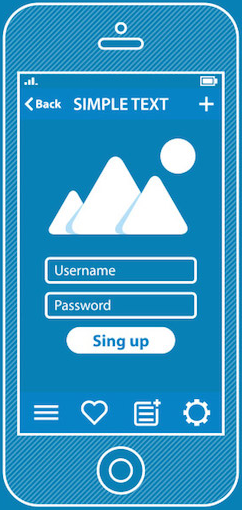
**Diseño de Interfaces de Usuario**



**Nombre**: Zabai Armas Herrera

**Curso**: 4º

Índice:

1. Reacciones y Actitudes del usuario
2. Reglas Empíricas de Diseño de Interfaces

**1. Reacciones y Actitudes del usuario**

**¿Qué objetivos desean alcanzar los usuarios con el uso de la aplicación?**

Es importante plantearse que se desea alcanzar y conseguir con la aplicación. Con esto en mente se consigue un software concreto, el cual se centra sólo en las tareas de un determinado tema o categoría. De esta manera se evita una aplicación muy ambigua, es decir, que intente abarcar demasiado y se desvíe de su objetivo principal, acabando en un mar de ideas que no terminen de funcionar tan bien como cabía esperar.

**¿Qué conjunto de tareas pretende apoyar la aplicación?**

**¿Qué tareas son comunes y cuáles son las más inusuales?**

Es posible identificar las tareas más comunes estudiando cuales son las más usadas. De la misma forma también se puede identificar las más inusuales.

Teniendo las más comunes en mente se puede poner en mejor disposición los accesos a dichas tareas en la interfaz.

Respecto a las inusuales habría que ver si es una tarea importante para intentar cambiarla o quitarla.

Proporcionando un entorno de trabajo más productivo, limpio y cómodo para el usuario final.

**¿Qué tareas son importantes y cuales son menos importantes?**

Hay que tener muy claro cuáles son las tareas importantes de la aplicación, en otras palabras, que generen un resultado relevante para el usuario o ayuden al objetivo de la misma. Esto se puede relacionar con la pregunta anterior, porque puede darse el caso de que una tarea de cierta importancia (como calcular un presupuesto) sea una tarea inusual debido a su escaso uso. Estas funciones deben estar depuradas al 100% con los casos más críticos que se lleguen a dar.

Las menos importantes son las que generan información adicional para ayudar al usuario o funcionalidades extra, que en realidad no aportan nada vital al objetivo del usuario. Actualmente se hacen estudios sobre el uso de las funciones de aplicaciones y, por mayoría un 80% del código de las mismas nunca se ejecutan. Por lo que los usuarios sólo usan el 20%, que son las tareas realmente importantes.

Controlando las tareas a diseñar se evita programar funciones innecesarias y centrarse en hacer las funciones fiables y rápidas.

**¿Cuáles son los pasos de cada tarea?**

Para diseñar las tareas es recomendable tener un diagrama de flujo, en el que ver plasmado todos los pasos necesarios que se necesitan ejecutar antes de cada tarea. Evitando así que la aplicación permita al usuario hacer tareas de una forma errónea.

Hoy en día existe el denominado el Lenguaje Unificado de Modelado (Unified Modeling Language → UML) que aporta un convenio para realizar diagramas de todo tipo (estructurales, comportamiento, interacción…). Tiene varios diagramas de comportamiento como, por ejemplo: el diagrama de flujo y el diagrama de historias de usuario.

**¿Cuáles son los resultados y salidas de cada tarea?**

Los resultados y salidas de cada tarea han de ser transparentes, que no den pie a confusiones por parte del usuario. Dejarlas bien claras en la interfaz para perder el mínimo de tiempo posible.

Hacerse esta pregunta antes de diseñar la aplicación aporta diversas ventajas, como tener una mejor visión de los resultados y llevar un desarrollo guiado por pruebas para la aplicación. Un método que consiste en desarrollar un código a base de pruebas, las cuales una vez superadas se re-factoriza el código a producción.

**¿De dónde viene la información de cada tarea?**

Se necesita saber de antemano de donde va a sacar la información que va a procesar cada tarea para obtener un resultado.

Si necesita ser introducida por el usuario, esa tarea no podrá ser ejecutada hasta que se la proporcione. Situación que cambiaría si se la proporcionara el propio sistema o una base de datos.

Estas preguntas pueden determinar la productividad del software, ya que no hace falta que el usuario intervenga para ciertas tareas. Así evitamos que la aplicación se quede ociosa durante un tiempo, aún más si tiene que “refrescar” las tareas cada cierto tiempo o tiempo real.

**¿Cómo se utiliza la información producida de cada tarea?**

¿La información que resulta de una tarea se muestra al usuario? ¿Se almacena en el sistema? ¿O la transmite a otra tarea? Este tipo de preguntas de diseño producen que la implementación pueda ser simple o algo más compleja. La transparencia de cómo tratar los datos ayudan a que el tiempo de implementación se reduzca y de un mejor servicio al usuario final.

**¿Qué personas realizan qué tareas?**

Si en una aplicación de gran escala estuvieran todas las tareas, el usuario final sería bombardeado con demasiada información, haciendo necesario la lectura de la documentación.

Una solución para este problema es introducir roles en la aplicación. Si tiene muchas tareas que abarcan distintas categorías como, por ejemplo: administración, economía, diseño, y planificación… Sería idóneo tener roles que permitiera ver funciones de las categorías asignadas a ese rol. Así, el que se encarga de temas financieros no tendría que ver otras tareas, teniendo una interfaz más clara y concisa.

Para esto se recomienda diseñar un diagrama UML de casos de uso, con el propósito de tener una visión general de los roles y las acciones que pueden realizar en la aplicación.

**¿Qué herramientas se usan para realizar cada tarea?**

**¿Qué problemas tiene la gente a la hora de realizar cada tarea? ¿Qué tipo de fallos son los más habituales? ¿Qué los causa? ¿Cuán dañinos son los fallos?**

Con el software en pruebas o una versión experimental se puede ver que problemas presentan la mayoría de usuarios con respecto a las funciones. Esto se traduce a pulir aspectos como la interfaz, las tareas y/o la documentación.

Una vez identificados, hay que priorizarlos según su impacto. Puede que un problema sea bastante común y los usuarios pierdan tiempo, pero otro que tenga un índice más bajo y pierda datos es más grave.

Los fallos pueden ser debidos a muchos factores: interfaz engorrosa con demasiada información o mala disposición de tareas. Instrucciones poco precisas que confundan al usuario o una documentación que no explique al completo todos los aspectos de la aplicación.

**¿Qué terminología usan las personas que hacen estas tareas?**

Un buen software es aquel en el que sus usuarios se sientan cómodos trabajando con él. Una forma de conseguirlo es utilizar una terminología determinada por el tipo de aplicación y a los clientes que está destinado.

Por ejemplo 2 aplicaciones de diseño gráfico: Paint y Paint.net.

* Paint es una aplicación básica incluida con casi cualquier versión de Windows. Dando a entender que sólo se va a usar para situaciones simples. Se puede apreciar en su interfaz simple y minimalista, su escasa cantidad de opciones, y en que usa un lenguaje que no requiere ningún conocimiento previo de diseño. No es una mala aplicación, sino que no está dirigida para diseñadores profesionales.
* Paint.net sin embargo es un software gratuito dirigidos para diseñadores con un determinado nivel. No hace falta nada más que abrirlo para darse cuenta de que la interfaz es mucho más compleja, personalizable y con muchísimas más opciones y configuraciones. Introduce muchos más recursos y herramientas a la hora de diseñar e incluso a la hora de exportar un diseño. Por lo que usan una terminología adecuada al público que tiene como objetivo.

**¿Qué comunicación es requerida con otras personas para hacer las tareas?**

Un software a gran escala tendrá muchos roles, y lo más posible es que necesiten comunicarse entre sí para sincronizar su trabajo conjunto. Por lo que da juego a los desarrolladores incluir algún sistema de notificaciones o chat en la propia aplicación para que los usuarios no tengan que hacer uso de una aplicación de terceros.

**¿Cómo se relacionan las diferentes tareas?**

En la fase de diseño del software se elaboran tanto el modelado de los datos, como el de la base de datos que pueda tener, y un diagrama de clases. Este último se caracteriza por ver como se relacionan los objetos que van a ser implementados en el lenguaje de programación elegido. Da a los desarrolladores una visión de cómo se relacionan los objetos y las operaciones que pueden ejecutar entre sí.

Al diseñarlo se va modificando hasta que todo quede perfecto y coherente, generando una mayor facilidad de comprensión tanto para los programadores como para el usuario final.

**2. Reglas Empíricas de Diseño de Interfaces**

**Anticipación**

“Las aplicaciones deberían intentar anticiparse a las necesidades y deseos del usuario.”

Si los usuarios tienen que buscar información adicional fuera de la aplicación va a perder productividad en el trabajo. Además de desarrollar cierta frustración por tener que “buscarse la vida”.

**Autonomía**

“El ordenador, la interfaz y el entorno de la tarea pertenecen al usuario, pero esto no significa que abandonemos todas las reglas.”

El usuario es un ser inteligente, por lo que el software debería tener una curva de aprendizaje asequible. No limitar las opciones, pero tampoco bombardearlo con ellas. Se resume a una interfaz clara con cierta profundidad.

“Mantén informado al usuario del estado del sistema.”

Tanto en la vida real como en la virtual, a nadie le gusta realizar un trámite y no saber su progreso o su estado. Siempre se debe informar del progreso de una tarea con una descripción del proceso que se lleva a cabo.

“Mantén la información de estado fácilmente visible y actualizada.”

La información proporcionada al usuario sobre el estado tiene que ser claramente visible y actualizarse cada cierto tiempo para evitar búsquedas innecesarias.

**Daltonismo**

“Si utilizas el color para transmitir información debes utilizar otros elementos complementarios para la gente con daltonismo.”

Además de no excluir a gente con este problema integrándolos más en la sociedad, se abre en menor medida el mercado del software.

**Consistencia**

“Niveles de consistencia”

Darles prioridad a unos aspectos de la interfaz frente a otros tiene un fuerte impacto en las impresiones del usuario final. Hay que

“Inconsistencia”

“La consistencia más importante es aquella que espera el usuario”

Como cabe esperar, la mejor consistencia es la que se espera la mayoría de usuarios. Por lo que conviene ponerse en el papel del cliente para

**Valores por defecto**

“Los valores por defecto deberían ser poder descartados con facilidad y rapidez.”

La única acción de teclear para reemplazar un valor por defecto le ahorra tiempo al usuario, a la misma vez que se evita crear un sentimiento de pesadez o de hacer un trabajo tedioso.

**Eficacia del usuario**

“Busca la productividad del usuario, no del ordenador”

Todas las personas saben que el tiempo es oro, por eso lo valoran tanto. Se desea acabar todas las labores de una manera rápida y sencilla para centrarse en aspectos personales. Por lo tanto, un software que está diseñado para maximizar la productividad del usuario

“Mantén ocupado al usuario”

Como se ha explicado anteriormente el tiempo es valioso, y tener a un usuario esperando una respuesta del sistema no es una buena práctica. Como se explicará más adelante el uso de multi-hilos permitirá al usuario realizar otras tareas mientras la aplicación trabaja en paralelo.

“Para maximizar la eficacia de un negocio u organización debes maximizar la eficacia de todos y no sólo de un grupo”

En un software que presente múltiples roles, no hay que centrarse en uno en especial, sino que deben pulirse todo para que den el máximo de eficiencia al trabajador.

“Los saltos cualitativos en eficacia se encuentran en la arquitectura del sistema, no en su superficie, en el diseño visual de la interfaz.”

En el diseño es bien sabido que la eficacia del programa viene dada por la forma de su implementación, y la productividad dependerá de la interfaz de usuario más otras variables. Mantener un equilibrio entre estos 2 conceptos es fundamental.

“Escribe mensajes de ayuda concisos y que ayuden a resolver el problema: un buen texto ayuda mucho en comprensión y eficacia.”

Un usuario que este perdido en una tarea está perdiendo productividad, por lo que es de gran ayuda mostrarle una pequeña guía con el tipo de problemas más común que puedan experimentar. Ahorrando una lectura al manual buscando la sección correspondiente.

“Menús y etiquetas de botones deben comenzar con la palabra más importante.”

Con solo leer la primera palabra de una opción las personas deberían de hacerse una idea de lo que podría realizar esa función. Deben ser claras, precisas y contundentes para intentar que el usuario pueda entender las acciones con un vistazo.

**Interfaces explorables**

“A veces es necesario ofrecer caminos bien profundos y marcados.”

Una interfaz consistente, legible, segura y familiar con el usuario da a pie a una navegación fluida. Por lo que sólo se tiene en mente terminar el trabajo sin distracciones.

“Da a los usuarios nociones estables para saber cómo llegar al inicio”

Mantener un icono de vuelta a la pantalla principal del programa siempre es de agradecer. Se evita confusiones y posicionar opciones en otros menús para navegar por la interfaz.

Últimamente es un convenio en las páginas webs utilizar el logo de la empresa para volver al “inicio”.

“Haz que las acciones sean reversibles”

Para aprender a controlar bien un software hace falta práctica, y para eso hay que probar que efecto tienen todas las funciones del mismo. Así que para fomentar el aprendizaje sin repercusiones irreversibles se debe implementar una acción de deshacer.

“Siempre permite el "deshacer"”

Respecto al tema anterior la opción de deshacer la función ejecutada previamente ha de permitirse en todo momento. De esta manera se evita bombardear al usuario con preguntas de confirmación que ralentizan el trabajo.

“Siempre deja una salida abierta”

Salir en cualquier momento de la aplicación debe ser un derecho. No hay que bloquearla bajo ningún concepto, ya que puede bloquearse o aun peor, que el usuario necesite salir en un momento determinado y no pueda.

“De todas formas, haz que sea fácil quedarse.”

Controlar de forma clara el flujo de trabajo hace que los usuarios sepan dónde están y a dónde quieren llegar. Provocando que se centren en terminar una tarea.

El software tiene que dar razones para que la comunidad tenga el menor número de quejas e inconvenientes. Por lo que no buscarían otras aplicaciones similares.

“Ley de Fitt”

Que la interfaz tenga sentido común, es decir, iconos más visibles para tareas más importantes y este tipo de prácticas, ayudan al usuario a la comprensión de la misma. Así no se genera un sentimiento de incomprensión total o de desubicación.

**Objetos humanos**

“Los objetos humanos de la interfaz se pueden ver, escuchar, tocar o percibir de otra manera.

Los objetos humanos visibles de la interfaz son bastante familiares. Aquellos que utilizan otros sentidos lo son menos.”

Usar iconos que representen objetos de la vida real (líneas, círculos, informes, papeleras) tiene un impacto directo en la comprensión de las tareas que ejecutan. Le facilitan al usuario identificar las funciones que tenga el software.

**Reducción de latencia**

“Cuando sea posible, utiliza el multihilo para dejar la latencia en un segundo plano.”

Sacar el máximo provecho a la computadora que se esté usando para evitar que el usuario se percate que el sistema está ocioso.

“Reduzca la percepción de latencia”

Indicar en todo momento al usuario cuando y cuando no puede seguir trabajando, o permitirle trabajar en otra tarea mientras se completa la anterior. El principal objetivo es que no baje ni la eficacia del programa ni la productividad del usuario impidiéndoselo con esperas.

**Aprendizaje**

“Limita las limitaciones”

Diseñando una buena interfaz, se puede lograr que la usabilidad y la facilidad de uso coexistan con armonía en el software. Dando al cliente una sensación de satisfacción. Una interfaz minimalista con funciones complejas es una buena combinación.

**Uso de metáforas**

“Dale vida a las metáforas apoyándote en su percepción -vista, sonido, tacto, kinestesia- y en sus recuerdos.”

Usar los recuerdos del usuario sobre un hito anterior puede crear unos sentimientos favorables. Lo que probablemente se traduzca a un mayor apego al software que se esté usando.

Incluso logra que los usuarios consigan un conocimiento instantáneo sobre cómo usar correctamente la interfaz. En resumen, hacen que la experiencia familiar ayude a comprender nuevos conceptos/prácticas.

**Protege el trabajo del usuario**

“Asegúrate de que el usuario nunca pierde su trabajo como resultado de un error suyo, los problemas de internet u otro tipo de problemas inevitables, como un apagón.”

Mantener guardados constantes cada cierto tiempo o backups de los mismos evita que cualquier fallo que ocurra, no pierda el progreso del fichero. Un software que procura que la información de sus usuarios no se pierda del todo tras un error gana mucha fiabilidad.

**Legibilidad**

“Utiliza texto con alto contraste.”

Hacer que la interfaz no canse la vista utilizando la fuente de letra correcta u cierto color sobre otro. Proporciona al usuario la posibilidad de estar más tiempo trabajando sin dolencias en la vista y consecuentemente, sin incomodidades físicas.

**Guardar el estado**

“La información de estado debe almacenarse en una cookie durante la sesión en el ordenador cliente. Luego se almacena en un servidor.”

Tener cierta información del usuario permite intuir lo que va a buscar o hacer. Sugiriéndole información similar a la que se ha recopilado le puede ahorrar tiempo o descubrir otras herramientas relevantes. Además de que en plena era de la información es imprescindible guardar el estado en el que ha dejado una aplicación.

**Navegación visible**

“Evita la navegación invisible”

Memorizar todas las ventanas que puede tener una interfaz no es un proceso rápido, llevadero y mucho menos agradable. Cuantos más elementos se necesite memorizar hace tedioso el uso del software. Por lo que diseñar un mapa con el menor número de transiciones posibles hace que el usuario se acostumbre más rápido a la aplicación.