Programmazione d'Interfacce

Federico Matteoni

Indice

1 Introduzione

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

Prof.: **Daniele Mazzei**, mazzei@di.unipi.it
Riferimenti web:

- *?*

Esame: compitini/scritto. **Facoltativo**: orale discorsivo dove si discute un software noto.
Possibile proporre un software personale da presentare all'esame orale, spiegando come si è applicati i rudimenti del corso sul software presentato.

Materiale didattico:

- **Google Classroom**, slide presentate a lezione e altro materiale didattico
  Codice **c14kiy** con le credenziali d'ateneo.
  La suite Google è attivabile a *start.unipi.it/gsuite*
  Non è autorizzata la divulgazione
- 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*

Ricevimento: Mercoledì 14.30-16, Stanza 366

2 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, ...

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

3 Design

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.

3.1 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...)

3.1.1 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 approccia al software come gli pare.

3.1.2 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.

3.2 Front-End Developer

Esegue il design della UI convertendolo in funzionalità del prodotto.

4 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 si 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 omesso: 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 utentze 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 **classficiate 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.

5 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 efficacie è disporre fornelli e pomelli in modo che sia evidente la correlazione fra essi.
> **Non sottovalutare il costo mentale dell'utente**.


5.0.1 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 autoconvincendosene.
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.

6 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 risolvendo 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.

7 Design Thinking vs HCD

Insieme al termine Human Centered Desing, 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 feeback**.

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 beneficiali, sul lungo termine, per le persone obiettivo.
Lo HCD quindi viene sovrapposto all 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...).

8 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.

8.1 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.

8.1.1 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 percievable**. 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**.

8.1.2 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 maggiorparte 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.

8.1.3 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.

8.1.4 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.

8.1.5 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'iterpretazione. 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.

8.1.6 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.

9 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 perchè 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.

9.1 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 sottotioli 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 si 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)

10 Constraints, Discoverability and Feedback

10.1 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:

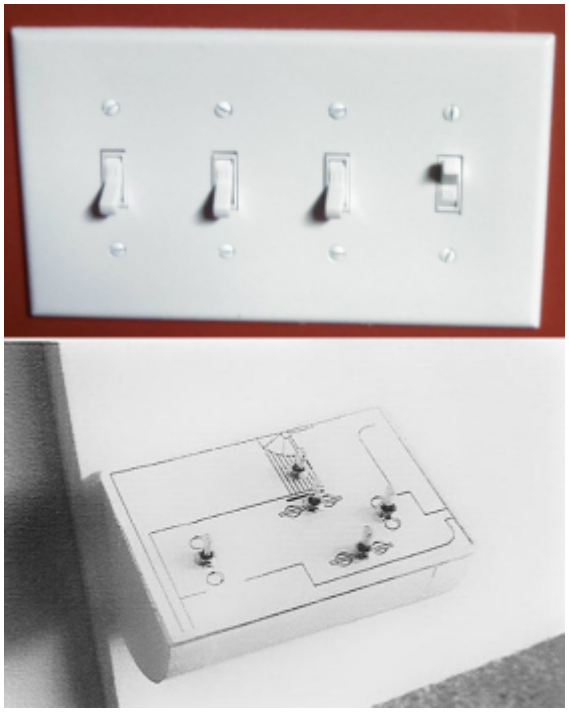
Fisici: ad esempio un pezzo Lego che entra solo in un determinato verso, il tappo di una biro entra solo in un verso. Sono vincoli concreti del mondo fisico

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 interuttori 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 interuttore.

10.2 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**

10.2.1 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.

10.2.2 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.

10.2.3 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.

10.3 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, cioù 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.

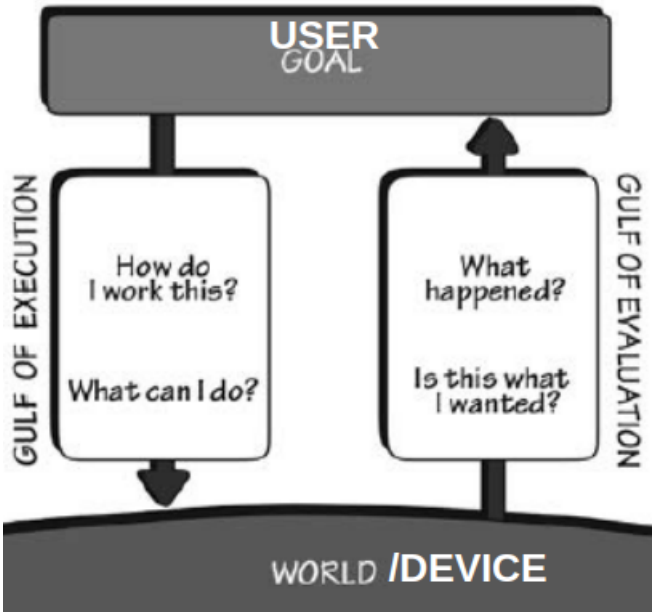
Un altro errore comune è creare un'apparente activity-centered control ma in realtà creare device-centered control.

11 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 fenomini 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.

11.1 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.

11.2 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

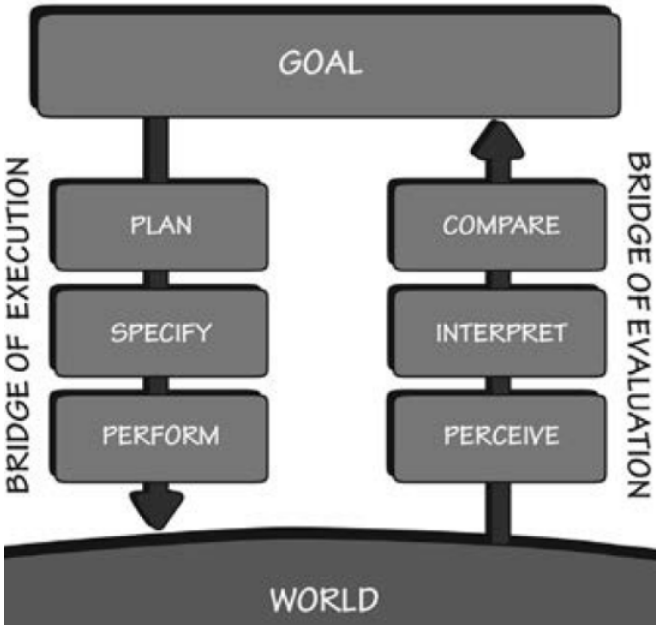
4: **Eseguo** i vari task pianificati: click del pulsante, riempimento dei campi...

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

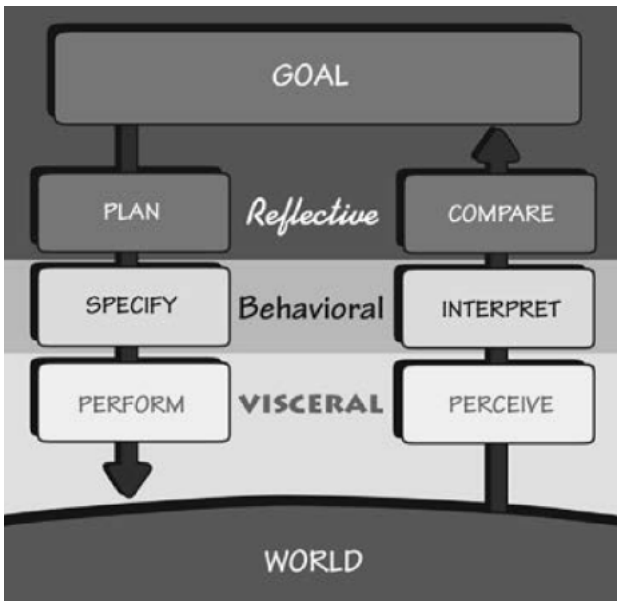
7: **Comparo** il risultato dell'interpretazione con il mio obiettivo, che non devono differire



`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 pardigma di analisi interazioni per capire se le nostre interfaccia performano bene o no.

11.2.1 Tre livelli di Processing



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

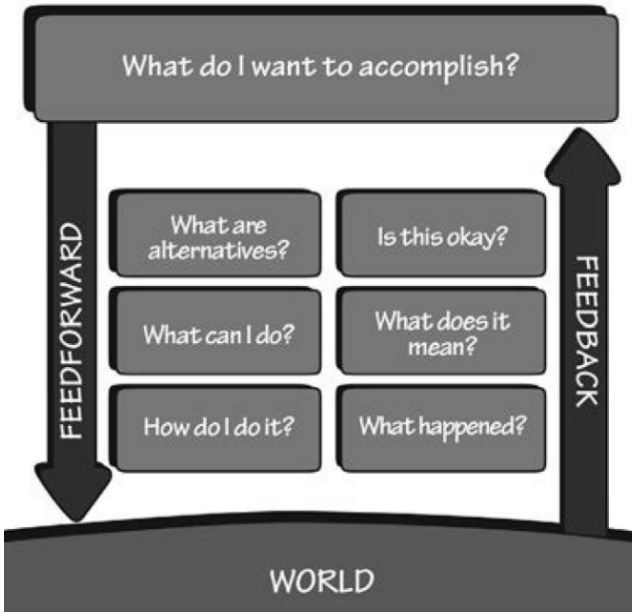
Viscerale: **comportamento basilare**, istantaneo e quasi del subconscio. *Clicco qua, vedo lo schermo.*

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.

12 Sette Principi Fondamentali della Progettazione

12.1 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?**

12.2 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*".

12.3 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 abilità 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 significante 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.

12.4 Opportunismo

Non sempre gli utenti eseguono delle azioni deliberatamente pianificate, ma alle volta 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**.

13 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.

13.1 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** $\longrightarrow$ 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**.

13.1.1 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 $\rightarrow$ aspetto $\rightarrow$ si ferma $\rightarrow$ entro $\rightarrow$ dico la destinazione $\rightarrow \ldots$
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

13.1.2 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

13.1.3 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.

13.1.4 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 buonsenso.
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.

14 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** (programmatori) invece che sui processi e sui tool (di sviluppo)
Software funzionante invece di grandi documenti
Capacità di **adattamento** al cambiamento.

14.1 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.

14.2 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 **perchè 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**.

14.2.1 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**.

14.2.2 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**.

14.3 Requirements

Un **requirement** è **una peculiarità** (proprietà, serivzio, 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 costraints**: un utente minorenne ha come requirement un costraint su certi siti che richiedono un lockout.

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 **requirement stanno uscendo di moda**.

14.3.1 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...*

14.4 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**.

14.4.1 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 dettagglio. 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
- ...

14.5 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 spacchettati, 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**, perché **gli scenarios sono stati approvati da lui stesso**.

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

14.5.1 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.

14.5.2 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.

Gl 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`

14.5.3 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 poché 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 dall personas → trovo elementi di contesto o particolarità da aggiungere.
Senza troppo dettaglio.

14.6 Use Cases

Evoluzione dello scnario, dove dato scenario ho la narrativa vista dal punto di vista dell'utente. Cioè che cosa utente fa nella nostra piattaforma per svolgere quello scneario.
Ogni use case è una serie di step monolitici. quindi use case sono nuova task list prodotto dal processo di design per innovare un metodo visto e descritto tramite la task analysis.
scenario si concentra su una situazione. use case invece è incentrato sulla personas: john e le cose che fa. Uno scenario diventa un set di use case.
Scenario: voglio prendere soldi al bancomat
Use Case: cliente della banca dello sportello, cliente di un'altra banca.

Fondamentale importanza perché vanno direttamente al team di sviluppo. Con use case ben fatti lo sviluppo è molto pià facile e semplice implementazione.
cosa c'è in use case: utente, cosa vuol fare, qual è goal finale, singoli step, feedback che l'intefaccia da durante step
cosa non c'è: qualsiasi dettaglio implementativo e di scelta tecnologica che non sia constraint o requirement, dettagli sull'ui

14.6.1 Elementi dello use case

...
main task (basic flow)
alternative paths (alternative flow)

14.6.2 Come scriverlo

1. identificare utenti personas
2. sceglierne uno
3. identificare i goal
4. discernere task principali (basi flow) dagli altri. legge 80-20: focalizzarsi sul main path perché 80% utenti necessita del 20% di features.
   Non gestisco carte smagnetizzate ma non posso permettermi di non dare soldi a nessuni (basi path, risolvere sempre)
5. con tutti i casi studio, si integrano eliminano ecc. ciò che si può
6. ...slides...

15 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.
alhpa.company costruito per i mooneshot.
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.

16 Fabio Viola, Gamification

Game Designer

16.1 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 persona → 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 paizzamento (gen precedenti premiazione piazzamtno 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 fette 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.

17 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 sottostitema grafico che può essere primitiva del S.O.
Kernel e sottostitema grafico insieme → winzozz
Applicazione e sottostitema 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.

17.1 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.

17.2 modelli

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

17.3 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
21/10/2019
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 visualliza *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)

18 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...)