Interaction Homme-Machine : Support de cours

Régis WITZ – rwitz@unistra.fr

Table des Matières

- La vision
  - Caractéristiques
  - Boucle perception - action
  - Interaction à deux mains
- La mémoire
- Résolution de buts
  - Algorithmie
  - Théorie de l'action
  - Réalité
- Périphériques
  - Écrans
  - Périphériques d'entrée
- Styles d'interfaces
  - Style conversationnel
  - Style par manipulation directe
  - Style par reconnaissance de traces
- Implémentation
  - Machine à états
  - Programmation événementielle
  - Fichiers de description
  - Constructeur d'interface graphique
- Conception
  - Les utilisateurs
  - Leurs tâches
  - Concepts
  - Vocabulaire
  - Scenarios
- Implémentation
  - Compatibilité
  - Guidage
  - Homogénéité
  - Souplesse
  - Contrôle explicite
  - Gestion des erreurs
  - Concision
- Exemples
  - Restrictions arbitraires
  - Mécanismes internes
  - Nombreuses fonctionnalités
  - Biais cognitif
  - Données
  - Affichage
- Couleurs
  - Usage
  - Palettes
- Interfaces multi-support
  - Responsive design
  - Adaptive design

- Choisir
- Techniques
- Égalité devant le contenu

Accessibilité

- Définition et contexte
- Enjeux
- Solution
- Principes

Internationalisation

- Traduction
- Localisation
- Internationalisation
- Validation

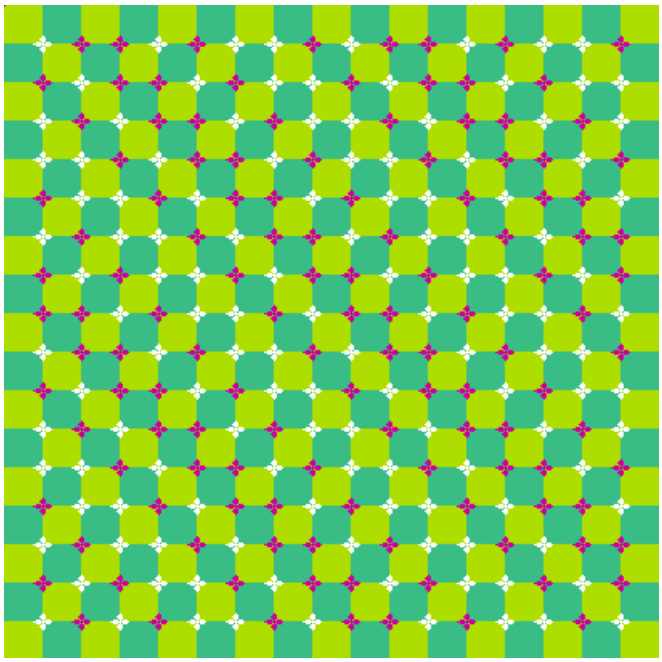
Tests

- Types de tests
- Outils de mesure
- Quand tester ?

La vision

Les indications apportées par les interfaces informatiques sont de nature essentiellement visuelle. Or, la vision humaine est imparfaite et facilement abusée : nous voyons parfois des choses qui n'existent pas, ou manquons à remarquer ce qui est devant nos yeux.

Ces imperfections de la vision humaine peuvent être exploitées à des fins récréatives, comme avec les illusions d'optique. Cependant, certaines dispositions d'objets affichées sur un écran peuvent être très désagréables suivant les situations.



Exemple d'illusion d'optique :© Akiyoshi Kitaoka, « Trick eyes ».

Caractéristiques

Son **champ de vision** est d'environ 180° ; cela signifie par exemple qu'outre son écran, un utilisateur perçoit aussi (quoique moins bien) sur les cotés. On en déduit aussi que l'utilisateur ne perçoit pas ce qui est derrière lui. Cela peut avoir son importance suivant l'**environnement** dans lequel se trouve l'interface : outre la réalité virtuelle, cela concerne par exemple les interfaces qui sont dans la rue (distributeurs, …) ou dont la nature est dépendante du lieu ou elles se trouvent (machines outils, …).

Son **acuité** donne la **taille minimale** des objets observables. En particulier, un seul pixel peut être très facilement vu. L'utilisateur d'une interface percevra souvent des artefacts graphiques ou des décalages d'objets les uns par rapport aux autres.

Sa sensibilité aux **couleurs** : la théorie des couleurs est complexe et parfois subjective. Sans entrer dans les détails pour l'instant, on retiendra que chaque couleur est porteuse d'une sémantique subjective, et que la perception de chaque couleur est influencée par les couleurs qui l'environnent.

Sa perception de la **profondeur**, grâce à la **vision stéréoscopique** : nous avons deux yeux qui voient chacun une image légèrement différente ; notre cerveau déduit la notion de profondeur de ces deux images. Cependant, d'autres **informations secondaires** peuvent aussi donner une impression de profondeur : on peut citer par exemple la taille des objets les uns par rapport aux autres, les ombres, la luminosité, … Ce sont ces informations qui nous permettent de « percevoir » la profondeur d'une scène en trois dimensions affichée sur un écran, par nature en deux dimensions seulement.

Sa sensibilité aux **mouvements**. Dans une interface graphique, le mouvement peut être utilisé au bénéfice de l'utilisateur, par exemple pour attirer son attention sur un objet **clignotant**, ou lui indiquer l'avancement d'une tâche grâce à une barre de **progression**. Cependant, trop d'éléments en mouvement à l'écran le font souvent au détriment de la compréhension de l'utilisateur.

Sa capacité à **reconnaître** et à **différencier**, grâce à la **vision active**, apte entre autres à isoler un élément particulier reconnaissable par une ou plusieurs **caractéristiques différentes** de celles des éléments environnants.

Votre vision active vous permet de détecter ces pandas dans leur milieu naturel (Panda cubs in Chengdu, inchengdu, CC BY 2.0)

Boucle perception - action

Notre vue, notre toucher et notre proprioception sont en interaction constante. Par exemple, pour saisir un objet, notre vue nous permet d'amorcer notre mouvement, de corriger la trajectoire de notre main, et de réajuster spécifiquement nos mouvements, tandis que notre toucher nous sert à confirmer que notre geste est correctement exécuté, ainsi qu'à affermir notre prise.

Interaction à deux mains

De manière générale, notre main dominante exécute l'**action**, tandis que notre main non-dominante nous permet de situer le **contexte** de cette action. Cette collaboration entre nos deux mains nous permet d'entreprendre des manipulations complexes.

Cependant, cette interaction entre les deux mains de l'utilisateur reste relativement peu exploitée dans le domaine des IHM. Citons comme exceptions certaines applications de réalité virtuelle et certains contrôleurs à reconnaissance de mouvements majoritairement utilisés dans le domaine des jeux vidéos.

La mémoire

Tout utilisateur d'IHM utilise sa mémoire pour **apprendre** à se servir de cette interface, mais aussi pour **exécuter une action** à l'aide de cette interface.

On peut en fait distinguer trois types de mémoires :

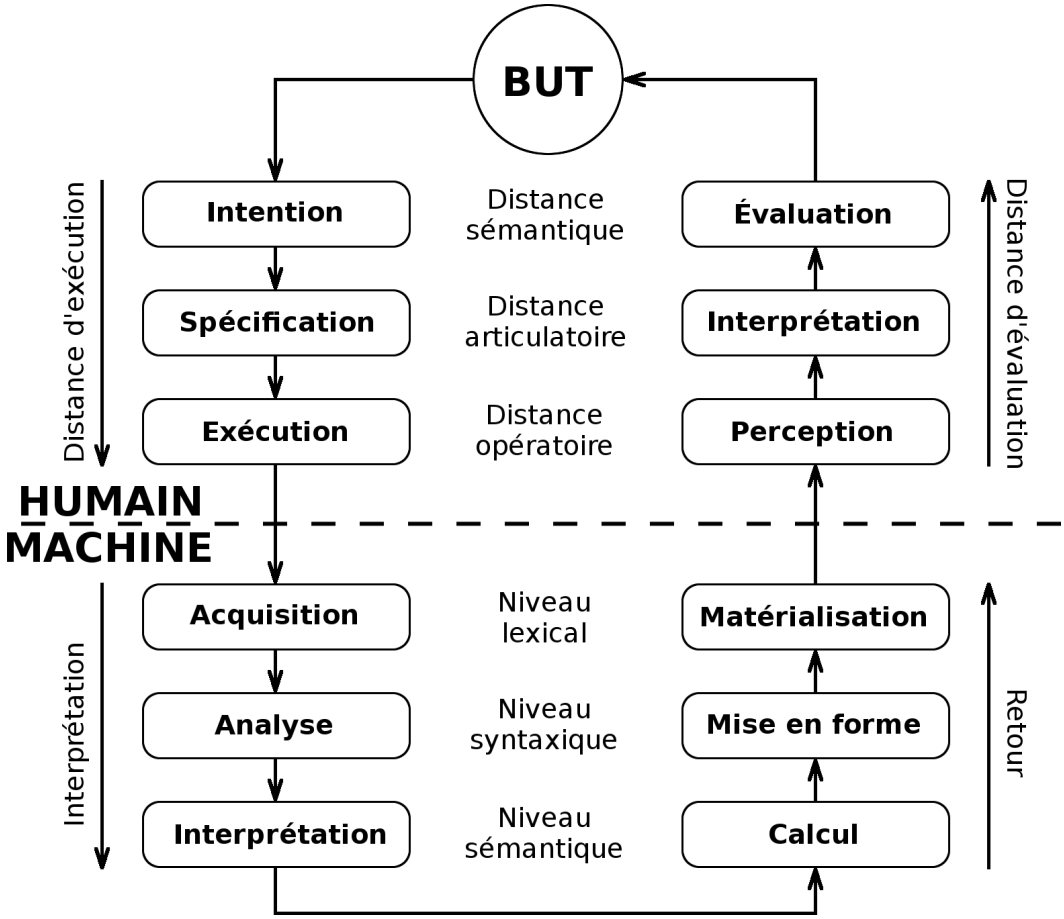
- La **mémoire sensorielle**, capable de retenir fidèlement mais très, très brièvement (souvent moins d'une seconde) les informations prodiguées par les différents sens. Seules les **informations significatives** pour l'utilisateur ont une chance d'être retenues.
  Cette vidéo externe peut vous donner une idée de la quantité importante d'informations que la vision perçoit mais que la mémoire sensorielle oublie très rapidement.
- La **mémoire à court terme**, capable de retenir environ 7 éléments mémoriels (on parle en général d'une capacité de **7 « mnèmes »**) pendant une durée maximale de quelques dizaines de secondes au maximum. Cette mémoire sert majoritairement à l'utilisateur d'une interface pour effectuer son action courante. La mémoire à court terme est de très faible capacité, et est très volatile. Une IHM ne devrait donc pas obliger son utilisateur à retenir trop d'informations (comme l'état actuel du système, ou les opérations effectuées auparavant) afin d'exécuter correctement une action.
  La limite de 7 mnèmes donne aussi une idée du nombre maximal d'éléments que devrait proposer un menu, par exemple.
- La **mémoire à long terme** permet en théorie de retenir un nombre infini de mnèmes, pour une durée infinie. À noter que la répétition d'une action ou d'une information permet une meilleure mémorisation. Cela a son importance pour la formation d'un utilisateur à une interface, ainsi que pour sa rapidité d'adoption.

Résolution de buts

Algorithmie

L'informatique enseigne que tout problème peut être décomposé en sous-problèmes ; ces sous-problèmes pouvant à leur tour être décomposés, jusqu'à atteindre une série de problèmes triviaux à résoudre. La résolution dans l'ordre de tous ces problèmes triviaux permet d'atteindre le but final.

Théorie de l'action



Théorie de l'action (Don Norman, 1986)

La théorie de l'action de Don Norman énonce que tout utilisateur franchit les étapes suivantes lorsqu'il manipule une interface dans un but précis :

❶ L'utilisateur traduit son but en formulant son **intention**, …

❷ … puis il **spécifie** actions à entreprendre pour satisfaire son intention, …

❸ … enfin, il **exécute** cette série d'actions.

❹ La machine **interprète** les données en entrée de l'utilisateur, puis matérialise son **retour** sous forme de données en sortie.

❺ L'utilisateur **perçoit** la sortie de la machine, …

❻ … il **interprète** ces sorties qu'il a réussi à percevoir sous forme de résultat, …

❼ … enfin, il **évalue** les résultats qu'il a interprété en regard de son but initial, pour savoir si celui-ci est atteint.

Réalité

La théorie ne donne qu'une vision partielle de la manière dont un humain tente d'accomplir ses buts à l'aide d'une IHM.

En pratique, il est nécessaire de garder en tête que :

- l'humain s'adapte de manière constante à son contexte,
- il tente souvent de résoudre ses problèmes de manière incrémentale, sans forcément planifier ses actions,
- le comportement des utilisateurs est très variable, surtout si la situation qu'ils subissent est inhabituelle pour eux.

En conséquence, il peut y avoir autant, voir davantage, de méthodes de résolution d’un même problème que d’utilisateurs.

Périphériques

Les périphériques de sortie, et surtout les périphériques d'entrée, déterminent la manière dont l'utilisateur interagit avec une machine donnée. En conséquence, ces périphériques fixent les contraintes, donc conditionnent fortement, la conception des interfaces graphiques.

Écrans

Le type de périphérique de sortie prévalent dans le domaine des IHM est bien évidemment l'**écran**. Les écrans sont aujourd'hui de plusieurs types : écrans LED, LCD, vieux écrans CRT, écrans amoled, retina, etc.

Leurs caractéristiques principales sont leur **résolution**, la **définition** de l'image qu'ils offrent, leur **persistance** ou rémanence, et leur vitesse de **rafraîchissement**.

Périphériques d'entrée

Le nombre de **périphériques d'entrée** disponible aujourd'hui est impressionnant : clavier, souris, molette, trackball, tablette (graphique), touchpad, joystick, joypad, … difficile d'être exhaustif. Un des périphériques d'entrée les plus utilisés, l'écran tactile (qu'il soit capacitif ou résistif), est à la fois un périphérique d'entrée et de sortie.

Tous ces périphériques d'entrée offrent chacun des sensations différentes, et sont donc réservés à des **usages différents**.

Il est possible de classer les périphériques d'entrée de différentes manières :

- ceux à **coordonnées absolues** (x,y) comme la tablette ou l'écran tactile donnent la position pointée ; d'un autre coté, ceux à **coordonnées relatives** (dx,dy) comme la souris ou le touchpad donnent le déplacement depuis la dernière position connue.
- les périphériques d'entrée peuvent être **directs** (par exemple l'écran tactile) ou **indirects** (comme le touchpad).
- leur **mode d'entrée** :
  - en mode **requête** : l'attente est bloquante.
  - en mode **échantillonage** : la réponse est immédiate
  - en mode **événementiel** : file d'attente

Une notion utile est aussi la notion d'**écho**. Il s'agit de la représentation graphique des actions de l'utilisateur. Par exemple lorque l'utilisateur utilise une souris, le curseur (en temps normal) ou le rectangle élastique (lors d'un redimensionnement) forment l'écho de cette souris.

Dispositif haptiques

Un périphérique d'entrée **haptique** est tactile (toucher, vibrations, …) et kinesthésique (retour de force, …). Il utilise des moyens comme les vibrations, les souffles d'air ou les ultrasons pour permettre à l'utilisateur de manipuler des objets virtuels comme s'il les tenait vraiment en main, sans ressentir ni le poids ni les mécanismes du périphérique proprement dit.

Par exemple, certains dispositifs haptiques permettent de ressentir la texture d'objets affiché par un écran lorsque l'utilisateur passe la main sur cet écran. Certains controleurs de jeu (comme les volants à retour de force) peuvent aussi être qualifiés d'haptiques en cela qu'il vibrent entre les mains de l'utilisateur pour lui faire ressentir des chocs, ou les vibrations d'un moteur par exemple.

Loi de Fitts

La loi de Fitts, publiée par Paul Fitts en 1954, est une manière d'estimer l'efficacité de l'utilisateur pour atteindre un élément particulier d'une interface graphique grâce à son périphérique d'entrée ; en d'autres termes, la rapidité avec laquelle il sera capable d'agir sur un élément précis de cette interface.

Cette rapidité est fonction de la distance séparant le centre de l'élément cible de son point de départ, mais aussi de la taille de l'élément cible.

Hystérésis

Les périphériques d'entrée permettent d'accomplir différentes opérations, dont le sens est donné par la manière dont ils sont utilisés.

Par exemple, pour le périphérique de pointage qu'est la souris :

- l’action se fait grâce au double clic ;
- la sélection se fait par un simple clic (éventuellement assorti de modificateurs issus de l’appui sur les touches control, alt ou shift du clavier) ;
- le tracé ou déplacement se fait en déplaçant le pointeur, bouton de clic enfoncé.

Il y a donc à priori conflit entre la sélection et le tracé/déplacement : lorsque l’utilisateur clique, est-ce une sélection ou le début d’un déplacement ? À l’inverse, s’il clique en se déplaçant, est-ce vraiment un tracé ? Les humains ne sont en effet pas des robots, et leurs mains peuvent trembler naturellement, ou simplement bouger suite au mouvement d’un des doigts.

En découle la notion d'**hystérésis** : un tracé ne sera par exemple reconnu après appui sur le bouton de la souris que lorsque le déplacement du curseur est supérieur à un seuil, qualifiant par là même un vrai déplacement, et non un simple faux positif.

Cette notion d’hystérésis s’applique à la plupart des périphériques d’entrée.

Styles d'interfaces

Ce chapitre détaille les principaux styles de communication entre l'homme et la machine. En d'autres termes, les principales manières de construire et d'utiliser une IHM.

À noter que le style réel d'une interface peut être difficile à reconnaître, car celle-ci fait un effort pour « dissimuler », avec plus ou moins de bonheur, ses mécanismes internes sous une couche d'ergonomie. Par exemple, les assistants vocaux peuvent sembler bien plus riches et complexes qu'un terminal, alors qu'ils communiquent au fond tous les deux de la même manière. De même, les formulaires web sont souvent éclatés en différentes pages, ou pré-remplis avec les informations concernant l'utilisateur que le site a glané au cours de sa navigation.

De plus, rares sont les interfaces qui n'utilisent qu'une seule de ces modes de communication. Rappelons que tous les moyens sont bons pour mieux faire se comprendre l'homme et la machine. En conséquence, la plupart des IHM sont hybrides, laissant à l'utilisateur plusieurs moyens de s'exprimer, ou utilisant un certain style pour certaines des fonctionnalités proposées, et un autre style pour d'autres.

Style conversationnel

Cette catégorie d'interface graphique regroupe elle-même trois sous-catégories : les interactions de type « console », celles de type « formulaire » et enfin les interfaces de navigation.

Console

Il s'agit là de la plus ancienne manière pour l'homme de communiquer « graphiquement » avec la machine. La séquence d'interaction est toujours la même :

❶ l'utilisateur entre une commande

❷ l'utilisateur valide sa commande

❸ la commande est exécutée

❹ les résultats de la commande sont présentés à l'utlisateur

La commande entrée par l'utilisateur doit évidemment être **exprimée dans un langage** et avec une **syntaxe** compris et attendus par l'ordinateur. Ce « langage console » est souvent **assez éloigné du langage naturel** de l'utilisateur.

Exemples

- Comme son nom l'indique, les interfaces de type **terminal**, les consoles de commandes entrent dans cette catégorie.
- Les **assistants personnels** fonctionnent aussi comme des consoles. Leur objectif est en effet d'**émuler une conversation** avec l'utilisateur : celui-ci formule ses commandes oralement, et sa machine (ie. son téléphone) présente ses réponses à l'écran, souvent en formulant ses réponses grâce à une synthèse vocale. Cependant, bien que la manière dont l'utilisateur doit formuler ses « demandes » à son assistant personnel se rapproche fortement du langage naturel, celle-ci doit quand même respecter une syntaxe précise à base de mot clefs. De plus, les assistants personnels actuels ne comprennent qu'un nombre de langage limité -voire parlent uniquement anglais. S'ajoute donc dans certains cas la barrière de la langue.

Avantages

- Pour l'utilisateur qui en maitrise la syntaxe, ce type d'interface est très puissant car elle lui permet de s'exprimer de manière très **rapide et concise**.
- Une interface console est **évolutive** : sa syntaxe peut être enrichie de méta-mots (scripts), de synonymes (*alias*), de conditions, …
- Bien qu'on sorte du domaine de l'interface homme-machine pur, ce type d'interface est aussi adapté à l'interaction entre programmes (« interaction machine-machine »), en raison justement de sa rigueur et de sa syntaxe clairement définie.

Inconvénients

- La syntaxe attendue par une console constitue une **barrière d'entrée**. En effet, plus celle-ci s'éloigne des habitudes de l'utilisateur, plus son temps d'apprentissage est important.
- Malgré la manière dont on peut l'enrichir (voir avantages), la **syntaxe** reste quand même **stricte**. En particulier, toute difficulté qu'aurait l'utilisateur à traduire ses intentions dans le langage console constitue la *distance d'exécution* décrite par Norman dans sa théorie de l'action, et est

une cause potentielle de problème durant l'utilisation d'une interface console.

- Il est souvent difficile pour l'utilisateur d'accomplir **plusieurs tâches en parallèle** avec une interface console.

Formulaire

Il s'agit là de tous les formulaires ou menus. La liste de commandes que peut exécuter une interface à base de formulaires est **explicite**.

Pour executer une action grâce à un formulaire, l'utilisateur en remplit les **champs** indispensable, puis valide sa saisie.

Exemples

- Les formulaires sont très présents sur les sites **web**. Par exemple, lorsque l'utilisateur décide de créer un compte, de commenter un article, de communiquer ses informations et ainsi de suite, il remplit quasiment systématiquement un formulaire.
- Les **applications de gestion** sont souvent à base de formulaires.
- L'interface du **minitel** était entièrement à base de formulaires.

Avantages

- Les actions de l'utilisateur sont fortement **guidées** : les champs sont libellés et expliqués pour qu'ils offrent une sémantique forte, certains champs pour lesquels seules certaines valeurs sont acceptables contraignent l'utilisateur via des menus à choix limités, certains champs peuvent être pré-remplis avec une valeur par défaut, et ainsi de suite. Tout cela fait que, si l'utilisateur est suffisament formé et dispose des informations requises, il lui est difficile de mal remplir un formulaire.
- La valeur des champs d'un formulaire est souvent **non imposée**, est reste **modifiable jusqu'à la validation** par l'utilisateur. Celui-ci a donc le temps de changer d'avis ou de rassembler les informations dont il a besoin.
- Un formulaire a une **forme plus graphique** qu'une console, par exemple. Entre autres, cela offre différents moyens supplémentaires d'orienter l'utilisateur et de guider sa saisie, par exemple par l'utilisation judicieuse des couleurs et des polices de caractères, ou par un placement logique des différents champs à l'écran.

Inconvénients

- Un formulaire donné permet d'accomplir une **tâche prédéfinie**, ce qui offre **peu d'évolutivité**. Faire évoluer un formulaire n'est parfois pas une bonne idée : en effet, l'utilisateur étant habitué à un certain formalisme (ordre de saisie, position des champs à l'écran, …) peut se sentir perdu et frustré, voire entrer involontairement des valeurs eronnées.
- Un formulaire permet rarement la **parallélisation** de plusieurs tâche. S'il le permet, cela se fait souvent au détriment de sa simplicité de compréhension.

Navigateur

Ce style d'interface est caractérisé par la capacité de l'utilisateur à naviguer entre différents **nœuds**, caractérisés par leur identifiant ou **adresse** hypertexte. Lorsque l'utilisateur y accède, chaque nœud offre son contenu, qui peut être une donnée ou un traitement. Tous les nœuds sont **reliés entre eux** directement ou indirectement. De plus, l'utilisateur conserve toujours la possibilité de **revenir en arrière** dans sa navigation.

Exemples

- Les **navigateurs internet**, évidement. Inutile d'en dresser une liste ici : vous en utilisez probablement un actuellement.
- Les **explorateurs de fichiers** offrent aussi une interface à base de navigation de nœud à nœud (fichier, dossier, …). Nautilus pour Gnome, Konqueror pour KDE, File explorer pour Windows sont quelques uns des nombreux exemples.

Style par manipulation directe

Un des fondateurs et défenseur des IHM par manipulation directe est Ben Shneiderman (*The future of interactive systems and the emergence of direct manipulation*, *Behaviour & Information Technology*, 1982).

L'idée sous-jacente à ce style d'interface est que les objets informatiques peuvent être manipulés

de manière similaire à des objets physiques, et que l'utilisateur peut faire des actions sur eux à l'aide de son périphérique d'entrée.

Dans sa forme la plus « pure », une IHM par manipulation directe respecte quatre grands principes :

❶ Les objets d'intérêt sont **disponibles en permanence**.

❷ L'utilisateur **agit directement** sur ces objets d'intérêt, plutôt que d'utiliser une syntaxe complexe ou un enchaînement d'actions.

❸ Chaque action de l'utilisateur sur un objet d'intérêt est :

- **incrémentale**,
- **réversible**,
- aux **résultats immédiatement visibles**.

❹ Une telle interface est **structurée en couches**.

Exemples

- Le paradigme actuel de la plupart des logiciels et des systèmes d'exploitation, **WIMP** (*Windows*, *Icons*, *Menus*, *Pointers*), est une application directe (bien que plus ou moins heureuse, voir Inconvénients) des principes de la manipulation directe.
- Les interfaces graphiques permettant d'éditer des données et de les visualiser telles qu'elles seront éditées (appliquant le principe **WYSIWYG**, pour *What You See Is What You Get*) permettent aussi souvent de les manipuler directement.

Avantages

- Ce style d'interface est **intuitif**. Cet avantage découle à la fois de ses qualités intrinsèques, mais aussi en raison du fait que l'écrasante majorité des interfaces graphiques utilise ce style depuis les années 70.
- Les tâches que peut accomplir l'utilisateur avec ce style d'interface sont **non-prédéfinies**. Une nouvelle tâche donne simplement lieu à un nouvel objet, ou à une nouvelle façon d'intéragir avec un objet existant.
- Combiné avec un système multi-fenêtres, ce style assure la **parallélisation** des tâches. L'utilisateur peut travailler en même temps dans deux fenêtres différentes.
- Ce style offre des **entrées et sorties riches** : déplacement d'icones, redimensionnement, *drag and drop*, …

Inconvénients

- Cette manière d'intéragir avec la machine étant avant tout pensée pour que l'utilisateur manipule directement et lui-même les objets, l'**automatisation** d'une interface par manipulation directe est difficile.
- Certaines tâches sont par nature complexes. Se pose donc la question de la manière de « résumer » ces tâches complexes en un seul objet, ou en une seule action sur un objet. Répondre de manière satisfaisante à cette question est bien souvent impossible. Cela a en particulier pour conséquence que le troisième principe est bien souvent mal appliqué, la tâche complexe étant souvent déléguée à une fenêtre dédiée avec laquelle l'utilisateur va passer beaucoup de temps à intéragir. Il est donc plus adapté, pour la plupart des systèmes *WIMP*, d'utiliser le qualificatif de « manipulation indirecte » …

Style par reconnaissance de traces

Cette catégorie d'IHM est rendue possible par le fait de détecter certaines données particulières issues d'un périphérique d'entrée considéré. Ces données sont ensuite comparées à un **vocabulaire** afin de leur donner une **sémantique** précise. Cette sémantique est ensuite traduite en opération(s) à effectuer par l'application.

Exemples

- La plupart des smartphones et des tablettes permettent, au moins dans une certaine mesure, des interactions dans ce style. Les capacités de l'écran tactile (sensibilité ? support du multitouch ?) ainsi que la nature des applications tournant sur ces supports sont cependant très variées, rendant difficile la définition du « vocabulaire tactile » exact supporté par un smartphone donné.
- Les capteur (caméras) à reconnaissance de mouvements permettent eux aussi de concevoir de telles interfaces.

Avantages

- Ce style d’interface est **intuitif**. Cet avantage découle à la fois de ses qualités intrinsèques, mais aussi grâce à la généralisation des smartphones dans la population depuis la fin des années 2000. Cela fait que les utilisateurs maîtrisent pour la plupart un minimum de « mots » d’interaction avec un écran tactile.
- Ce style d’interface est, peut-on dire, « dans l’air du temps », ou à la mode. Cette caractéristique n’est pas à négliger, car elle rend plus facile la possibilité de susciter l’adhésion des utilisateurs à une IHM.

Inconvénients

- Une telle interface nécessite un **périphérique d’entrée spécifique**. Rien ne sert de concevoir une IHM très intuitive par reconnaissance de traces si la machine cible ne permet pas ce type d’interaction (voir le chapitre sur l'*Adaptive design* pour quelques pistes de solution).
- Même si elle est intuitive, une telle interface attend quand même un vocabulaire précis, une syntaxe à apprendre qui constitue néanmoins une **barrière d’entrée** que l’utilisateur doit franchir avant de pouvoir utiliser l’interface.
- S’il simple pour l’utilisateur d’accomplir une action avec un simple geste, se pose la question de comment **annuler** cette action.
- Le style par reconnaissance de traces souffre du même inconvénient que le style par manipulation directe : comment traduire des actions complexes par un simple geste ?

Implémentation

Historiquement, dans les années 70-80, les interfaces graphiques étaient caractérisées par le fait d'être utilisées par un utilisateur averti ou « expert », qui était souvent le seul à utiliser un ordinateur donné. De plus, les performances étaient faibles et les entrées/sorties peu nombreuses, surtout comparées aux caractéristiques de l'informatique d'aujourd'hui. Tout cela se traduisait par des IHM relativement pauvres. Donc, il y avait peu de code dédié spécifiquement à l'IHM dans les applications de l'époque.

Au tournant des années 90, les choses ont commencé à changer. L'informatique s'est tourné toujours plus vers le grand public. Les interfaces graphiques ont suivi, et sont devenues toujours plus simples d'accès. Elles ont aussi profité de machines toujours plus puissantes, qui permettaient des affichages plus colorés, détaillés et complexes, et des utilisations diversifiées. De plus, la volonté actuelle des concepteurs d'IHM est de les rendre toujours plus intuitives, mais aussi toujours plus riches.

En conséquence, la proportion du code spécifique à l'IHM dans le code des applications est en général plus grande que par le passé. Au delà de ce fait, le temps accordé au design, à la conception de l'interaction homme-machine est lui aussi toujours plus important.

Ce chapitre détaille les différentes manières d'implémenter une IHM aujourd'hui.

Machine à états

Toutes les fonctionnalités de l'interface sont décomposées en tâches élémentaires. Chacune de ces tâches correspond à l'implémentation d'un **automate à états finis**. Cet automate est enrichi d'actions associées :

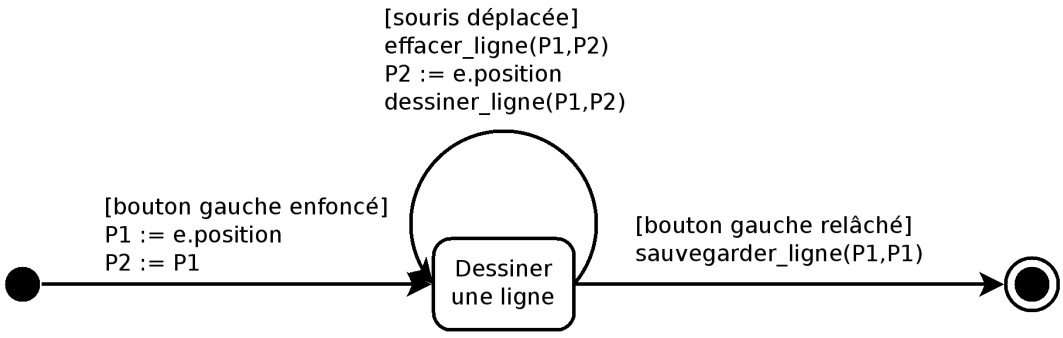
- aux **états** : l'action est exécutée lorsque l'état considéré devient l'état courant ;
- aux **transitions** : l'action est exécutée lorsque la transition considérée est empruntée.

Exemple

Voici un exemple de logique d'implémentation d'une tâche élémentaire : dessiner une ligne, dans un logiciel de dessin par exemple.

- partant d'un état initial de l'application,
- lorsque le bouton gauche de la souris est enfoncé, on mémorise la position actuelle *P1* de la souris et on passe dans l'état *Dessiner une ligne*.
- si la souris est déplacée, on efface la ligne précédemment dessinée (si elle existe), on mémorise la position actuelle *P2* de la souris, et on redessine une ligne allant de *P1* à *P2*.
- lorsque le bouton gauche de la souris est relâché, on valide la dernière ligne dessinée.

Le schéma d'implémentation de la machine à états finis correspondante est ici représentée en utilisant le formalisme UML (diagramme d'activités).



Exemple de machine à états : Dessiner une ligne

Avantages

- Implémenter un automate est aisé et réaliste pour des **systèmes simples**.
- Cette possibilité offre une **flexibilité** inégalée qui peut être indispensable dans certains cas, notamment en terme de configuration des actions à exécuter.

Inconvénients

- Tout implémenter soi-même est à éviter absolument si le système est **complexe**.
- Comment représenter et maîtriser l'état global du système ?

Programmation événementielle

Ce paradigme de programmation est caractérisé par sa non-linéarité. Pour chaque objet *O* de l'interface : si l'événement *E* se produit sur l'objet *O*, alors le traitement *T* est exécuté.

Exemple

La plupart des librairies graphiques se basent sur de la programmation événementielle. Les exemples sont nombreux: Qt, swing, node.js, …

Avantages

- L'aspect « local » de l'implémentation de chaque traitement fait que les composants sont souvent bien **isolés**, et favorise la **modularité**.
- Une fois ses principaux principes acquis, la programmation événementielle est relativement **simple** d'utilisation et **adaptée** au domaine des IHM.

Inconvénients

- La programmation événementielle n'est pas forcément simple à se représenter pour un développeur.
- C'est à chaque traitement implémenté de vérifier lui-même si il doit s'interrompre et, si c'est le cas, de le faire proprement. Implémenter une tâche de longue durée sans prendre cette précaution élémentaire est la cause la plus courante de « freeze » de l'IHM.

Fichiers de description

Plutôt que d'implémenter directement une interface graphique en utilisant un langage de programmation, il est possible d'utiliser un **langage de balisage** d'interface graphique (ou *User Interface Markup Language*).

Les fichiers créés sont alors destinés :

- soit à être compilés, éventuellement en passant par une étape de traduction en un langage de programmation plus générique ;
- soit à être interprétés par un *runtime*.

Exemples

- QML, qui fait partie du projet Qt ;
- XUL, supporté par la fondation Mozilla ;
- UIML, le précurseur, forme un standard ouvert ;
- XAML, supporté par Microsoft.

Avantages

- La plupart de ces langages de description formant un sous-ensemble d'XML, il est possible de créer des interfaces grâce à eux sans avoir de compétence avancée en programmation.
- Conséquence directe, cette solution peut former une base commune pour le designer et le développeur pour communiquer et se comprendre.

Inconvénients

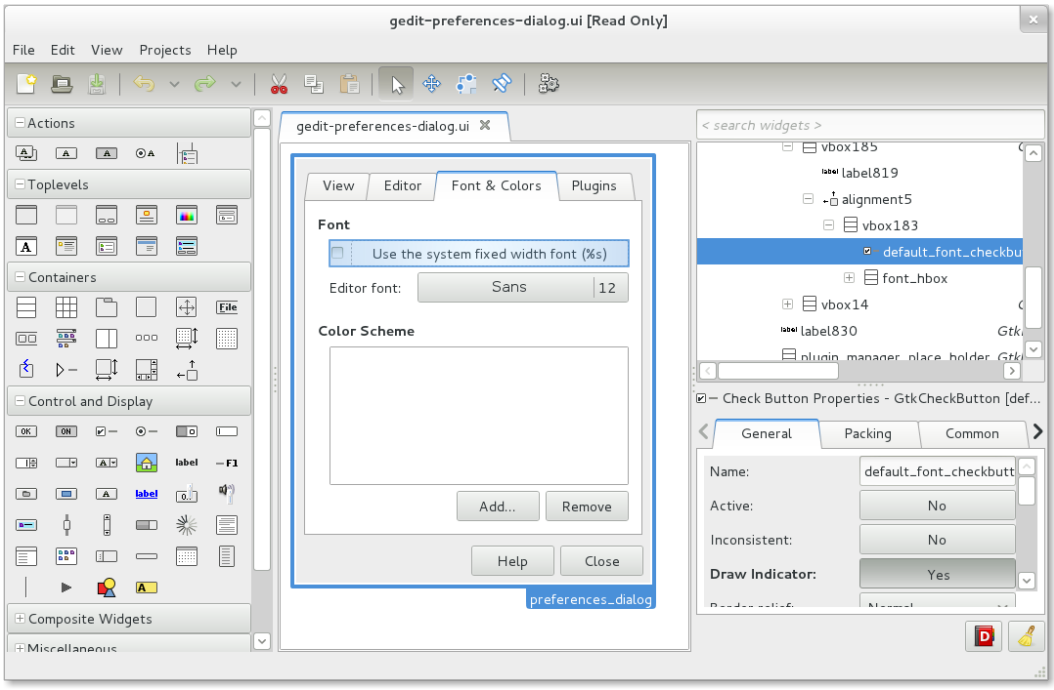
- L'interprétation de ces fichiers par leur runtime induit forcément une certaine latence qui peut, dans le pire des cas, dégrader la réactivité de l'IHM ainsi créée.

Constructeur d'interface graphique

Il est enfin possible d'utiliser un **constructeur (builder) d'interface graphique**. Il s'agit d'un logiciel WYSIWYG de création d'interface graphique.

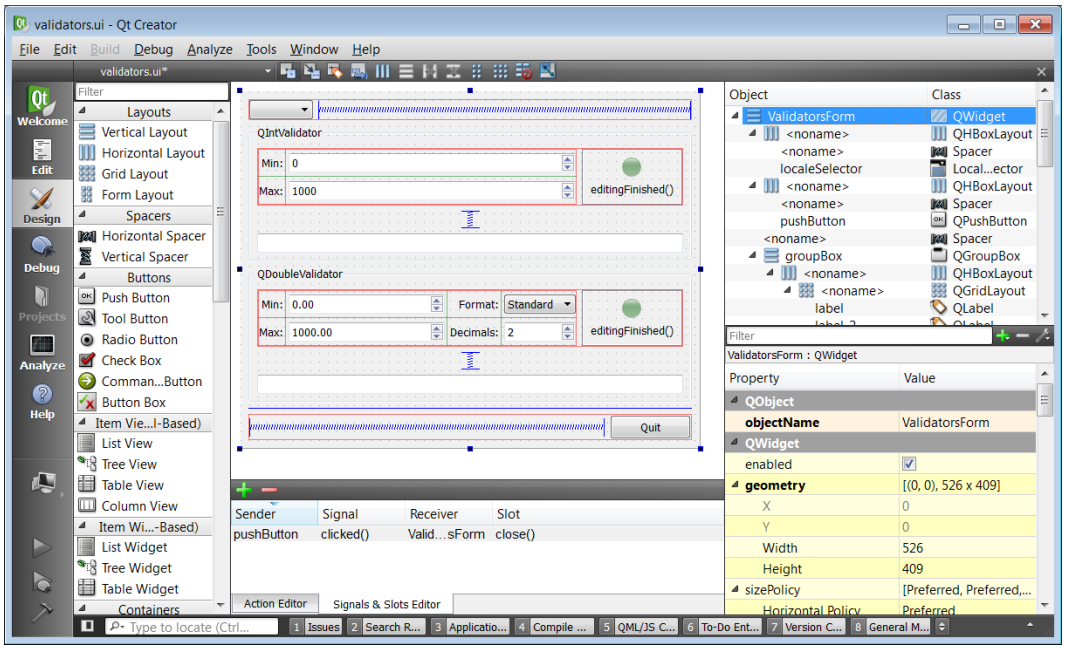
Exemple

- Glade, qui sert à créer des interfaces GTK+ :



Glade 3.6.7

- Qt Creator, qui fait partie de Qt :



Qt Designer (Qt Creator 3.1.1, Qt 5.3)

- La plupart des Environnements de Développement Intégrés (*IDE*) intègrent un constructeur d'interface graphique.

Avantages

- Ces technologies étant **WYSIWIG** (*What You See Is What You Get*), elles permettent de voir rapidement (voire de tester) le résultat d'un ou plusieurs changements dans le design de l'IHM. Cela est particulièrement intéressant pendant les phases de conception, ou un **prototype** est particulièrement utile.
- La quantité de code à réaliser est réduite.

Inconvénients

- En fait de WYSIWIG, il est plus explicite de parler de **WYSIOWYG** (*What You See Is Only What You Get*). En d'autres termes, l'utilisation de constructeur d'interface graphique peut donner un faux sentiment de sécurité ou de compétence, et faire négliger la quantité de code métier à implémenter pour rendre l'IHM fonctionnelle.
- Si le constructeur produit du code source, ce code doit être maintenu par l'équipe de développement. Or, il est souvent malaisé d'analyser et de corriger du code généré par un

algorithme …

Conception

Au bout du compte, il y a deux principales manières d'approcher la conception d'une IHM :

- l'approche **technocentriste** pense en premier lieu à la machine : Quelles sont ses contraintes de fonctionnement ? De quelle forme sont les données qu'elle traite ? Quels sont ses périphériques d'entrée ?
- l'approche **anthropocentriste** pense en premier lieu à l'homme : Quel est la problématique à résoudre ? Quel est le contexte, l'environnement auquel est soumis l'utilisateur ? Comment maximiser son envie d'utiliser le logiciel, et son efficience à le faire ?

Bien que la première approche puisse éventuellement faire gagner du temps de conception, cela n'est aucunement garanti et se fera quasiment systématiquement au détriment de l'ergonomie de l'interface produite. Si on désire concevoir une interface conviviale pour l'utilisateur, l'approche anthropocentriste est donc à privilégier dans tous les cas.

Pour cela, une règle d'or est à respecter : "" Concentrez-vous sur les utilisateurs et les tâches qu'ils ont à accomplir, pas sur la technologie. ""

Les utilisateurs

L'utilisateur, ou plutôt les utilisateurs, sont d'un intérêt crucial, puisque ce sont eux qui décideront au cours de leur usage si le système créé est convivial ou pas. Il est donc incontournable de bien cerner *qui* utilisera l'interface, de bien **identifier ses utilisateurs cibles**. Cela peut se faire à travers certaines questions précises, dont une liste figure ci-dessous à titre d'exemple.

- Qui est l'utilisateur cible ? Est-ce l'acheteur, ou quelqu'un d'autre ?
- Quelles sont ses compétences, ses connaissances ? Est-il formé au système ? Est-il même *disposé à se former* au système ?
- Quels problèmes a l'utilisateur dans son activité ? Est-il satisfait ou non de sa situation actuelle ? Quelle est aujourd'hui l'approche qu'il a de son activité ?
- Quels services est sensé rendre l'interface ? Quels problèmes aidera-t'elle à résoudre ? Quel est l'intérêt du système pour l'utilisateur ? Comment notre service s'inscrira-t'il dans son activité ? De quelle manière changera-t'il son activité ?
- Quels sont les autres facteurs qui peuvent influer sur l'utilisabilité du système ?
  - facteurs physiques ou physiologiques
  - facteurs psychologiques ou socio-culturels
  - savoirs-faire et compétences annexes

Répondre à ces questions n'est ni facile ni rapide, mais c'est vital, et les résultats de concevoir une IHM sans y répondre sont en général … décevants.

La meilleure manière de répondre à ces questions sur les utilisateurs est de *leur parler*. Éventuellement en parlant à leur management. L'utilisateur doit être associé au design de l'interface qu'il utilisera. Il n'est ni un objet abstrait, ni un simple sujet d'étude.

Une autre maxime du design d'interfaces utilisateur est qu'un système ne doit être conçu ni *pour* les utilisateurs, ni *par* eux, mais *avec* eux. Ce qu'il faut en comprendre est qu'un designer n'est pas réellement capable de penser à la place des utilisateurs, mais que ceux-ci sont le plus souvent incapable de créer leurs propres outils. Le mieux est que tous travaillent ensemble.

Profils utilisateur

Essayer de satisfaire tout le monde, destiner son système au grand public au sens le plus large, ne permet en général de ne satisfaire personne totalement. Imaginer, donner un visage à l'utilisateur cible donne des contraintes, un cadre qui aide à mieux construire l'IHM d'après les besoin réels de notre utilisateur en particulier. Certains designers vont même jusqu'à créer des **profils** d'utilisateurs types. Ces profils constituent autant de personnes fictives qui ont un nom, un métier, une famille, des hobbies, et ainsi de suite. Il est effectivement possible d'aller jusque là ; cependant, ces profils doivent être créés d'après des données réelles, collectées, factuelles … et non simplement inventés !

Leurs tâches

Les tâches de chaque utilisateur doivent être accomplies à cause d'un certain **contexte**. Ce contexte dépend de beaucoup de facteurs :

- l'entreprise cliente, sa stratégie, son historique
- ses employés, leur métier et ses caractéristiques, leurs compétences
- le marché où ils évoluent, ou -pour être précis- la perception qu'ils ont de ce marché

De la même manière qu'il est crucial de bien cerner les utilisateurs d'un système grâce à des questions pertinentes, cerner la nature exacte des tâches qu'ils ont à réaliser l'est tout autant :

- Quelles tâches particulières sont concernées par le système ?
- Quelles tâches sont les plus habituelles ? Lesquelles s'entreprennent plus rarement ?
- Quelles tâches sont les plus importantes ? Lesquelles le sont moins ?
- Comment se décompose chaque tâche en détail ? Que produit-elle ?
- Quels sont ses prérequis (données, outils, communications, …) ?
- Quel utilisateur fait chaque tâche ?
- Quels sont les problèmes rencontrés habituellement, ou les erreurs régulièrement commises au cours de l'exécution de chaque tâche ? Quelles sont les causes de ces problèmes ou erreurs ?
- Quelles tâches sont en lien l'une avec l'autre ?

Le système que vous concevez (ou achetez) n'est pas le centre de l'univers. Le contexte est important. Et la technologie seule ne résout pas tous les problèmes.

Concepts

Avant d'envisager à quoi ressemblera l'interface, il est plus profitable de l'envisager à travers les **concepts** qu'elle permettra de manier. Ses concepts doivent être correctement définis. Un bon concept doit être adapté à l'utilisateur et à la tâche. Il faut notamment faire attention aux traditions ou habitudes qu'on manie sans réfléchir.

Exemples

- La commande `mv` permet de déplacer, mais aussi de renommer un fichier. Est-ce un concept adapté à un débutant dans le monde UNIX ?
- La plupart des traitements de texte nécessitent de Sauver/Enregistrer son travail. Est-ce le meilleur concept qu'on puisse imaginer ? Ne pourrait-on pas annuler les opérations de n'importe quel document, même si on vient de le réouvrir ?
- Google est une entreprise qui brasse des milliards de dollars et emploie des dizaines de milliers de personnes. Son activité est très diversifiée.
  Or, quelle est la page d'accueil de son produit phare ? L'évolution de l'entreprise a-t'elle impliqué un changement de son concept de base ?

Vocabulaire

Lors de la conception d'une interface il est important de **clairement définir** chaque terme qu'on emploie. Il faut savoir de quoi on parle. En effet, si la définition même des termes utilisés par l'interface est floue et/ou changeante dès la conception, comment demander à l'utilisateur de s'y retrouver ?

Il est de plus nécessaire d'employer toujours le même terme dans une situation donnée. Utiliser des synonymes ou inventer de nouveaux termes (par exemple pour « éviter les répétitions ») est à éviter. L'utilisateur doit être en terrain connu : une IHM n'est pas un roman ou une rédaction.

Enfin, il faut rester conscient que le vocabulaire utilisé par l'application (ou par le designer de l'application), n'est pas toujours celui de l'utilisateur.

Scenarios

Décrire les différentes fonctionnalités de l'interface ne doit pas se faire dès le début en se référant à des éléments graphiques. En phase de design, on étudie les différentes opérations supportées par une interface afin d'en maximiser l'ergonomie, mais on ne s'intéresse pas encore, en pratique, à quoi ressembleront *graphiquement* ces opérations.

Exemple

La première manière de décrire un scénario donné manie des concepts qui pourront se traduire de n'importe quelle manière dans une interface graphique. La seconde commet l'erreur de penser trop tôt à la technologie qui supportera graphiquement le même scénario.

❶ *Alice consulte le solde de son compte courant. Elle dépose ensuite de l'argent sur son compte.*

❷ *Alice double clique sur l'icône « Mon compte courant ». L'interface affiche le statut de son compte ainsi que les dernières opérations sous forme d'un tableau. Elle clique ensuite sur « Effectuer un versement ». …*

Implémentation

L'ergonomie (*usability*) d'un système renferme plusieurs aspects. Par exemple, la norme ISO 25010 définit un système comme ergonomique s'il respecte les attributs de qualité suivants :

- **Attrait** (*Attractiveness* / *User interface aesthetics*)
  Est-ce que l'utilisateur a envie d'utiliser l'interface ?
- **Facilité de compréhension** (*Understandability* / *Appropriateness recognizability*)
  Est-ce qu'il est (intuitivement) facile pour un utilisateur de comprendre et retenir comment réaliser une opération ?
- **Facilité d'apprentissage** (*Learnability*)
  Est-ce qu'il est facile de former de futurs utilisateurs à l'utilisation du logiciel ?
- **Facilité d'exploitation** (*Operability*)
  Est-ce qu'il est facile pour l'utilisateur d'accomplir sa tâche ?
- **Protection contre les erreurs d'utilisation** (*User error protection*)
  Comment réagit le logiciel si son utilisateur a un comportement inatendu ?
- **Accessibilité** (*Accessibility*)
  Est-ce que le logiciel est utilisable par des personnes en situation de handicap ?

Mais comment, en pratique, implémenter les concepts d'un système afin que son interface ait ces qualités ?

Une manière est de faire en sorte de respecter les sept principes de base détaillés dans ce chapitre. Cependant, pour chacun, il existe souvent autant de manières de « bien faire » que de gâcher l'ergonomie de votre système. Le but n'est pas d'être complètement dogmatique dans l'application de ces principes. Rappelez-vous l'importance d'être adapté aux utilisateurs et à leurs tâches : le contexte de chaque projet est roi.

Compatibilité

Il est utile d'exploiter les connaissances que l'utilisateur possède déjà et qui concernent des interfaces similaires, que ce soit dans leur apparence ou dans leurs buts. Cela peut se traduire, par exemple, par :

- S'assurer de la cohésion des différents produits proposés par une même compagnie. Cela passe par respecter la **charte graphique** (ou *look & feel*) de l'entreprise, c'est à dire l'ensemble des règles qui lui sont propres et qui régissent l'apparence et les comportements des différents éléments de ses interfaces graphique.
- Respecter les **conventions de la plateforme** qui accueille notre interface.
- Exploiter un univers familier.

Guidage

L'utilisateur ne doit jamais être perdu ou livré à lui-même. Il doit pouvoir, en permanence :

- connaître l'**état** du système
- comprendre quel **impact** ont eu ou auront ses actions sur cet état
- pouvoir en déduire quelles actions entreprendre

Correctement guider l'utilisateur facilite son apprentissage du système et sa facilité à s'y repérer. De même, cela contribue à prévenir d'éventuelles erreurs d'utilisation dues à une mauvaise compréhension de l'état du système ou d'une opération particulière.

Il existe deux types de guidage complémentaires :

- Le **guidage explicite** se fait via des messages d'avertissement ou de confirmation, des boîtes de dialogue spécifique, un manuel utilisateur ou une aide en ligne, des codes d'erreur précis, …
- Le **guidage implicite** passe par une **structuration appropriée** de l'affichage, l'utilisation d'**éléments différenciés** que ce soit par leur couleur, leur police de caractère, le fait de les regrouper par catégorie, …

Homogénéité

Les outils informatiques cultivent et entretiennent les habitudes de leurs utilisateurs. Les utilisateurs recherchent eux-même cela : ils veulent pouvoir au plus vite tomber dans une sorte d'utilisation

inconsciente de leurs outils, afin de pouvoir les « oublier » et se concentrer sur les tâches qu'ils ont à accomplir. Plus un système informatique est **homogène**, mieux ils y arriveront.

Un système est homogène si, par exemple :

- son **look & feel** est cohérent
- sa conception est **stable**
- sa **logique d'utilisation** est apparente

Ces qualités rendent le système plus prévisible, plus facilement exploitable, plus compréhensible. Tout ceci contribue à rendre les détails propres au système « oubliables », afin que l'utilisateur puisse se concentrer sur *ce* qu'il a à faire, au lieu de *comment* le faire.

Souplesse

Un système **souple** s'adapte à l'utilisateur, à sa tâche et au support sur lequel elle tourne. Il peut, par exemple :

- proposer **plusieurs manières** de réaliser une même opération
- offrir des moyens de **personnalisation** de sa configuration

Contrôle explicite

L'utilisateur doit toujours **garder le contrôle** sur le système. De même, il doit comprendre de quelle manière ce contrôle s'exerce.

Un système ergonomique offrira un **feedback** immédiat aux actions de l'utilisateur, afin que celui-ci puisse se rendre compte de l'effet de ses actions. Ce **retour** offert à l'utilisateur doit être présent même si la commande est longue. Cela peut se faire via des messages de confirmation, des barres de progression, des voyants de statut, et ainsi de suite. À l'inverse, un *freeze* de l'IHM pendant la durée d'une opération longue est à proscrire.

Tout cela contribue à rendre les opérations permises par un système **prédictible**. Outre faciliter l'apprentissage, cela prévient les erreurs d'utilisation dues à une mauvaise compréhension de ses actions par l'utilisateur, ou à la simple impatience.

Gestion des erreurs

Lorsqu'un erreur survient, l'utilisateur doit pouvoir la percevoir et l'identifier comme une erreur. S'il est nécessaire que l'utilisateur fasse une opération explicite pour corriger une erreur, il doit être guidé afin de pouvoir faire des choix éclairés, et non livré à lui-même. Enfin, lorsqu'une erreur survient, le système doit être suffisamment **robuste** (ie. fiable) pour que cette erreur n'impacte pas la suite du fonctionnement du système.

Il faut garder à l'esprit que toute erreur n'est pas **bloquante**. Il est possible de notifier l'utilisateur de manière **non-bloquante**, c'est à dire sans que celui-ci doive interrompre ce qu'il est en train de faire, afin qu'il puisse identifier ce qui s'est mal passé plus tard, quand il le jugera utile.

Un système **tolérant** aux erreurs qui surviennent et qui offre les outils pour analyser et corriger précisément tout type d'erreur, met son utilisateur **en confiance**.

Concision

Les informations fournies à l'utilisateur doivent être **concises** et **précises**. Attention, le langage utilisé pour fournir ses information, ainsi que leur niveau de précision doivent être **adapté à l'utilisateur**. En effet, un message exposant le fonctionnement interne du système, bien que compréhensible par un développeur, *n'est pas* adapté à un utilisateur !

Cette concision s'applique aussi à la facilité d'exploitation du système. Il est important de **minimiser le nombre et la durée des actions** à réaliser par l'utilisateur pour accomplir une opération particulière.

Exemples

Restrictions arbitraires

De trop nombreuses IHMs imposent arbitrairement des restrictions à leurs utilisateurs. Par exemple :

- Limiter le nombre d'annulations d'opérations (« *undo* ») maximum possibles.
  Pour quelle raison ? Stocker un nombre indéfini d'opérations nécessite-t'il tant de ressources machine que ça ?
- Limiter la taille d'un champ de texte d'un formulaire à un certain nombre de caractères.
  Y a t'il une limitation à la base de données sous-jacente ? Si oui, pourquoi cette limitation existe-t-elle ?
- Forcer à remplir *tous* les champs d'un formulaire.
  Comment l'utilisateur est-il sensé faire s'il ne dispose pas d'une partie des informations ? Toutes ces informations sont-elles réellement indispensables ?

Parfois, ces limitations ont une raison d'être due aux limitations techniques d'un système donné. Cependant, elles sont toujours perçues par l'utilisateur comme arbitraires, frustrantes, et faciles à oublier.

En conséquence, il est préférable de **limiter** à tout prix ces restrictions, que ce soit dans leur nombre ou leur impact.

Mécanismes internes

La conception d'un système devrait prendre comme point de départ l'utilisateur et les tâches qu'il a à accomplir (voir approche anthropocentriste). Il est nécessaire, afin de maximiser l'efficience avec laquelle l'utilisateur accomplit ses tâches, qu'il puisse se concentrer sur elle, sans avoir à se préoccuper de détails d'implémentation de l'interface.

De plus, les concepts qu'utilise ce système doivent être clairement définis. Clairement définir ses concepts implique de savoir lesquels sont réellement utiles à l'utilisateur, et lesquels sont, à l'inverse, utiles au seul concepteur/développeur de l'interface.

En conséquence, les concepts et le vocabulaire propres à un système, mais non utiles à l'exécution d'une tâche, doivent être **toujours invisibles pour l'utilisateur**. En d'autres termes, il faut garder les mécanismes internes du système … à l'intérieur.

Nombreuses fonctionnalités

Lorsqu'un système offre toujours plus de fonctionnalités ou d'options, il promet davantage de puissance à son utilisateur. Cependant, ces améliorations fonctionnelles doivent nécessairement se retrouver dans l'IHM, au risque de rendre celle-ci toujours plus complexe. Il faut donc éviter que, pour l'utilisateur, de grands pouvoirs impliquent … une grande confusion !

Une IHM offrant pléthore de fonctionnalités peut être intimidante. Surtout pour un utilisateur qui n'utilise vraiment qu'une portion, parfois minime, de ces fonctionnalités.

Il vaut mieux ne pas s'aliéner les utilisateurs débutants, en cours de formation ou occasionnels. De plus, il ne faut pas freiner l'utilisateur, même expert, dans ses opérations les plus courantes en le forçant à considérer d'autres opérations offertes par le système.

En conséquence, seules les fonctionnalités les plus importantes doivent être aisément accessibles et visibles. À l'inverse, les fonctionnalités moins couramment utilisées doivent pouvoir être ignorées.

- Utiliser de **valeurs par défaut**, par exemple en :
  - initialisant automatiquement certains champs de formulaire
  - positionnant certaines options dans un menu de paramètres à part
- Utiliser de **templates** ou de **wizards** pour que l'utilisateur n'ait pas à apprendre ou à maîtriser toutes ses possibilités avant de pouvoir accomplir une opération
- Ne pas afficher les contrôles ou widgets qui ne sont pas pertinents pour l'action en cours
- Utiliser des noms de fonctionnalités génériques. Cela peut signifier que deux fonctionnalités permettant de manier deux concepts nommés différement porteront le même nom aux yeux de l'utilisateur. Par exemple, « créer », « ouvrir », « déplacer », « copier », « sauv(egard)er » ou «

supprimer » sont des appellations généralement bien comprises par les utilisateurs, qui peuvent être réutilisées dans des contextes différents, de manière à ce que l'utilisateur n'ait pas l'impression de manier ou de devoir apprendre de nouvelles fonctionnalités.

Biais cognitif

Un designer ne doit pas partir du principe que l'utilisateur connait le fonctionnement du programme, et tout spécialement pas son fonctionnement interne. L'utilisateur ne connait ni le langage, ni surtout les intentions du designer, et n'a en général pas connaissance du contexte nécessaire pour comprendre les messages d'erreur.

À titre d'exercice, avant de cliquer sur chacun des liens suivant (chacun mène vers un article de presse), lisez-en l'intitulé relativement loufoque, et essayez de vous imaginer la situation qu'ils décrivent. Ensuite seulement, cliquez et lisez l'article. Est-ce que les faits décrits correspondaient tous exactement à ce que vous aviez imaginé ?

- Rennes: Sébastien a couru après son bus pendant 4 kilomètres
- Un babouin ressucite grâce à « Staying Alive » des Bee Gees
- Il réécrit la Bible car il la trouve « mal écrite »
- Il danse en léchant un crapaud : la police l'arrête

Pour les simples utilisateurs d'un système, la situation est semblable. Il ne sont probablement pas plus idiots que le designer ou le développeur. Ils en savent même probablement davantage sur la réalité des tâches que le logiciel permet d'accomplir. Mais ils ne connaissent pas vraiment le logiciel lui-même dans le détail, et ce contexte qui leur fait défaut leur empêche par exemple de comprendre un message d'erreur qui détaille le fonctionnement interne du logiciel.

Typiquement, un développeur ou un designer pensent « *inside-out* » : il connaissent les détails internes du système, et s'en servent, même inconsciemment, pour évaluer l'ergonomie de l'interface externe. À l'inverse, un utilisateur lambda pense « *outside-in* » : il ne connaît que le logiciel de l'extérieur, et doit se servir de cette connaissance partielle pour comprendre ce qui se passe vraiment.

C'est ce biais cognitif, intrisèque aux fonctions de designer ou de développeur, qui doit être évité pour maximiser entre autres la facilité d'apprentissage et d'exploitation d'un logiciel par son utilisateur réel.

Données

L'informatique est la traitement automatique de l'information. L'utilisateur à besoin que le système qu'il utilise lui fournisse des **informations utiles**, pas qu'il se contente de lui afficher des données sans traitement. Les meilleures IHMs n'embrouillent pas l'utilisateur avec des données dont il n'a pas besoin à ce moment, mais lui présentent les informations réellement nécessaires, compilées à partir de ces données.

En particulier :

- Les messages d'erreur, ou fenêtres de confirmation ne doivent pas décrire un problème, mais proposer une solution la plus concise et utile possible, dans un langage compréhensible par l'utilisateur.
- L'esthétique d'une IHM doit certes être attrayante et cohérente. Cependant, il est parfois nécessaire de faire des compromis pour rendre la prise d'informations par l'utilisateur plus aisée.
- La quantité d'informations, leur forme et l'ordre dans lequel elles sont présentées doivent toutes être appropriées au périphérique de sortie considéré. En particulier, le contenu doit être adapté à la navigation sur mobile (voir Adaptive Design).

De manière générale, la convivialité des données présentées réside dans l'attention portée par le concepteur aux détails.

Affichage

Une maxime qui est pour beaucoup dans le fait que l'utilisateur s'approprie une interface et la trouve intuitive peut être traduite par « l'écran appartient à l'utilisateur ». Elle signifie que quand un changement survient à l'écran, l'utilisateur doit le percevoir, s'y attendre, et le maîtriser.

En pratique, cela veut dire :

- Les éléments d’une interface ne doivent pas bouger unilatéralement, sans le consentement de l’utilisateur.
- Une interface ne doit pas « trop » changer d’un seul coup (notion anglaise de*display inertia*).
  - Par exemple, si l’utilisateur effectue une « petite » opération localisée, cela ne doit se traduire à l’écran que par un « petit » changement localisé.
  - Si, à l’inverse, il est nécessaire de grandement changer ce que l’interface affiche à l’écran, ce changement doit être d’une part annoncé, attendu par l’utilisateur, et d’autre part le moins disruptif possible pour l’interface.
- Les changements doivent apparaître à l’écran de manière visible par l’utilisateur. Ils ne doivent pas être si infimes que l’utilisateur ne les remarque pas. Ils ne doivent pas non plus être dissimulés par d’autres éléments de l’interface elle-même, comme une boîte de dialogue par exemple.
- Une interface ne doit pas (ré)ordonner ses éléments différement de ce que l’utilisateur attend, ou de ce qu’il a configuré. Même si la volonté est d’aider l’utilisateur, en lui proposant par exemple un tri automatique « plus logique » : la logique du concepteur n’est pas forcément celle de l’utilisateur.
  Des opérations de tri ou de classement peuvent évidemment être proposées, mais elles doivent être explicites et au choix de l’utilisateur.
- Une interface ne doit *jamais* déplacer le pointeur de souris elle-même. En aucun cas.

Couleurs

Usage

Dans le domaine des IHM, les couleurs sont utilisées pour :

Attirer l'attention

Les éléments d'intérêt gagnent à être représentés par une couleur qui se détache du reste de l'interface. De plus, il est possible d'utiliser tout un assortiment d'effets de couleurs (incluant des mouvements, des dégradés, du clignotement, …) pour attirer une attention particulière sur un élément particulier.

Indiquer un état

Les utilisateurs sont habitués à associer certaines couleurs à un statut particulier. Par exemple, le rouge sera associé aux erreurs ou à un danger, tandis que le vert sera perçu comme un succès ou un marqueur de sécurité.

Organiser les relations entre éléments

Différents éléments de même nature peuvent être facilement identifiés par la réutilisation du même code couleur. De même, un dégradé de couleurs pourra permettre de hiérarchiser les éléments par leur ordre d'importance, par exemple.

Palettes

Il est important pour l'attrait d'une interface d'utiliser une **palette de couleurs** bien choisie. C'est là le travail des artistes, mais il est possible d'isoler certaines règles de base :

Less is more

Il vaut mieux minimiser le nombre de couleurs différentes. Multiplier le nombre de couleurs fait courir le risque d'aboutir à une concurrence entre ces couleurs pour l'attention de l'utilisateur.

Le confort visuel doit primer

En effet, même si une palette de couleurs attrayante est évidemment préférable à une autre moins esthétique, cette notion d'esthétique est souvent nébuleuse et subjective. Par contre, il est vital de garder en tête qu'une interface graphique *n'est pas* quelque chose qu'on accroche au mur au-dessus d'un canapé, mais une chose qui sera affichée par un écran, et devant laquelle l'utilisateur passera des heures : il est important donc de **minimiser la fatigue visuelle**.

Différences selon l'utilisateur

Une palette de couleurs sera non seulement perçue différemment, mais devra souvent être affichée différemment selon la personne (daltonisme, …) et le matériel (type & réglage de l'écran) considérés.

Relations de voisinage

Les couleurs qui « vont le mieux ensemble » sont souvent liées par des relations de voisinage particulières sur le cercle chromatique. On peut distinguer plusieurs relations intéressantes :

- Les objets rendus dans des **couleurs chaudes** apparaissent plus proches, plus grands, plus accueillants, tandis que ceux affichant des **couleurs froides** apparaissent plus éloignés, plus petits, plus professionnels.
- Les **couleurs complémentaires** sont **diamétralement opposées** sur le cercle chromatique. Pour davantage de variété, plutôt que prendre la couleur diamétralement opposée à une couleur, on pourra choisir ses voisines.
- Il est possible de jouer sur les **harmonies triadiques**, en prenant trois couleurs équidistantes, qui dessinent un triangle équilatéral sur le cercle chromatique. Les **harmonies tetradiques**, elles, associent quatre couleurs qui, si on les relie, forment un carré ou un rectangle sur le cercle chromatique.
- Les **couleurs analogues** sont voisines sur le cercle chromatique.
- Enfin, on pourra utiliser des **monochromes** de couleurs situées au même endroit du cercle chromatique, mais offrant des niveaux de saturation ou de luminosité différents.

Interfaces multi-support

Aujourd'hui, une IHM n'a plus forcément vocation à être affichée sur un seul type d'écran. L'heure est aux produits **multi-supports**, et les IHM doivent suivre. Parmi les supports les plus courants, on peut citer :

- l'« ordinateur » classique (*desktop* ou *laptop*) : support historique, encore souvent nécessaire ;
- le téléphone mobile (smartphone,…) : sa part de marché a explosé ces dernières années ; pourtant, l'expérience utilisateur d'Internet sur mobile est encore loin d'être parfaite
- les tablettes : la taille de leur écran les rapproche d'un PC, mais leur modèle de navigation tactile les rapproche davantage d'un téléphone mobile … pourtant, la navigation tactile sur tablette offre des différence notables ;
- une foule d'autres supports : montres connectées, TV connectées, etc.

Aujourd'hui, même la conception d'une interface qui se limite au duo « PC et smartphone » est bien souvent obsolète.

Il y a certaines spécificités des téléphones mobiles à garder en tête lors de la conception d'une IHM :

- La relativement petite taille de l'écran conditionne les éléments qui peuvent être affichés. Mais comme l'écran est tactile, les contrôles (boutons,…) ne peuvent pas être inférieurs à une certaine taille.
- La navigation sur téléphone mobile se fait souvent « à une main », voire avec le seul pouce. La postion des contrôles doit donc être pensée en fonction de cette contrainte.
- Sur mobile, l'utilisateur a parfois un besoin très spécifique à remplir, et/ou peu de temps à y consacrer. Une IHM approprié à l'usage mobile doit donc offrir uniquement ce dont l'utilisateur a besoin.

Recycler à l'identique une IHM déjà existante sur PC afin de se contenter de l'afficher sur un écran plus petit n'est donc pas une solution. Pourtant, par exemple, trop de sites web forcent encore leurs utilisateurs à zoomer/dézoomer et à faire défiler sur deux axes une page qui n'a pas été pensée pour la navigation mobile.

Responsive design

Ethan Marcotte a défini la notion de Responsive Design en 2010 dans un article pour le magazine A List Apart.

Cette conception implique un design unique qui s'adapte **automatiquement** pour tous les écrans (c'est à dire pour toutes les tailles de viewport) grâce à une mise en page **fluide**. Peu importe l'appareil sur lequel est affichée l'interface graphique, seule la taille de l'écran compte.

Par exemple, des éléments d'un menu disposés horizontalement sur un écran large « flottent » verticalement lorsque la largeur de l'écran se réduit. Une grille de 4×1 éléments pourra ainsi se voir successivement redimensionnée en 2×2, puis 4×1.

Avantages

- Tous les formats d'affichage sont supportés : les formats présents, mais aussi (théoriquement) à venir sont pris en compte. Cette conception est donc très **pérenne**.
- Cette solution offre un avantage au niveau de la maintenance.

Inconvénients

- Cette manière de faire pose des questions de **performance** et de **dimensionnement**.
- L'enfer est dans les détails : envisager une solution purement responsive est souvent illusoire.
- Ce design a évidemment un coût.

Adaptive design

Le terme d'adaptive design a été utilisé pour la première fois par Aaron Gustafson dans son ouvrage *Adaptive Web Design*.

Il implique de concevoir un design adapté, **optimal pour chaque appareil** sur lequel doit s'afficher une IHM. Cela entend une stratégie de contenu propre à chaque support et ce, dès la conception. L'adaptive design est centré sur l'expérience utilisateur ; d'où la nécessité de comprendre les

besoin et habitudes de navigation (potentiellement différentes) des utilisateurs de chaque appareil. Puisque l'accent est mis sur les phases d'analyse du besoin et de conception, une grande rigueur est nécessaire.

Avantages

- Ce type de conception offre davantage de contrôle, une maîtrise fine de ce qui est affiché sur quel appareil et comment.
- Les contenus étant adaptés à chaque appareil, l'adaptive designe offre en général de meilleures performances.

Inconvénients

- Puisqu'il n'y a pas un seul design mais plusieurs, les coûts de maintenance peuvent être plus importants.
- Qu'en est-il des appareils non prévus ?

Choisir

Gradient Chart

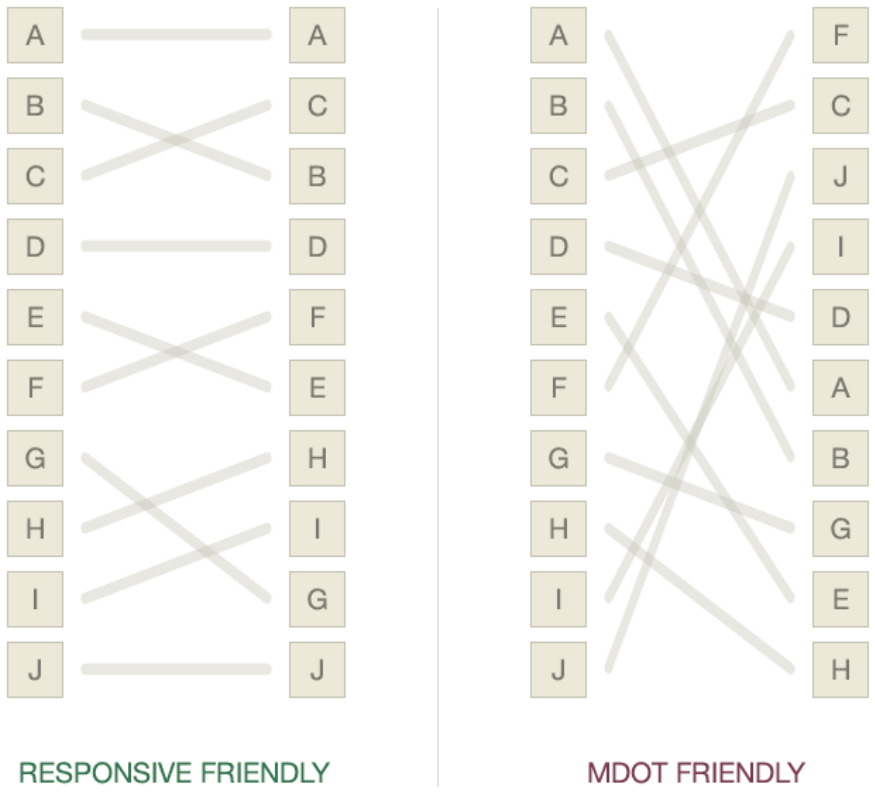
Une méthode préliminaire est celle du « Gradient Chart » de Cennyd Bowles :

❶ Premièrement, lister toutes les fonctionnalités que l'interface doit remplir

❷ Puis, les ordonner en fonction de leur importance pour les utilisateurs de chaque appareil

❸ Enfin, relier chaque fonctionnalité de même nom par une ligne :

- si les lignes ne se croisent pas tellement, un design responsive pur peut à priori être adapté.
- si toutes les lignes sont entremêlées, il vaudra mieux partir sur un adaptive design.



Exemple : Gradient Chart, Cennyd Bowles

Amélioration progressive

La méthode d'amélioration progressive consiste à débuter avec la base la plus simple (par exemple *mobile first* ou *desktop first*), puis à construire par dessus, à améliorer l'interface en fonction des capacités de chaque appareil.

Comme toujours, meilleure est la connaissance de l'utilisateur, meilleure sera l'interface destinée à remplir ses besoins.

Techniques

Voici un aperçu rapide des techniques de bases qui rendent possible un design multi-support.

Unités absolues, Unités relatives

Il est important de manier correctement les différentes unités disponibles. Ces unités, parfois nombreuses, se répartissent en deux catégories :

- les unités **absolues** : pixels, centimètres, pouces, …
- les unités **relatives** : pourcentage de la taille de l'élément parent ou du viewport, fonction de la taille du texte, …

Mise en page fluide

C'est le contenu qui doit décider de la mise en page, et pas seulement la taille de l'écran. Cela signifie qu'un élément qui prend davantage de place va naturellement « pousser » les élements voisins de manière harmonieuse.

Dimensions maximum

C'est une chose que d'avoir une interface qui s'adapte au plus grands écrans. Cependant, il faut aussi penser au delà de quelle taille les éléments ne doivent pas grandir, pour éviter les effets d'étirement par exemple.

Images

Les images affichées peuvent être de deux types : **bitmaps** (*.jpg*, *.png*, *.gif*, …) ou **vectorielles** (_.svg, …). Une image vectorielle offre l'avantage de pouvoir être affichée à n'importe quelle résolution sans perte de qualité, alors qu'une image bitmap agrandie apparaitra pixelisée. La question de la taille mémoire de l'image est à garder à l'esprit. De même, il est nécessaire de s'assurer du bon rendu des couleurs (en particulier sur le web : attention aux palettes et aux optimisations).

Polices de caractères

- Les polices **web** offrent un rendu souvent intéressant et original, mais doivent être téléchargées, ce qui induit une latence d'affichage.
- Les polices **système** permettent un affichage rapide, mais elle peuvent ne pas être installées sur le système hôte.

Dans tous les cas, une bonne pratique est de prévoir une ou plusieurs **polices de remplacement** (*fallbacks*). C'est pourquoi on parle plutôt aujourd'hui de **familles de polices** (*font families*) affectées à un contenu particulier.

Breakpoints

Fondements du *adaptive design*, les breakpoints permettent de fixer une limite pour affecter une caractéristique donnée à une interface. On distingue notamment :

- les breakpoints **majeurs**, qui altèrent complètement la mise en page de l'interface ;
- les breakpoints **mineurs**, qui lui apportent de plus légers changements : par exemple, au niveau de espacement entre les éléments :
- les breakpoints **partiels**, qui sont … entre les deux. Ils apportent en général un changement important, mais limité à un **groupe d'éléments** (*nested objects*) particulier.

Égalité devant le contenu

La notion d'égalité devant le contenu (*content parity*, ou **One Web*(W3C)) consiste à rendre disponible les mêmes informations et services à tous les utilisateurs. Et cela, **quel que soit l'appareil** qu'ils utilisent pour accéder à ces contenus. Cela ne signifie cependant pas que les contenus seront affichés exactement de la même manière quel que soit l'appareil. Au contraire, les contenus devraient être présentés de la manière **la plus appropriée** à chaque appareil.

Ces buts sont évidemment à poursuivre dans la limite du raisonnable. En particulier, des points comme les capacités techniques ou d'input des appareils ou, dans un contexte mobile, les considérations de réseau peuvent affecter la présentation du contenu.

Accessibilité

Définition et contexte

L'accessibilité numérique est la mise d'un à la disposition de tous les individus, quels que soient leur matériel ou logiciel, leur infrastructure réseau, leur langue maternelle, leur culture, leur localisation géographique, ou leurs aptitudes physiques ou mentales, des ressources numériques.

L'accessibilité est généralement associée à la notion de handicap. En effet, le handicap de l'utilisateur d'un logiciel influe directement sur sa capacité à l'utiliser.

Il existe en fait bien des types de handicaps différents, temporaires ou permanents, qui ont pour la majorité différents dégrés ou formes spécifiques :

- les handicaps **visuels** (cécité totale ou partielle, daltonismes, épilepsie, …) empêchent de percevoir tout ou partie des aspects graphiques d'une interface.
- les handicaps **auditifs** (surdité totale ou partielle, hyperacousie, acouphènes, …) empêchent de percevoir tout ou partie des aspects sonores d'une interface.
- les handicaps **cognitifs** (autisme, dyslexie, dyscalculie, TDAH, …) freinent l'apprentissage, la compréhension et/ou l'exploitation d'une interface.
- les handicaps **moteurs** (paralysies, atrophies, dyspraxie, syndrome du canal carpien, foulures, fractures, …) entravent la faculté d'exploitation d'une interface.

Cependant, les personnes en situation de handicap ne sont pas les seules concernées par l'accessibilité d'une interface. On peut citer par exemple le cas d'une personne distraite par autre chose qui aura ses capacités de réaction et d'opération décrues, celui d'une personne opérant dans un environnement bruyant, dont la carte son est cassée ou qui a tout simplement ses enceintes éteintes qui bénéficiera de contraintes semblables à celles d'une personne sourde, le cas évident d'une personne ne maîtrisant pas la langue de l'interface, et ainsi de suite. L'accessibilité d'un logiciel concerne donc tout le monde.

Enjeux

Les enjeux de l'accessibilité sont nombreux :

- **économiques** : certains handicaps sont plus courants qu'on ne croit. Par exemple le daltonisme, qui concernerait 1 homme sur 7, ou encore le vieillissement actuel de la population des pays développés et tout ce qui l'accompagne (devenir « dur d'oreille », presbyte, souffrir de DMLA, de tremblements ou de pertes de mémoire). Dans ce cadre, il parait difficile pour une entreprise d'ignorer les (parfois nombreuses) part de marché que représentent les personnes en situation de handicap, qui peuvent constituer autant de clients potentiels.
- **légaux** : en France, le cadre légal est encore relativement peu contraignant pour les entreprises du numérique. Cependant, le secteur d'activité (transports, administration, …) d'une entreprise peut quand même la contraindre à prendre certaines mesures pour améliorer l'accessibilité de ses produits.
- **techniques** : rendre un logiciel accessible consiste avant tout à respecter certaines bonnes pratiques de conception et d'implémentation. Ces bonnes pratiques ne découlent bien souvent pas directement du sujet de l'accessibilité, et leur bénéfice premier est bien souvent autre (pérennité, maintenabilité, interopérabilité, …).
- **sociétaux** : l'informatique, en étant accessible, peut devenir un fantastique outil d'autonomie et d'intégration à la société. Et, puisque l'accessibilité d'un logiciel ne concerne pas les seules personnes en situation de handicap, il peut bénéficier à tous. Par exemple, la télécommande a initialement été conçue pour les personnes handicapées !

Solution

Les situations dans lesquelles les utilisateurs ont besoin d'une interface accessible sont diverses et variées. De plus, des stratégies, produits ou aides spécifiques existent déjà pour adresser certains problèmes. Parmi eux, on peut citer entre autres :

- les agrandissements d'éléments (loupe, …)
- les lecteurs d'écran utilisant la synthèse vocale
- les plages braille ou périphériques d'entrée spécifique
- les générateurs de sous-titres

- les personnalisations de l’affichage

Il parait donc difficile de répondre de manière exhaustive et appropriée à chacun de ces usages. La solution est de respecter certains standards :

- les **normes**, par exemple issues d’organismes comme l’ISO ou l’AFNOR
- les **guidelines** spécifiques à chaque plateforme, comme par exemple les recommandations du W3C, qui forme la base de plusieurs référentiels nationaux.
- les **standards** et **usages**, tacites ou non, en termes de protocoles de communication ou d’implémentation
- concevoir son application de manière **modulaire**, en *séparant le fond de la forme*

Toutes ces sources ne sont bien souvent pas centrées sur l’accessibilité, mais comportent soit une partie qui lui est dédiée, soit un tas de bonnes pratiques qui permettent naturellement l’accessibilité.

Principes

Internationalisation

L'adaptation d'une IHM à des usagers issus d'une ou plusieurs cultures différentes recouvre plusieurs notions :

- La **traduction** des éléments textuels affichés
- La **localisation** (*localization* ou *l10n*, parfois aussi appelée régionalisation) est l'adaptation de l'interface à une autre langue, et même au-delà, à une autre culture. Ce processus est bien plus vaste que la simple traduction.
- L'**internationalisation** (*internationalization* ou *i18n*, parfois aussi appelée globalisation) est l'ensemble des techniques de conception et d'implémentation rendant possible la localisation. L'internationalisation est un prérequis à une bonne localisation.

Traduction

- L'**Unicode** étant le seul encodage unifiant toutes les écritures informatiques, il est particulièrement intéressant qu'une application fonctionne principalement dans cet encodage
- Les règles de **pluriel** et de **grammaire**, par exemple la manière dont une phrase est construite, varient largement d'une langue à l'autre.
- De manière générale, il vaut mieux éviter de recourir à des expressions imagées, qui peuvent être difficile à traduire en dehors d'un contexte linguistique donné.
- Davantage que la notion de seule langue, il est important de manier une **locale**, qui est un couple (langue, pays) représentant le fait qu'une langue peut être parlée dans plusieurs pays, et qu'un même pays peut compter des locuteurs de plusieurs langues. Une locale est représentée par un code standardisé, tel que `fr_FR` (le français parlé en France), `en_US` (l'anglais parlé aux États-Unis), `fr_CA` (le français parlé au Canada), `en_CA` (l'anglais parlé au Canada), et ainsi de suite.
- Outre les mots et les phrases, il est nécessaire d'actualiser le format des **nombres** entiers ou décimaux, des **dates** et des heures, les **unités** de poids et de mesure, …
- Attention à n'oublier aucun endroit dans lequel apparaissent des mots ou des phrases : au sein des images et icônes, ainsi qu'au niveau sonore (au niveau d'une synthèse vocale par exemple).
- Traduire les libellés en change fatalement la longueur. Il est donc indispensable de valider la **robustesse de la mise en page** de l'interface, car les éléments affichés peuvent l'altérer grandement.

Localisation

- Il est souvent nécessaire de mettre à jour les monnaies utilisées ainsi que les coordonnées demandées par le système.
- Il est aussi utile de s'intéresser à la manière dont changer de culture influe sur les **tris** ou l'**ordonnancement** des éléments au sein de l'interface, en particulier si c'est l'ordre alphabétique qui est utilisé.
- Les **types de clavier** (QWERTY, claviers en langues asiatiques, …) ainsi que les **habitudes** des utilisateurs qui les utilisent doivent être pris en compte.
- Certains **symboles** ou **icônes**, de même que certaines **syntaxes** ou **couleurs** sont propres à une culture donnée.
- De même, attention à la **formulation** des textes et images, qui peut heurter la **sensibilité** de certains utilisateurs issus d'une culture spécifique.
- Alors que nos habitudes d'Européen peuvent ne nous faire considérer que les langues qui se lisent de gauche à droite (on parle de langues *L2R*, pour *Left to Right*), certaines langues se lisent de droite à gauche (*R2L*, pour *Right to Left*), voire de haut en bas. Ces autres **sens de lecture** ne concernent pas que les textes, mais peuvent nécessiter d'altérer la position à l'écran des différents éléments d'une interface.
- Les **réglements et lois** spécifiques d'un pays donnés peuvent avoir une influence sur ce que peut faire ou pas un système, et cela concerne aussi les IHM. En particulier, il faut s'intéresser à la légalité de demander certaines informations relatives à la vie privée dans un formulaire d'inscription ou d'achat, par exemple. Au delà de la stricte légalité, certaines cultures peuvent ne pas être habituées à ce qu'elles considèrent comme des intrusions dans leur vie privée, ou encore n'être pas habituées ou simplement équipées pour des fonctionnalités liées aux réseaux sociaux.

- Les utilisateurs d'une culture donnée peuvent avoir certaines **attentes** ou habitudes concernant la mise en page et/ou la densité d'informations affichées par certains types d'interface. Par exemple, alors qu'en Europe, la tendance est plutôt aux images et à un certaine baisse de la lecture, en Asie, la densité de texte affichée est en général bien plus importante.

Internationalisation

L'internationalisation consiste à **préparer** son adaptation à des langues et des cultures différentes. C'est un travail essentiellement technique, de conception et d'implémentation. Pour qu'un logiciel puisse être considéré comme étant internationalisé

- Il doit de manière générale **éviter les à-priori** concerna la construction des informations textuelles affichées par l'interface. En particulier, on peut citer comme exemple :
  - n'effectuer **aucune concaténation** de chaînes de caractères pour construire un message affiché à l'utilisateur ; en effet, toutes les langues n'ont pas la même façon de construire leurs phrases.
  - éviter toute **dépendance** du code envers les libellés affichés ; par exemple, comparer le libellé d'un bouton appuyé avec des valeurs codées en dur pour savoir quelle opération déclencher est la pire des manières de faire.
- gerer correctement les **encodages** de chaînes de caractères produites en sortie et attendues en entrée.
- détecter les **préférences culturelles** de l'utilisateur chaque fois que cela est possible, et les rendre configurables dans tous les cas.
- **séparer le fond et le forme**, afin que la forme (constituée par l'interface) puisse être remplacée ou altérée par configuration ou par des outils tiers.

Validation

Il existe différentes manières complémentaires pour valider l'accessibilité d'un logiciel :

- Suivre un check-list basée sur les recommandations de la plateforme cible
- Utiliser des lecteurs d'écrans et/ou d'autres outils, les mêmes utilisés par les utilisateurs en situation de handicap
- Les interfaces web peuvent bénéficier de visualisations par un navigateur uniquement textuel, comme lynx
- Faire appel directement à un panel approprié de testeurs en situation de handicap lors des tests d'utilisabilité.

Tests

Même un système qui satisfait toutes les exigences du client (c'est à dire qui respecte totalement sa spécification) n'est qu'utile *potentiellement*. En effet, un système qui possède toutes les fonctionnalités requises ne sert à rien si ces fonctionnalités ne sont pas exploitables, que le système n'est pas utilisable *en pratique*. Un système réellement utilisable, réelement convivial fera preuve de **qualités ergonomiques** telles que la facilité d'apprentissage, la facilité d'exploitation, la protection contre les erreurs d'utilisation, et ainsi de suite. D'où l'importance de **tester**, de **valider** que le système possède effectivement ces qualités.

Valider l'ergonomie d'un système se fait via des *usability tests* (ou « tests d'utilisabilité »).

Durant ces tests, ce n'est jamais les designers/développeurs d'un système qui l'utilisent eux-mêmes : ils en savent trop sur l'objet à tester, et cette connaissance induit un biais cognitif qui risque de les faire tomber dans une vue technocentriste du système.

Les tests ont un rôle double :

- informatif : le but premier d'un test est de trouver ce qui est perfectible afin d'améliorer l'interface ;
- social : faire tester un système par une personne extérieur permet de convaincre plus facilement les designers/développeurs qu'il y a réellement un problème dans le système.

Types de tests

Il existe différentes manières de tester l'ergonomie d'un système :

- Un **test informel** a pour principe de laisser l'utilisateur « seul » avec le système. On le libre de choisir quelles tâches il entreprend et de la manière dont il s'y prend. Éventuellement, on peut lui demander d'exécuter les tâches d'une journée de travail habituelle.
- Un **test formel** guide davantage l'utilisateur. Il est possible de lui faire exécuter une même tâche avec différentes interfaces (pour comparer différents designs), de lui imposer d'accomplir des tâches particulières, dans un certain ordre, et ainsi de suite.
- Un **test A/B** vise à comparer deux designs d'interface différents. Si le but est de comparer plus de deux designs, on appelle cela un **test A/Z**.

Une règle à respecter absolument dans tous les cas est que le designer/développeur ne doit **jamais intervenir**, ne jamais guider l'utilisateur, même pour « essayer de l'aider ». Par contre, il faut **toujours observer** l'utilisateur, prendre des notes sur ce qu'il fait, observer ses réactions, car on **apprend toujours** quelque chose d'un tel test sur la qualité du système.

Il est aussi utile de s'assurer que l'utilisateur est conscient que ça n'est pas *lui* qui est testé, mais le système, afin de s'assurer de la sincérité de ses actions et de ses réponses.

Outils de mesure

- Il est possible de fournir à l'utilisateur des questionnaires de satisfaction, de lui demander son appréciation … Cependant, ce type d'information est souvent difficile à quantifier. De plus, l'utilisateur peut, de bonne foi, oublier certains détails pourtant vitaux pour la compréhension de l'ergonomie du système.
- On peut aussi demander à l'utilisateur de **penser à voix haute** (« *think aloud* »), au fur et à mesure qu'il intéragit avec le système. Cette façon de faire donne souvent des résultats plus sincères et utiles que la première, mais aussi plus « brutes », plus difficiles à exploiter. À noter que cette manière de mener le test peut aussi être approprié pour valider le contenu d'un manuel ou une documentation.
- Il existe aussi des **méthodes analytiques** de validation d'un système, dans les détails desquels ce cours n'entrera pas.

Quand tester ?

On peut tester à toutes les étapes du développement, mais pas de la même manière ni avec les mêmes enjeux.

- *avant le développement*, en présentant à l'utilisateur des maquettes sur papier, des slides informatiques, voire même une personne qui dialogue avec l'utilisateur en « jouant le rôle » de l'interface ;
- *pendant le développement*, en utlisant des prototypes plus ou moins fonctionnels, des *mockups*,

ou une interface graphique représentant l'état des développements et ce, même si les fonctionnalités réelles ne sont pas encore implémentées ;

- *après le développement*, il est rarement possible de changer grand chose puisque le logiciel est pour ainsi dire dans sa forme finale. Cependant, on peut quand même être à la recherche de « *showstoppers* », des graves problèmes qui gênent gravement l'utilisateur. De plus, tester un logiciel terminé permet toujours d'apprendre quelque chose pour les *releases* futures.

Last updated 2019-09-04 08:42:24 UTC