, stomp your foot, then as he hesitates, say "Good" and not his head. Take a cat treat away from the



(Left) Proximity defines groups of related text (paragraphs and sections) and helps scanning. (Right) These groupings are discernible even without viewing the actual text

Pics from <https://www.nngroup.com/>

Proximity: forms

The principle of proximity is especially useful in forms

Page with a very long form - □ ×

Last name	
Email	
Re-type Email	
Password	
Re-type Password	

- In the example on the left, it's not entirely clear to which field each label corresponds
- Labels are equi-distant from fields

Proximity: forms

The principle of proximity is especially useful in forms

Page with a very long form

- □ ×

Last name	<input type="text"/>
Email	<input type="text"/>
Re-type Email	<input type="text"/>
Password	<input type="text"/>
Re-type Password	<input type="text"/>

- Using spacing better can help us make a more intuitive form!

13.7.4 Prossimità: moduli

Moduli lunghi con molti campi di input possono sembrare **opprimenti**. Raggruppare i campi correlati aiuta gli utenti a comprendere le informazioni da inserire. Lo spazio è un modo per raggruppare i campi correlati.

13.7.5 Prossimità: posizionamento delle etichette nei moduli

L'approccio più sicuro è posizionare le etichette sopra i campi di input. Se vuoi moduli più compatti, considera di posizionare le etichette a sinistra dei campi.

- Le etichette dovrebbero avere lunghezza simile e non essere troppo distanti dai campi
- Le etichette allineate a destra sono note per ostacolare la scansione

Attenzione a non raggruppare elementi non correlati! Può nascondere elementi non correlati, rendendoli meno visibili.

Proximity can backfire: example

The screenshot shows the California Employment Development Department (EDD) website. At the top, there's a navigation bar with the CA.GOV logo, the EDD State of California logo, and links for Skip to main content, Help, Benefit Programs Online, and Log Out. Below the header is a main menu with links for Home, Inbox, File a New Claim, Continue a Saved Draft, Manage My Profile, and My Claim History. The main content area is titled 'Employment Summary' and shows a five-step process: Personal Information, Initial Questions, Employment Information (which is circled in blue), Additional Information, and Certification. Below this, a message says 'You are currently on Step 3 Employment Information'. A section titled 'Section 4A - List of Employers' contains a message: 'Please click the "Add" button to add information about your last or current employer. You must add at least one employer.' A note below says 'No Results Found'. At the bottom right are buttons for Previous, Next, Add, Save as Draft, and Cancel.

CA.gov website. Pic from <https://www.nngroup.com/>

Proximity can backfire: example

- In the CA.gov website page, the «add» button, required to add employment information, is placed near unrelated buttons (move to the next step, save submission as draft, and cancel).
- When looking around the page, users may **only look at one item within a perceived grouping and use that to make a judgement about what the other items in that group must be.**

Proximity can backfire: example

CA.GOV Employment Development Department State of California Skip to main content Help | Benefit Programs Online | Log Out

MAIN MENU

- Home
- Inbox
- File a New Claim
- Continue a Saved Draft
- Manage My Profile
- My Claim History

Employment Summary

1 → 2 → 3 → 4 → 5

Personal Information Initial Questions Employment Information Additional Information Certification

You are currently on Step 3 Employment Information

Section 4A - List of Employers

Please click the "Add" button to add information about your last or current employer. You must add at least one employer.

No Results Found

Previous **Next** **Add** **Save as Draft** **Cancel**

Proximity can backfire: fixing the example

The screenshot shows the California Employment Development Department (EDD) website. At the top, there are links for CA.GOV, EDD, Employment Development Department, State of California, Skip to main content, Help, Benefit Programs Online, and Log Out. On the left, a main menu includes Home, Inbox, File a New Claim, Continue a Saved Draft, Manage My Profile, and My Claim History. The main content area is titled "Employment Summary". It shows a process flow with five steps: 1 Personal Information, 2 Initial Questions, 3 Employment Information, 4 Additional Information, and 5 Certification. Step 3 is highlighted with a larger circle. Below this, a message says "You are currently on Step 3 Employment Information". A section titled "Section 4A - List of Employers" contains the message "Please click the 'Add' button to add information about your last or current employer. You must add at least one employer." A "No Results Found" message is displayed. At the bottom, there are "Previous" and "Next" buttons, and a "Save as Draft" and "Cancel" button.

Principle of Connectedness

- Elements that are connected (or share a border) are perceived as related or part of the same group
- How many groups do you see in the shapes below?



Principle of Connectedness

- Elements that are connected (or share a border) are perceived as related or part of the same group
- How many groups do you see in the shapes below?



Principle of Connectedness

- Elements that are connected (or share a border) are perceived as related or part of the same group
- How many groups do you see in the shapes below?



Principle of Connectedness

- Elements that are connected (or share a border) are perceived as related or part of the same group
- How many groups do you see in the shapes below?



Principle of Connectedness

- Elements that are connected (or share a border) are perceived as related or part of the same group
- How many groups do you see in the shapes below?



Principle of Connectedness

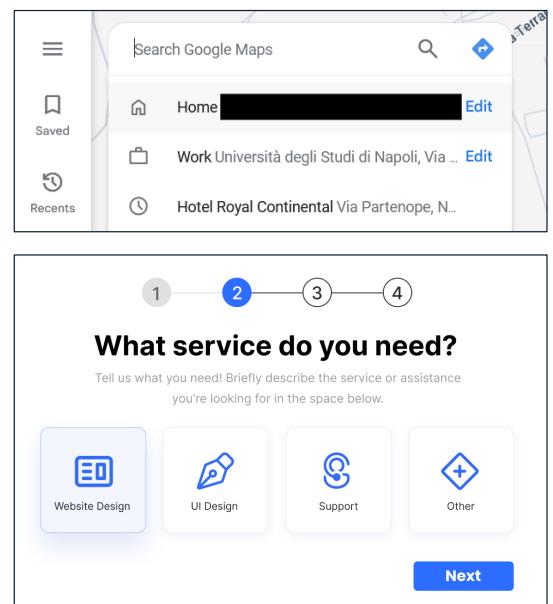
- Elements that are connected (or share a border) are perceived as related or part of the same group
- How many groups do you see in the shapes below?



- Connectedness **overrides** proximity and similarity

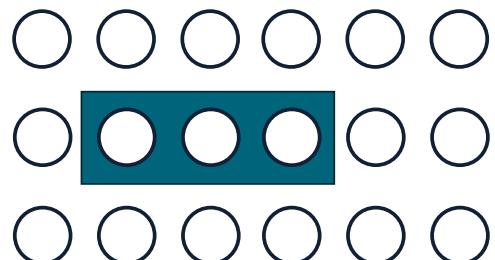
Principle of Connectedness in UI Design

- In Google Maps (web version) the search box is connected (share a border) to recent queries and saved locations
 - This suggests that those features are related
- Long forms can also be split in multiple phases or steps. The steps in the indicator on top of the figure (from fluentforms.com) are connected
 - This conveys the fact that these step belongs to the same, larger process



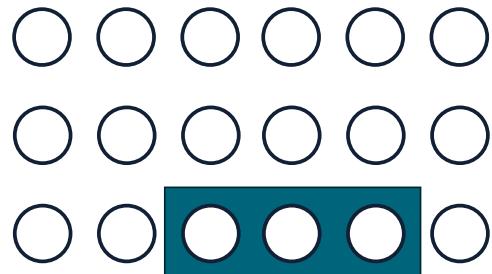
Principle of Common Region

- Items within a boundary are perceived as a group and assumed to share common characteristics or functionality



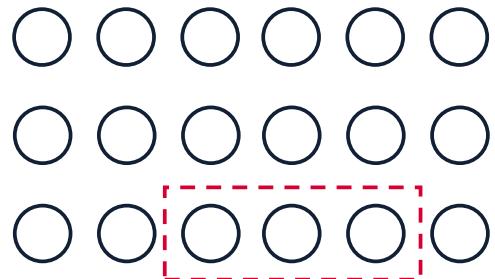
Principle of Common Region

- Items within a boundary are perceived as a group and assumed to share common characteristics or functionality

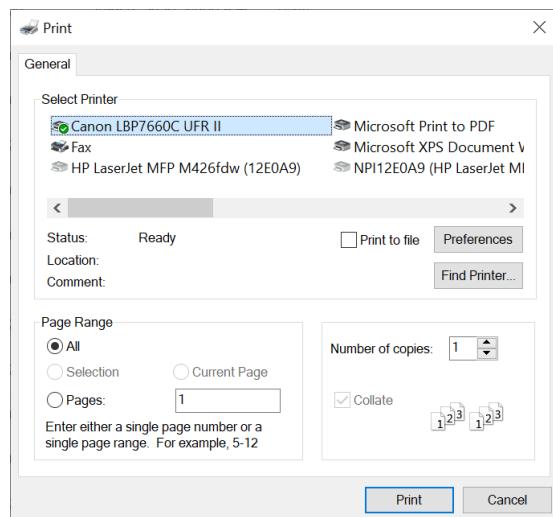


Principle of Common Region

- Items within a boundary are perceived as a group and assumed to share common characteristics or functionality



Principle of Common Region: examples



Printing dialog on Windows 11.
Pic from <https://www.nngroup.com>

Principle of Common Region: examples

2023

GUI Testing of Android Applications: Investigating the Impact of the Number of Testers on Different Exploratory Testing Strategies

Joint work with S. DI MARTINO, A. FASOLINO, and P. TRAMONTANA.

Journal of Software: Evolution and Process.

2021

Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing

Joint work with S. DI MARTINO, A. FASOLINO, and P. TRAMONTANA.

Software Testing, Verification and Reliability.

[BACK TO TOP](#) [ABOUT THIS WEBSITE](#) [PRIVACY POLICY](#) [CONTACTS](#)

Designed and developed with ❤ by Luigi L. Starace

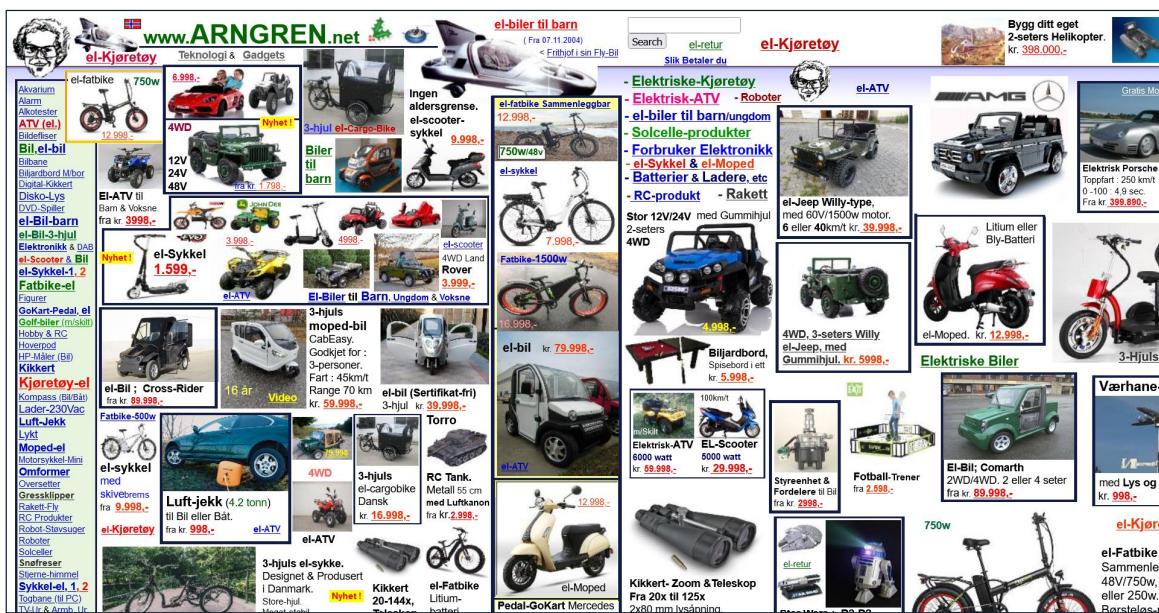
[R²](#) [✉](#) [✉](#) [✉](#) [✉](#) [✉](#) [✉](#)

Footer in the teacher's personal website: <https://luistar.github.io>

Visual Hierarchy in UI Design

- Have you ever seen a website or an app presenting a screen full of information, and you don't even know where to start looking?
- When that happens, it's likely that the layout is missing a clear **visual hierarchy**
- **Visual hierarchy** (of a 2D layout) refers to the organization of the design elements on the screen/page so that the eye is guided to consume each design element in the order of intended importance.

Lack of a Visual Hierarchy on arngren.net



Lack of a Visual Hierarchy (SPSB website)



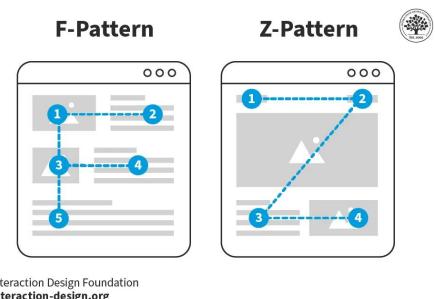
<https://www.scuolapsb.unina.it/programmi-di-internazionalizzazione/>

Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Software Engineering Course - Lecture 15 - Colors, Typography and Gestalt

84

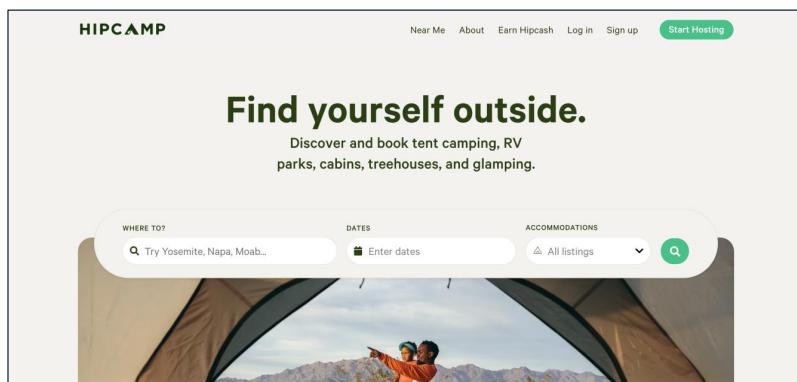
Creating a Visual Hierarchy

- You should use what you learned about colors, typography, and Gestalt principles to ensure that your designs have a clear visual hierarchy
- Designs should guide users so that they consume contents in the desired order
- Keep in mind that western users typically scan a UI using a **F- and Z-pattern**
 - You can reinforce these natural patterns
- **Scale, colors and groupings** are powerful tools to create a visual hierarchy



Creating a Visual Hierarchy: Scale

- Users pay more attention to big things than to small things
- More important elements should be larger than less important ones
- Users will notice larger elements first

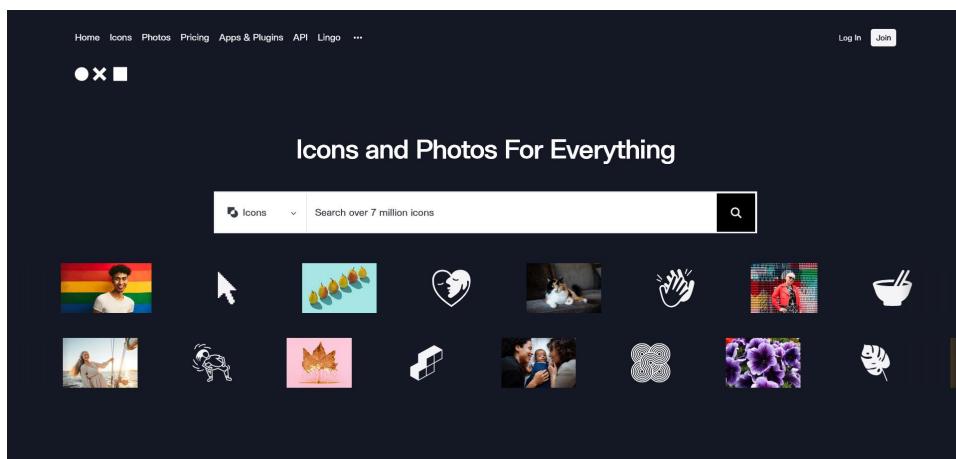


Hipcamp.com: *The visual hierarchy is communicated through font size. The eye is drawn first to the “Find yourself outside” text due to its large, bolded size. This text gives you a general idea of what you can do on this website.*

From <https://www.nngroup.com/>

Creating a Visual Hierarchy: Color / Contrast

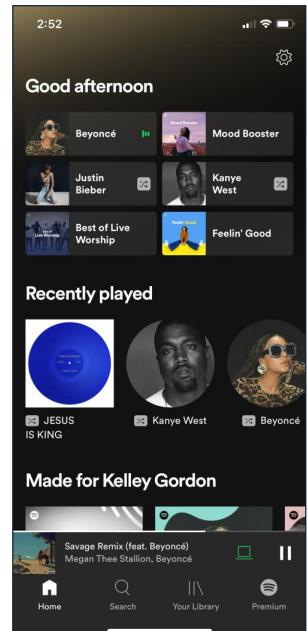
- Color and contrast is a good way to make some elements appear in advance while others recede



In [theNounProject.com](http://thenounproject.com), attention is drawn to the search field both because of its size and because of its high contrast w.r.t. the dark background.

Creating a Visual Hierarchy: Grouping

- Implicit and explicit groupings help us see the bones or the structure of a layout and allow us to direct attention to those areas of the screen that are likely to be relevant to our goal
- Gestalt principles (proximity, similarity, common region) can help convey groupings
- In the Spotify app (pic from nngroup.com), we immediately see 3 different groupings.



Capitolo 14

Lezione 17: Linee guida e principi nell'Interazione Uomo-Macchina

14.1 Linee guida e principi nell'HCI

Linee guida e principi per un buon design delle interfacce sono stati sviluppati nel corso degli anni. Queste linee guida sono applicabili alla maggior parte dei sistemi interattivi. Derivano dall'esperienza e sono state affinate nel tempo. Non pretendono di essere complete o universali. Richiedono validazione e adattamento per domini di design specifici. Sono comunque utili per studenti e professionisti.

Le linee guida e i principi possono essere utili:

- Per guidare la fase di progettazione
- Per valutare una UI e trovare problemi di usabilità (vedremo tra qualche lezione)

Esistono sovrapposizioni tra i diversi insiemi di linee guida/principi, come temi ricorrenti: prevenire errori, minimizzare il carico di memoria, ...

14.2 Le otto regole d'oro di Shneiderman

1. Cerca la coerenza
2. Punta all'usabilità universale
3. Offri feedback informativo
4. Progetta dialoghi che portino a una conclusione
5. Previeni gli errori
6. Permetti la facile reversibilità delle azioni
7. Mantieni l'utente al controllo
8. Riduci il carico di memoria a breve termine

14.2.1 Le dieci euristiche di usabilità Nielsen-Molich

1. Dialogo semplice e naturale
2. Parla la lingua dell'utente
3. Minimizza il carico di memoria dell'utente
4. Coerenza
5. Feedback
6. Uscite chiaramente indicate
7. Scorciatoie
8. Buoni messaggi di errore
9. Prevenzione degli errori
10. Aiuto e documentazione

14.2.2 Dialogo semplice e naturale

La UI dovrebbe essere il più semplice possibile (ma non troppo!).

- Meno è meglio: ogni funzionalità/informazione aggiuntiva è qualcosa in più da imparare, fraintendere o cercare
- Gli utenti inesperti possono sentirsi sopraffatti da troppe informazioni
- Legge di Hick!

La UI dovrebbe corrispondere il più naturalmente possibile al compito dell'utente (mapping e metafore!).

- La versione digitale di un modulo è organizzata come quella cartacea
- Un'app bussola funziona come una bussola reale

14.3 Parla la lingua dell'utente

Il dialogo dovrebbe essere espresso con parole, frasi e concetti familiari all'utente, non con termini orientati al sistema (design centrato sull'uomo!). I dialoghi dovrebbero essere nella lingua madre dell'utente (localizzazione), non solo nel testo ma anche negli elementi non verbali come le icone! Attenzione alle parole che usi.

- Se progettisti per il pubblico generale, usa parole che tutti comprendono, con il loro significato standard
- Se progettisti per un gruppo con una terminologia specifica, usa i termini specializzati

14.3.1 Minimizza il carico di memoria dell'utente

I computer ricordano perfettamente, la memoria di lavoro umana molto meno! (Ricordi MHP?) La UI dovrebbe sollevare l'utente dal peso della memoria il più possibile. Come?

- Il riconoscimento è meglio del richiamo
- Quando chiedi input agli utenti, descrivi il formato richiesto e fornisci un esempio. Indica esplicitamente i valori ammessi (se ci sono)

14.3.2 Minimizza il carico di memoria dell'utente: esempi

Gli utenti devono inserire il codice a due lettere di una provincia italiana.

- Accettare l'input tramite campo di testo richiede agli utenti di ricordare il codice della provincia desiderata
 - Qual è il codice per Lecce?
 - E per Lecco?
- Usare una lista a discesa con nomi e codici di tutte le province solleva l'utente dal dover ricordare i codici delle 110 province italiane!

14.3.3 Coerenza

Per essere usabile, un sistema dovrebbe mostrare coerenza interna ed esterna. **Coerenza interna** (all'interno del prodotto o della famiglia di prodotti):

- Sequenze di azioni coerenti dovrebbero essere richieste in situazioni simili: cancellare un cliente e cancellare un fornitore dovrebbero richiedere una sequenza simile
- Le stesse informazioni dovrebbero essere presentate nello stesso modo e nella stessa posizione su tutte le schermate

Coerenza esterna (con le convenzioni consolidate)

Internal (In)consistency: Examples

In the [docenti.unina](#) platform, the (sic) **Next Results** control changes position

- In the first page (top figure), it's the leftmost control
- In the subsequent pages (bottom figure), it's shifted to the right

Class schedules
Registration lessons
Register of lessons
Course Materials
Registration groups / test

7/3/24 Sessione di Discussione Progetti di Tecnologie Web di Luglio 2024 Updates
Published on in TECNOLOGIE WEB

6/26/24 Sessione di Discussione Progetti di Tecnologie Web di Giugno 2024 Updates
Published on in TECNOLOGIE WEB

[Next Results](#)

Results 1-5 of 19

Class schedules
Registration lessons
Register of lessons
Course Materials
Registration groups / test

6/14/24 Annullamento ricevimento studenti del 18 giugno 2024
Published on in AVVISI DI CARATTERE GENERALE

6/14/24 Risultati Seconda Prova Intercorso del 13 giugno 2024
Published on in TECNOLOGIE WEB

[Previous Results](#) [Next Results](#)

Results 6-10 of 19

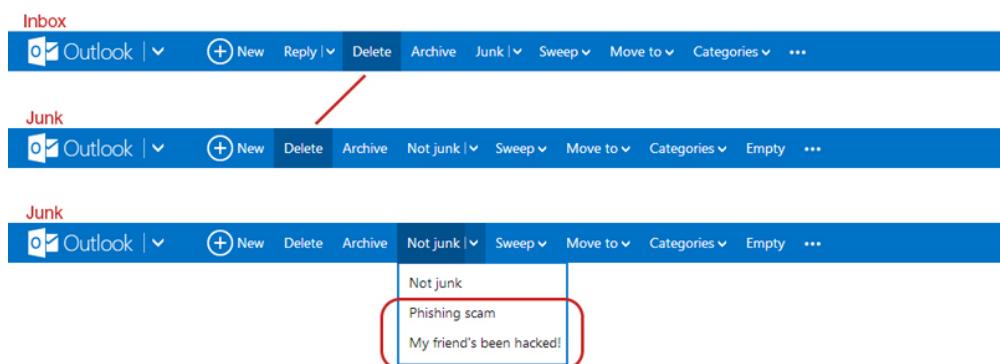
Internal (In)consistency: Examples

The figure consists of three side-by-side screenshots from the Sutter Health eZ Arrival mobile application. Each screenshot shows a different step in the appointment checkin process.

- Screenshot 1:** Shows the "Responsibility for Payment" screen. It displays a list of names under "Guarantor" and asks if the person on file will pay for costs. Buttons for "Yes" and "No" are shown. Below this, it asks if insurance should be used to pay for the appointment, with options "Use Insurance" and "Do not bill insurance". At the bottom are "NEXT" and "FINISH LATER" buttons.
- Screenshot 2:** Shows the "Send Message to Care Team" screen. It asks the clinician to verify changes before sending a non-urgent message. A note says "Your clinician must verify any changes before health patient message to verify when your medical record is request is appropriate." It includes a timestamp "Added 11/3/2010" and a "Learn more" link. Below this is a section titled "Health Issues You've Asked to be Added". At the bottom are "BACK", "NEXT", and "FINISH LATER" buttons.
- Screenshot 3:** Shows the same "Send Message to Care Team" screen as Screenshot 2, but with a checked checkbox labeled "This information is correct". The "FINISH LATER" button is highlighted with a red border.

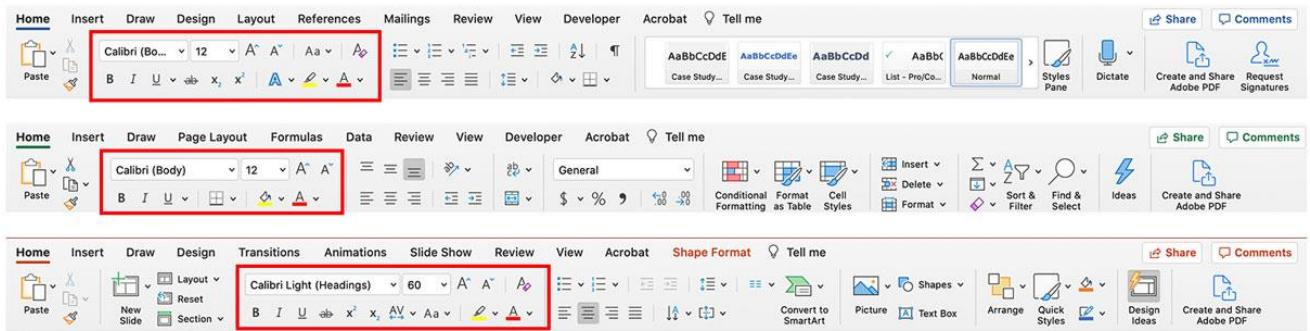
Sutter Health appointment checkin process, from <https://www.nngroup.com/articles/consistency-and-standards/>

Internal (In)consistency: Examples



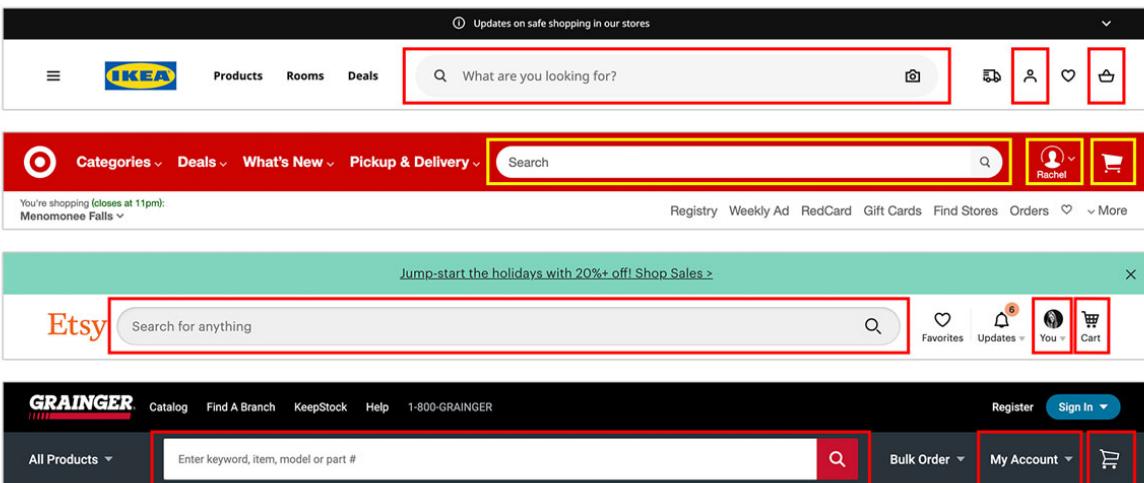
Outlook.com new UI Inconsistencies, from <https://www.elwinlee.com/blog/outlook-coms-ui-inconsistency/>

Internal Consistency



Internal consistency in the Microsoft Office Suite. From top to bottom: Word, Excel, PowerPoint
<https://www.nngroup.com/articles/consistency-and-standards/>

External Consistency



Navigation in different e-commerce websites.

<https://www.nngroup.com/articles/consistency-and-standards/>

14.4 Feedback

Il sistema dovrebbe informare continuamente l'utente su cosa sta facendo e come interpreta l'input dell'utente.

- Non solo quando si verificano errori
- Il feedback positivo è importante quanto quello negativo
- Quando possibile, dai feedback anche in caso di errori di sistema
- Il peggior feedback possibile è nessun feedback!
- Il feedback non dovrebbe essere troppo astratto o generico

14.4.1 Persistenza del feedback

Tipi diversi di feedback possono richiedere diversi livelli di persistenza.

- Alcuni feedback sono rilevanti solo per la durata di un fenomeno o sono semplici conferme di un'operazione
- Possono scomparire automaticamente (es: messaggi toast)
- Altri (soprattutto avvisi o errori) possono richiedere un riconoscimento esplicito da parte dell'utente
- Altri ancora possono richiedere alta persistenza e diventare parte permanente della UI

14.4.2 Feedback e tempi di risposta del sistema

Il feedback è cruciale quando i sistemi hanno tempi di risposta lunghi.

1. Meno di 0,1 secondi: le reazioni sono percepite come istantanee
 - Nessun feedback richiesto, tranne mostrare il risultato o confermare l'esito
2. Meno di 1 secondo: il flusso di pensiero dell'utente resta ininterrotto
 - Nessun feedback speciale richiesto (ma non si ha la sensazione di reazione istantanea)
3. 10 secondi: limite per mantenere l'attenzione dell'utente
 - Il feedback è cruciale per ritardi superiori a 10 secondi
 - Fornisci una stima di quando il compito sarà completato (gli utenti vorranno fare altro mentre aspettano)
 - Aggiorna frequentemente l'indicatore di progresso

14.4.3 Labor Perception Bias

Il Labor Perception Bias: le persone si fidano e apprezzano di più ciò che percepiscono come frutto di lavoro. Tutti odiano aspettare. Ma se le aspettative sono alte (es: gestione di denaro, backup o migrazione di dati importanti, analisi e report...), possono diventare scettici se il tempo di attesa è troppo breve!

- Aggiungere una schermata di lavoro subito dopo un'azione chiave può migliorare l'esperienza utente
- Talvolta vengono aggiunti dai designer dei "inganni benevoli" (es: tempi di caricamento finti): link all'articolo

14.4.4 Uscite chiaramente indicate e reversibilità delle azioni

Gli utenti vogliono sentirsi in controllo dell'interazione. Gli utenti commetteranno comunque errori durante l'uso del sistema. Il sistema dovrebbe offrire una via d'uscita semplice dalla maggior parte delle situazioni.

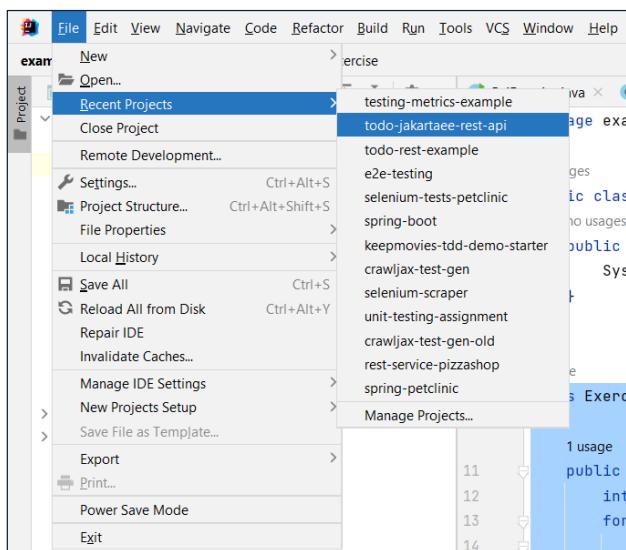
- Se il sistema non riesce a completare l'azione entro 10 secondi, l'utente dovrebbe poter interrompere l'operazione e annullare l'azione
- In operazioni con effetti collaterali, le uscite possono essere offerte tramite una funzione di «Undo» che riporta il sistema allo stato precedente

14.4.5 Scorciatoie

In generale, l'uso di una UI dovrebbe richiedere la conoscenza di poche regole. Gli utenti esperti dovrebbero poter eseguire rapidamente azioni frequenti tramite scorciatoie e acceleratori.

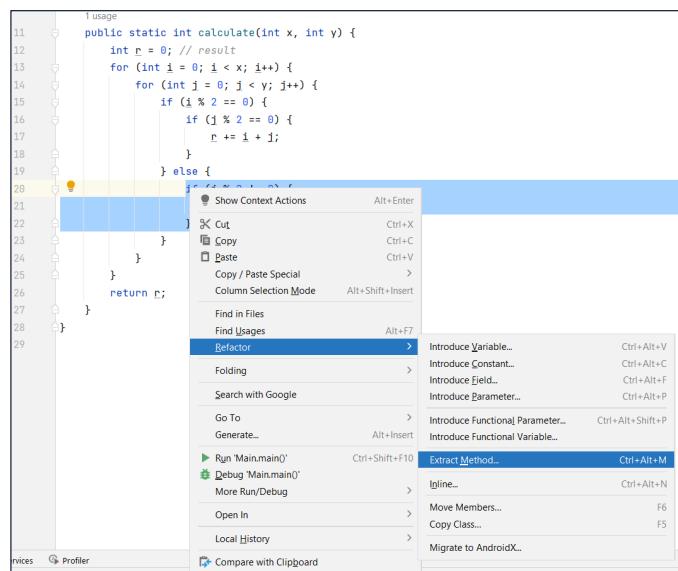
- Tasti funzione o combinazioni che eseguono un intero comando con una pressione
- Doppio clic su un oggetto per eseguire l'azione più comune su quell'oggetto
- Pulsanti specifici per accedere direttamente a funzioni importanti dove sono più necessarie
- Riutilizzo della cronologia delle interazioni (ripetere rapidamente gli stessi comandi)
- Fornire valori predefiniti nei moduli, quando possibile

Shortcuts



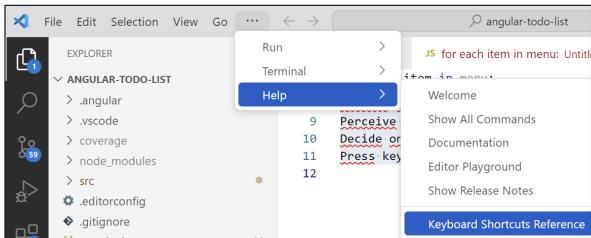
Recent Projects accelerator in IntelliJ IDEA

Shortcuts



Keyboard shortcuts also shown in context menus in IntelliJ IDEA

Shortcuts



The screenshot shows the 'Keyboard shortcuts for Windows' page from Visual Studio Code. The page is titled 'Visual Studio Code' and 'Keyboard shortcuts for Windows'. It is divided into sections: 'General', 'Basic editing', and 'Multi-cursor and selection'. Each section lists keyboard shortcuts with their descriptions. For example, in the 'General' section, 'Ctrl+Shift+P, F1' is listed as 'Show Command Palette'. In the 'Basic editing' section, 'Ctrl+X' is listed as 'Cut line (empty selection)'. In the 'Multi-cursor and selection' section, 'Alt+Click' is listed as 'Insert cursor'.

General	Description
Ctrl+Shift+P, F1	Show Command Palette
Ctrl+P	Quick Open, Go to File...
Ctrl+Shift+N	New window-instance
Ctrl+Shift+W	Close window-instance
Ctrl+,	User Settings
Ctrl+K Ctrl+S	Keyboard Shortcuts

Basic editing	Description
Ctrl+X	Cut line (empty selection)
Ctrl+C	Copy line (empty selection)
Alt+↑ / ↓	Move line up/down
Shift+Alt+↑ / ↓	Copy line up/down
Ctrl+Shift+K	Delete line
Ctrl+Enter	Insert line below
Ctrl+Shift+Enter	Insert line above
Ctrl+Shift+`	Jump to matching bracket
Ctrl+] / [Indent/outdent line
Home / End	Go to beginning/end of line
Ctrl+Home	Go to beginning of file
Ctrl+End	Go to end of file

Multi-cursor and selection	Description
Alt+Click	Insert cursor
Ctrl+Alt+↑ / ↓	Insert cursor above / below
Ctrl+U	Undo last cursor operation
Shift+Alt+I	Insert cursor at end of each line selected
Ctrl+L	Select current line
Ctrl+Shift+L	Select all occurrences of current selection
Ctrl+F2	Select all occurrences of current word
Shift+Alt+→	Expand selection
Shift+Alt+←	Shrink selection
Shift+Alt+drag mouse	Column (box) selection
Ctrl+Shift+Alt+↑ / ↓	Column (box) selection
Ctrl+Shift+Alt+PgUp/PgDn	Column (box) selection page up/down

Keyboard Shortcuts reference in VS Code

Shortcuts Guidelines

- Keyboard shortcuts should also be **learnable** and **memorable**

Introduce <u>Variable</u> ...	Ctrl+Alt+V
Introduce <u>Constant</u> ...	Ctrl+Alt+C
Introduce <u>Field</u> ...	Ctrl+Alt+F
Introduce <u>Parameter</u> ...	Ctrl+Alt+P

- You can't just use any unique combinations of keys!
- If users need to check the reference everytime they want to use a shortcut, then it is no shortcut at all!

14.5 Messaggi di errore

I messaggi di errore sono fondamentali per l'usabilità.

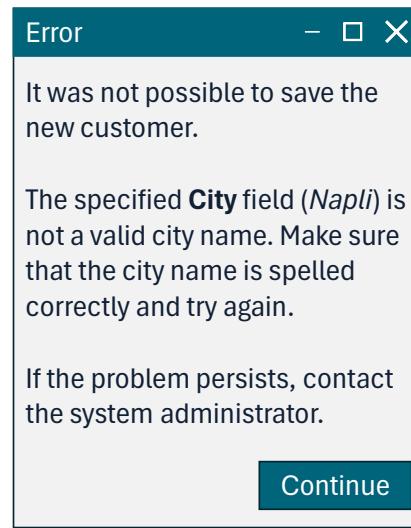
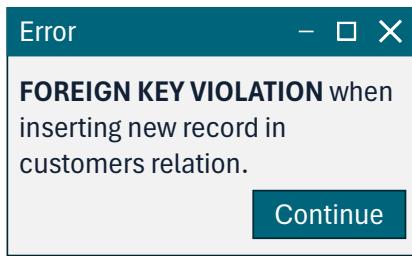
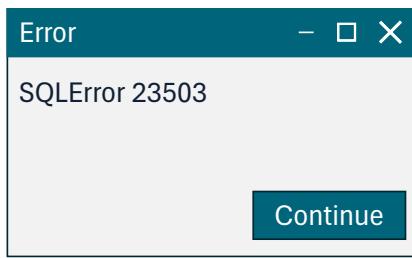
- Rappresentano situazioni in cui gli utenti sono in difficoltà e potrebbero non riuscire a raggiungere i propri obiettivi
- Offrono opportunità per aiutare gli utenti a comprendere meglio il sistema. Gli utenti sono generalmente più motivati a prestare attenzione al contenuto dei messaggi di errore

14.5.1 Buoni messaggi di errore

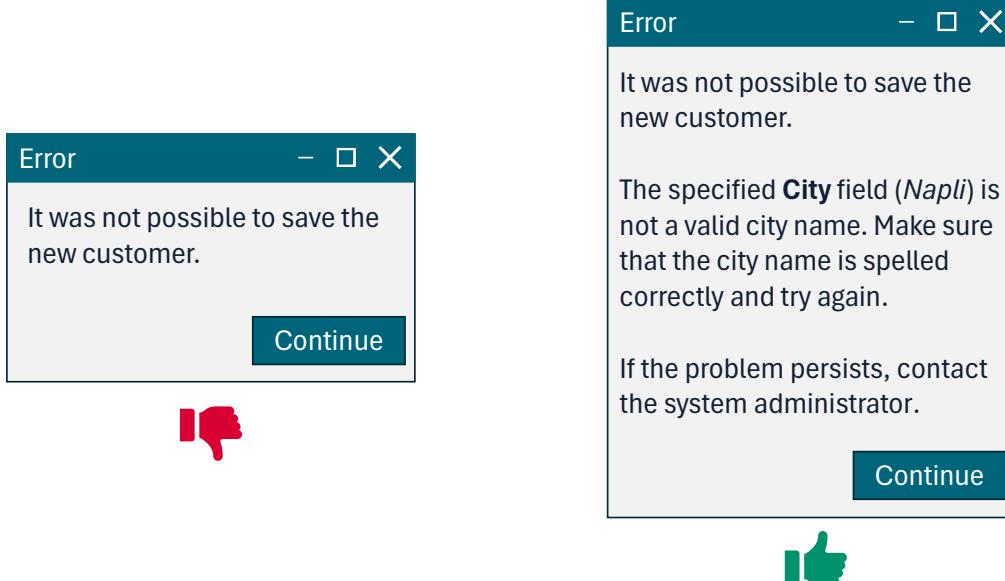
Secondo Shneiderman, i messaggi di errore dovrebbero seguire quattro regole:

1. Devono essere formulati in linguaggio chiaro ed evitare codici oscuri
2. Devono essere precisi, non vaghi o genericci
3. Devono aiutare costruttivamente l'utente a risolvere il problema
4. Devono essere cortesi e non intimidire o colpevolizzare l'utente

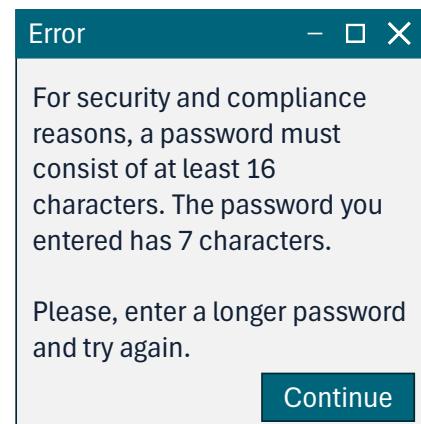
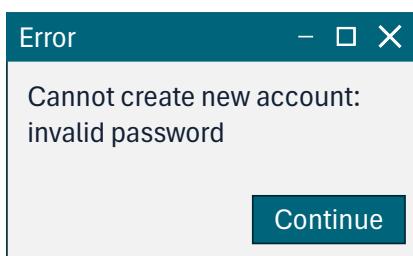
Good Error Messages: Clear Language



Good Error Message: Precise and Not General



Good Error Messages: Be Constructive



14.5.2 Buoni messaggi di errore: sii cortese

Non intimidire o colpevolizzare l'utente.

- Gli utenti già si sentono frustrati quando non riescono a raggiungere i propri obiettivi, non serve peggiorare la situazione!
- Evita termini offensivi come: «AZIONE UTENTE ILLEGALE!», «LAVORO ANNULLATO», «PROCESSO TERMINATO», «ERRORE FATALE»
- Cerca di formulare il messaggio di errore in modo da suggerire che il problema è del sistema (in fondo, una buona UI avrebbe potuto prevenire quell'errore!)

14.5.3 Buoni messaggi di errore: livelli multipli

I messaggi di errore sono utili sia per gli utenti che per il personale tecnico.

- Gli utenti normalmente non comprendono dettagli tecnici (es: codici di errore o stack trace), ma il personale tecnico può averne bisogno per la risoluzione dei problemi
- I messaggi di errore possono dover contenere entrambi i livelli di informazione

Spesso è preferibile separare le viste dei diversi livelli:

- Gli utenti normali non sono intimiditi da messaggi strani
- Il personale tecnico può accedere alle informazioni di troubleshooting
- Le finestre di errore possono includere link a siti di supporto

14.5.4 Prevenire gli errori

Ancora meglio che avere buoni messaggi di errore, è evitare gli errori! Cerca di non mettere gli utenti in situazioni soggette a errore.

- Se chiedi all'utente di digitare il nome di una città, c'è il rischio di errori di ortografia
- Se l'utente deve inserire un intervallo di date nel futuro, formattato in modo specifico, c'è il rischio che non formatti correttamente la data o inserisca una data non valida

Progetta la UI per evitare (o minimizzare) questi errori: non è solo positivo per l'usabilità, ma significa anche meno lavoro per formalizzare i casi d'uso e meno codice per gestire situazioni di errore!

14.6 Tipi di errori

Secondo Don Norman, esistono due categorie di errori:

1. **Slips:** l'utente intende eseguire un'azione, ma ne compie un'altra
 - Premere il tasto «Invio» invece di «Backspace»
 - Cliccare sul pulsante «Minimizza» invece di «Massimizza»
2. **Mistakes:** l'utente forma obiettivi non appropriati per il problema/compito corrente
 - Il responsabile di un sito e-commerce vuole eliminare tutti gli articoli di una certa categoria. Crede che eliminando la categoria verranno eliminati anche gli articoli associati. In realtà, gli articoli vengono spostati implicitamente nella categoria «Altro».

14.6.1 Tipi di errori: Slips

Se gli utenti formano obiettivi corretti, ma sbagliano l'esecuzione, hanno commesso uno *slip*. Gli slip derivano tipicamente da comportamenti automatici. Sono più frequenti nei comportamenti esperti (gli utenti prestano più attenzione quando stanno ancora imparando). Gli slip sono spesso «errori di cattura»: quando due sequenze di azioni hanno un prefisso comune, e una viene usata molto più spesso dell'altra, gli utenti finiscono inconsciamente per seguire la sequenza più frequente, anche se volevano eseguire quella meno frequente. Gli slip sono il motivo per cui permettere la facile reversibilità delle azioni è generalmente preferibile rispetto a fare affidamento solo su una finestra di conferma.

14.6.2 Tipi di errori: Mistakes

- I mistake sono molto più critici
- Spesso derivano dal fatto che l'utente ha formato un modello mentale errato del sistema
- Possono essere molto più difficili da rilevare (e quindi più pericolosi!)
- Ripensa all'esempio precedente:
 - Il responsabile di un sito e-commerce vuole eliminare tutti gli articoli di una certa categoria. Crede che eliminando la categoria verranno eliminati anche gli articoli associati. In realtà, gli articoli vengono spostati implicitamente nella categoria «Altro».
 - Quando se ne accorgerà?

14.6.3 Aiuto e documentazione

Idealmente, un sistema dovrebbe essere così facile da usare da non richiedere aiuto o documentazione aggiuntiva. Questo obiettivo purtroppo non è sempre raggiungibile. A parte i sistemi veramente «walk-up-and-use», la maggior parte delle UI ha abbastanza funzioni da giustificare un manuale e possibilmente un sistema di aiuto. Un manuale può essere utile anche agli utenti regolari per acquisire maggiore competenza e aumentare la produttività. Nota: avere un buon manuale e sistema di aiuto non riduce i requisiti di usabilità! «È tutto spiegato nel manuale!» non è una buona scusa per una UI poco usabile!

14.6.4 La verità fondamentale sui manuali utente

Gli utenti non leggono i manuali utente. Preferiscono dedicare tempo ad attività che li fanno sentire produttivi. Tipicamente iniziano a usare il sistema senza aver letto le istruzioni. Corollario:

- Se gli utenti vogliono leggere il manuale, probabilmente sono in difficoltà e hanno bisogno di aiuto immediato
- I manuali online con ricerca orientata ai compiti e funzioni di ricerca personalizzata sono particolarmente utili in questi casi

14.6.5 Puntare all'usabilità universale

Cerca l'usabilità per tutti! Bisogna considerare: Differenze tra utenti principianti ed esperti, fasce d'età, disabilità, variazioni internazionali. Progettare per tutti non significa ottenere un prodotto meno efficace. Spesso molte categorie di utenti beneficiano di accorgimenti pensati per una categoria specifica. Pensa alle rampe sui marciapiedi! (curb cut effect)

14.6.6 Progettare dialoghi che portino a una conclusione

Le sequenze di azioni dovrebbero essere organizzate in gruppi con inizio, sviluppo e fine. Un feedback informativo al termine di un gruppo dovrebbe dare all'utente soddisfazione, senso di sollievo, e indicare che può prepararsi al prossimo gruppo di azioni. Ad esempio, i siti e-commerce guidano i clienti attraverso una serie di passaggi chiari:

- Aggiunta degli articoli al carrello
- Specifica del metodo di pagamento, indirizzo di consegna, ecc.
- Pagamento

14.7 Letture e riferimenti

1. Shneiderman, B., & Plaisant, C. (2010). *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education.
2. Molich, R., & Nielsen, J. (1990). Improving a human-computer dialogue. *Communications of the ACM*, 33(3), 338-348.
<https://dl.acm.org/doi/10.1145/77481.77486>
3. Holcomb, R., & Tharp, A. L. (1991). What users say about software usability. *International Journal of Human-Computer Interaction*, 3(1), 49-78.
<https://doi.org/10.1080/10447319109525996>

4. Polson, P. G., & Lewis, C. H. (1990). Theory-based design for easily learned interfaces. *Human–Computer Interaction*, 5(2-3), 191-220.
<https://doi.org/10.1080/07370024.1990.9667154>
5. Carroll, J. M., & Rosson, M. B. (1992). Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Transactions on Information Systems (TOIS)*, 10(2), 181-212.
<https://dl.acm.org/doi/abs/10.1145/146802.146834>
6. Nielsen, J. (1994, April). Enhancing the explanatory power of usability heuristics. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems (pp. 152-158).
<https://dl.acm.org/doi/10.1145/191666.191729>

Architetture

- L'architettura di un sistema software viene definita nella prima fase di System Design (progettazione architetturale)
- Lo scopo primario è la scomposizione del sistema in sottosistemi:
 - la realizzazione di più componenti distinti è meno complessa della realizzazione di un sistema come monolito.
 - Permette di parallelizzare lo sviluppo
 - Favorisce modificabilità, riusabilità, portabilità, etc...

Architetture

- Definire un'architettura significa mappare funzionalità su moduli
 - Es: Modulo di interfaccia utente, modulo di accesso a db, modulo di gestione della sicurezza, etc...
- Anche la definizione delle architetture deve seguire i concetti di Alta Coesione e Basso Accoppiamento
 - Ogni sottosezione dell'architettura dovrà fornire servizi altamente relativi tra loro, cercando di limitare il numero di altri moduli con cui è legato

Definizione dell'architettura

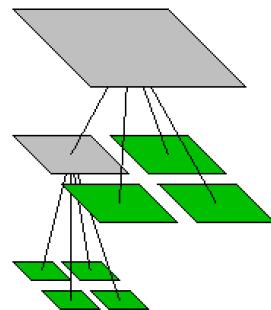
- La definizione dell'architettura viene di solito fatta da due punti di vista diversi, che portano alla soluzione finale:
 - Identificazione e relazione dei sottosistemi
 - Definizione politiche di controllo
- Entrambe le scelte sono cruciali:
 - è difficile modificarle quando lo sviluppo è partito, poiché molte interfacce dei sottosistemi dovrebbero essere cambiate.

Identificazione sottosistemi



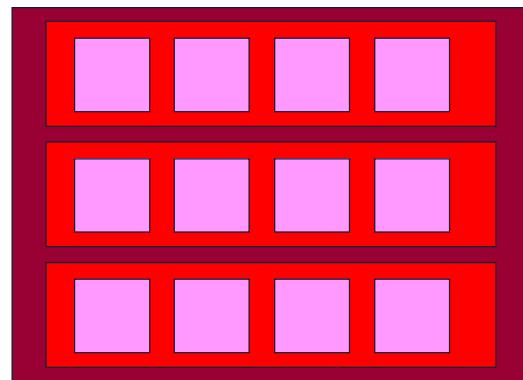
Design Principle: Divide and conquer

- Trying to deal with something big all at once is harder than dealing with a set of smaller things
 - Each individual component is smaller, and therefore easier to understand
 - Parts can be replaced or changed without having to replace or extensively change other parts.
 - Separate people can work on separate parts
 - An individual software engineer can specialize

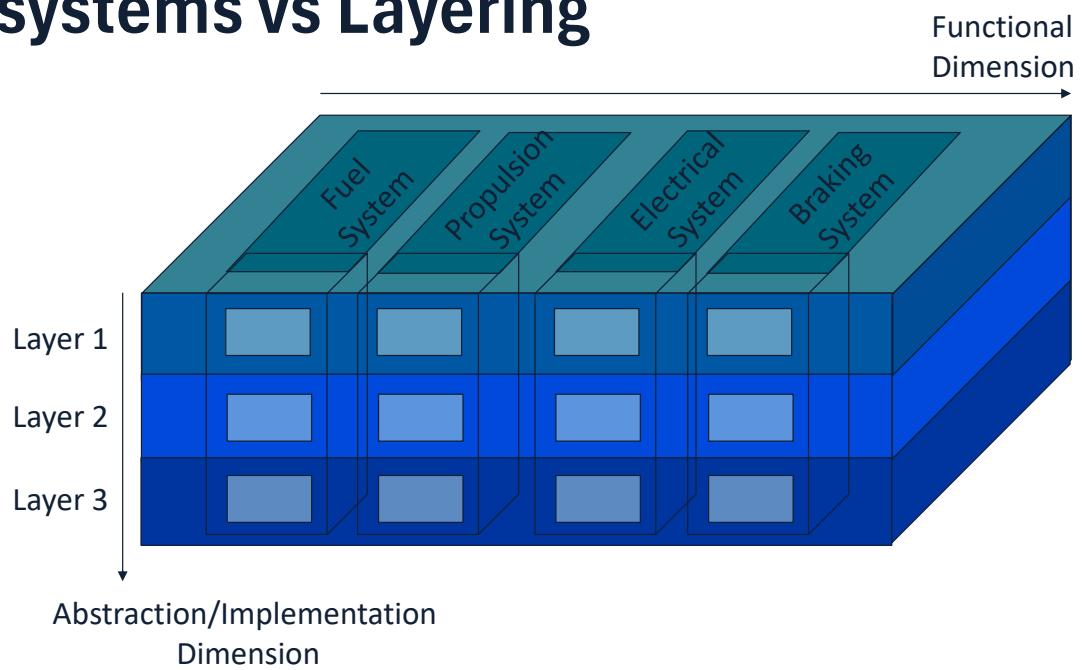


Ways of dividing a software system

- A system is divided up into
 - Layers & subsystems
 - A subsystem can be divided up into one or more packages
 - A package is divided up into classes
 - A class is divided up into methods



Subsystems vs Layering

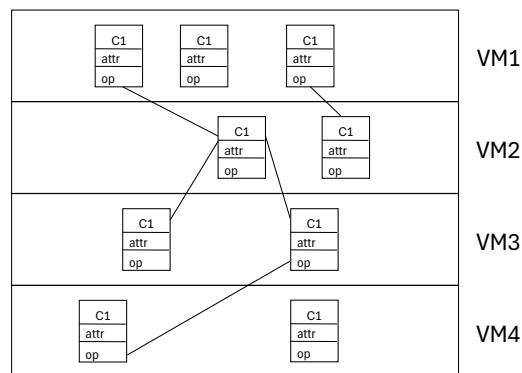


Layers

- Una decomposizione gerarchica di un sistema consiste di un insieme ordinato di layer (strati).
 - Un layer è un raggruppamento di sottosistemi che forniscono servizi correlati, eventualmente realizzati utilizzando servizi di altri layer.
 - Un layer può dipendere solo dai layer di livello più basso
 - Un layer non ha conoscenza dei layer dei livelli più alti
- **Architettura chiusa:** ogni layer può accedere solo al layer immediatamente sotto di esso
- **Architettura aperta:** un layer può anche accedere ai layer di livello più basso

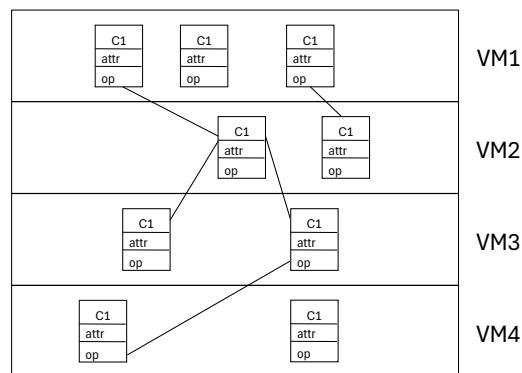
Macchina Virtuale (Dijkstra, 1965)

- Prima formalizzazione di architettura a layers
- Un sistema dovrebbe essere sviluppato da un insieme di macchine virtuali, ognuna costruita in termini di quelle al di sotto di essa.

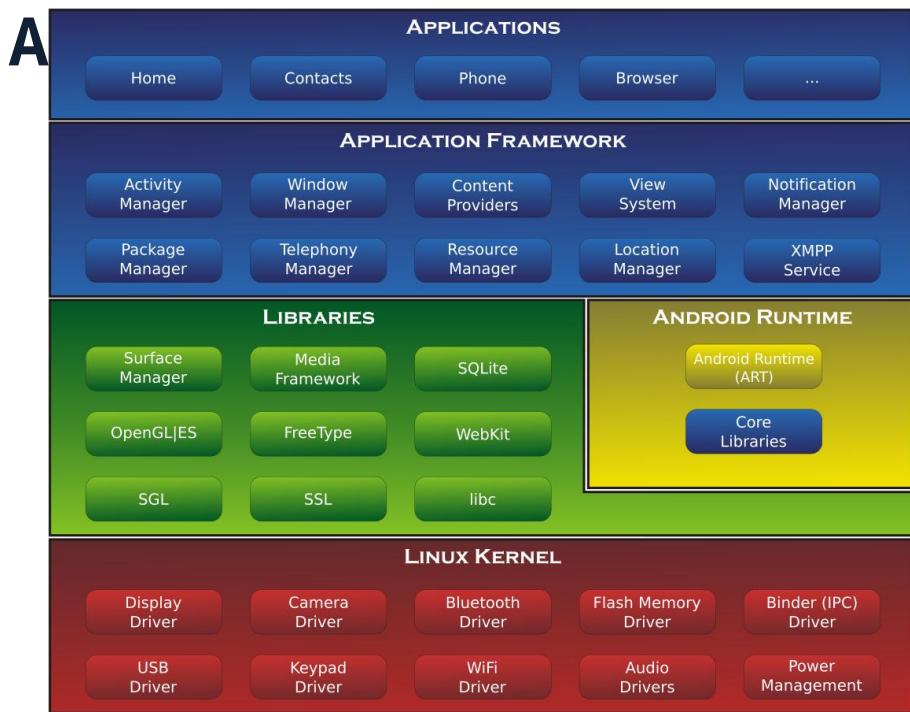


Architettura Chiusa

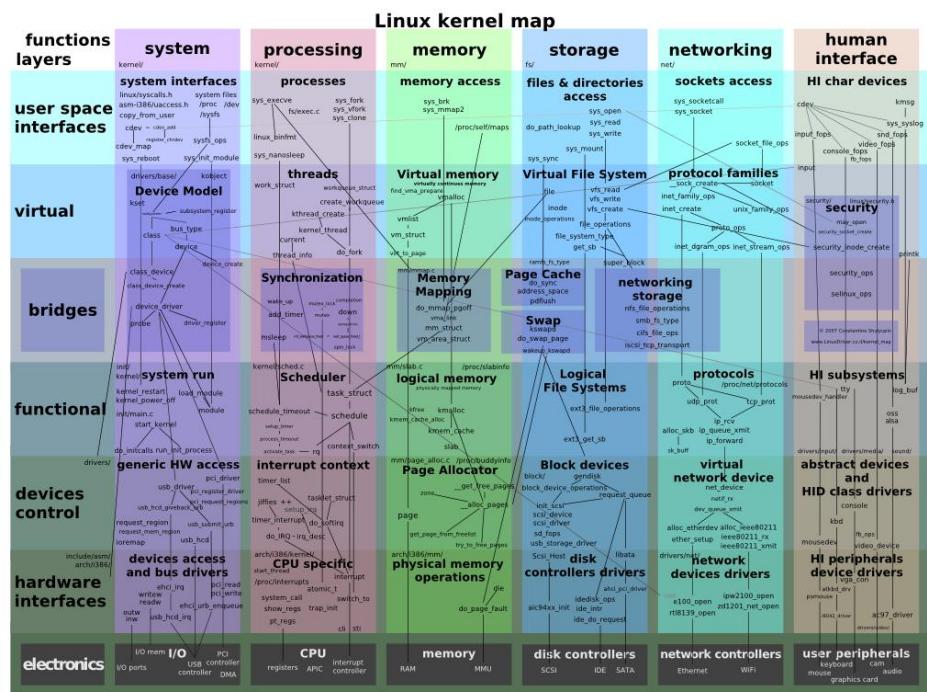
- Una macchina virtuale può solo chiamare le operazioni dello strato sottostante
- Design goal: alta manutenibilità e portabilità



Esempio di Architettura chiusa:

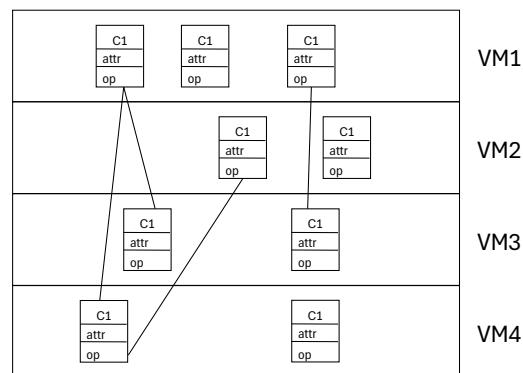


Consistency in two dimensions

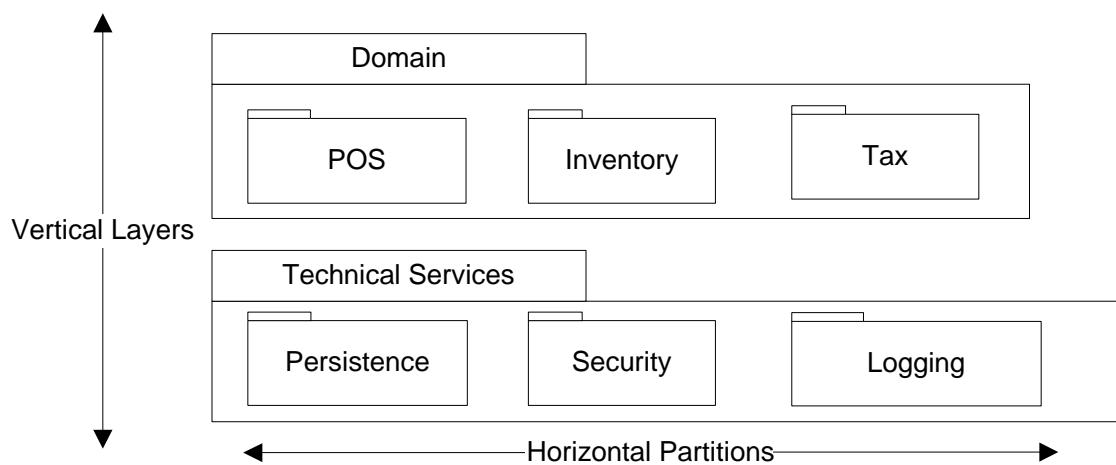


Architettura Aperta

- Una macchina virtuale può utilizzare i servizi delle macchine dei layer sottostanti
- Design goal: efficienza a runtime



Architettura Logica: Layers e Partizioni



Principali Architetture

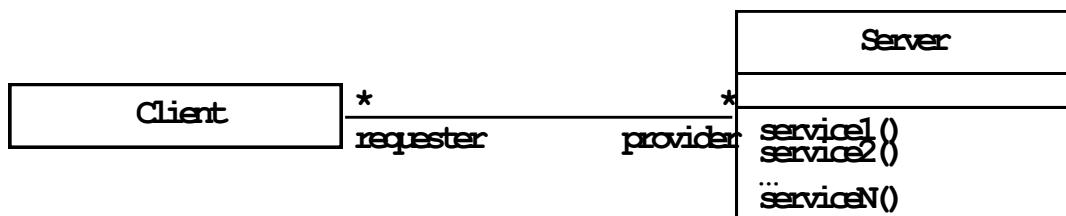


Architettura Client-server

- E' una architettura distribuita dove dati ed elaborazione sono distribuiti su una rete di nodi di due tipi:
 - I server sono processori potenti e dedicati: offrono servizi specifici come stampa, gestione di file system, compilazione, gestione traffico di rete, calcolo.
 - I client sono macchine meno prestazionali sulle quali girano le applicazioni-utente, che utilizzano i servizi dei server.
- I Client devono conoscere i nomi e la natura dei Server;
- I Server non devono conoscere identità e numero dei Clienti.

Client/Server Architecture

- Un sottosistema, detto server, fornisce servizi ad istanze di altri sottosistemi detti client che sono responsabili dell'interazione con l'utente.
- I Client chiamano il server che realizza alcuni servizi e restituisce il risultato.
 - I Client conoscono l'interfaccia del Server (i suoi servizi)
 - I Server non conoscono le interfacce dei Client
 - La risposta è in generale immediata
- Gli utenti interagiscono solo con il Client

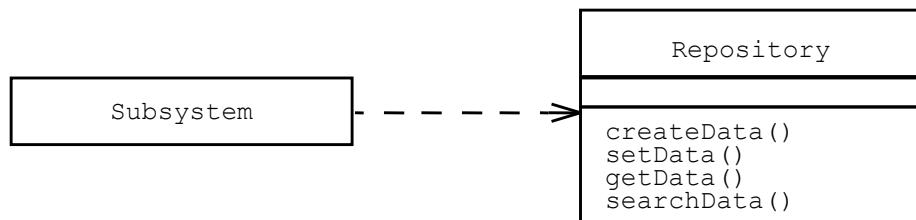


Principali Architetture Software

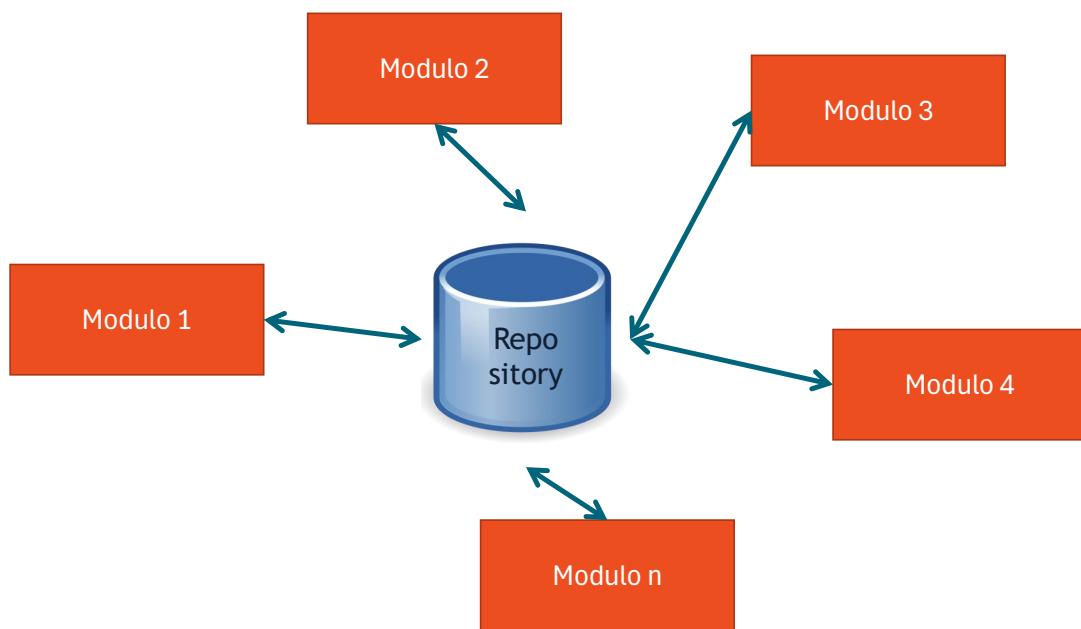
- Nell'ingegneria del sw sono stati definiti vari stili architetturali che possono essere usati come base per uno specifico sistema software da sviluppare:
 - Client/Server Architecture
 - Repository Architecture
 - Peer-To-Peer Architecture
 - Model/View/Controller

Repository Architecture

- I sottosistemi accedono e modificano una singola struttura dati chiamata repository.
- I sottosistemi sono “relativamente indipendenti” (interagiscono solo attraverso il repository)
- Il flusso di controllo è dettato o dal repository (un cambiamento nei dati memorizzati) o dai sottosistemi (flusso di controllo indipendente)

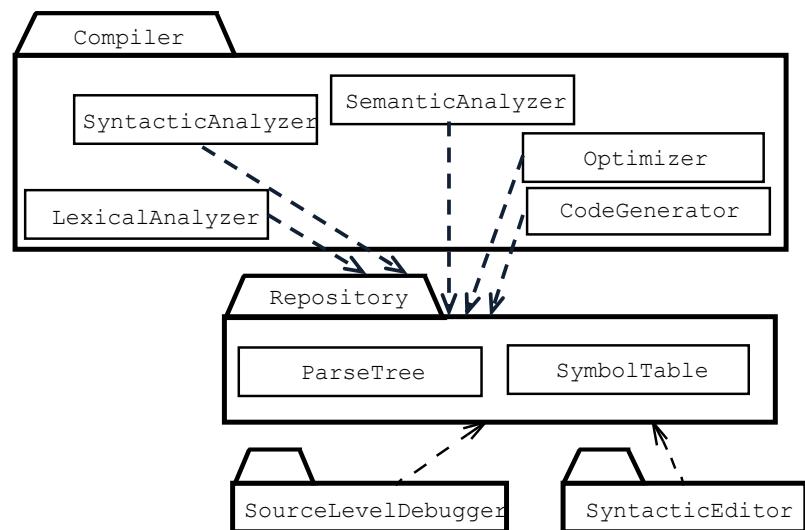


Repository Architecture



Esempio di Repository Architecture

- Database Management Systems
- Modern Compilers



Vantaggi dell'architettura a repository

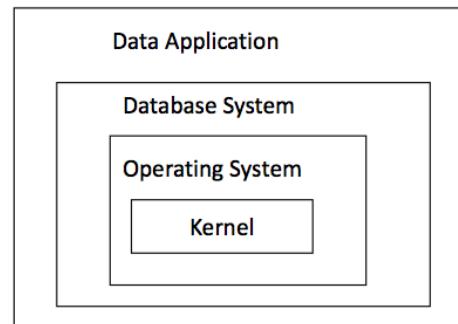
- Modo efficiente di condividere grandi moli di dati: write once for all to read
- Un sottosistema non si deve preoccupare di come i dati sono prodotti/usati da ogni altro sottosistema
- Gestione centralizzata di backup, security, access control, recovery da errori...
- Il modello di condivisione dati è pubblicato come repository schema
→ facile aggiungere nuovi sottosistemi

Svantaggi dell'architettura a repository

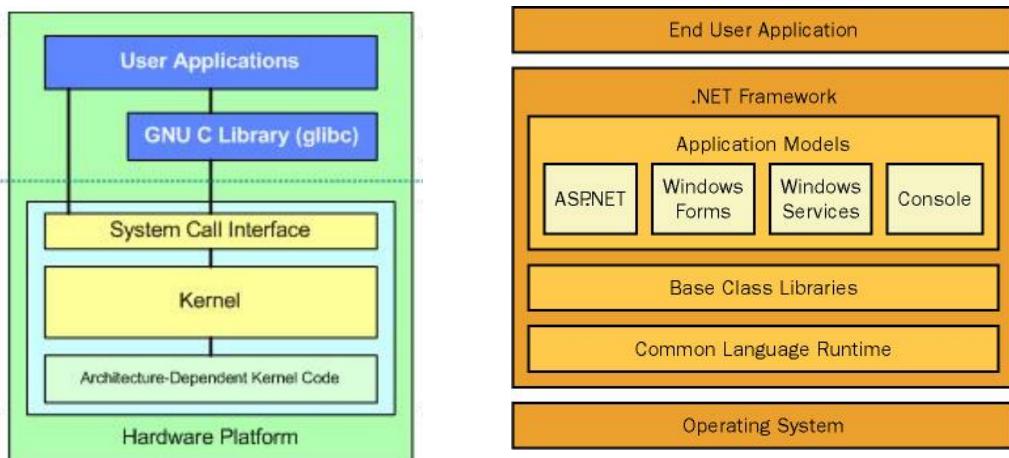
- I sottosistemi devono concordare su un modello dati di compromesso
→ minori performance
- Data evolution: la adozione di un nuovo modello dati è difficile e costosa:
 - esso deve venir applicato a tutto il repository,
 - tutti i sottosistemi devono essere aggiornati
- Diversi sottosistemi possono avere diversi requisiti su backup, security... non supportati
- E' difficile distribuire efficientemente il repository su piu' macchine (continuando a vederlo come logicamente centralizzato): problemi di ridondanza e consistenza dati.

Layered Architecture

- Sub-systems are organized into layers
- Each layer:
 - uses the services of the layer below
 - provides services to layer above through well-defined interfaces
- The terms “tier” and “layer” are interchangeable, for most people



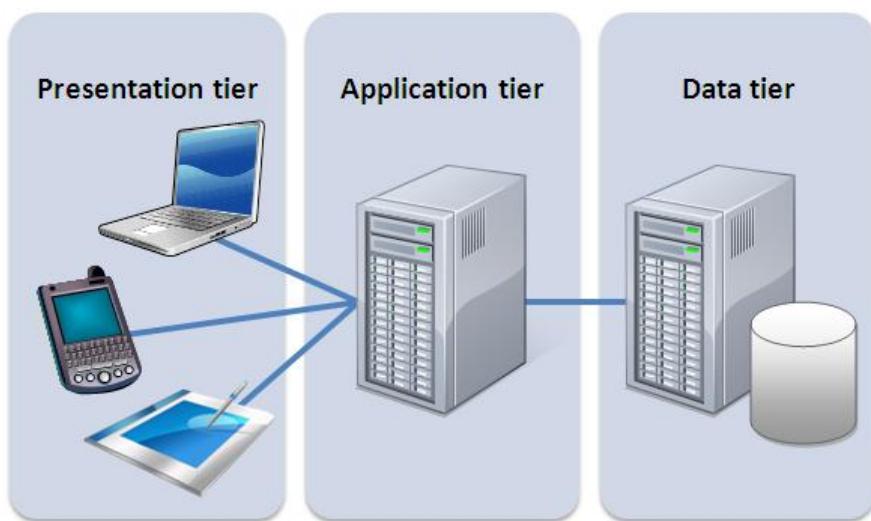
Layered Architecture Examples



Layered Architecture

- + abstraction of concerns
- + isolation of lower and higher levels from one another (i.e. decoupled)
- + reusability
- - overhead of going through different layers
- - complexity

Architetture a 3 livelli



Architetture three-tier - I

- Introdotte negli anni 90
- Business logic trattata in modo esplicito:
 - livello 1: gestione dei dati (comunicazione verso DBMS, file XML,)
 - livello 2: business logic (processamento dati, ...)
 - livello 3: interfaccia utente (presentazione dati, servizi)
- Ogni livello ha obiettivi e vincoli di design propri
- Nessun livello fa assunzioni sulla struttura o implementazione degli altri:
 - livello 2 non fa assunzioni su rappresentazione dei dati, né sull'implementazione dell'interfaccia utente
 - livello 3 non fa assunzioni su come opera la business logic..

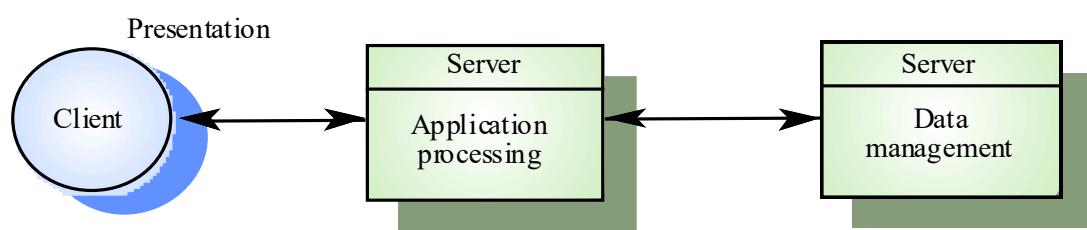
Architetture three-tier - II

- Non c'è comunicazione diretta tra livello 1 e livello 3
 - Interfaccia utente non riceve, né inserisce direttamente dati nel livello di data management
 - Tutti i passaggi di informazione nei due sensi vengono filtrati dalla business logic
- I livelli operano senza assumere di essere parte di una specifica applicazione
 - ⇒ applicazioni viste come collezioni di componenti cooperanti
 - ⇒ ogni componente può essere contemporaneamente parte di applicazioni diverse (e.g., database, o componente logica di configurazione di oggetti complessi)

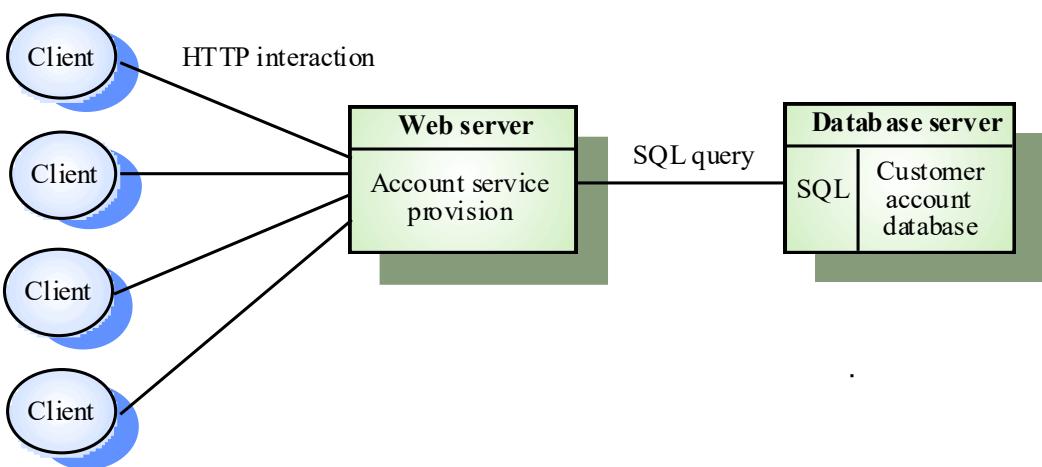
Vantaggi di architetture three-tier

- Flessibilità e modificabilità di sistemi formati da componenti separate:
 - componenti utilizzabili in sistemi diversi
 - modifica di una componente non impatta sul resto del sistema (a meno di cambiamenti nelle API)
 - ricerca di bug più focalizzata (separazione ed isolamento delle funzionalità del sistema)
 - aggiunta di funzionalità all'applicazione implica estensione delle sole componenti coinvolte (o aggiunta di nuove componenti)

A 3-Tier Client-Server Architecture



An Internet Banking System



Vantaggi di architetture three-tier

- Interconnettività
 - API delle componenti superano il problema degli adattatori del modello client server → N interfacce diverse possono essere connesse allo stesso servizio etc.
 - Facilitato l'accesso a dati comuni da parte di applicazioni diverse (uso dello stesso gestore dei dati da parte di business logics diverse)
- Gestione di sistemi distribuiti
 - Business logic di applicazioni distribuite (e.g., sistemi informativi con alcuni server replicati e client remoti) aggiornabile senza richiedere aggiornamento dei client

Svantaggi di architetture three-tier

- Dimensioni delle applicazioni ed efficienza
 - Pesante uso della comunicazione in rete ⇒ latenza del servizio
 - Comunicazione tra componenti richiede uso di librerie SW per scambio di informazioni ⇒ codice voluminoso
- Problemi ad integrare Legacy software
 - Molte imprese usano software vecchio (basato su modello monolitico) per gestire i propri dati ⇒
 - difficile applicare il modello three-tier per nuove applicazioni
 - introduzione di adapters per interfacciare il legacy SW

Architetture n-Tier

- Evoluzione delle 3-tier, su N livelli
- Permettono configurazioni diverse.
- Elementi fondamentali:
- Interfaccia utente (UI)
 - gestisce interazione con utente
 - può essere un web browser, Mobile App, interfaccia grafica, ...
- Presentation logic
 - definisce cosa UI presenta e come gestire le richieste utente
- Business logic
 - gestisce regole di business dell'applicazione
- Infrastructure services
 - forniscono funzionalità supplementari alle componenti dell'applicazione (messaging, supporto alle transazioni, ...)
- Data layer:
 - livello dei dati dell'applicazione

Introduzione

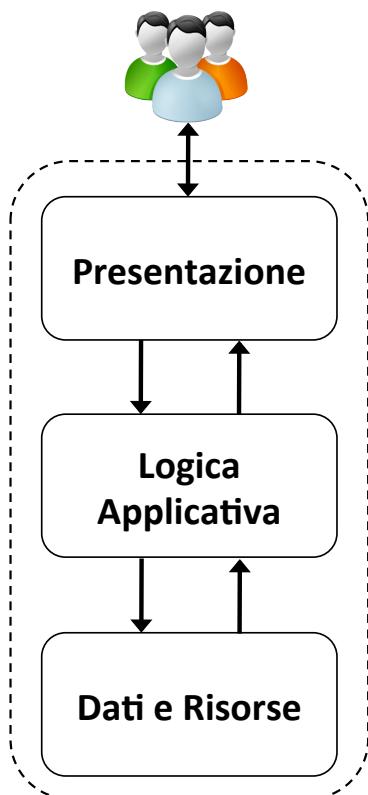
Obiettivo: definire uno schema generale di riferimento (*pattern architettonico*) per la progettazione e strutturazione di applicazioni di tipo interattivo

Pattern Architetturale

- definisce il più alto livello di *astrazione* di un sistema software
- descrive la *struttura* di un sistema software in termini di
 - *sottosistemi* e relative responsabilità
 - linee guida per gestire le *relazioni* e l' *interazione* tra sottosistemi
- la scelta del pattern architettonico è una scelta fondamentale e influenza direttamente le fasi di progettazione e realizzazione

Struttura di una Applicazione Software

La struttura di una applicazione software, e più in generale di un sistema informativo, è caratterizzata da **tre livelli**



- **Presentazione:** insieme dei componenti che gestiscono l'interazione con l'utente
- **Logica Applicativa:** insieme dei componenti che realizzano la logica applicativa, implementano le funzionalità richieste e gestiscono il flusso dei dati
- **Dati e Risorse:** insieme dei componenti che gestiscono i dati che rappresentano le informazioni utilizzate dall'applicazione secondo il modello concettuale del dominio

Model – View – Controller (MVC)

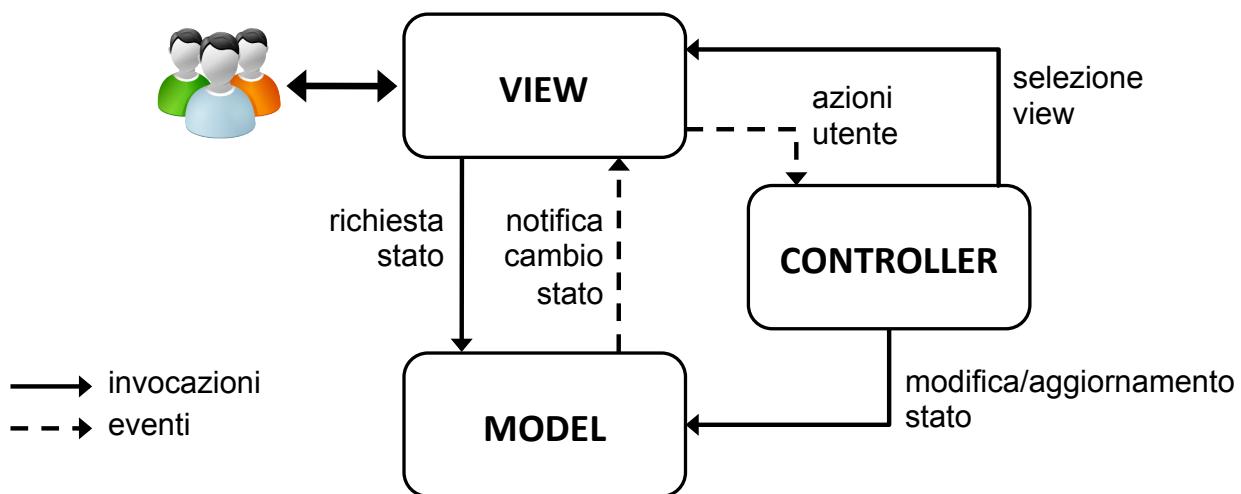
- **Pattern architetturale** per la progettazione e strutturazione modulare di applicazioni software *interattive*
- Originariamente introdotto da Trygve Reenskaug (sviluppatore Smalltalk presso lo Xerox Palo Alto Research Center) nel 1979
- Consente di separare e disaccoppiare il modello dei dati (**model**) e la logica applicativa (**controller**) dalle modalità di visualizzazione e interazione con l'utente (**view**)
 - l'applicazione deve separare i componenti software che implementano il **modello delle informazioni**, dai componenti che implementano la **logica di presentazione** e la **logica di controllo** che gestiscono tali informazioni

MVC: Struttura e Responsabilità (1/2)

Il pattern MVC identifica tre componenti di base

1. **Model:** rappresenta il modello dei dati di interesse per l'applicazione
 - incapsula lo stato dell'applicazione
 - gestisce l'accesso ai dati
 - fornisce le funzionalità per l'aggiornamento dello stato e l'accesso ai dati
 - notifica al *view* i cambiamenti di stato
2. **View:** fornisce una rappresentazione grafica ed interattiva del *model*
 - definisce le modalità di presentazione dei dati e dello stato dell'applicazione
 - consente l'interazione con l'utente
 - riceve notifiche dal *model* e aggiorna la visualizzazione
3. **Controller:** definisce la logica di controllo e le funzionalità applicative
 - gestisce gli eventi ed i comandi generati dall'utente
 - opera sul *model* (modifiche, aggiornamenti, inserimenti) in base agli eventi ed ai comandi ricevuti
 - può selezionare/aggiornare il *view* in base al risultato del processamento o alle scelte dell'utente

MVC: Struttura e Responsabilità (2/2)



- **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 generati dall' interazione con l' utente
- **Controller**
 - ha riferimento al **model** e al **view**

MVC: Interazioni Fondamentali (1/2)

Nella fase di inizializzazione dell' applicazione

1. viene creato il **model**
2. viene creato il **view** fornendo un riferimento al **model**
3. viene creato il **controller** fornendo riferimenti al **model** e al **view**
4. il **view** si registra come *listener* (o *observer*) del **model**
 - per ricevere notifiche di aggiornamento dal **model** (*observable*)
5. il **controller** si registra come *listener* (o *observer*) del **view**
 - per ricevere dal **view** (*observable*) gli eventi generati dall' utente

MVC: Interazioni Fondamentali (2/2)

Quando un utente interagisce con l'applicazione

1. il **view** riconosce l'azione dell'utente (es. pressione di un bottone) e notifica il **controller** registrato come *listener*
2. il **controller** interagisce con il **model** per realizzare la funzionalità richiesta ed aggiornare/modificare lo stato o i dati
3. il **model** notifica al **view** registrato come *listener* le modifiche e gli aggiornamenti
4. il **view** aggiorna la visualizzazione sulla base del nuovo stato
 - il nuovo stato e le info aggiornate per modificare la visualizzazione possono essere ottenuti dal view con
 - **approccio push:** il model notifica al view sia il cambiamento di stato che le informazioni aggiornate
 - **approccio pull:** il view riceve dal model la notifica del cambiamento di stato e poi accede al model per ottenere le informazioni aggiornate

MVC: Considerazioni

- Il pattern Model-View-Controller definisce una **architettura concettuale** di riferimento
 - *indipendente* dal linguaggio di programmazione
 - utile per impostare la *struttura generale* dell' applicazione in fase di progettazione
 - non definisce in maniera univoca schemi realizzativi ed implementazione
 - le modalità implementative possono dipendere dal linguaggio di programmazione e dal contesto applicativo
- In applicazioni interattive complesse l' architettura software è spesso costituita da un **insieme di componenti** con relazioni di tipo MVC
 - i componenti **model** encapsulano dati e funzionalità
 - possono esserci più componenti **view** per uno stesso **model**
 - ad ogni **view** può essere associato un componente **controller**
- Numerosi *framework* per diversi linguaggi di programmazione sono riconducibili al pattern MVC
 - Java Swing
 - Apple Cocoa
 - PureMVC (per C++, Flex, JavaScript, C#, Perl, PHP, Python, Ruby...)
 -

MVC e Qualità del Software (1/2)

Una progettazione architetturale che

- si basa sulla **separazione dei ruoli** dei componenti software
 - rende **strutturalmente indipendenti** moduli con funzionalità differenti
- favorisce qualità esterne ed interne del software

Qualità esterne

✓ **estendibilità**

- semplicità di progetto e decentralizzazione dell' architettura
- software facilmente estendibile agendo su moduli specifici

✓ **riusabilità**

- possibilità di estrarre e riutilizzare componenti

✓ **interoperabilità**

- interazione tra moduli con ruoli differenti
- possibilità di creare gerarchie tra componenti

MVC e Qualità del Software (2/2)

Qualità interne

✓ strutturazione

- struttura del software riflette le caratteristiche del dominio applicativo (dati + controllo + interazione e visualizzazione)

✓ modularità

- organizzazione del software in componenti con funzionalità definite

✓ comprensibilità

- ruoli e funzioni dei componenti sono facilmente identificabili

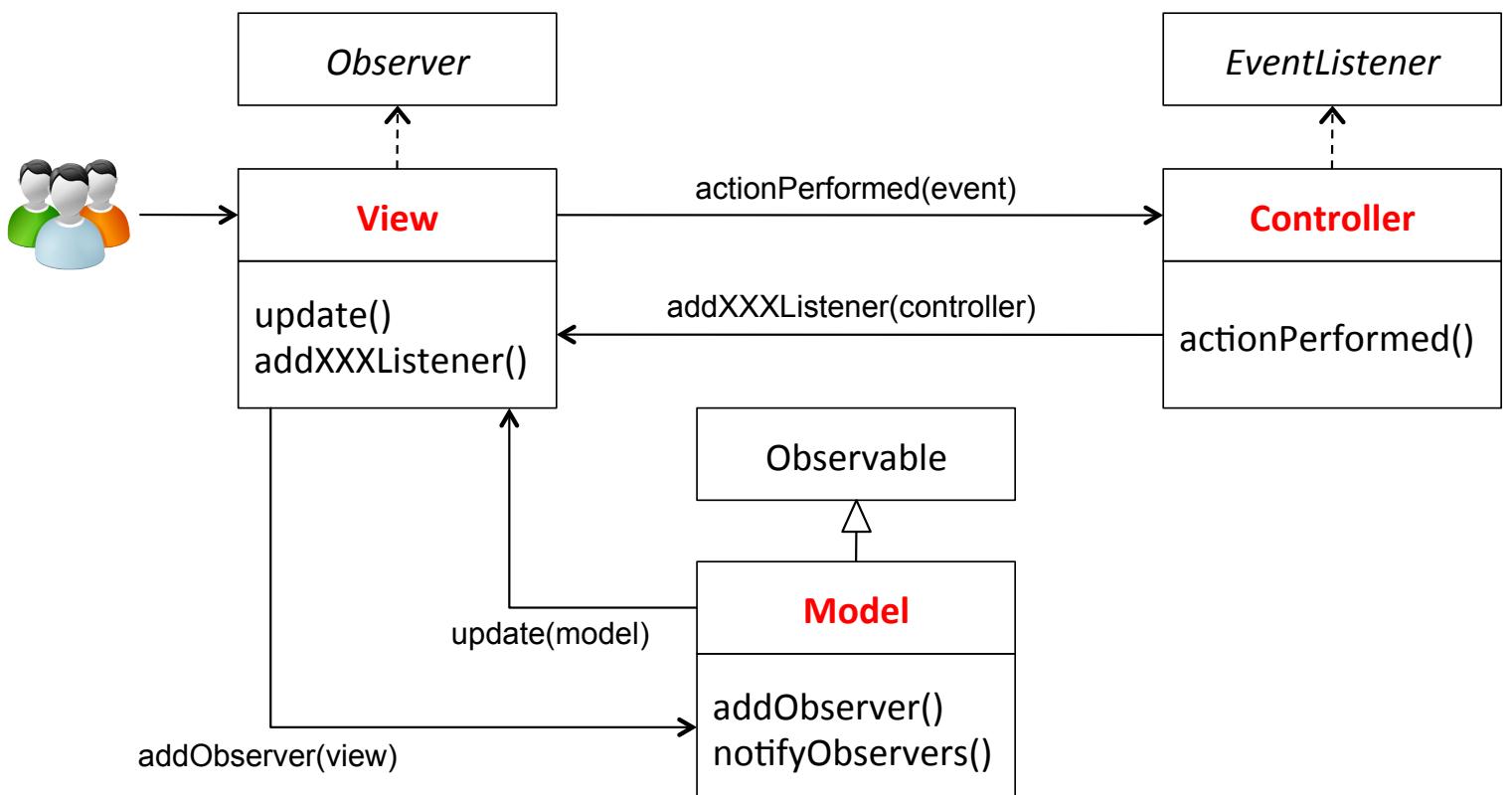
✓ manutenibilità

- possibilità di intervenire su componenti specifici con effetti nulli o limitati su altri componenti

MVC ed Eventi in Java (1/2)

- Nel paradigma MVC l' interazione tra componenti è basata su meccanismi di propagazione e gestione di **eventi**
 - componenti **view** notificano le azioni dell' utente ai componenti **controller**
 - componenti **model** notificano cambiamenti di stato ai componenti **view**
- In Java si ha che
 - l' interazione tra **view** e **controller** avviene in base al meccanismo di propagazione e gestione **eventi Swing/AWT**
 - i componenti **controller** sono **EventListener** (es. **ActionListener**, **MouseListener**...) associati ai componenti grafici **view** (es. **JButton**)
 - l' interazione tra **model** e **view** avviene secondo il pattern **Observer-
Observable**
 - i componenti **model** estendono la classe **Observable**
 - i componenti **view** implementano l' interfaccia **Observer** e si registrano presso i componenti **model** (**model.addObserver(view)**)
 - i componenti **model** notificano i cambiamenti ai componenti **view** registrati come **observers** (**notifyObservers()**)
 - i componenti **view** ricevono le notifiche (**update(model)**) e aggiornano la visualizzazione

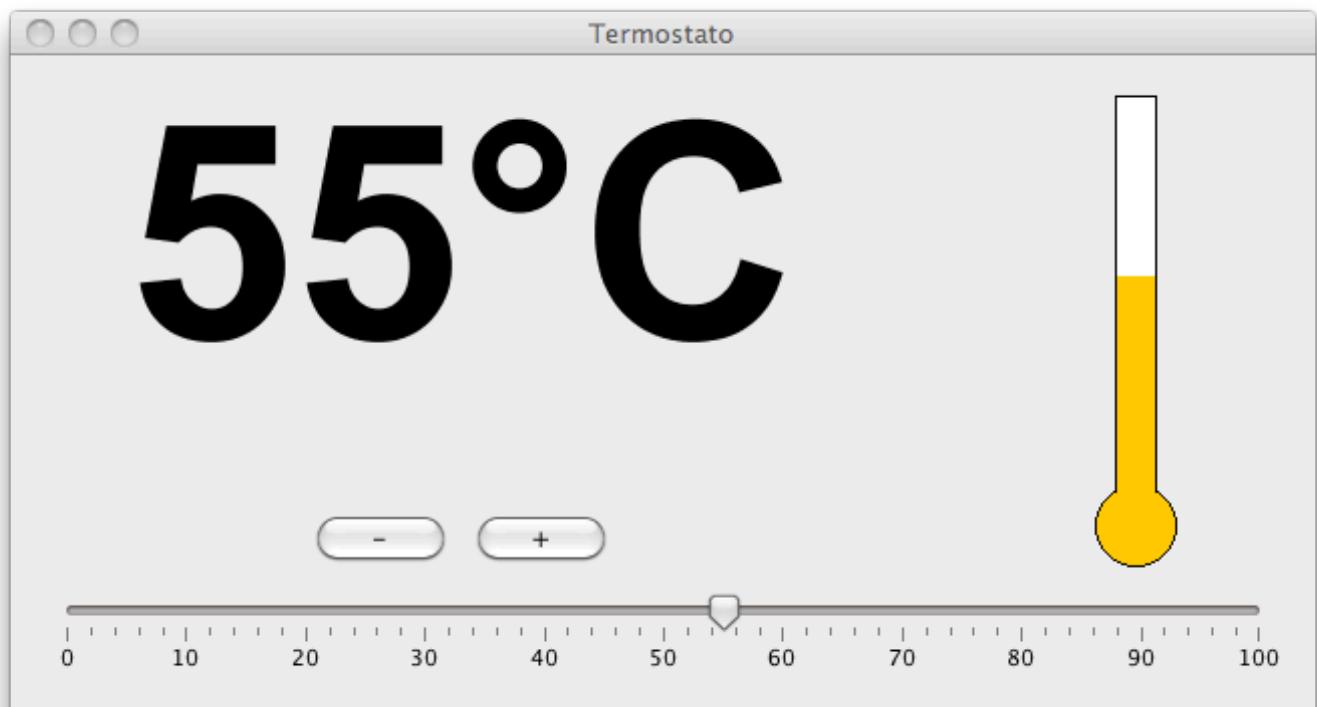
MVC ed Eventi in Java (2/2)



Esempio: Termostato (1/2)

- Si vuole realizzare un' applicazione interattiva che consente di impostare la temperatura tramite un termostato
 - **model**: rappresenta lo stato del termostato con il valore di temperatura impostato
 - **view**: consente all' utente di selezionare il valore di temperatura e visualizza con modalità differenti il valore che viene impostato
 - **controller**: riceve i comandi dall' utente per impostare la temperatura e aggiorna il model di conseguenza

Esempio: Termostato (2/2)

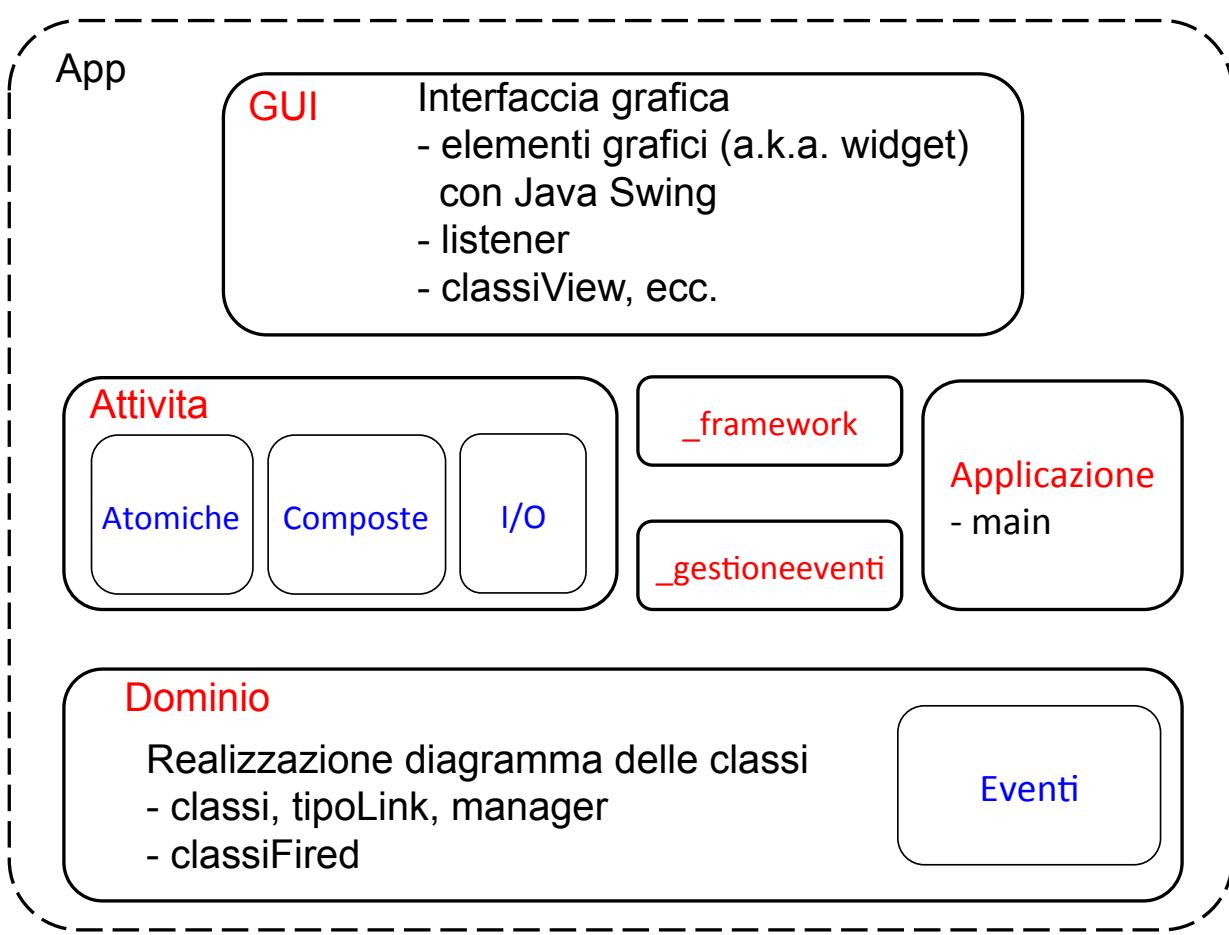


- 1 model
- 3 diverse view
- 2 controller
- Codice disponibile sul sito del corso

Applicazioni e Livelli

- Gli argomenti e la metodologie di progettazione e realizzazione presentati in questo corso possono essere ricondotti ad una **struttura a tre livelli**
 1. il **diagramma delle classi** (e corrispondente realizzazione) rappresenta il **dominio applicativo** di interesse
 - lo stato dell' applicazione è definito dalle istanze delle classi, dai link presenti tra di esse e dallo stato degli oggetti reattivi (secondo il **diagramma stati e transizioni** associato)
 2. il **diagramma delle attività** (e corrispondente realizzazione) descrive il comportamento dell' applicazione e definisce la **logica di controllo**
 - l' esecuzione delle attività opera sullo stato e produce modifiche (creazione nuove istanze e link, rimozione link, transizioni di stato, etc.)
 - l' esecuzione di attività può determinare l' interazione con l' utente (attività I/O grafiche) e la visualizzazione di schermate specifiche
 3. un' **interfaccia grafica** (realizzata con Java Swing) consente la **visualizzazione** dello stato dell' applicazione e l' **interazione** con l' utente
 - consente all' utente di interagire con l' applicazione per attivare e guidare la logica applicativa e di controllo
 - deve riflettere le modifiche e gli aggiornamenti che avvengono sullo stato (nuovi oggetti, eliminazioni oggetti, transizioni di stato, etc.)

La Struttura delle nostre Applicazioni



La Struttura dei Package

```
app
|
+-- dominio
|   |-- NomeClasse.java
|   |-- NomeClasseFired.java
|   |-- TipoLinkNomeLink.java
|   |-- ManagerNomeLink.java
|   |-- EccezioneCardMinMax.java
|   |-- EccezionePrecondizioni.java
|   |-- ...
|   \-- eventi
|       |-- NomeEvento.java
|       \-- ...
+-- attivita
|   |-- AttivitaIO.java
|   +-- atomiche
|       |-- NomeAttivita.java
|       \--...
|   +-- complesse
|       |-- AttivitaPrincipale.java
|       |-- NomeAttivitaComplessa.java
|       \-- ...
|   \-- ...
|       \-- ...
+-- _framework
|   |-- Executor
|   |-- Task
+-- _gestioneeventi
|   |-- Environment
|   \-- ...
+-- applicazione
|   \-- Main.java
|
\-- gui
    |-- ClasseGraficaSwing.java
    |-- XXXListener.java
    |-- NomeClasseView.java
    |-- ErrorNotifier.java
    \-- ...
```

Note

- Nel package **applicazione** è presente la classe **Main.java**
 - definisce ed implementa il metodo **main**
 - si occupa di inizializzare e avviare l' applicazione
- Nel package **gui** è presente la classe **ErrorNotifier.java**
 - definisce ed implementa il metodo

```
public static void notifyError(String message)
```
 - consente di mostrare all' utente un messaggio di errore al verificarsi di una eccezione
 - se in una porzione qualsiasi del codice si verifica una eccezione
 - l' eccezione viene catturata e gestita localmente
 - se è necessario visualizzare un messaggio di errore viene invocato il metodo **ErrorNotifier.notifyError** specificando il messaggio da visualizzare
 - la specifica modalità di visualizzazione del messaggio di errore può dipendere dalla applicazione (es. output su console, finestra, etc.) ed è definita nell' implementazione del metodo

MVC per Classi di Dominio e Widget (1/2)

- Durante la fase di progettazione vengono identificati i concetti e le entità che costituiscono il dominio applicativo
 - si definisce il **diagramma delle classi**
 - si identificano gli oggetti reattivi ai quali associare un **diagramma degli stati e transizioni**
- La logica applicativa e di controllo viene descritta tramite il **diagramma delle attività**
- In un' applicazione interattiva, un' opportuna **interfaccia grafica** consente all' utente di visualizzare lo stato ed i dati
- Durante l' esecuzione delle attività è spesso necessario **aggiornare** e **modificare** le informazioni visualizzate a causa di
 - creazione oggetti, link tra oggetti, etc.
 - transizioni di stato di oggetti reattivi

MVC per Classi di Dominio e Widget (2/2)

- Affinché l' interfaccia grafica rifletta i cambiamenti degli oggetti del dominio è necessario
 - identificare gli oggetti del dominio (**model**) cui corrisponde un elemento di visualizzazione grafica (**view**)
 - definire uno o più componenti grafici (**widget**) utilizzati per la rappresentazione e visualizzazione degli oggetti di dominio
 - stabilire una relazione **Model-View** (cioè **observable-observer**) tra gli oggetti del dominio e i corrispondenti componenti grafici
- Da un punto di vista realizzativo, per ogni classe **NomeClasse** del dominio cui è associata una visualizzazione
 - definire nel package **gui** una classe **NomeClasseView** che rappresenta il componente grafico (**widget**) associato all' oggetto di dominio **NomeClasse** (es. un pannello, una finestra, etc.)
 - registrare **NomeClasseView** come *observer* del corrispondente oggetto **NomeClasse** (secondo il pattern **observer-observable**)
 - identificare i cambiamenti di stato (es. inserimento nuovo link, aggiornamento variabile istanza, transizione stato) cui deve corrispondere un aggiornamento del componente **NomeClasseView** e notificare l' avvenuto cambiamento di stato (tramite **notifyObservers ()**) affinché venga aggiornata la visualizzazione
- I dettagli realizzativi dipendono dalla logica applicativa e di controllo della specifica applicazione (si vedano esercitazioni/esercizi di esame)