

Programmazione d'Interfacce

Federico Matteoni

Indice

1	Introduzione	3	12	Disruptive Innovation	17
1.1	Il Corso	3	12.1	Root Cause Analysis	18
2	Design	4	12.1.1	Why root cause analysis	18
2.1	XX Designer	4	12.1.2	Tipi di Task Analysis	19
2.1.1	UX Designer	4	12.1.3	Come fare la Task Analysis	19
2.1.2	UI Designer	4	12.1.4	Livello di dettaglio	19
2.2	Front-End Developer	4	13	Agile	19
3	Interfacce Utente	4	13.1	12 principi dell'agile	20
4	Good and Bad Design	5	13.2	Personas	20
4.0.1	Design of Useful Things	6	13.2.1	Scrivere le personas	20
5	Human Centered Design	6	13.2.2	Quante personas scrivere?	20
6	Design Thinking vs HCD	7	13.3	Requirements	20
7	Principi Fondamentali dell'Interazione	8	13.3.1	Tipi di requirement	21
7.1	Sei Fondamenti	8	13.4	User Stories	21
7.1.1	Affordance	8	13.4.1	Come scrivere le user stories	21
7.1.2	Signifiers	9	13.5	Scenarios	22
7.1.3	Mapping	9	13.5.1	Cosa tenere in considerazione	22
7.1.4	Feedback	9	13.5.2	Come scrivere uno scenario	23
7.1.5	Conceptual Model	10	13.5.3	Tre metodi	23
7.1.6	System Image	10	13.6	Use Cases	23
8	Cambiare le convenzioni	11	13.6.1	Elementi	24
8.1	Rethinking OS	11	13.6.2	Come scrivere uno use case	24
9	Constraints, Discoverability and Feedback	11	13.6.3	Takeaway	24
9.1	Constraints	11	14	Human Error and Mitigation Strategies	25
9.2	Forcing functions	13	14.1	Root Cause Analysis	25
9.2.1	Interlock	13	14.2	I 5 Perché	26
9.2.2	Lock-In	13	14.3	Attitudine delle persone verso gli errori	26
9.2.3	Lock-Out	13	14.4	Errore	26
9.3	Activity-Centered Controls	14	14.4.1	Lapsus e Errori cognitivi	26
10	How People do Things	14	14.4.2	Prevenzione dell'errore	27
10.1	I Golfi	14	14.5	Interruzioni	27
10.2	Sette Stati dell'Azione	15	14.6	Feedback errati	27
10.2.1	Tre livelli di Processing	16	14.6.1	Voce come feedback	28
11	Sette Principi Fondamentali della Progettazione	16	15	Design lessons	28
11.1	Sette Domande	16	15.1	Aggiungere vincoli per bloccare gli errori	28
11.2	Feedforward e Feedback	17	15.2	Undo	28
11.3	Sette Principi	17	15.3	Messaggi d'errore e di conferma	28
11.4	Opportunismo	17	15.4	Controlli di sensibilità	28
			15.5	Ridurre i lapsus	29
			16	Mitigare l'errore	29
			16.1	Principi di design per gestire l'errore	29
			16.2	Principi chiave di design	30

17 Prototyping	30	20 Alberto Betella, Moonshots and Disruptive Innovation	39
17.1 Thoughtland	30		
17.2 Falsi Positivi	31	21 Fabio Viola, Gamification	40
17.3 Falsi Negativi	31	21.1 Gamification	40
17.4 Thought Without Data Are Just Opinions	31		
17.5 Prototyping Manifesto	31	22 Vincenzo Gervasi, Sottosistema Grafico	41
17.6 Prototype	32	22.1 sottosistema grafico	41
17.7 I Sette Pilastri del Prototyping	32	22.2 modelli	41
17.8 Flusso del Prototyping	33	22.3 event loop	41
		22.4 Il file che descrive l'interfaccia utente	42
18 UX for Connected Devices	34	22.5 DOM	42
18.1 Internet of Things	34	22.6 Flutter	43
18.2 Industria 4.0	34	23 Antonio Cisternino, Realizzazione di una GUI	44
18.3 Digital Twin	35		
19 Ubiquitous Technology e Big Data	36	24 Nicola Melluso, Natural Language User Interfaces	45
19.1 Dispositivi Smart	36	24.1 Speech recognition e natural language understanding	45
19.2 Perché UX per IoT è diversa?	36		
19.3 Sensori e attuatori	36	25 Diego Colombo, Notebooks Hands on Session	46
19.4 Contesto di utilizzo	37		
19.5 Diversi livelli di sviluppo della UX	38		

1 Introduzione

Appunti del corso di **Programmazione d'Interfacce** presi a lezione da **Federico Matteoni**.

Prof.: **Daniele Mazzei**, mazzi@di.unipi.it

Materiale didattico:

- **Google Classroom**, slide presentate a lezione e altro materiale didattico
- La Caffettiera del Masochista, Donald A. Norman
Eng: The Design of Everyday Things
- Designing the User Interface, Ben Shneiderman
- *www.usability.gov*
- *interaction-design.org*

1.1 Il Corso

Interface Development in 2020 diventa **Interface Design in 2020**

Diviso in due parti

- **UX e UI** con introduzione, UI vs UX, HCI, paradigmi, gamification...
- **Strumenti per lo sviluppo dell'interfaccia utente** presentati da vari ospiti: Unity, Zerynth, Ubidots, Angular, Amazon Lex, ...

2 Design

Interfaccia è qualsiasi metodo utilizzato da una persona per **interagire** con un dispositivo.

Cos'è il design Il design è la **pianificazione o la specifica per la costruzione di un oggetto o sistema** o per l'implementazione di un'attività o processo. Diventa l'esatto opposto della decomposizione del problema in sottopassaggi, cioè del pensiero computazionale. Il design parte dalla base del problema e **identifica soluzioni per la causa del problema**. Si può avere anche il design di una strategia di implementazione.

„Bisognerebbe progettare le applicazioni come se fossero persone che ci piacerebbe frequentare”. Ad esempio Netflix, o il frigorifero.

2.1 XX Designer

Discernere tra Graphic Design, User Experience Design (UX Design) e User Interface Design (UI Design).

UX : come l'utente si sente per interagire e cosa vuole fare. Aspetto più psicologico, guida la UI design in base a statistica fatta su gruppi di utenti. Manda "l'output" a chi fa UI e al marketing.

UI : come l'utente interagisce col prodotto (shortcut, sottomenu...)

2.1.1 UX Designer

Si deve porre il problema di quali approcci usare per risolvere problemi evidenziati da analisi di mercato.

Chi paga non è detto che sia chi usa il servizio. Ad esempio Netflix viene pagato da una persona, ma lo stesso account viene usato anche da altre persone (anzi, in particolare **il 90% del tempo** chi usa l'account non è chi paga). Uno dei metodi usati per fare UX è quello della **definizione delle personas** (cioè un archetipo di utente). Una persona può assumere diverse personas.

User Experience Con User Experience si parla del prodotto e di come si comporta nel mondo reale, che è fatto di *personas*. **Non si può progettare una user experience, si può progettare per la user experience**. La user experience è ciò che fa l'utente, e lo sviluppatore non ha controllo su ciò. L'utente si appropria al software come gli pare.

2.1.2 UI Designer

Dalla UX si crea lo **sketch** dell'interfaccia. Non viene prodotto subito il wireframe ma bisogna partire da altro, ad esempio dai **casi di studio**. Esistono più casi di studio per ogni personas (casalinga voghera che fa bonifico, casalinga voghera che cambia password, ecc.). Ogni caso di studio è **specifico per personas**, poiché personas diverse hanno capacità diverse (non conoscere alcuni concetti, non saper fare determinate operazioni...).

L'UI design è un procedimento diverso dal front-end developing, quindi possono essere persone separate. Il designer progetta le guideline che istruiscono il developer.

Si può dire che la UI design è sottoarea di UX design.

2.2 Front-End Developer

Esegue il design della UI convertendolo in funzionalità del prodotto.

3 Interfacce Utente

L'interfaccia utente (UI) L'UI di un sistema è **lo spazio dove avviene l'interazione uomo-macchina**: lo schermo, le casse, il mouse e quant'altro.

L'obiettivo dell'interfaccia è far sì che **l'utente possa controllare la macchina, e non il contrario**. L'interfaccia può però influenzare il comportamento dell'utente, ad esempio se voglio guidare l'utente in un particolare modo l'interfaccia deve dare un feedback tale da guidare l'utente.

L'altro obiettivo dell'interfaccia è **rendere fruibile in maniera piacevole le funzionalità che una macchina eroga** verso l'utente. Il termine **user-friendly** non può essere omissivo: tra un'app facile e piacevole da usare e una solo facile da usare, l'utente medio preferirà sempre la prima.

L'interfaccia è strutturata a layer. lo HID (Human Interface Device) è la periferica con cui l'umano interagisce col sistema. Questo server per usare più HID per interagire con diverse applicazioni.

HMI (Human Machine Interface) è più astratta rispetto a HCI (Human Computer Interface), quindi in HMI è più teorica la cosa.

Diversi tipi di interfacce Abbiamo 5 sensi, quindi diverse **categorie d'interfaccia**: le più comuni sono **grafiche** e **tattili** (**GUI**, Graphical User Interface). Se si aggiunge anche il suono diventano **MUI** (Multimedia User Interface). Il concetto di GUI è stato coniato in un tempo in cui l'audio era raro. Adesso **praticamente tutte le interfacce sono MUI**.

Esempio di MUI riprogettata in GUI: Facebook. I video partivano in automatico con l'audio attivo, mentre ora sono mutati. Poi sono stati aggiunti i sottotitoli automatici: questo è un esempio di tecnica ideata per le utenze disabili e riusata per poter far fruire il prodotto a quelle personas che in quel momento non possono usufruire dell'audio. *Meglio un sottotitolo sbagliato che niente.*

Categorizzare interfacce Le CUI (Composite User Interfaces) sono le UI che interagiscono con due o più sensi. Esistono tre diverse macrocategorie di CUI:

Standard, che utilizzano dispositivi standard come tastiere, mouse e monitor

Virtuale, che **bloccano il mondo reale e creano un mondo virtuale** e tipicamente utilizzano dei caschi VR

Aumentata, che **non blocca il mondo reale e eroga contenuti non completamente digitali**, ma che prendono dalla realtà esterna che circonda l'utente

Le CUI possono anche essere **classificate per il numero di sensi** con cui esse interagiscono. Per esempio, lo *Smell-O-Vision* è una CUI standard 3S (3 sensi) con un display, suono e odori. Se si aggiungesse la vibrazione della poltrona, diventerebbe 4S poiché si aggiunge il tatto.

Si parla di **Qualia Interfaces** quando si stimolano tutti i sensi.

Mancata evoluzione Le UI **sono le stesse di 10 anni fa**. Bisogna mettere in discussione i paradigmi attuali. L'industria ha convertito l'ambiente fisico della scrivania in ambiente digitale, prendendo ispirazione dall'abitudine dell'utente per rendere più semplice il passaggio. Ora l'utente è abituato, la realtà da cui si prende spunto non esiste più. Bisogna cambiare.

4 Good and Bad Design

Il buon design non esiste, poiché si fa design *per* la user experience **di una determinata personas**. Le due caratteristiche più importanti su cui misurare il buon design sono:

Discoverability: è la **capacità innata di un sistema di veicolare i possibili usi e dire come si usa**. Non è detto che una volta che si è capito cosa si può fare si riesca a farlo.

Per avere buona discoverability si usa tipicamente la visibilità: un rubinetto con i pomelli bene in vista incrementa la discoverability. Nel software, **questo lavoro lo fanno i pulsanti**.

Understanding: è la **capacità di comprendere i possibili usi**. Ad esempio: il fornello, è in cucina quindi so che si usa per scaldare ecc., il problema maggiore però è il mapping pomello → fornello. Si può risolvere con l'icona del fornello corrispondente, ma non risolve effettivamente il problema. Una soluzione efficace è disporre fornelli e pomelli in modo che sia evidente la correlazione fra essi.

Non sottovalutare il costo mentale dell'utente.



4.0.1 Design of Useful Things

Il paradosso di TripAdvisor „Quando la gente mangia bene, non recensisce. Quando mangia male, recensisce”.

Sensazioni Quando le cose vanno bene, si dimenticano subito. Questo perché, in qualche modo, l'uomo pensa che **le cose vadano bene per definizione**. Quando qualcosa va storto, invece, **l'amigdala crea un ricordo con un peso molto maggiore**.

Il design deve quindi preoccuparsi di come funzionano le cose, come vengono controllate e della natura delle interazioni. Quando la progettazione è fatta bene, crea prodotti piacevoli e brillanti. Quando è fatta male, i prodotti sono inutilizzabili e ciò porta a notevole frustrazione e irritazione.

Marcatore somatico: ricordo le esperienze in base alle sensazioni che provavo durante esse. **Più forte è la sensazione più si cementifica il ricordo**. Ad esempio se faccio un incidente ad una curva, la ricorderò bene per molto tempo. La strada che faccio per andare in vacanza non la ricordo più già al ritorno.

Con il software si applica lo stesso discorso. Se non riesco ad usare un programma inizio a provare frustrazione. Gli umani non informatici tengono a ritenere le macchine come superintelligenti, quindi associano alla frustrazione l'incapacità personale: **se credo di non essere in grado di usare il software non ci riprovo**.

Confrontando IA e intelligenza umana, l'IA risulta strettamente limitata a computazione e risoluzione di problemi logici. Al contrario, **la mente umana non funziona ad algoritmi ma procede per deduzione**. Per lo più generando ipotesi senza fondamento e autoconvincendosi.

Le macchine seguono regole semplici: gli algoritmi. Essi non hanno la flessibilità (**common sense**) tale da assecondare l'utente. Per esempio, se chiedo telecomando per l'aula D2 ma non esiste o non c'è il proiettore, la signora mi corregge in D1 e dà il telecomando corretto. La macchina dice semplicemente che non esiste l'aula D2 o il proiettore in aula D2.

Le macchine non hanno buonsenso. La maggiorparte delle regole sotto il software sono note solo agli sviluppatori. Potrebbe andare bene, basta renderle discoverable.

Bisogna invertire il paradigma attuale: se qualcosa va storto è **colpa dello sviluppatore** e non dell'utente. **Il dovere della macchina è essere comprensibile** da parte dell'utente.

Bisogna accettare che il comportamento umano è com'è e non come vogliamo che sia.

5 Human Centered Design

Alla fine di ogni passaggio c'è l'utente.

Si tratta di una norma ISO 9241-210:2010(E).

Un approccio Lo HCD è un **approccio di design** specificamente orientato allo sviluppo di sistemi interattivi con l'**obiettivo di fare sistemi utili, altamente usabili e che si focalizzino sull'utente**. Il metodo è orientato all'**efficienza ed all'efficacia**, per aumentare la soddisfazione dell'utente ed evitare il più possibile gli effetti negativi.

Prima l'utente, poi le features Lo HCD mette **i bisogni, comportamenti e capacità umane prima di tutto, e progetta in funzione di esse**.

Significa che se devo risolvere un problema, non mi interessa risolverlo completamente ma raggiungere il miglior risultato che posso far ottenere all'utente che usa il mio software. Se il *70% degli utenti raggiungono il proprio scopo* col nostro software, allora esso ha un'*efficacia del 70%*. Posso puntare ad un'efficienza maggiore magari resolvendo una parte minore del problema.

Less is more. Meglio una feature in meno che una in più. Ogni volta che aggiungi una feature devi dimostrare perché e a cosa serve, perché tale feature va: spiegata, testata, mantenuta oggi e domani (**backward compatibility**).

Il problema principale delle UI è un **problema di comunicazione** in particolare dalla macchina verso la persona. Una buona interfaccia sa comunicare con l'utente.

Progettare interfacce che funzionano egregiamente fintanto che le cose vanno bene è relativamente facile, ma **la comunicazione è ancora più importante quando le cose non vanno bene**: entrano in gioco le **strategie di mitigazione dell'errore**. Si focalizza l'interazione soprattutto nel **comunicare ciò che è andato storto**, in quel momento devo aiutare l'utente frustrato a risolvere il problema perché se lo aiuto a risolvere il problema da solo proverà una sensazione positiva di successo per aver capito cosa non funzionava. Ciò **crea empatia col sistema**.

Quindi bisogna **evitare la frustrazione**, e **aiutare a risolvere** quando insorge un problema.

Capire l'utente Lo HCD è una filosofia di design che parte dalla **comprensione delle persone e dei bisogni** che si intende soddisfare. Spesso gli utenti non si rendono conto dei loro effettivi bisogni e nemmeno delle difficoltà che incontrano.

Per capire l'utente la tecnica più utilizzata è l'osservazione. Non è detto sia sempre possibile. Versioni alpha e beta non servono solo debuggare il software, ma servono anche a capire ciò che fanno gli utenti. Diventa utile avere statistiche sull'utilizzo effettivo del sistema: quanti click su un determinato pulsante, quante volte una determinata procedura finisce e così via.

Le specifiche dello HCD, quindi, **nascono dalle persone** e per questo **non si possono scrivere**. Quindi risulta essere un paradigma che si sposa bene con la computer science perché va avanti per iterazioni: si esegue una specifica ad alto livello, ne implemento una parte, la testo sull'utente reale e tramite il feedback modifico la parte implementata e ri-testo. Quando ritengo buono ciò che ho prodotto lo congelo, e passo ad implementare un'altra parte dell'interfaccia.

Il ruolo dello HCD nel design

Experience design	Area di focus
Industrial design	Area di focus
Interaction design	Area di focus
Human Centered Design	Il processo che assicura che la progettazione incontra i bisogni e le capacità degli utenti che useranno il sistema

Possiamo progettare per esperienza utente, il design industriale e progettare per l'interazione. lo HCD non è area di focus del processo di design ma è metodo.

Utilizzo l' HCD per progettare tutto il resto.

6 Design Thinking vs HCD

Insieme al termine Human Centered Design, spesso si può vedere il termine **Design Thinking**. I termini vengono da due scuole di pensiero molto forti ma con visioni diverse.

Cos'è il Design Thinking Il **Design Thinking** segue il filone Stanford, dove è nato: è un **processo di design** con cui **progettare nuovi prodotti** che verranno **effettivamente adottati dalle persone**. Come processo è più vicino alla disruptive innovation che all'antropocentricità.

Metodo, strumento per sviluppare prodotti innovativi. Per sviluppare qualsiasi modello di business orientato all'essere profittevole.

Si suddivide in 5 fasi iterative.

Empathize **Studiare** il proprio pubblico. Progettare il prodotto in modo che stabilisca un collegamento empatico con l'utente.

Define Delineare meglio le **domande chiave**, cioè quali sono i bisogni a cui assolvere.

Ideate **Brainstorming**, creare soluzioni.

Prototype **Costruire** una o più idee.

Test **Testare** le idee e **ricevere feedback**.

HCD e DT Lo HCD è un mindset che viene sovrapposto al design thinking, il quale è orientato a garantire che le idee siano rilevanti e beneficiari, sul lungo termine, per le persone obiettivo.

Lo HCD quindi viene sovrapposto al design thinking: identificato il modello di business, uso lo HCD per sincerarmi che la famiglia di soluzioni identificate venga "pulita", attraverso un processo che **garantisce l'usabilità da parte di soggetti umani**.

Design Thinking	Human Centered Design
Processo iterativo a 5 fasi che porta all'effettivo sviluppo di prodotti/soluzioni che verranno adottate dall'utente finale desiderato	Mentalità e strumento da applicare insieme al Design Thinking che crea un impatto a lungo termine positivo, per gli utenti della soluzione

Quando l'ispirazione (divergente: produrre idee) cala, si passa all'ideazione (convergente: unire le idee simili, scartare idee ridondanti...).

7 Principi Fondamentali dell'Interazione

Life is made of experiences Bravi designer producono **esperienze** piacevoli. L'esperienza è molto importante, perché determina quanto bene gli utenti si ricorderanno l'interazione.

Cognizione ed Emozione Quando la tecnologia si comporta in maniera inaspettata, proviamo confusione, frustrazione e rabbia: **emozioni negative**. Quando invece comprendiamo il comportamento della tecnologia, abbiamo una sensazione di controllo, bravura e persino orgoglio: **emozioni positive**. **Cognizione ed emozione sono profondamente legate**. Se non metto l'utente in un **mood positivo** farà più fatica ad apprendere l'interfaccia. Più mi arrabbio meno sono predisposto a comprendere e riutilizzare il prodotto.

7.1 Sei Fondamenti

La **Discoverability**, cioè il grado di facilità con cui un utente **scopre come funzione l'interfaccia**, è il risultato della corretta applicazione di sei principi psicologici.

7.1.1 Affordance

Il termine **affordance** si riferisce alla **relazione tra un oggetto fisico e una persona**: precisamente la relazione tra **le proprietà di un oggetto e le capacità dell'utente che determinano i possibili utilizzi dell'oggetto**. **Questa proprietà determina il modo con cui l'oggetto può essere usato**.

"Cosa posso fare sull'interfaccia".

Esempi Un **pulsante di una UI**, da premere con uno HID che sia il dito o il mouse, è un **oggetto fisico**. Il pulsante *afforda* (**consente**) l'essere premuto.

Una sedia *afforda* il sostenere, quindi *afforda* di sedercisi.

Un potenziometro *afforda* l'essere ruotato.

Tipi di affordance Ci sono affordance **innate nel cervello**, forme che il sistema visivo e il cervello interpretano automaticamente.

L'affordance è una **proprietà scaturita da una relazione con un particolare soggetto** (quindi è peculiarità della relazione). Ad esempio, una poltrona *afforda* il sostenere per quasi tutti, ma lo spostamento non è detto sia *affordato* per tutti (per esempio una persona debole non può spostare la poltrona).

Anti-affordance: prevenzione dell'interazione. Ad esempio degli spunzoni per evitare che piccioni si posino su un cornicione, **prevengono l'affordance che il cornicione ha verso i piccioni di sedersi**.

Affordance e anti-affordance **devono essere discoverable e perceivable**. Questo fatto non è scontato: il vetro *afforda* l'essere attraversato dalla luce e non *afforda* l'essere attraversato dalla materia, ma si può non vedere e **percepire una falsa affordance** di passarci attraverso... e ci batto.

Un altro esempio: anche a schermo spento, lo smartphone ha comunque l'*affordance* di essere premuto.

Assolutamente sbagliato dire che "metto un *affordance*". Posso dire che "metto un **significante**", ma solo se ho un'*affordance*. I tre pallini per il tasto menu sono un **significante**.

7.1.2 Signifiers

I designer hanno problemi pratici: devono sapere come progettare le cose per renderle understandable. Un **significante** è un **modo per indicare dove applicare un determinato affordance per ottenere un risultato**

Esempi Un box quadrato in una GUI (un pulsante) è un **significante**: se applichi l'*affordance* "tocco" qua ottieni un determinato risultato.

L'*affordance* del touch, lo slide, il pinch... esiste su tutto lo schermo. L'*affordance* dice **cosa** posso fare, il **significante** dice **dove** fare l'azione.

A volte i **significanti sono indispensabili** perché la maggioranza delle *affordance* sono invisibili. I significanti servono per fare capire le *affordance* che non si vedono. Per esempio, le porte scorrevoli: se non vedo i cardini, quando vedo la maniglia decido di spingere la porta ma essa non si muove perché è scorrevole. La spinta è un'*affordance* **percepita** che non esiste.

Nel design i **significanti sono molto più importanti delle affordance**, perché **comunicano come usare il design**. Questo perché viviamo in un mondo in cui le affordance sono state già presentate in genere. Creare nuove affordance è molto molto difficile.

Convenzioni Come associare l'affordance e il significante ad azioni reali? Nella maggiorparte dei casi tramite **convenzioni**. La comprensione di un'affordance percepita è dovuta alle convenzioni culturali.

Tipi di signifiers I significanti possono essere **voluti** o **accidentali**.

Voluto Ad esempio un'etichetta, una stringa, un'icona.

Accidentale Ad esempio delle persone in fila alla stazione.

7.1.3 Mapping

Il **mapping** è di grande importanza nel progettare le interfacce e stabilire i significanti. La **disposizione** dei significanti, a parità di significanti, può dire di più sull'interfaccia e le funzionalità.

Il **mapping** è la **relazione tra elementi di due insiemi**. Il modo migliore per fare mapping è **quello naturale**, perché è un'attività in cui il nostro cervello è molto bravo, ed il mapping di forme geometriche è la prima cosa che si impara da bambini.

7.1.4 Feedback

Un altro elemento fondamentale per il design delle interfacce è il **feedback** inteso come **risposta dell'interfaccia verso l'utente**.

Immediato Il feedback **deve essere immediato**. Il sistema sensoriale è parte integrante del sistema cognitivo, e l'uomo usa i propri sensi per guidare i propri ragionamenti. Se progetto un'interfaccia che non abilita i miei sensi a capire cosa sto facendo, inizio a fare più fatica a usare il prodotto o non ci riesco proprio. Un esempio: una pagina web che **non mostra se sta caricando la procedura richiesta**.

Uno dei **problemi principali** del feedback quindi è il **tempo**. Se faccio un'azione, **devo avere un feedback entro un certo lasso di tempo**. Se questo tempo è superato, il mio cervello non è più in grado di associare il feedback all'azione compiuta e ho così due pessimi risultati: **non ho dato feedback** e **ho mandato in confusione l'utente**. Il feedback deve avvenire **entro massimo 100 ms dall'azione, altrimenti non sarà efficace**. Meglio un buon feedback che un **bel** feedback.

Informativo Inoltre, il feedback **deve essere informativo**. Questo non significa che deve portare con sé tanta informazione, ma che deve **assolvere al proprio obiettivo**. Un esempio: se premo un pulsante non ho bisogno di fare grandi cose come feedback, posso **semplicemente** farlo diventare grigio. Non servono messaggi del tipo "*ok pulsante premuto*" ecc., sono superflui.

Colorare il pulsante di rosso o di verde **non è più informativo**: creo confusione a causa del mapping naturale tra il colore e il significato (rosso → errore, verde → successo) e l'utente non capirà se ha ottenuto un errore o se la richiesta è stata ricevuta correttamente.

Il feedback deve quindi essere **informativo nell'accezione dell'azione a cui è associato**. *Meglio nessun feedback rispetto ad un feedback errato.*

Semplicità **Non bisogna essere troppo pedanti**. Se il feedback è eccessivo, l'interfaccia utente diventa pesante. Altro problema che può insorgere è un feedback non allineato con il contesto dell'utilizzo del dispositivo. Per esempio, non posso usare lo stesso beep delle cinture per segnalare la riserva. Il beep delle cinture è fastidioso perché *deve esserlo*, ma la riserva, quando viene segnalata, non è in un contesto urgente. Se il beep è fastidioso, o spaventa, posso mettere in pericolo la vita dell'autista se viene spaventato mentre guida. **Non limitarsi alla tecnologia disponibile** "*ho solo quel buzzer, non posso fare altrimenti*". Nell'esempio non sono obbligato a far partire un beep quando si entra in riserva, posso **semplicemente** fare lampeggiare la spia.

Esempi Il **feedback** della luce del pulsante dell'ascensore quando viene premuto.

Un messaggio "*Pagamento eseguito*" a termine di una procedura pagamento.

7.1.5 Conceptual Model

Un **modello concettuale** è una **descrizione estremamente semplificata delle funzionalità del sistema**. L'esempio classico sono i file e le cartelle. Come racconto a qualcuno com'è organizzata memoria di archiviazione di un computer? Uso un **modello concettuale noto**, cioè *fogli di carta con contenuti, vengono raccolti in raccoglitori e quest'ultimi raccolti in schedari*.

Agli utenti **non interessano come funzionano** le cose, ma che **funzionino**. Perché l'hanno comprato.

Il modello concettuale è **come il designer vuole che l'utente percepisca la piattaforma**. Sarebbe l'*ambizione* di progettare (la comprensione) della UX.

Per l'utente I modelli concettuali servono per **andare incontro all'utente**, per convertire i vari aspetti di complessità tecnica in aspetti **comprensibili da chiunque**. I **modelli concettuali già in commercio sono difficili da mettere in discussione**. Questo perché, in caso si esca con un prodotto concorrente ad uno già affermato ma che funziona in modo diverso (diverso modello concettuale), l'utente diventa costretto a confrontare i due prodotti. **Mai far valutare "uno contro uno" agli umani**, perché non esistono le sfumature ma la valutazione si risolverà in **vivo o morto**. Inventare un nuovo sistema di streaming musica/film è pressoché **impossibile**. La gente ha Spotify e Netflix, non importa cosa fanno o come.

Modello concettuale "*film non comprati su DVD*" → **Netflix**

Modello concettuale "*musica non comprata su CD*" → **Spotify**

Modello Mentale Una volta pensato e progettato il modello concettuale si **implementa l'interfaccia in modo che il modello concettuale venga veicolato all'utente** tramite i significanti.

Quando persona si interfaccia con un sistema, sviluppa un **modello mentale**. Se il modello mentale e quello concettuale sono **allineati**, la persona è **in grado di usare il sistema**. **Più è grande la differenza** tra il modello mentale e quello concettuale, **più la persona farà fatica** ad usare il sistema. Inoltre, l'utente può sviluppare **modelli mentali diversi per diverse funzionalità** dello stesso sistema.

Il **modello concettuale viene trasferito all'utente per spiegare come funziona l'interfaccia**, non com'è fatta.

Il pomello che comanda un determinato fornello è una questione di *mapping*, ma **che il fornello spruzzi fuoco o acqua è modello mentale dell'utente**.

Telefono senza fili Solitamente, e idealmente, la gente apprende i modelli concettuali direttamente dal device andando per tentativi. La **problematica è quando lo apprende per passaparola**. Seguendo la filosofia del *telefono senza fili*, quando avviene il passaparola da persona a persona cambia l'interpretazione. Per ogni disallineamento tra modello concettuale e mentale, l'interpretazione cambia di conseguenza. Per questo vi è **necessità che il modello concettuale sia pressoché univoco con quello mentale**. Più piccolo è il delta tra i due, meno rischio di creare comportamenti assurdi col passaparole.

Less is more, se la feature è difficile da veicolare non metterla.

Quando gli utenti di una determinata feature la usano con successo l'85% delle volte, e il 70% degli utenti del sistema usano quella feature, allora posso inserirne un'altra.

7.1.6 System Image

Le persone creano **modelli mentali di sé stessi, degli altri, dell'ambiente che hanno intorno e delle cose con cui interagiscono**. Questi modelli mentali sono creati attraverso l'**esperienza, l'allenamento e l'istruzione**. Nell'immagine di sistema troviamo tutti questi componenti. Essa è il **modello concettuale che l'utente si crea dell'intero sistema grazie all'esperienza**. Si può quasi descrivere come l'insieme dei modelli mentali, racchiude il modello concettuale e quello mentale e descrive come stanno nel sistema complesso.

Scoperta L'utente non può chiedere allo sviluppatore come funziona il sistema, ma **deve scoprire le funzionalità da solo**. La **teoria dell'immagine di sistema** dice che l'utente sviluppa un proprio **user model** grazie all'oggetto e agli elementi di contorno (il manuale d'istruzioni, la pubblicità, il passaparola ecc.). Quindi **il modello concettuale è solo parte dell'immagine di sistema**. Devo **aiutare l'utente a sviluppare un modello mentale vicino a quello concettuale**, dotandolo di altre informazioni, come ad esempio il blog del software.

Il modello concettuale è **ciò che voglio che l'utente pensi**, il modello mentale **ciò che l'utente pensa**. L'**immagine di sistema è insieme delle informazioni** (UI, materiali di contorno) **che consente all'utente di formarsi il modello mentale più consono**. Bisogna essere bravi a farlo in modo che tutti i modelli generati siano compatibili con il modello concettuale.

8 Cambiare le convenzioni

Viene naturale pensare che l'innovazione debba essere un segno di discontinuità con il passato (**disruptive innovation**), ma far digerire questo agli utenti spesso è un **problema**. Ogni volta che si **viola una convenzione**, sia essa

culturale, legale, tecnologica, o anche frutto di pessime abitudini, si **chiede all'utente di fare un nuovo passaggio di apprendimento**: questa richiesta genera **attrito** con l'utente perché il cambiamento mette stress, indipendentemente dai meriti del nuovo sistema. Se devo cambiare abitudini ci penso due volte, *perché cambiare fa fatica*. Quindi il processo del cambiamento delle convenzioni **dovrebbe essere graduale**.

Consistenza Bisogna cercare di trovare il **modo per rimanere consistenti**. Diventa necessario regolare la quantità di innovazione introdotta ad ogni passo in modo che sia **abbastanza da portare l'utente avanti** ma **non troppo da essere percepita come diversa**. In altri termini, il **modello mentale** che l'utente si è costruito **deve rimanere tale, solo leggermente allargato**. La volta dopo lo si allargherà ancora e così via.

No Sistemi Misti! La cosa peggiore che si può fare è quella di **innovare lasciando il vecchio sistema presente per un po'**, in modo da "*facilitare il passaggio*". Al contrario, ciò genera solo confusione per l'utente.

I sistemi misti confondono perché l'utente è portato a creare un nuovo modello mentale che è un mix dei due. L'utente non si chiederà il perché del nuovo modello, quindi se gli diamo la possibilità di scegliere andrà sul sicuro, cioè il vecchio sistema. Quando il vecchio sistema verrà tolto, l'utente entrerà in crisi.

8.1 Rethinking OS

Mercury OS (link slide)

Tanta innovazione abilitata da richiesta per consentire accesso a categorie meno fortunate si è trasformata nell'ottimizzazione dell'intero paradigma. es sottotitoli prima per sordi ora tutti i video sono sottotitolati. Altro es riscrivere un'interfaccia per deficit attenzione, diventa meno faticosa anche per gli altri.

Reinventare SO facendo sì che comunicazione tra applicazioni esista e sia fluida (nella realtà l'interoperabilità fra le app si crea grazie all'utente)

PEnsa SO a chi usa pc come strumento di lavoro, computer come sistema di ingresso uscita, interfaccia verso il proprio lavoro. SO in cui l'elemento cardine non siano finestre ma il tempo. Flussi composti da moduli, elemento atomico base. In uno spazio diversi flussi, flussi diversi condividono moduli. il concetto di app rientra nel concetto di spazio. spazi/flussi generabili con moduli taggati.

nomi oggetti (cose tangibili, funzionali), verbi affordance (cosa posso fare con nome), modifiers (quasi)significanti (azioni specifiche che consentono di attuare quei verbi e ottenere risultati)

9 Constraints, Discoverability and Feedback

9.1 Constraints

Ricordando che *non si può progettare LA UX* ma *si progetta PER la UX*, **non si può vincolare** più di tanto l'utente a fare specifiche azioni.

Vincolare I **constraint** in un certo senso vanno esattamente contro la precedente frase. Si possono usare i constraint perché quando l'utente vede la nostra interfaccia per la prima volta si fa un modello mentale **mischiando la propria conoscenza pregressa**. Si limita quindi l'utente nella libertà d'azione, impedendo eventuali affordance e signifier percepiti grazie alla sua conoscenza.

Il **vincolo** più famoso è la **pila stilo**: si può inserire in un solo verso, e se si prova ad inserirla al contrario si fa fatica a spingere la molla. Inoltre, pur presentando esempi chiari di affordance, significanti (+ e -, disegno ecc.), si è reso necessario un sistema di vincoli che evita l'inversione batterie.

Perché creare dei **vincoli fisici**? Perché non è detto che un determinato vincolo culturale sia onnipresente. Il **vincolo fisico è l'ultima spiaggia**. Per progettare una UX a volte è possibile usare anche solo vincoli.

Categorie di vincoli I vincoli che tipicamente si trovano in nelle interfacce si dividono in 4 macrocategorie:

Fisici: ad esempio un pezzo Lego che entra solo in un determinato verso, il tappo di una biro entra solo in un verso. Sono vincoli concreti del mondo fisico

Culturali: ad esempio la guida a destra, indossare la t-shirt sul torace

Semantici: vincoli relativi ai significati, ad esempio il limite di velocità (cioè la semantica del cartello stradale con il numero)

Logici

L'assenza di vincoli e mapping genera frustrazione.



Vincoli e mapping alle volte si confondono fra loro. Nell'esempio, posizionare gli interruttori in corrispondenza delle luci sulla piantina della stanza è un mapping così forte che è quasi un vincolo logico: non ti puoi permettere di sbagliare interruttore.

9.2 Forcing functions

Forzare le funzioni è una forma di **vincolo fisico**.

Interlock: azione **dopo serie di passi**

Lock-In: azione **prima di concludere**

Lock-Out: azione **prima di iniziare**

9.2.1 Interlock

L'**interlock** consiste nell'**obbligare l'utente ad eseguire una successione di azioni** per raggiungere un certo stato/obiettivo (ad esempio premere pedale e due pulsanti distanti per azionare una pressa idraulica) oppure anche per guidare il learning (ad esempio un tutorial).

Esempi "Verifica la tua mail" prima di poter accedere con un account appena creato.

9.2.2 Lock-In

Con un **lock-in** invece **metto in pausa l'attuale situazione fino a che l'utente non ha fatto una determinata cosa**.

Esempi Non puoi uscire dall'editing di un documento se consciamente non mi hai detto di salvarlo o buttarlo.

A volte una lock-in **funziona così bene che diventa una shortcut**: chiudo e uso la finestrella di lock-in per salvare, invece di andare col mouse a premere tasto in alto a sinistra.

9.2.3 Lock-Out

Una **lockout** è l'opposto della lock-in, cioè **chiude fuori l'utente finché non compie una determinata azione**

Esempi Finestra *v.m.* 18, assolve alla richiesta legale di vietare l'accesso ai minorenni. Io utente posso **dichiarare il falso** ed entrare comunque, ma così il reato l'ho fatto io e non il sito.

9.3 Activity-Centered Controls

In molti casi è comodo avere **controlli associati alle attività** piuttosto che alle funzioni. Particolare modalità di interazione utente che usa tutti i principi precedentemente elencati.

Un esempio Invece di avere un pulsante "abbassa telo", un altro pulsante "alimenta proiettore", un altro "accendi proiettore" e così via, posso fare un pulsante "**presentazioni con slide**" che esegue tutte quelle operazioni. Altro esempio sono i preset per le impostazioni di schermi e audio: si possono ottenere i medesimi risultati manualmente. Il preset lo rende semplicemente più veloce.

Teoria vs Pratica Nella teoria le activity-centered controls sono eccellenti, ma nella pratica sono difficili da costruire bene. Se fatte male, creano **difficoltà**. Per questo **le activity-centered controls devono essere disegnate sugli utenti** e non sullo sviluppatore.

L'elettricista programmerà il pulsante "presentazioni con slide" con la sua idea di presentazione: probabilmente questa idea sarà super completa ma **talmente specifica che non andrà bene a nessun utente**. Invece **bisogna fare user activity-centered control**: tutta la parte sullo HCD si applica anche qua, cioè chiedersi chi è l'utente, le sue caratteristiche e così via. Ciò deve portare a **creare quella scena di presentazione che abilita tutti gli utenti**. Eventualmente per qualcuno andrà bene così com'è, mentre per un altro potrebbe essere incompleta e premerà altri pulsanti.

Un altro errore comune è creare un'apparente activity-centered control ma in realtà creare device-centered control.

10 How People do Things

Come fanno le persone a fare cose? Finchè le cose vanno bene allora sembrano essere semplici, l'utente crede di aver chiaro come ha ottenuto un risultato e allo sviluppatore sembra chiaro come l'utente esegue le attività. Invece, **quando le cose vanno male allora l'interpretazione è opposta**, cioè l'utente non capisce perché siano andate male e diventa frustrato: questo accade perché **il modo in cui le persone fanno le cose è complesso**. I fenomeni mentali in atto quando si fanno le cose sono complessi.

Quando le cose vanno bene non ce ne rendiamo conto, ma **quando ci sono problemi ce ne rendiamo conto in una serie di fasi**, che sono le fasi eseguite durante l'ottenimento di un obiettivo.

Come ottenere un obiettivo Si dà per scontato che le persone eseguano delle azioni, ma **prima di tutto persone scelgono le azioni da compiere**. Si dà per scontato che l'utente scelga l'azione scelta dal progettista.

10.1 I Golfi



Immagino l'utente e il mondo fisico/device contrapposti. Considero due "flussi": quello dell'esecuzione di una serie di attività e quello della valutazione della risposta ottenuta dal sistema.

I due flussi vengono descritti come due golfi, seguendo l'analogia che **tanto più profondo è un golfo tanto più lunga è la strada da percorrere per giungere dall'altra parte**.

- Nel **golfo dell'esecuzione** gli utenti cercano di capire cosa fare e, una volta capito, eseguono le azioni
- Nel **golfo della valutazione** l'utente interpreta e valuta il feedback ricevuto dal sistema

Il golfo della valutazione è ritenuto dai designer il più semplice da superare. Nel golfo della valutazione c'è una quantità di effort mentale che l'utente usa per comprendere l'interfaccia e capire se l'azione che aveva deciso di fare è stata eseguita come da lui pianificato. **Tanto più il modello mentale differisce da quello concettuale, tanto più il golfo valutazione sarà grande.**

Mitigare Gli elementi del design che contribuiscono alla mitigazione del **golfo dell'esecuzione** sono: **significanti**, **constraint**, **mapping** e **modello concettuale**.

Gli elementi che contribuiscono a alla mitigazione del **golfo della valutazione**: **feedback** e **modello concettuale**.

Fare ed Interpretare Le **due parti di un'azione** sono: **fare** l'azione e **valutare** i risultati. In entrambe le parti bisogna **garantire l'understanding**, cioè la **discoverability** e la **visibility**: ad esempio se non so quando ottengo punti ma vedo aumentarli apparentemente a caso non posso capire perché li ottengo se non è facilmente discoverable.

10.2 Sette Stati dell'Azione

Sono i sette stati attraverso i quali passano i due golfi:
Per prima cosa **(1)** si **specifica il proprio obiettivo**.
Dopodiché si passa ai 3 stati dell'esecuzione:

2: Pianifico ciò che devo fare

3: Specifico la mia pianificazione in task

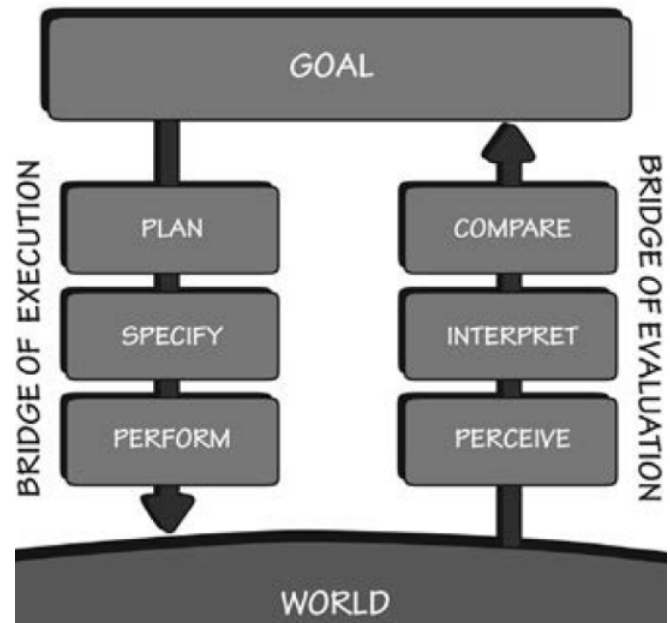
4: Eseguo i vari task pianificati: click del pulsante, riempimento dei campi...

Anche la valutazione ha 3 stati:

5: Percepisco cosa è accaduto (il feedback)

6: Interpreto ciò che ho percepito. Non è detto che io abbia lo strumento corretto per interpretare o che il feedback sia sufficiente da essere interpretato

7: Comparo il risultato dell'interpretazione con il mio obiettivo, che non devono differire



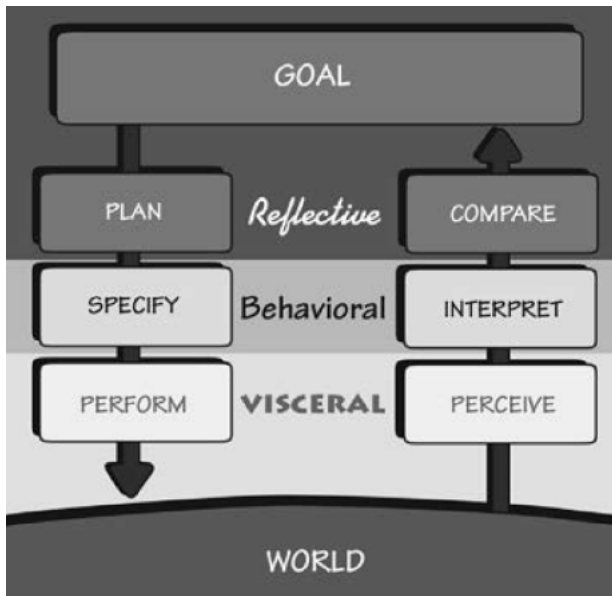
Goal → Plan → Specify → Perform → Percieve → Interpret → Compare

Non è detto che tutti i comportamenti debbano passare da tutte le fasi sopra descritte. Per esempio, l'abitudine porta ad abilitare altre parti cervello e non rende più necessaria la specify o consente di saltare la pianificazione, perché al goal associo già la sequenza di azioni più veloce che conosco già.

Un'altra situazione è il caso in cui il feedback non si possa comparare al goal, che quindi abilita nuovo processo. Per esempio quando vado a concludere un acquisto ma non sono registrato, il feedback è "inserisci i dati per la registrazione" che non è compatibile con il mio obiettivo "acquista prodotto" → Inizia un nuovo processo)

Il trucco è sviluppare delle skill per utilizzare questo paradigma di analisi interazioni per capire se le nostre interfaccia performano bene o no.

10.2.1 Tre livelli di Processing



Gli stati dell'azione possono essere associati con tre livelli di processing mentale:

Viscerale: **comportamento basilare**, istantaneo e quasi del subconscio. *Clicco qua, vedo lo schermo.*

Comportamentale: guidato dalle **aspettative** durante l'esecuzione e guidato dalle emozioni durante l'attesa di conferma di tali aspettative. **Decide in che modo faccio** un determinato task e **in che modo interpreto** un determinato feedback.

Riflessiva: relativo alle emozioni, che valuta i risultati in termini di presunti agenti causanti e le loro conseguenze. Essendo la **parte più emotiva**, è qui che avviene la soddisfazione e l'orgoglio, ma anche la frustrazione e il senso di colpa.

11 Sette Principi Fondamentali della Progettazione

11.1 Sette Domande



Il modello dei sette stati dell'azione può essere un utile strumento di progettazione, poiché **fornisce una checklist basilare di domande da porsi** durante il design. In generale, **ogni stato dell'azione richiede strategie di design specifiche** e offre le proprie opportunità di disastro. Derivano quindi 7 domande, a cui **dovrebbe poter rispondere chiunque** stia usando quel prodotto.

- **Cosa voglio ottenere?**

Ad esempio, dalle personas individuate genero come obiettivo "voglio selezionare le scarpe divise per modello" → vedo che nell'interfaccia non ho inserito la scelta per modello.

- **Quali sono le sequenze d'azione alternative** che posso compiere per raggiungere comunque l'obiettivo?

- **Che azione posso fare ora?**

- **Come posso fare quest'azione?**

- **Cosa è successo?**

- **Cosa significa?**

- **Va bene? Ho raggiunto il mio obiettivo?**

11.2 Feedforward e Feedback

Feedforward Tutta l'informazione che aiuta a rispondere alle domande riguardo l'esecuzione, quindi l'insieme di affordances, significanti e mapping.

Feedback Tutta l'informazione che aiuta a **capire cos'è successo, macchina → uomo**

Feedback e feedforward vengono realizzati correttamente attraverso i **sette principi di design**. Occorre far notare che i sette principi non sono mappati uno ad uno con i sette stati del processo d'interazione: il fatto che siano entrambi *sette* è una casualità, non perché sono correlati. I sette principi costituiscono una "*checklist da fare prima del commit*".

11.3 Sette Principi

1. **Discoverable**: il design deve **mettere utente nella posizione di capire cosa può fare** e capire lo stato del dispositivo con cui interagisce
2. **Feedback**: un sistema che dà feedback sbagliati, non esplicativi, in ritardo, che annoiano ecc. è un sistema che ha problemi. Deve esserci un flusso continuo di informazione sui risultati delle azioni e sullo stato attuale del prodotto/servizio: **deve essere facile determinare il nuovo stato dopo aver compiuto un'azione**.
3. **Modello Concettuale**: il modello concettuale che ho pensato è percepibile? Regge? Lo capisco solo io o tutti? Il design deve **proiettare tutte le informazioni necessarie a passare il corretto modello** all'utente.
4. **Affordances**: il dispositivo/interfaccia abilita la proprietà dell'interazione **con le categorie di utenti con cui sto lavorando**. L'affordances **esiste laddove è percepibile**, se non so se lo schermo è touch o no potenzialmente per me l'affordances non esiste.
5. **Significanti**: importanti perché **se usati bene abilitano la discoverability funzionalità** e la corretta percezione ed interpretazione dei feedback.
6. **Mapping**: ho messo significanti che va contro le capacità mentali utente? Un buon mapping è quello naturale.
7. **Constraint**: usati con parsimonia, posizionare vincoli fisici, logici, semantici e culturali **guida le azioni e facilita l'interpretazione**

Con questi sette principi si conclude la parte dedicata a strumenti, metodi ed elementi per il design dello human-computer interaction.

11.4 Opportunismo

Non sempre gli utenti eseguono delle azioni deliberatamente pianificate, ma alle volte eseguono **azioni di tipo opportunistico**.

L'opportunismo **rompe** lo schema d'interazione Obiettivo → Mondo.

Azioni opportunistiche Queste azioni sono quelle in cui il **comportamento scaturito dalle circostanze prevale sulle pianificazione** e quindi su modo di essere, di conseguenza **prevarica tutto il discorso precedente**.

Ci sono quindi situazioni in cui può accadere che abbia progettato tutto bene seguendo i principi, ma l'utente non si comporta in un modo che avevo previsto. Questo **può succedere**, perché gli **utenti agiscono in maniera incontrollata specialmente se guidati dalle opportunità**.

12 Disruptive Innovation

La **maggior parte dell'innovazione è fatta migliorando incrementalmente prodotti già esistenti**. La **disruptive innovation** invece riguarda le **idee radicali**, cioè l'**introdurre nuove categorie di prodotto nel mercato**.

Innovazione Lineare Si migliora il sistema esistente: abbasso i consumi, miglioro l'efficienza...

Disruptive Innovation Si tratta più di un cambio di binario.

Ad es. prima si guardavano i film andando fisicamente a prenderli a noleggio, mentre adesso basta accedere ad un catalogo già "presente nel nostro televisore". **Cambia il modello di business.** Ciò che nasce da un processo disruptive tipicamente fa fallire i metodi di business classici.

"La gente non vuole un trapano di 5 millimetri, vuole un buco di 5 millimetri" – Theodore Levitt

Metodi Ci sono dei metodi per arrivare alla disruptive innovation, non la si "sogna". Il cliente del nuovo modello di sistema e di business non è tipicamente conosciuto, così come il mercato: impossibile eseguire la user experience analysis.

12.1 Root Cause Analysis

Una volta che si è capito che l'utente non vuole il trapano, magari ci si accorge che **non vuole nemmeno un buco ma vuole installare la libreria: perché non sviluppare metodi che non richiedono buchi nel muro? O ancora, libri che non richiedono librerie?**

La **RCA** è, letteralmente, l'analisi dell'origine delle cause. La gente non vuole uno scaffale ma desidera un **modo per contenere i libri** → e-Book Reader.

Si pensa a **cosa fa** l'utente e **perché lo fa**, non per migliorare il libro. Il libro è un contenitore di testo.

Nella **disruptive innovation** viene scomposta la **RCA**.

12.1.1 Why root cause analysis

Il processo di RCA più tipico, nel quale **non ci si ferma a dire cosa è successo, si vanno a comprendere le cause alla base del perché è capitato.**

La root cause analysis produce la **task analysis**: la RCA produce la **serie di attività** (task) da eseguire per raggiungere un obiettivo.

Questa analisi può essere suddivisa in 4 passaggi:

1. **Identificare e descrivere** chiaramente il problema.
Ad esempio: spostare le persone da un punto A ad un punto B
2. **Creare una timeline** che porti **sequenzialmente** dalla identificazione del bisogno al suo compimento.
Chiamo taxi → aspetto → si ferma → entro → dico la destinazione →...
3. **Distinguere le vere cause** (root cause) dai fattori collaterali.
Non c'entra la dimensione della pensilina d'attesa, è un problema locale.
4. **Creare un grafico di causa-effetto** tra il problema e la root cause

Perché serve? Avere una struttura così solida aiuta a discutere in azienda, in team..., idee su come innovare. Si discute sulla forma e sul metodo ma non su come andare dal punto A al punto B. **Questo aspetto unico per tutti va scritto.**

I task estratti sono di **tipo utente** e **non tecnologico**, si parla della **user experience**. La task analysis è il processo di apprendimento di un fenomeno fatto tramite l'esperienza, **l'osservazione di come sono eseguite le attività in esame.**

Si consiglia di **estrarre più task e il più dettagliati possibile** al fine di:

Comprendere nel profondo gli **obiettivi**

Cosa fanno gli utenti **per raggiungere** quell'**obiettivo**

Quale esperienza (personale, sociale, culturale...) l'**utente porta** all'attività

Come gli utenti **sono influenzati dall'ambiente** circostante

Quali conoscenze ed esperienze precedenti influenzano

ciò che pensano del loro operato

la sequenza di attività che seguono per portare a termine le proprie attività

12.1.2 Tipi di Task Analysis

Ci sono **due metodi** che danno **due tipologie** di task analysis:

Cognitive task analysis: concentrata sul capire i task che richiedono **decision-making, problem-solving, memoria, attenzione e giudizio**

Hierarchical task analysis: butto giù dei macro-task e li scompongo in sotto-task

12.1.3 Come fare la Task Analysis

Il processo per decomporre task di alto livello segue i seguenti passi

1. **Identifica** il task da analizzare
2. **Scomponi** questo task in **4–8 sottotask**.
I sottotask dovrebbero essere specificati in termini di obiettivo e, tra tutti, **coprire l'intera area d'interesse**.
3. Disegna un **diagramma a strati** per ogni sottotask, assicurandoti che sia completo
4. Scrivi un resoconto e la decomposizione del diagramma
5. **Presenta l'analisi a qualcuno** che non sia stato coinvolto nella decomposizione ma che conosca l'attività abbastanza bene da verificarla.

12.1.4 Livello di dettaglio

Quando mi fermo? Non c'è una risposta, è un'arte. Non c'è un metodo o una formula. Richiede esperienza e buon senso.

Sarebbe bene **definire un criterio di arresto**, anche se all'inizio non si hanno gli strumenti per farlo. Agli inizi è meglio realizzare un'analisi più dettagliata possibile, poiché no esperienza sull'eliminare.

13 Agile

"Design thinking è come esploriamo e risolviamo i problemi;

Lean è il nostro framework per testare le nostre idee e imparare a ottenere gli obiettivi;

e Agile è come ci adattiamo ai cambiamenti nelle condizioni del software."

Non un semplice framework Si pensa che l'Agile sia semplicemente uno strumento di sviluppo, ma in verità l'**Agile è un modo di pensare** alla stregua del design thinking, fa parte della famiglia dei metodi di lavoro.

Il design thinking è come ci poniamo nella soluzione dei problemi

l'Agile è come applichiamo i metodi del design thinking allo sviluppo software.

L'Agile è una forma mentis, poi ad esempio lo scrum è una tecnica di organizzazione team **di tipo Agile**.

Individui Il focus è sugli individui e l'**interazione** (programmatore) invece che sui processi e sui tool (di sviluppo)
Software funzionante invece di grandi documenti

Capacità di **adattamento** al cambiamento.

13.1 12 principi dell'agile

<https://www.agilealliance.org/agile101/the-agile-manifesto/>

Il concetto più importante è quello della **adattabilità**. La mancanza di struttura e di documenti **non è anarchia**, anzi è molto complesso garantire al resto del team di aver compiuto qualcosa che non penalizzi i colleghi, in assenza di un "grande capo" che controlla la produzione.

13.2 Personas

Identificato correttamente e precisamente il problema (dalla **task analysis**), come creo gli elementi individuali?

Identificare le personas Il primo passo da fare dopo aver concluso la task analysis è **identificare le personas**. Una **personas** è l'**archetipo di uno dei nostri utenti**: *la casalinga di voghera, l'utente anziano, l'infante*. Identifichiamo le personas subito dopo la task analysis **perché durante di essa ci si rende conto che non tutti si comportano nello stesso modo**. Si rende **necessario creare degli archetipi**. Durante la macro task analysis, quindi, posso appuntarmi delle note da cui poi genero le personas.

Debugging Nel produrre le personas si **innesca un processo** durante il quale, **mentre scrivo le personas, affino la task analysis**.

Mettersi nei panni dell'utente Definire le personas mette a fuoco la userbase e **colma la distanza tra azienda e cliente**. Questo perché ci si prova a **immedesimare in un archetipo dei potenziali clienti, descrivendone la figura**. Questo è l'**unico modo per provare davvero a capire l'utente**.

13.2.1 Scrivere le personas

Creare una **personas** tipicamente parte dalla **user research**: come la task analysis, ci sono altre metodologie a partire dai **feedback** (questionari e focus group) fino ai **prototipi**, cioè sperimentando con le idee prima di svilupparle. Tipicamente le personas sono caratterizzate da un **goal**.

13.2.2 Quante personas scrivere?

Uno dei principi fondamentali al mondo è il **Principio di Pareto**: *l'80% degli effetti è generato dal 20% delle cause*. Anche la progettazione delle personas dovrebbe seguire questo principio. Bisogna concentrarsi sul **20% della userbase che userà/comprerà l'80% delle funzionalità/prodotti o che sarà responsabile dell'80% degli introiti**.

13.3 Requirements

Un **requirement** è una **peculiarità** (proprietà, servizio, funzione...) **che il mio sistema deve avere al fine di assolvere al need dell'utente**.

Persona \Rightarrow Need \Rightarrow Requirement

Oltre a funzionalità **possono anche essere dei constraints**: un utente minorenne ha come requirement un constraint su certi siti che richiedono un logout.

Lavoro complesso Scrivere requirements è molto **complesso**, oneroso e va **in contrasto col metodo Agile**: si richiede di **essere sempre pronti al cambiamento**, ma se faccio requirement-driven development si rischia di essere invece poco flessibili, per questo i **requirements stanno uscendo di moda**.

13.3.1 Tipi di requirement

I requirement sono divisi in:

Functional Requirements o FR: i requirement **funzionali** esprimono una **funzione o funzionalità che il sistema deve avere. Non esprimono come deve essere fisicamente ottenuta una soluzione.**

Esempi: *ottenere lo storico degli ordini, accedere al sito del cliente...*

Entrambi possibili in tanti modi, con database, link ipertestuali, frame embedded ecc.

Non-Functional Requirements o NFR: i requirement **non funzionali** definiscono **quanto bene, o a quale livello, una soluzione deve comportarsi.** Sono i cosiddetti **vincoli di qualità:** legislativi, di efficienza, sicurezza, formato... Descrivono quindi **gli attributi di una soluzione.**

Esempi: *rispondere entro 2 secondi, essere conforme alla GDPR...*

13.4 User Stories

Il prossimo passo è **la scrittura delle user stories.**

Cosa sono Una **user story** è una **breve descrizione che identifica l'utente insieme al suo obiettivo.** Determina **chi è l'utente**, di **cosa ha bisogno** e **perché ne ha bisogno.**

Tipicamente c'è **una o più user story per personas**, **non ci possono essere più personas per user story** – significherebbe l'aver introdotto troppe personas.

Se una user story non copre tutte sfaccettature della singola personas allora dovrei suddividere tale personas, perché forse ho definito una personas troppo ampia.

Una **user story** è un **requirement espresso dalla prospettiva dell'utente.**

Debugging Definire le user stories aiuta a **migliorare le personas.**

13.4.1 Come scrivere le user stories

Scrivere una user story è molto semplice

As a <role>, I want <feature> because <reason>.

I vare elementi sono presi dalla personas in esame. Notare che il role non è per forza la personas in esame, ma *potrebbe* esserlo.

Chi scrive le user stories **Chiunque** può scrivere le user stories. Nonostante sia responsabilità del product owner assicurarsi che siano definite le user stories, ciò non significa che sia necessariamente compito suo scriverle.

Durante lo sviluppo di un buon progetto Agile, **ogni membro del team** dovrebbe produrre una user story.

Inoltre, **chi scrive la user story è molto meno importante rispetto a chi è coinvolto nella sua discussione.**

Livello di dettaglio Uno dei vantaggi delle user stories è che **possono essere scritte a livelli di dettaglio variabili.** Si possono scrivere user stories che coprono ampi aspetti delle funzionalità, e sono chiamate **epics** o epiche.

Un esempio di **epic agile user story** di un software per backup: *"As a user, I can backup my entire hard drive."*

Come aggiungere il dettaglio Poiché una epic è solitamente **troppo grande per essere completata in una iterazione** da un team Agile, viene solitamente **divisa in user stories più piccole** prima di iniziare a lavorarci.

L'epica precedente può essere suddivisa in dozzine, o addirittura centinaia, di altre user stories, tra cui:

"As a power user, I can specify files or folders to backup based on file size, date created and date modified."

"As a user, I can indicate folders not to backup so that my backup drive isn't filled up with things I don't need saved."

Il dettaglio può essere aggiunto ad una user story in due modi:

Suddividendo una user story in diverse user stories più piccole.

Quando una user story relativamente grande viene suddivisa in diverse user stories più piccole, viene naturale pensare che si sia aggiunto dettaglio. Dopotutto, si è scritto di più.

Aggiungendo "condizioni di soddisfacimento".

Le **condizioni di soddisfacimento** sono semplicemente test di accettazione ad alto livello che saranno verificati dopo che la user story è completa.

"As a vice president of marketing, I want to select a holiday season to be used when reviewing the performance of past advertising campaigns so that I can identify profitable ones."

Posso **aggiungere dettaglio** a questa user story **aggiungendo le seguenti condizioni** di soddisfacimento:

- Assicurarsi che funzioni con le festività più importanti: Natale, Pasqua, Festa della Mamma. . .
- Supportare feste che avvengono a cavallo di due anni
- . . .

13.5 Scenarios

Uno **scenario** è l'evoluzione delle user stories.

Con una user story **sintetizzo in brevi frasi associate alle personas ciò che tali personas fanno con il mio software**. Lo scenario **estende a partire da questo**: inizia a catturare anche *come* l'utente si comporterà sul sistema. Questo "come" non è da intendersi nell'accezione dei task spaccettati, ma **più in generale**. Inoltre permangono i goal delle personas e li si estendono, venendo **descritti in maniera più ampia**. Gli scenario sono poi divisi in singoli use case.

A cosa servono? Le user stories sono sintetiche, brevi e dicono cosa sviluppare. Gli scenarios invece sono **utili per interloquire con gli stakeholder**. Questo perchè **per gli esterni al progetto le user stories non sono facilmente comprensibili**. Al contrario, gli scenarios **convogliano immediatamente l'immagine dell'obiettivo**.

Gli scenarios diventando quindi **fondamentali per allinearsi con le richieste del cliente**. Senza di essi si rischia di presentare un progetto finito senza che il cliente sia soddisfatto, poiché le user stories che avevamo definito non confluivano nel risultato che il cliente si auspicava. **Una volta approvati gli scenario ho raggiunto metà dell'opera**: non ho fissato i requirements, quindi **sono ancora flessibile agli eventuali cambiamenti**, e il **software che realizza gli scenarios approvati sarà sicuramente soddisfacente per il cliente**, perchè gli scenarios sono stati approvati da lui stesso.

Stakeholder Colui che è interessato all'output del progetto: cliente, capo, project manager. . .

13.5.1 Cosa tenere in considerazione

Un buon scenario è **conciso ma risponde alle seguenti domande chiave**:

1. **Chi è l'utente?** Usa le personas sviluppate
Es: *il tecnico*
2. **Qual è la motivazione che lo ha spinto ad arrivare da me?** Cosa motiva l'utente ad arrivare al mio sistema e le sue aspettative all'arrivo
Es: *perché sono un sistema che fa report automatico*
3. **Qual è il suo obiettivo?** Attraverso la task analysis, puoi comprendere meglio cosa voglia l'utente dal tuo sistema, quindi cosa deve avere il sistema per soddisfare l'utente
Es: *fare report automatico*

In alcuni casi uno scenario potrebbero includere aspetti di **come** fare cose, ma ciò appartiene al mondo degli **use case**. Sono preferibili scenarios privi di tale aspetto.

13.5.2 Come scrivere uno scenario

Prima di iniziare a definire gli scenario, è necessario **mapparli**. Bisogna partire avendo già le personas e le relative user stories, e individuare per ogni personas un **key task** che tale personas vuole ottenere.

Scrivo gli scenarios racchiudendo una o più user stories relative alla personas. Tipicamente per ogni personas descrivo almeno uno scenario.

Focalizzarsi sul key task Il key task diventa il task fondamentale di quello specifico scenario, ad esempio *l'utente che compra il biglietto*.

Gli scenarios quindi **contestualizzano i goal dell'utente**: il mio goal è **comprare il biglietto**, lo contestualizzo sul mobile, desktop, tablet, biglietteria...

Mindset Sviluppare gli scenarios richiede un **mindset dedicato** a risolvere questo problema. Nella scrittura di uno scenario è importante **riuscire a esprimere qual è il goal dell'utente e com'è influenzato dal contesto dello specifico utente**, dalla **prior knowledge** e dal **background** – evincendo ciò dalla personas.

Gli scenarios portano quindi le user stories al livello successivo e sono relativi all'**esperienza della personas, integrando con l'interazione che avviene tra l'utente e il prodotto o il servizio** della story.

<https://uxknowledgebase.com/scenarios-43e05671b07>

13.5.3 Tre metodi

Ci sono tre metodi principali per scrivere gli scenarios

(<https://www.usability.gov/how-to-and-tools/methods/scenarios.html>):

1. **Goal/Task Oriented**
2. **Elaborated Scenarios**
3. **Full Scale Task Scenarios**: da evitare poiché diventano del tutto uguali agli use cases.

Conviene **partire scrivendo uno scenario goal/task-oriented** e aggiungendo elementi di contesto.

Ricapitolando Parto dall'estensione di un paio di user stories.

Percorso di debug a partire dalle personas → trovo elementi di contesto o particolarità da aggiungere.

Senza troppo dettaglio.

13.6 Use Cases

Gli **use cases** sono l'**evoluzione degli scenarios**. In uno **use case** ho la **narrativa, in step monolitici**, che partono dall'obiettivo dell'utente e finiscono ad obiettivo raggiunto, **del comportamento del sistema in risposta alle richieste** visto dal punto di vista dell'utente.

Essendo una serie di step monolitici, gli use case **sono nuova task list, prodotta dal processo di design** per innovare un metodo visto e descritto in precedenza tramite la task analysis.

Un attore Uno scenario si concentra su una situazione. **lo use case** invece è **incentrato sulla personas**: *John e le cose che fa*. **Uno scenario si trasforma in un set di use case.**

La differenza principale è la **prospettiva**.

Esempio di scenario: *voglio ritirare una somma dal bancomat*

Use Case relativi:

- *Cliente della banca che possiede lo sportello*
- *Cliente di un'altra banca*

Diretti allo sviluppo Gli use case sono di fondamentale importanza perché **vanno direttamente al team di sviluppo**. Con degli use case ben fatti lo sviluppo è molto più facile e si riduce a semplice implementazione, poiché **ho già delineato precisamente cosa deve fare il sistema**, fornendo una lista di obiettivi che può essere usata anche per stabilire costo e complessità del sistema. Si può quindi anche negoziare quali funzioni entreranno effettivamente in sviluppo.

13.6.1 Elementi

Cosa è incluso in uno use case

L'utente
Cosa vuole fare
Qual è il goal finale
I singoli step necessari
I feedback del sistema durante gli step

Cosa non è incluso in uno use case

Qualsiasi dettaglio implementativo e di scelta tecnologica, che non sia un constraint o un requirement

Dettagli a proposito dell'interfaccia utente

A seconda di quanto livello di dettaglio si desidera raggiungere, gli use case descrivono una combinazione dei seguenti elementi

Attore – Chiunque o qualsiasi cosa che esegue un comportamento (che sta usando il sistema)

Stakeholder – Qualcuno o qualcosa interessato al comportamento del sistema

Attore Primario – Stakeholder che inizia l'interazione con il sistema per raggiungere un obiettivo

Pre/postcondizioni – Cosa deve essere verificato o deve succedere prima e dopo l'esecuzione dello use case

Triggers – Gli eventi che scatenano lo use case

Basic Flow – La **sequenza principale degli eventi**, ovvero la sequenza di azioni nel caso in cui niente vada storto

Alternative Flow – La **sequenza alternativa degli eventi**, sequenze di azioni che variano la basic flow e che accadono quando qualcosa va storto a livello di sistema

13.6.2 Come scrivere uno use case

<https://www.usability.gov/how-to-and-tools/methods/use-cases.html>

1. **Identificare gli utenti** (personas)
2. **Sceglierne uno**
3. **Identificarne i goal**
4. **Discernere i task principali** (basic flow) dagli altri.
Legge 80-20: **focalizzarsi sul basic path** perché l'80% degli utenti necessita del 20% delle features.
Devo sempre risolvere il basic path.
5. **Considera le sequenze alternative** e aggiungi quelle che estendono lo use case
6. **Cerca i punti in comune tra i vari use case** e accorpali, riducendone il numero ove possibile
7. Ripeti per tutti gli utenti

13.6.3 Takeaway

Se eseguiti correttamente tutti questi passaggi ti consentono di identificare informazioni chiave sui tuoi utenti, utili a costruire prodotti che soddisferanno i tuoi utenti spesso e volentieri.

Ogni passo che ci porta più vicino all'utente è un passo nella giusta direzione.

14 Human Error and Mitigation Strategies

La maggior parte degli incidenti nell'industria **sono causati da errori umani**: la stima varia tra il 75% e il 95%. Come mai **le persone sono così incompetenti**?

La risposta è che **non lo sono. Sono problemi di design.**

Perché? Progettiamo strumenti che richiedono alle persone di **essere attente e ricettive per ore ed ore** o di **ricordare procedure articolate e confuse** che spesso vengono usate solo saltuariamente.

Mettiamo persone in **ambienti noiosi**, senza **niente da fare per ore ed ore** e **all'improvviso** viene loro **richiesto di rispondere velocemente e accuratamente**.

Oppure le mettiamo in **ambienti complessi e con alto carico di lavoro**, dove vengono continuamente **interrotte mentre eseguono molte attività in contemporanea**.

Interruzioni Le interruzioni sono una comune causa d'errore. Questo non è mitigato dai design che spesso richiedono completa attenzione e non rendono facile la ripresa dopo un'interruzione.

Comportamento verso l'errore "*Abbiamo trovato il colpevole.*". Ciò non risolve il problema, e lo stesso errore si ripresenterà ancora e poi ancora.

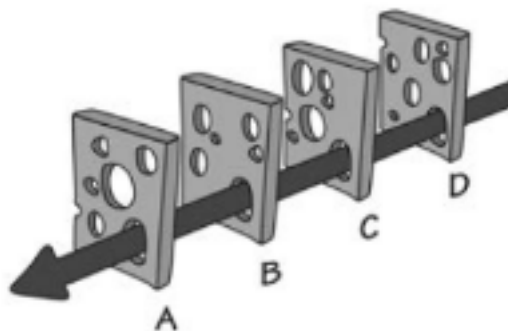
Invece, **quando si presenta un errore**, bisogna individuare il **perché** per poi **riprogettare il prodotto** o le procedure **in modo che l'errore non si possa più ripresentare** o almeno rendere il suo impatto minimo.

14.1 Root Cause Analysis

Investigare l'incidente fino a che non viene scovata la singola causa. Ciò significa che, quando le persone compiono decisioni o azioni sbagliate, bisogna **determinare cosa** ha portato la persona a sbagliare. Questo è lo **scopo della root cause analysis**, non fermarsi quando ha trovato la persona che ha sbagliato.

Multiple cause La maggior parte degli incidenti non ha una singola causa, ma spesso sono causati da **diverse cose che sono andate male**.

Questo è ciò che James Reason ha definito "*Modello degli incidenti a groviera*".



14.2 I 5 Perché

Origine In origine sono stati sviluppati da Sakichi Toyoda, usati dalla Toyota per migliorare la qualità dei propri prodotti.

Oggi Oggi sono ampiamente usati. In pratica, significa che quando si cerca una ragione non bisogna fermarci quando se ne trova una. **Bisogna chiederci perché.** Poi bisogna chiedercelo di nuovo. Continuare a chiedere finché non si trova la vera causa alla base. Non sono necessariamente 5, ma chiamare questa procedura "I 5 Perché" enfatizza la necessità di continuare a chiedere il perché anche dopo che una causa è stata trovata.

Problema: "Il veicolo non si accende"

Perché 1 → La batteria è morta

Perché 2 → L'alternatore non funziona

Perché 3 → La cinghia dell'alternatore è rotta

Perché 4 → La cinghia dell'alternatore era ben oltre il suo tempo di servizio e non è stata sostituita

Perché 5 → Il veicolo non è stato mantenuto secondo le tempistiche raccomandate

powernoodle.com/models-article/5-whys

14.3 Attitudine delle persone verso gli errori

Non si possono risolvere i problemi a meno che le persone non riconoscano la loro esistenza. Se si dà la colpa alle persone diventa **difficile convincere le organizzazioni** a ristrutturare la progettazione per eliminare questi problemi.

Perché le persone commettono errori? Perché i design si focalizzano sui requisiti del sistema e delle macchine e **non sui requisiti delle persone.** La maggior parte dei macchinari richiedono comandi e controlli molto precisi, forzando le persone a inserire le informazioni in maniera impeccabile.

Ma le persone non sono molto precise. Le persone sono creature creative, costruttrici ed esploratrici. Sono particolarmente brave a trovare novità, nuovi modi di fare le cose, e vedere nuove opportunità.

Requisiti noiosi, ripetitivi e precisi vanno contro questi tratti.

14.4 Errore

L'errore umano è definito come qualsiasi deviazione dal comportamento "appropriato".

La parola "appropriato" è tra virgolette perché **in molte circostanze il comportamento "appropriato" non è conosciuto** o è determinato solo dopo il fatto. Comunque, l'errore è definito come una deviazione dal comportamento generalmente accettato o appropriato.

Errore è il termine generico per tutte le azioni sbagliate.

14.4.1 Lapsus e Errori cognitivi

Ci sono due classi principali d'errore

Lapsus (slips)

Un **lapsus avviene quando una persona che intende fare un'azione finisce per farne un'altra.** Con un lapsus, l'azione svolta non è quella che si aveva intenzione di svolgere.

Lapsus di azione (action based): viene compiuta l'azione sbagliata

Esempio: ho versato il latte nella tazza e poi ho messo la tazza in frigo. Azione corretta applicata all'oggetto sbagliato.

Lapsus di memoria (memory lapse): la memoria fallisce, quindi l'azione che si intendeva compiere non è compiuta o i suoi risultati non sono valutati.

Esempio: mi sono dimenticato di spegnere il gas dopo aver preparato cena.

Errori cognitivi (mistakes)

Un **errore cognitivo** avviene quando viene stabilito il **goal sbagliato** o viene ideato il piano sbagliato. Da quel punto in poi, **anche le azioni corrette fanno parte dell'errore**, perché le azioni in sé non sono appropriate e fanno parte del piano sbagliato.

Con un errore cognitivo, le azioni che vengono compiute rispecchiano il piano: **è il piano ad essere sbagliato**.

Rule based mistakes: la persona ha diagnosticato correttamente la situazione, ma ha ideato un piano d'azione sbagliato. Viene seguita la **regola sbagliata**.

Esempio: un meccanico diagnostica un difetto sulla batteria della macchina ma decide di non sostituirla perché funziona ancora al 50% della capacità.

Knowledge based mistakes: la persona ha diagnosticato male il problema a causa di **conoscenza sbagliata o mancante**

Esempio: il peso del carburante viene calcolato in pound invece di chilogrammi.

Memory lapse mistakes: avvengono quando vi è una **dimenticanza** durante le fasi di **goal, plan o evaluation**.

Esempio: un meccanico si distrae e non porta a termine la diagnostica.

Slips: specify/perform, interpret/perceive

Mistakes: plan, compare

I nuovi utenti sono più portati a compiere **errori cognitivi** piuttosto che lapsus.

Gli utenti esperti compiono più lapsus.

Gli **errori cognitivi** spesso **nascono da informazioni ambigue o poco chiare** riguardo lo stato corrente del sistema, la **mancanza di un buon modello concettuale** e procedure inappropriate.

14.4.2 Prevenzione dell'errore

Non dovrebbe essere possibile, per un semplice errore, causare danno elevato. Ecco cosa si dovrebbe fare:

Capire le cause dell'errore e progettare in modo da minimizzare tali cause

Fare controlli di sensibilità. Le azioni superano il test del "buon senso"?

Rendere possibile annullare tali azioni – *undo* – o rendere più difficile fare ciò che non può essere annullato (es. "*Sei sicuro di...?*", lock).

Rendere più semplice la scoperta di errori da parte delle persone e rendere più facile la loro correzione.

Non trattare l'azione come errore, piuttosto aiuta l'utente a compiere correttamente l'azione. Pensa all'azione come un'approssimazione di ciò che si desidera fare.

14.5 Interruzioni

L'interruzione è una delle più grandi cause d'errore, specialmente di errori memory-lapse.

Quando un'attività è interrotta da qualche altro evento, il **costo dell'interruzione è molto maggiore della perdita di tempo richiesta per avere a che fare con l'interruzione**.

Per riprendere, è necessario **ricordare precisamente il precedente stato dell'attività**: qual'era l'obiettivo, a che punto del ciclo d'azione ero e lo stato rilevante del sistema.

La maggior parte dei sistemi rende difficile la ripresa a seguito di un'interruzione.

14.6 Feedback errati

Avvisi fastidiosi e non necessari si presentano in numerose occasioni. Cosa fanno le persone? Li disattivano, silenziosamente...

Il problema si presenta **dopo che questi avvisi vengono disabilitati**, sia quando le persone si **dimenticano** di riattivarli (lapsus di memoria) o se **un incidente differente avviene mentre l'avviso è scollegato**.

A quel punto, **nessuno se ne accorge**.

Avvisi e metodi di sicurezza **vanno usati con cura e intelligenza**, prendendo in considerazione i tradeoff per le persone coinvolte.

La progettazione degli avvisi è sorprendentemente complessa.

14.6.1 Voce come feedback

Un numero sempre maggiore di macchine offrono informazioni attraverso la voce, ma come tutti gli approcci anche questo ha pro e contro.

Pro Consente di fornire informazioni precise, specialmente quando l'attenzione visiva di una persona è diretta da qualche altra parte

Contro Se ci sono molti avvisi vocali contemporaneamente, o se l'ambiente è rumoroso, tali avvisi vocali possono non essere compresi. Oppure se sono necessarie conversazioni tra gli utenti o gli operatori, gli avvisi vocali interferiscono.

Gli avvisi vocali possono essere efficaci, ma solo se usati intelligentemente.

15 Design lessons

Dallo studio degli errori possono essere ricavate diverse lezioni sulla progettazione. Una per prevenire gli errori prima che accadano, e l'altra per rilevarli e correggerli quando avvengono.

15.1 Aggiungere vincoli per bloccare gli errori

La prevenzione spesso riguarda **aggiungere specifici vincoli** (constraints) **alle azioni**. Nel mondo fisico questo può essere realizzato attraverso un uso intelligente di **dimensioni e forme** (es. bocchettone della benzina / diesel). I sistemi elettronici hanno un'ampia selezione di metodi che possono essere usati per ridurre l'errore. Uno è quello di **segregare i controlli**, così che controlli facilmente confondibili tra loro vengano piazzati lontani uno dall'altro. Un altro è di **separare i moduli**, così che qualsiasi controllo non direttamente rilevante all'operazione corrente non sia visibile a schermo ma richieda uno sforzo extra per essere raggiunto.

15.2 Undo

Forse lo strumento più potente per minimizzare l'impatto degli errori è il comando "undo" nei sistemi elettronici moderni, che **annulla le operazioni effettuate dal precedente comando** ove possibile.

I sistemi migliori hanno più livelli di "undoing", così che sia possibile annullare intere sequenze di azioni.

15.3 Messaggi d'errore e di conferma

Molti sistemi cercano di prevenire l'errore **richiedendo conferma** prima di eseguire un comando, specialmente quando l'azione distruggerà qualcosa di importante. Ma queste richieste sono spesso mal temporeggiate, perché dopo aver richiesto un'operazione le persone sono solitamente certe di volerle compiere.

Un controllo migliore sarebbe visualizzare sia l'azione da compiere e l'oggetto affetto, con l'opzione "annulla" o "fallo". **I messaggi di avviso sono sorprendentemente inefficaci contro gli errori.** Cosa può fare il designer?

Rendere più evidente l'oggetto su cui si agisce: cambiarne dimensione, colore, apparenza per renderlo più visibile

Rendere l'operazione reversibile. Se la persona salva il contenuto, non c'è nessun danno oltre al fastidio di dover riaprire il file. Se la persona seleziona "non salvare", il sistema può segretamente salvare i contenuti e, al prossimo avvio, proporre di ripristinare lo stato precedente.

15.4 Controlli di sensibilità

I sistemi elettronici hanno il vantaggio di poter **controllare che l'operazione richiesta sia sensibile**. Ad esempio verificare che l'importo indicato sia nominale o considerevolmente più grande, quindi esporre eventuali avvisi.

15.5 Ridurre i lapsus

I lapsus avvengono più frequentemente in situazioni dove la mente è distratta, che sia da un altro evento o semplicemente perché l'azione compiuta è stata appresa così bene che è divenuta automatica. Come risultato, la persona non presta sufficiente attenzione all'azione o alle sue conseguenze.

Si può pensare che un modo per minimizzare i lapsus sia assicurarsi che le persone prestino sempre attenzione alle azioni svolte. **Sbagliato!**

Il comportamento "skilled" è nel subconscio, il che significa che è veloce, privo di sforzo e solitamente accurato.

Prevenire Molti lapsus possono essere **ridotti rendendo le azioni e i loro controlli il più differenti possibile**, o almeno fisicamente il più lontano possibile.

Il miglior modo per mitigare i lapsus è fornire un feedback percettibile a proposito della natura dell'azione in esecuzione, quindi feedback percettibile che descriva il nuovo stato risultante, insieme ad un meccanismo che consenta di annullare l'errore.

Per esempio: l'uso di codici a barre leggibile da macchine ha portato ad una significativa riduzione nella somministrazione di farmaci sbagliati ai pazienti.

Le scansioni aumentano il carico di lavoro ma solo leggermente. Altre tipologie d'errore sono ancora possibili.

16 Mitigare l'errore

La metafora del groviera suggerisce diversi modi per ridurre gli incidenti:

Aggiungere più fette

Ridurre il numero di buchi, o rendere più piccoli i buchi esistenti

Allertare l'operatore umano quando diversi buchi si allineano

Ognuna di queste ha implicazioni operazionali:

Più fette significa più **linee di difesa (controlli e verifiche)**

Ridurre il numero di punti critici dove può avvenire l'errore è come ridurre il numero o la dimensione dei buchi del groviera (**ridurre le distrazioni e progettare per evitare l'errore**)

16.1 Principi di design per gestire l'errore

Le persone sono flessibili, versatili e creative.

Le macchine sono rigide, precise e relativamente fisse nelle loro operazioni.

C'è un disaccordo tra i due.

Collaborazione Le difficoltà nascono quando non consideriamo che **le persone e le macchine sono sistemi collaborativi**, ma assegniamo qualsiasi task automatizzabile alle macchine e lasciamo il resto alle persone.

Questo finisce per richiedere alle persone di comportarsi come le macchine, in modi che differiscono dalle capacità umane.

Ci aspettiamo che le persone monitorino le macchine, e che **rimangano in allerta per lunghi periodi**. Ma le persone non sono brave in questo.

Errore umano Ciò che definiamo "**errore umano**" spesso è semplicemente un'azione umana inappropriata per i bisogni della tecnologia. Come risultato, **segnala una mancanza nella nostra tecnologia**. Non dovrebbe essere considerato un errore.

Bisognerebbe **eliminare il concetto di errore**. Invece, dovremmo realizzare che **le persone hanno bisogno di assistenza nel tradurre i propri obiettivi** nelle forme appropriate per la tecnologia.

I migliori design tengono ciò in considerazione e cercano di **minimizzare le opportunità di errore**, al contempo **mitigandone le conseguenze**. Si assume che ogni possibile sbaglio possa avvenire, e ci si protegge di conseguenza.

16.2 Principi chiave di design

Rendere disponibile al mondo la conoscenza richiesta per operare la tecnologia.

Non richiedere all'utente di tenere a memoria tutta la conoscenza necessaria. Consentire l'uso efficiente una volta che le persone hanno imparato tutti i requisiti, cioè quando sono esperti che possono operare senza fare affidamento sulla conoscenza nell'ambiente, ma rendere possibile ai non esperti di usare la tecnologia nell'ambiente. Questo aiuta anche gli esperti che devono eseguire operazioni rare e infrequenti, o che ritornano dopo un periodo di lontananza.

Usare il potere dei vincoli naturali e artificiali: fisici, logici, semantici e culturali. Sfruttare il potere dato dal forzare le funzioni e dal mapping naturale

Mettere in comunicazione i due golfi, quello dell'Esecuzione e quello della Valutazione. Rendere le cose visibili, sia per l'esecuzione che per la valutazione.

Per l'esecuzione, fornire informazione feedforward, rendere le opzioni facilmente disponibili.

Per la valutazione, fornire feedback, rendere evidenti i risultati di ogni azione.

Rendere possibile determinare lo stato del sistema prontamente, facilmente, accuratamente e in una forma consistente con gli obiettivi, i piani e le aspettative dell'utente

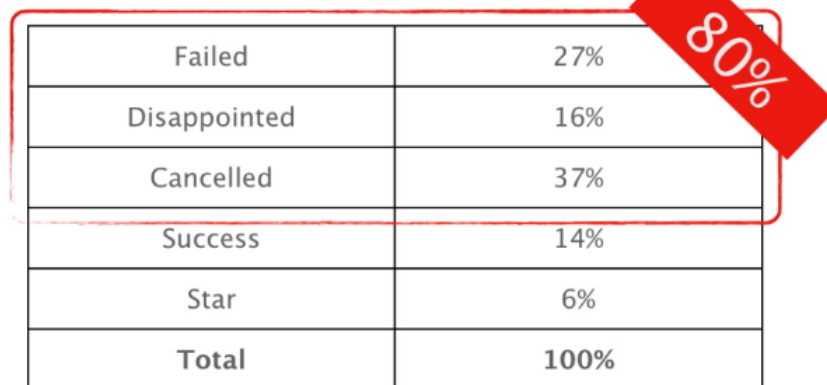
17 Pretotyping

Pretotype It: *assicurati di star costruendo bene prima di costruirlo bene.*

Ogni anno le compagnie lanciano migliaia di nuovi prodotti di tutti i tipi e in tutti i mercati, con ogni team che crede e spera che il proprio sia il prodotto definitivo. Tutti questi lanci sono monitorati e seguiti da compagnie di ricerca di mercato.

Una di queste compagnie, la Nielsen, fa questo lavoro da tanto tempo. Ecco un sommario da un report annuale:

One year: 24,543 new products*



Failed	27%
Disappointed	16%
Cancelled	37%
Success	14%
Star	6%
Total	100%

* Nielsen

La Legge del Fallimento di Mercato La maggior parte dei nuovi prodotti fallirà nel mercato anche se vengono eseguiti in maniera competente.

Nella legge, una persona è considerata innocente fino a prova contraria. Nella legge di mercato, bisogna supporre ogni nuovo prodotto come un fallimento, almeno fino a quando non abbiamo collezionato abbastanza prove oggettive che ci permettano di pensarla diversamente.

17.1 Thoughtland

Il "Lost in Translation Problem" Un'idea è un'astrazione soggettiva. Qualcosa che puoi immaginare o visualizzare nella tua testa. Nel momento in cui provi a comunicare ciò che vedi nella tua mente a qualcun altro incontri un problema di traduzione, specialmente quando la tua idea è nuova e diversa da qualsiasi altra cosa abbia visto il tuo interlocutore. Il modo in cui immagini un nuovo prodotto e i suoi usi può essere completamente diverso da come loro lo immaginano.

Il problema della predizione Anche se la comprensione astratta della tua idea che il tuo pubblico ha è molto vicina alla tua, le persone sono notoriamente pessime a prevedere se qualcosa lo vogliono o gli piacerebbe senza prima provarlo, o se e come lo userebbero.

17.2 Falsi Positivi

Webvan, fine anni '90.

Ha incassato più di 1Mld\$

Bancarotta a Luglio 2001

Motorola Iridium, telefonia via satellite.

6Mld\$ per 66 satelliti

Segway

180M\$ di investimento

Google Wave, piattaforma di messaggistica istantanea

20-30M\$

John Carter della Disney

275M\$ + 100M\$ di marketing

New Coke, Pepsi Clear

50M\$ stimati

17.3 Falsi Negativi

SMS

Twitter

Google

Xerox PARC

17.4 Thought Without Data Are Just Opinions

I **falsi positivi** possono **portare a credere che la tua idea sia immune alle Legge del Fallimento di Mercato**, quindi a investire troppo e troppo presto in un prodotto nuovo che fallirà.

I **falsi negativi** possono **spaventare e far evitare di dare una chance alla tua idea**, finendo per scartare prematuramente il prossimo Twitter, Google o Tesla.

Dati Per **minimizzare le possibilità** di ottenere falsi positivi o negativi devi **ottenere qualcosa di più sostanzioso e oggettivo delle opinioni**. L'unico modo è **portare la tua idea dalla thoughtland ad un ambiente più concreto chiamato actionland**.

Toughtland Usi le tue **idee astratte** per porre **domande ipotetiche** e ottenere **opinioni**.

Idee \longrightarrow Domande \longrightarrow Opinioni

Actionland Usi **artefatti** per scatenare **azioni** e collezionare **dati**.

Artefatti \longrightarrow Azioni \longrightarrow Dati

17.5 Pretotyping Manifesto

Innovators beat ideas

Pretotypes beat productypes

Building beats talking

Simplicity beats failures

Now beats later

Commitment beats committees

Data beats opinions

17.6 Pretotype

I **prototipi** ti aiutano a **fallire più in fretta**, ma spesso **non abbastanza in fretta** o spendendo troppo. Più si investe in qualcosa e più diventa difficile lasciarla andare e ammettere che era la cosa sbagliata.

Una volta ottenuto un buon prototipo che funziona, è facile pensare di lavorare su di esso un po' di più e investire un po' di più: *"se aggiungiamo questa feature sono sicuro che la gente finalmente lo userà"*.

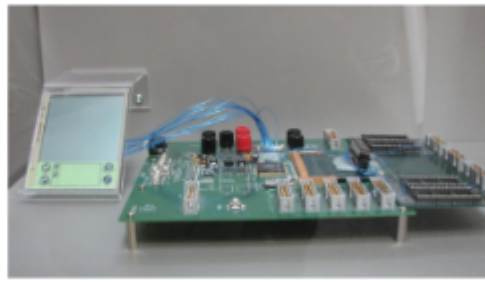
I prototipi spesso si trasformano in prodotti, un **prototipo andato troppo oltre**.

Pretotype Tra le idee astratte e i prototipi funzionanti ci sono i **pretotype**. Quest'ultimi **rendono possibile ottenere preziose informazione d'uso e di mercato**, per poter prendere decisioni su cosa fare e cosa non fare, **ad una frazione del costo di un prototipo**: ore o giorni invece di settimane o mesi, e centesimi invece di euro. Il **pretotyping** ti consente di **fallire in fretta e recuperare in fretta** ma lasciandoti comunque **molto tempo per esplorare nuove modifiche o idee** fino a che non trovi l'idea definitiva che le persone vogliono.

Pretotype



Prototype



Product



Un **pretotype** è un **mock-up del prodotto o servizio voluto** che può essere **costruito in minuti, ore o giorni invece di settimane, mesi o anni**.

Il pretotyping ha l'obiettivo di aiutare gli innovatori a:

Identificare le funzionalità chiave e l'esperienza chiave del nuovo prodotto

Decidere quali funzionalità chiave possono, e dovrebbero, essere inserite nel mock-up (o venire semplificate).

Usare i mock-up per **testare sistematicamente e collezionare feedback e dati d'uso**

Analizzare i dati d'uso per determinare il prossimo passo

17.7 I Sette Pilastri del Pretotyping

1. Obbedire alla Legge del Fallimento di Mercato
2. Assicurati di stare costruendo il prodotto giusto
3. Non perderti nella thoughtland
4. Fidati solo dei tuoi dati (**YODA**, Your Own DAta)
5. Fai i pretotype
6. Dillo coi numeri
7. Pensa globalmente, testa localmente

17.8 Flusso del Pretotyping

1. **Isola l'assunzione chiave.** Qual è quell'assunzione sulla tua idea che, se falsa, significa che non è assolutamente la strada giusta?
2. **Scegli un tipo di pretotype.** Quale tipo di pretotype ti permette di isolare e testare la tua assunzione chiave?
3. **Fai un'ipotesi di coinvolgimento di mercato.** Quante e quali tipi di persone faranno quello col tuo preotype? La tua ipotesi può essere qualcosa di semplice come "X% di Y farà Z". Un'ipotesi solida rimuove le opinioni dal testing.
4. **Testa il pretotype.** Ora metti il tuo pretotype nel mondo reale e vedi quante persone ci interagiscono. Parti dal basso: un posto, una volta.
5. **Impara, refinisci, hypozoom.** Valuta i tuoi risultati. Refinisci il pretotype con i nuovi dati. Se la tua ipotesi ha retto, decidi quali altre situazioni devi testare per ottenere una visione completa (**hypozooming**)

18 UX for Connected Devices

Come cambia il mondo della UX quando si parla degli oggetti interconnessi?

Nel momento in cui iniziamo a ragionare con i dispositivi IoT le cose cambiano notevolmente rispetto a ciò che è stato visto fin'ora. **Quando si approccia l'interfaccia di un dispositivo interconnesso si tende a seguire la filosofia classica**, studiata fin'ora, e a **concentrarsi sull'estetica del prodotto e della sua interfaccia**. Però questi elementi non sono sufficienti. Un prodotto interconnesso può avere un'**ottima UI ma pessima UX**. Questo perché un **prodotto IoT non è limitato né alla sua forma né alla sua interfaccia**, ma è **l'insieme delle due**. Quindi **conta tantissimo l'immagine di sistema** che risulta più complessa di un servizio o prodotto classico.

Internet Nato dalla necessità di scambiarsi **informazioni** velocemente. Informazione = dato elaborato. **Internet è un mezzo di comunicazione**.

18.1 Internet of Things

Internet è stato creato per **far comunicare gli umani**, perché **creare un mezzo di comunicazione per i sistemi**? Sembra quasi di voler creare un altro sistema di comunicazione.

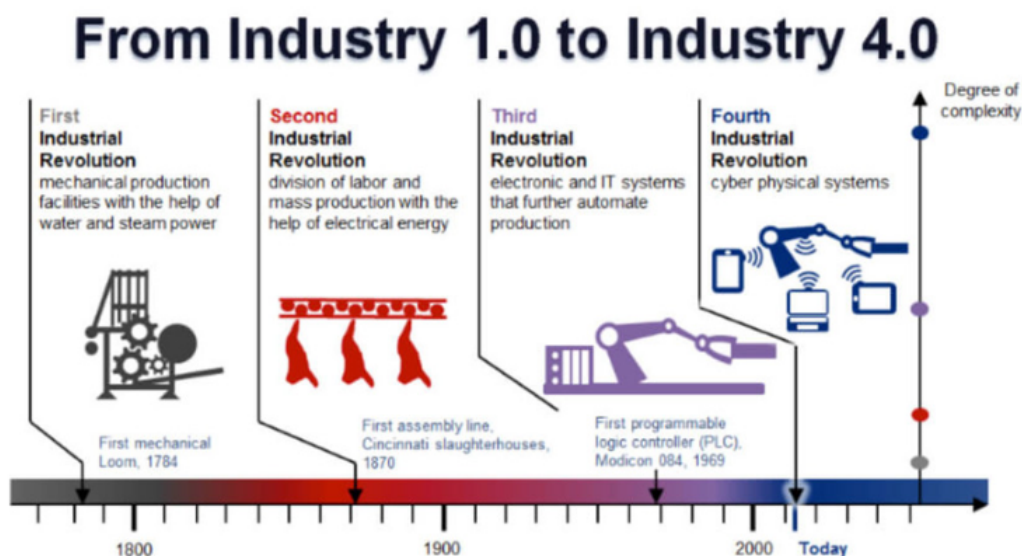
Quando si lavora agli IoT ci dimentichiamo la comunicazione umana: si creano sistemi che si interfacciano perfettamente tra loro ma che non comunicano efficientemente l'informazione agli utenti. Questo perché **peccano di UX**, il cui obiettivo sarebbe rendere facile e piacevole l'interazione all'utente, ma in molti casi ingrandiscono i golfi di valutazione ed esecuzione.

18.2 Industria 4.0

Quarta rivoluzione industriale: **smart factory**. La differenza con la terza è che prima c'era l'IoT tra sistemi **chiusi verso l'utente**. Nella 4.0 i sistemi sono collegati alle interfacce verso gli operatori, ai software gestionali ecc...e **comunicano attivamente con l'utente**.

Quindi nella 3.0 la produzione veniva pianificata in anticipo, mentre nella 4.0 può essere aggiornata in corso d'opera in base a dati e statistiche (**lean manufactory**).

Anche la **complessità informatica** è cambiata nel tempo. Sostituire l'asino col motore a vapore ha una bassa complessità. Con la produzione in serie e successivamente l'automazione ci sono stati dei grossi salti. Ma il cambiamento che avverrà con il 4.0 è pari al totale del cambiamento cumulativo delle 3 precedenti rivoluzioni. Questo perché **cambia completamente il paradigma industriale**, i **processi produttivi diventano servizi** e la **quantità di informazione aumenta**.



Perché è importante IoT nell'industria? Quattro motivi principali:

Si potranno **aumentare i profitti ottimizzando la produzione**

Si potranno **creare nuovi business model**

Sfruttare **le tecnologie smart per alimentare l'innovazione**

Trasformare la forza lavoro: l'operaio del futuro potrà modificare il processo, non solo l'azione. Ciò richiede interfacce.

18.3 Digital Twin

Il cuore di un sistema interconnesso, che sia industriale o consumer, è il **digital twin**: una **copia digitale di un bene/macchina/asset/processo fisico**. La **comunicazione** tra elemento reale e digital twin è **continua e bidirezionale**. Ogni azione, parametro... vengono spedite in tempo reale da macchina a digital twin, inoltre ogni modifica sul digital twin deve essere istantaneamente applicata sul dispositivo fisico.

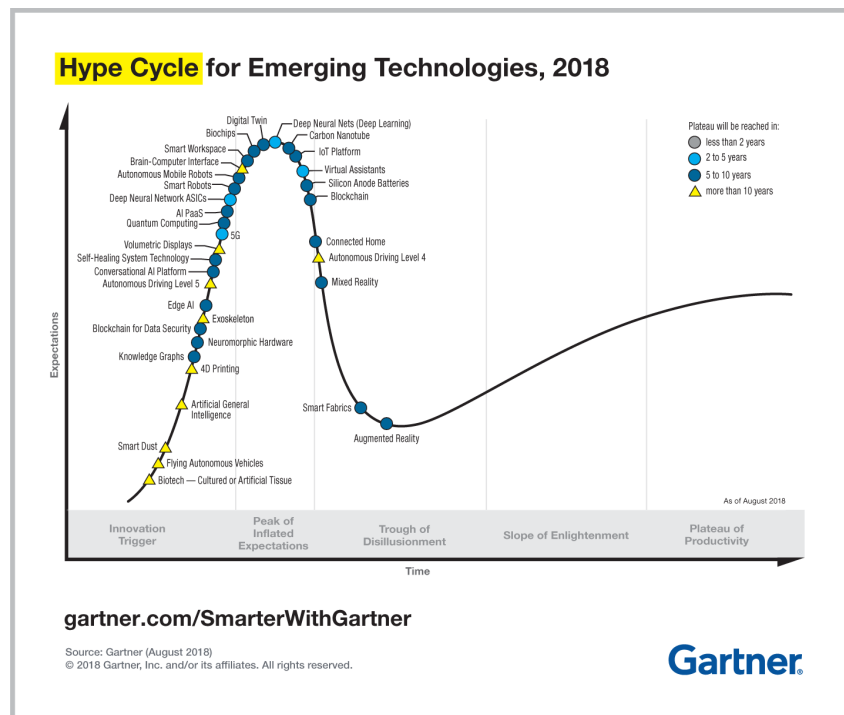
Simulazioni Il vantaggio del digital al twin è anche **simulare**. Prima per provare qualcosa (es. un nuovo modello da produrre) bisognava sfruttare una macchina fisica, potenzialmente bloccando la produzione e fare i test. Ora invece, tramite il digital twin, si hanno tanti dati a disposizione che consentono la creazione di un modello virtuale della macchina fisica. Trovati i parametri giusti, si possono trasferire subito sulla macchina e partire – risparmiando tempo e avendo solo bisogno di ultimi ritocchi dovuti alla distanza seppur piccola tra modello fisico e reale.

Manutenzione predittiva La manutenzione preventiva è ad es. ogni tot km il tagliando poiché statisticamente ogni tot+x km si rompe dato componente.

La **manutenzione predittiva** invece consiste nel **sfruttare sensori e dati dalla macchina per poter agire quando se ne presenta il bisogno**: es. un sensore che segnala che le vibrazioni sono aumentate del 5%. Si risparmiano tanti soldi.

Curva di Gartner Quando qualcosa supera la cima della curva è considerata **tecnologia stabile**: l'IoT è tra queste, quindi da un punto di vista tecnologico non appartiene più al futuro ma è una delle **comodities** quindi bisogna allargare la fetta d'utenza. Quindi bisogna pensare al prodotto come dispositivo di tutti i giorni.

Quindi **l'IoT è una tecnologia abilitante**. Non sviluppo l'IoT, ma sviluppo sistemi interconnessi che sfruttano l'IoT dando per scontato che un domani sarà tecnologia di tutti i giorni. Svilupperei "l'IoT" se stessi sviluppando i protocolli e i paradigmi in laboratorio.



A sinistra della curva è emergente: esce dai laboratori. A destra della curva è tecnologia di tutti i giorni.

19 Ubiquitous Technology e Big Data

Anni fa esistevano i computer come **centri di aggregazione tecnologica**, adesso i dispositivi sono sparsi per casa: Alexa, sensori CO₂ e soprattutto lo **smartphone**. **Intelligenza distribuita e ubiquitaria con interfaccia personale**. Questo cambiamento dal punto di vista UX è molto importante.

Non esisterà mai il "robot" che: laverà i piatti, porterà i figli a scuola e la colazione a letto. Invece avremmo: lavastoviglie smart, letto smart, pulmino smart, app della scuola per il rendimento...

Non si può più pensare ai dispositivi general purpose. Ma a dispositivi con **funzioni altamente specifiche veicolate da interfacce generiche**: smartphone, interfaccia vocale...

Come pensare? Bisogna pensare che **gli smart product sono servizi remoti emanati attraverso un'interfaccia fisica** che ottimizza l'interazione con l'utente.

Amazon Echo e Google Home sono **solo interfacce vocali**. Lo smart assistant è Alexa, ma lo si trova anche su smartphone, PC e così via. Echo è l'interfaccia di Alexa ottimizzata per il contesto domestico.

Ecosistema Quindi si ha un **ecosistema di servizi** (es: Google Nest) e **funzionalità erogati da dispositivi specifici** (es: Nest Camera) **attraverso una interfaccia generica per tutti**.

19.1 Dispositivi Smart

Un dispositivo smart, nel 2020, è un **prodotto che fa ciò che vuole l'utente senza che lui sappia come lo fa**.

Esempio Frullatore con un unico pulsante che si rende conto se c'è il ghiaccio, quindi fare un cocktail frozen, o se ci sono gli ingredienti per un frullato e **decide autonomamente velocità e pulsazione** è un frullatore del 2020.

Questo tipo di interazione è **molto semplice da progettare**: i golfi di valutazione ed esecuzione sono molto piccoli, praticamente non c'è niente da capire. La **gestione della UX invece è estremamente complessa** se fatta bene **perché deve tenere di conto le aspettative dell'utente**. Posso solo premere, se non va come voglio provo frustrazione.

Quindi progettare per IoT e dispositivi interconnessi è enormemente più complesso rispetto al classico approccio.

Non è ancora così ovvio per la base utente che gli oggetti siano interconnessi, ma lo è per il progettista perché IoT è ormai una delle comodities come visto prima.

Questi **aspetti di design fisico, UX e interconnettività non possono essere gestiti separatamente se fanno parte di un ecosistema**.

19.2 Perché UX per IoT è diversa?

Il primo problema tra tutti è che **gli oggetti IoT hanno una natura specializzata**. Questo è diverso dal sviluppare siti e app perché in quei casi si dà per scontato che l'elemento che veicola l'interfaccia sia general purpose. Qui invece si lavora con oggetti con natura altamente specializzata. Quindi sono **oggetti specifici che tagliano i riferimenti culturali e la conoscenza pregressa dell'utente in fatto di affordance e immagine mentale**. Quindi devo veicolare gli utilizzi.

Inoltre questi dispositivi fanno da **ponte tra mondo fisico e digitale**. Spesso inoltre sono anche distribuiti, formati da una moltitudine di dispositivi separati. Quindi l'interfaccia deve veicolare anche questa informazione: se la caldaia smart non va il problema potrebbe essere il relè dall'altra parte della casa.

Inoltre hanno **potenza di calcolo estremamente limitata**. Quindi si fa affidamento al **cloud** (quando c'è rete, collegamento...)

19.3 Sensori e attuatori

Le "interfacce" nel mondo dei dispositivi fisici sono i sensori e gli attuatori.

Sensore Ciò che converte una variabile fisica in un segnale elettrico, **input**

Attuatore Ciò che converte un segnale elettrico in una variabile fisica, **output**

Inoltre **non esiste l'undo nel mondo fisico**.

19.4 Contesto di utilizzo

Nel web il contesto di utilizzo è "di per sé": esiste un mondo che è quello della fruizione di internet. Si può assumere che l'utente sul nostro sito/app sia in una specifica modalità: psicologica, funzionale...

Ciò non vale per **il contesto dei dispositivi IoT, che è molto più specifico**. Es: il forno smart deve tenere in considerazione l'utente con le mani sporche e in mezzo alla preparazione di qualche piatto, con la mente da un'altra parte. La capacità cognitiva è ridotta e la **probabilità di errori e lapsus è notevolmente più alta**. Questo perché **l'interazione non avviene in un contesto dedicato all'interazione con la tecnologia ma è tutt'altro**, quindi **gli oggetti altamente tecnologici falliscono**. Bisogna creare oggetti molto semplici da capire.

Interusabilità Proprietà del sistema di **essere usato attraverso tutti i dispositivi o interfacce che lo compongono**. Oggetto fisico, app e sito web, ad esempio.

Importante perché è necessaria un'esperienza uniforme nel passare tra le varie interfacce. Questo perché l'utente usa queste interfacce quando le servono, dobbiamo consentire l'accesso il più facile possibile, dove vuole lui e il più vicino alla modalità dov'è in quel momento. Es: se mi collego all'account Nest da PC probabilmente voglio vedere più dati, perché magari è interessato a dedicare del tempo alla configurazione. Se invece interagisce col termostato nel corridoio non lo devo tormentare con tutti i dati ma solo quelli adatti a quel contesto, quindi posso nascondere determinate impostazioni – o rimuoverle e renderle disponibili solo da PC.

Inoltre **l'esperienza deve essere molto più univoca** e la **UI design fa molto più parte della UX rispetto a prima**. Questo perché a livello elettronico una determinata interfaccia può essere più difficile da realizzare, quindi è importante trovare un punto d'incontro tra interfaccia fisica e digitale e che siano il più simili possibile.

Network Bisogna anche considerare che non è sempre possibile dare per scontata la connessione alla rete, a differenza di contesti web e mobile.

Il problema si presenta in particolare con i feedback: se ho un ritardo sulla rete ho un ritardo anche sul feedback all'utente. Questo non è un problema facilmente risolvibile.

Progettare per la **mancanza di rete, rovesciare il paradigma** e pensare che internet è presente **ogni tanto**.

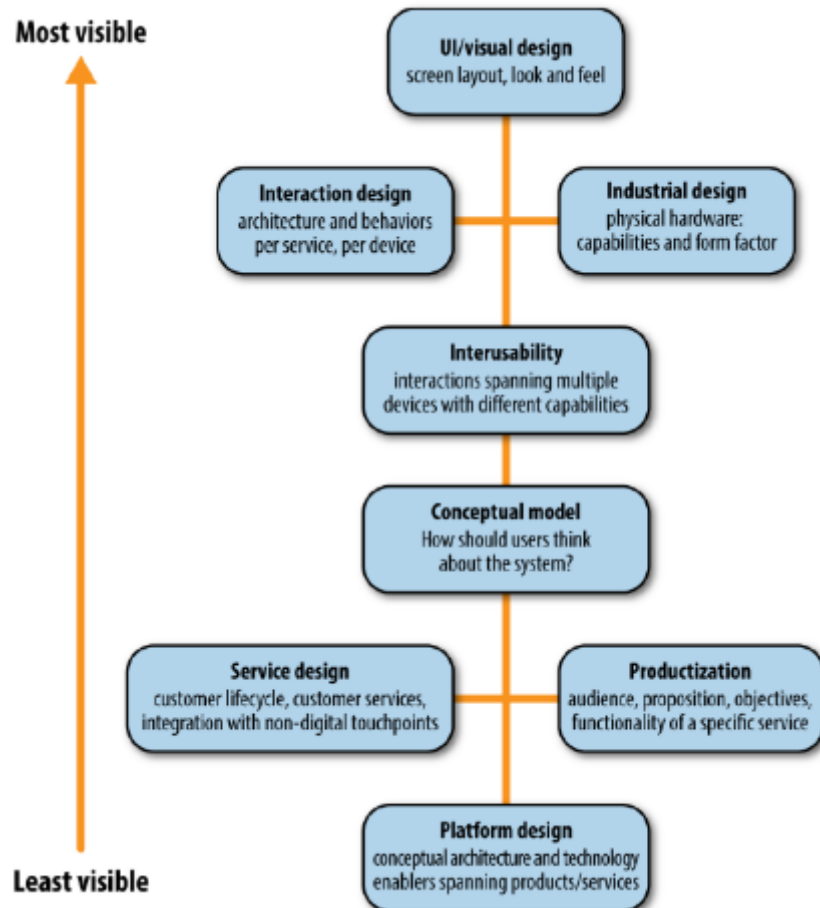
Power Saving Un dispositivo IoT sta spento e si accende quando richiesto. Ad esempio allo scadere di un timer alimento il dispositivo ed eseguo una serie di operazioni: leggo la temperatura, la converto ecc., incremento un contatore. Dopo tot volte faccio la media e spedisco il dato al cloud *se ce la faccio*, altrimenti la mantengo nel buffer.

Manutenzione La UX va considerata anche nel mondo della **manutenzione**. Per esempio se devo aggiornare un firmware OTA (over the air): il dispositivo può essere spento, necessitare di una versione leggermente diversa del firmware perché è di un modello in particolare ecc...

Productization Prodotti dello stesso brand con design simili: aiuta per immagine concettuale, di sistema...

Si collega anche agli abbonamenti: non per il prodotto ma per il sistema. Quindi non abbonamento per la IP camera ma abbonamento NEST che consente di collegare tot telecamere, tot termostati...

19.5 Diversi livelli di sviluppo della UX



20 Alberto Betella, Moonshots and Disruptive Innovation

Alberto Betella, PhD e CTO, Health Moonshot @ Alpha

googlex, moonshot: studi per innovazioni future a lungo termine. Telefonica studio per moonshot di salute, rivoluzionare la salute.

aloha.com company costruito per i moonshot.

terapie cognitive, esercizi dati alla persona giusta momento giusto posto giusto e usano sensori per estrarre info implicite nell'utente: es segni di pre-depressione, se esco tutti i weekend (rilevo da gps, celle telefoniche) ma poi smetto e non chiamo nessuno può essere segnale d'allarme.

21 Fabio Viola, Gamification

Game Designer

21.1 Gamification

Il modo di esprimermi in casa visto da poche persone, se spendo pochi euro per esprimermi in un gioco online lo vedono decine di migliaia di persone → status sociale.

Oggi **dover fare** viene meno rispetto al **voler fare**.

I software sono ormai quasi sempre personalizzati in qualche modo: prodotti suggeriti, post in bacheca... Non più produzione seriale (prodotti tutti identici, variazione viene eliminata) e utilità per azienda ma **utilità per l'utente** (Human Centered Design)

Engagement Centered Design: progetto per le persone ma anche per **conquistare il loro tempo**, *il tempo è denaro*. **Sistemi partecipativi**, story-doing contrapposto allo story-telling. Persona parte fondamentale e attiva in grado di modificare e alterare l'esperienza (dosi di coinvolgimento elevate, senso di protagonismo). **"Io** ho salvato la principessa" contrapposto a **"Arthur** ha sparato a spoiler". 1a persona vs 3a persona. Prodotto migliore se si adatta alle volontà dell'utente.

Progettare per le singole persone, nuove gen premiate per la singola partecipazione e non per il piazzamento (gen precedenti premiazione piazzamento primi 3). Molto più valore sulla collaborazione invece che competizione.

Dato microsoft 2013: media d'attenzione è 8s.

Libro: Small Data

Desiderio di completare muove grossa fetta di utenti. è un driver, tra gli altri: pressione sociale, paura di perdere... Storydoing: pubblico modella attivamente narrativa con decisioni anche a modificare il finale. Videogiochi: regno delle decisioni. Non funzionano senza persona che costantemente prende decisioni. Scelte significative sono quelle in cui persone sono: consapevoli, permanenti, con conseguenze e da fare ricordare.

sony robottino cane famiglia giapponese

creatività, coinvolgimento, collettivi (non si lavora più da soli), contaminazione (inutile 8 ingegneri, ma collettivi misti) *dimmi io dimentico, mostrami io ricordo, coinvolgimi io imparo.* – Benjamin Franklin

Libro: Bowling Alone, di Putnam.

22 Vincenzo Gervasi, Sottosistema Grafico

Dopo aver progettato interfaccia bisogna implementarla. Tra disp in e disp out (GUI) c'è la nostra interfaccia, da implementare.

Insieme alla nostra app c'è un sottosistema grafico che può essere primitiva del S.O.

Kernel e sottosistema grafico insieme → winzozz

Applicazione e sottosistema grafico insieme (S.O. non ha parte grafica) → unix <3

sottosist grafico fornisce serie di operazi da fare tra in e outp in forma digeribile per le applicazioni.

22.1 sottosistema grafico

Vari gruppi di operazioni

da qualche parte si prendono gli input: interfaccia esterna del sttosistema grafico che prende info da disp input
Kernel – Driver I/O implementati sul kernel – dispositivi input – sottosistema grafico

sottosistema grafico offre interfaccia a applicazioni

oppure il contrario applicazione offre interfaccia al sottosist. grafico (inversion of control) il sistema chiama l'app quando ha qualcosa da mostrare (push, appl aspettano che sistema chiami loro interfaccia)

Sottosistema grafico ha tante applicazioni da gestire (**multiplexing**).

Multiplexing in tempo dei disp input

Multiplexing in spazio bidimensionale del dispositivo di output

Segnale da input processato e finisce ad un componente chiamato **event manager**.

Event manager compatta informazioni relative al singolo evento in una struttura dati generata

struttura dati evento inserita in coda eventi e ho finito di processare l'evento

tutto questo è successo sul thread del S.O. che ha gestito l'interruzione del driver I/O

event loop componente che periodicamente estrae eventi dalla coda degli eventi e lo gestisce

la coda è importante perché crea disaccoppiamento temporale (momento in cui inserisco diverso dal momento in cui estraggo) quindi thread/processo che inserisce è separato da chi estrae. Coda importante anche perché consente in maniera pulita di gestire concorrenza perché si creano condizioni di racing solo all'accesso alla coda, ma sono facilmente gestibili.

22.2 modelli

a finestre, splitscreen, le tab dei browser, alt-tab (multiplexing schermo tra applicazioni schermo intero **nel tempo**)

22.3 event loop

```
while (!done) {  
    msg = cde.get(); //estraggo dalla coda degli eventi facendo i controlli sulla concorrenza  
    switch (msg) {  
        case KeyDown: if (isQualifier(msg.keyCode)) keyFlags |= flag(msg.keyCode)  
                        else if (isValue(...)) ...  
                        ...  
                        break;  
        case MouseMove ...  
        ...  
    }  
}
```

processing immediato, differito (mi segno dati imposto timer quando arriva agisco) o ignorati

X Server server che gira sul client che offre delle interfacce grafiche tramite il X11 protocol (protocollo di rete) per cui host (server) su cui girano applicazioni chiama funzioni dell'X server sul client che visualizza *cosa* sullo schermo.

X server fornisce istruzioni "accendi pixel a coordinate xy" quindi librerie da usare per astrarre.

librerie : java(awt, swing, gdt), obj-c, c++ ne hanno n. Qt, Cocoa su linux. W3c dom

programmazione ad eventi paradigma di programmazione distinto da a oggetti, imperativa, funzionale, logica,

dichiarativa. Senza multithreading, ognuno degli event loop è un thread, non mi preoccupo di deadlock, race, fairness ecc. . . perché tutto cade nell'accesso lettura/scrittura nella coda eventi
hw – so – app (chrome) – tab – dom – javascript – element (button, img, link. . .) – event handler (onClick = ". . .")
Dalla grafica si scopre che coda a messaggi e gestione asincrona funziona bene in altri contesti (Es. I/O asincrono in Java NIO)

Publish–subscribe

In origine ogni applicazione sviluppava internamente le strutture necessarie per la UI: righe di testo, liste, strutture dati. Ciò creava metodi da decine di migliaia di righe di codice ed era un **problema**, inoltre spesso ogni programmatore finiva per programmare cose già viste (pulsanti, tabelle e controlli vari) che potevano essere riutilizzati invece che riscritti copia-incollando.

Così invece di scrivere decine di migliaia di righe di codice ogni volta, si passa a scrivere qualche migliaio di **chiamate di libreria** a toolkit che, ad esempio, creano il pulsante o il controllo.

Trainability L'uso di **metodi comuni** favorisce l'**uniformità** e la **familiarità dell'interfaccia utente**: questo perché lo stesso metodo dello stesso toolkit disegnerà tale componente sempre nel solito modo, e ciò vale solitamente anche per toolkit diversi poiché tendono ad uniformarsi tra loro. La **trainability** è l'**addestrare l'utente ad aspettarsi dei comportamenti specifici**. Nel momento in cui **qualcosa di familiare non si comporta "come dovrebbe" causa irritazione**.

Tutto è quindi stato ridotto ad una semplice serie di chiamate di sistema che, **astruendo**, può essere indicata con `initUI(nomefile)`. Posso fare questo perché **tutta la sequenza di istruzioni per scrivere l'interfaccia può essere sostituita da un interprete**. Come passare dall'assembler al C, sostituisco le mie righe di codice con delle istruzioni a più alto livello che verranno interpretate.

In pratica, per costruire una UI si: istanziano degli oggetti, li si piazzano sullo schermo in qualche modo (coordinate, algoritmi. . .) e dopodiché si **esegue l'event loop** – estrazione messaggi dalla coda degli eventi e a seconda del messaggio si fa qualcosa.

L'interprete ovviamente cambia a seconda dell'ambiente: per esempio sul web è il browser, sul S.O. è una parte dello stesso.

22.4 Il file che descrive l'interfaccia utente

Tipi I vari componenti sono strutturati in una gerarchia di classi, di solito abbastanza profonde – storicamente la OOP è nata dalle gerarchie dei componenti UI.

La gerarchia parte da una qualche classe "madre" – ad esempio **View** – da cui si specializzano i componenti – **Button**, **List**. . .

Sono anche ottimi esempi di scomposizione di comportamenti complessi, ed un ottimo esempio di strutturazione delle gerarchie di classe.

Fissati da chi scrive il framework.

Posizioni Ogni componente sta nella gerarchia, con relazione di ereditarietà, ma anche **all'interno della UI in una gerarchia di contenimento**: avremo un contenitore **parent** – ad esempio il box della finestra – che **conterrà al suo interno altri componenti**, anch'essi conterranno a loro volta altri componenti. . .

Definiti dal programmatore in relazione alle scelte del framework.

Comportamenti Valori, eventi e azioni. Per esempio una lista deve contenere del testo, dei **valori**. Anche gli **eventi**, come lo scrolling che permette di vedere parti dell'UI nascoste. Infine le **azioni**: click, doppio click. . .

Definiti liberamente dal programmatore.

22.5 DOM

Document Object Model Modello dei documenti che sono pagine web. Una pagina web è un albero di oggetti in relazione di contenimento. Il linguaggio ovviamente è l'HTML, una specializzazione dell'XML.

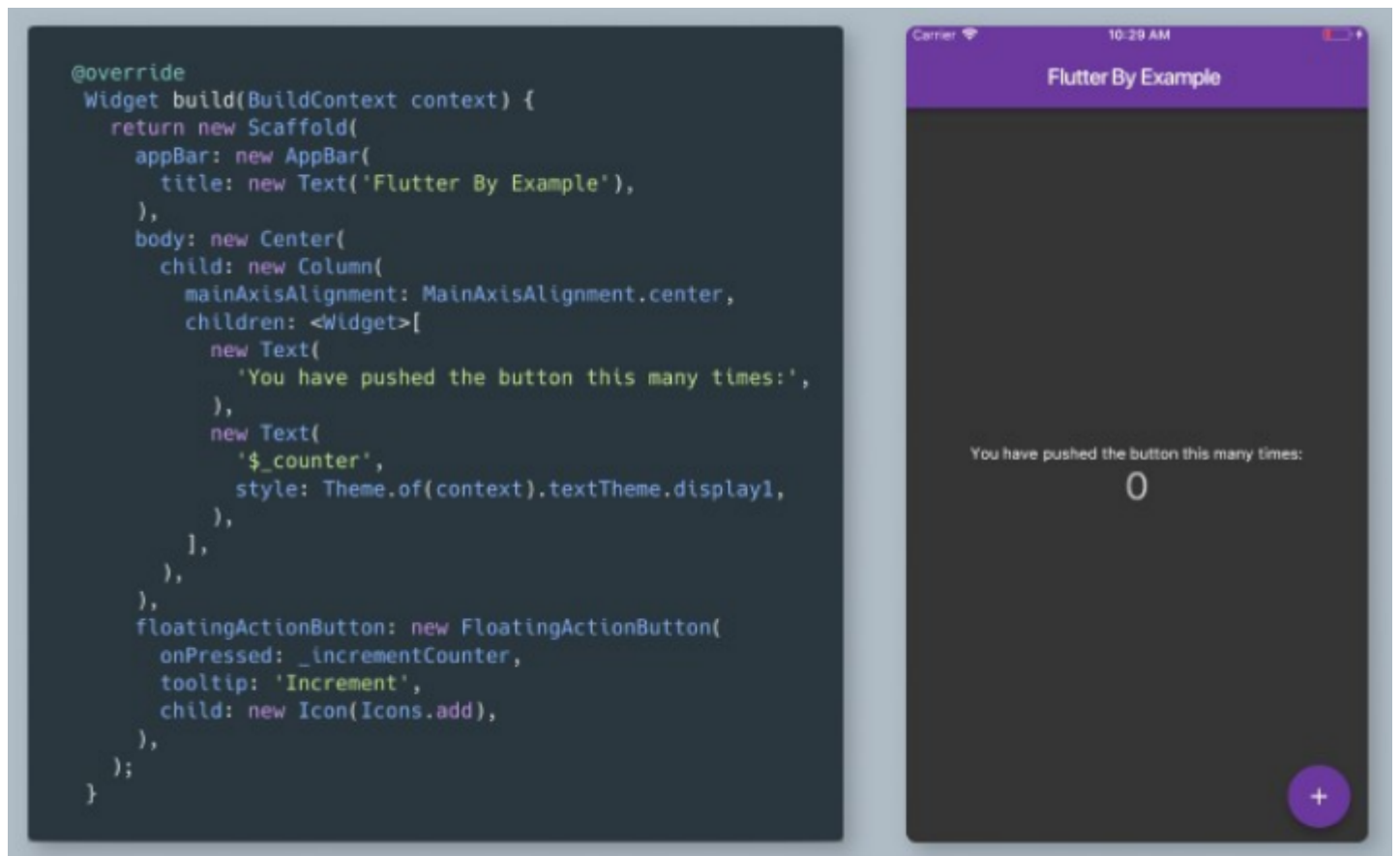
Gli oggetti sono specificati con i propri attributi, che è un modo per indicare valori, eventi (specifici del framework, es. `onClick`) ed azioni (spesso con un trucco cioè dando l'azione da eseguire come valore di un attributo che ha come nome l'evento associato) dei singoli oggetti.

L'interprete deve quindi navigare l'albero e, in caso di evento, cercare l'azione da eseguire (che spesso è il nome di un

metodo js scritto in un tag script) ed eseguirlo.
Ma anche questo è tanto lavoro, per questo sono nati i **framework**.

22.6 Flutter

Framework recente, sistema in cui sviluppare applicazioni mobile e web native partendo da unica base. flutterbyexample.com



23 Antonio Cisternino, Realizzazione di una GUI

grafica retantion o immediata

windows forms e windows presentation foundation libreria grafiche windows

contesto grafico: tavolozza con operazioni grafiche fattibili sui pixel della finestra.

finestra solitamente è ciò che trascini chiudi ecc., nel sistema grafico la finestra è un rettangolo di pixel. un bottone tecnicamente è una finestra (vive di vita propria e riceve eventi dal sistema grafico). Le finestre si possono organizzare in alberi.

pixel = 1/96 di pollice

frame buffer: area di memoria che mappa ogni pixel dello schermo su zona di memoria (solitamente 4Byte, 1 per canale di colore rosso verde blu e 1 byte per il canale alpha cioè trasparenza). 2 gigabyte al secondo per trasmissione 60fps su 4k. Flickering: applicazione scrive nel frame buffer mentre viene letto senza sincronizzazione (scheda video legge quel che c'è e manda non aspetta che il sistema abbia finito il disegno) – **double buffering** per risolverla, scrivo già "i pixel giusti" senza aver disegna rettangolo, disenga ellissi ecc in sequenza. contesto grafico non dal frame buffer ma da altra fonte (immagine, stampante...)

24 Nicola Melluso, Natural Language User Interfaces

Conversational System Interfaccia uomo macchina che **avviene tramite una conversazione – testuale o vocale**. La conversazione diventa **interessante per l'utente perché è naturale**. L'idea è che nella vita di tutti i giorni scambiamo input con la conversazione, quindi perché non usarlo anche per comunicare con i servizi che usiamo? Inoltre **è accessibile**: molte volte è difficile capire quale tasto bisogna premere o individuare i task da fare. Invece, se l'utente deve semplicemente dire ciò che vuole fare allora **non farà fatica a capire come funziona l'interfaccia**. La difficoltà di questo approccio sta nelle sfide per il programmatore: il **linguaggio naturale è ambiguo**, con tantissimi modi diversi per dire la stessa cosa. Il processo è complicato perché pieno di difficoltà tecniche.

24.1 Speech recognition e natural language understanding

Intent AmazonLex definisce "**intento**" come l'**obiettivo dell'azione che la macchina deve eseguire**. Bisogna riconoscere le **informazioni chiave** all'interno della frase.

DataType: gli slot delle **informazioni** che vengono **fornite** nella conversazione.

Intento: l'**obiettivo** della nostra conversazione.

Utterances: le **frasi** pronunciate, dalla **diversa struttura ma stesso intento**

Slots: i **dati necessari** per formare l'azione

Prompts: **risposta** che si dà ad ogni azione

Fulfillment:

Lex Bot structure Prima definisco le frasi (utterances), poi gli slot presenti in tali frasi.

25 Diego Colombo, Notebooks Hands on Session

Programmazione Interattiva Oltre alla programmazione classica *scrivi* → *compila* → *esegui* vi sono altri paradigmi:

REPL, Read Evaluate Print Loop.

Usata da strumenti come: IPython, Spark, Node, Mongo DB, FSI

Notebook

Documenti mark-down con linguaggio che descrive delle funzioni e possibilità di plot. Partito con mathematica.

La **programmazione interattiva** è **facile da usare** come strumento per esplorare vari tool ed è un **processo iterativo**.

Notebook I notebook hanno come obiettivo **collezionare pezzi di codice ed il loro output**. Quando un utente apre un notebook, può **vedere lo stato precedente senza aver bisogno di rieseguirlo**.

Un punto fondamentale è che il **contenuto creato come notebook possa essere usato sia come read-only sia come artefatto eseguibile**.

Inoltre è print-friendly con formattazione apprezzabile: facile l'esportazione in PDF ad esempio.

Incarnazioni moderne:

Wolfram Mathematica e **Matlab** sono notebook molto conosciuti

Jupyter è un notebook implementato su Python open-source e leggero. Ha guadagnato popolarità come tool per creare e condividere dati scientifici.

Concetti alla base I notebook sono **organizzati in celle**. Ognuna di esse può contenere **codice**, **output**, input, widgets, markdown o **semplice informazione**.

Inoltre devono essere **grafici**, **interattivi** e supportare **L^AT_EX**

Il concetto è analogo ai PDF, ma **richiedono più software**: fronted, language kernels, moduli ed estensioni.

Altri aspetti L'aspetto fondamentale da tenere in considerazione quando si creano notebook è che l'input e l'output è tutto lì, l'**utente vede il codice che si scrive**. Il documento dovrebbe dire la propria storia e dare il proprio contesto di ciò che parla. Se l'utente ha bisogno di sapere perché sto facendo passi sii esplicito, no commenti perché è disutile.