UT4- Interfaz de Usuario

Interface de Usuario

• Es el método que permite interactuar a los usuarios con máquinas, aplicaciones, dispositivos, etc.

Usabilidad

• **Efectividad** – Precisión y plenitud con que los usuarios alcanzan los objetivos deseados
• **Eficiencia** – En los recursos empleados para llegar a la efectividad
• **Satisfacción** – Actitud positiva en el uso del producto y ausencia de incomodidad

Heurísticas de Jackob Nielsen

• Las heurísticas de Nielsen establecen que **la mayor parte del tiempo las personas usan productos digitales distintos al tuyo, por lo tanto,sus expectativas están basadas en dichos productos**. Los usuarios no deben preguntarse si diferentes situaciones o acciones significan lo mismo.

1 – Visibilidad del Status del Sistema

– Hacer **que el usuario sepa que está pasando**
• **Mensajes** “El formulario fue enviado correctamente”
• **Opción seleccionada resaltada**
• **Barra de Progreso**
• **Cambio del Cursor**
• **Animación**

2 – Alineación entre el Sistema y el mundo Real

– ¡Cuanto más claro mejor!
– **El sistema tiene que hablar con el usuario en su mismo lenguaje**
• Imágenes claras
• **Iconos representativos y claros** (papelera para eliminar)
• Mensajes que el usuario pueda entender
• Límite de caracteres identificado
• Seguir la convención del sistema

3 – Control y Libertad para el Usuario

– **No forzar al Usuario a seguir un camino determinado**
– Darle al usuario la posibilidad de **corregir errores**
– Evitar los callejones sin salida
– **Opciones de deshacer o volver atrás**

4 – Consistencia y Estándares

– **Los usuarios no deberían cuestionarse las acciones**
• El mismo menú muestra opciones diferentes en distintas páginas, pero las opciones hacen los mismo (carro / cesta)
– **Cosas distintas deben ser visiblemente distintas**
– Usar diferentes diseños para la misma cosa
– **Color de botones**
– Posición del menú

– Tres tipos de Consistencia

• **Interna** – Consistencia entre páginas o pantallas de un mismo sistema
• **Externa** – Consistencia entre aplicaciones de una misma plataforma (Instalación de aplicaciones en Windows, Mac OS)
• **Metafórica** – Es la forma de hacer intuitiva una interface **basándose en lo que el usuario ya conoce** (ej. Calculadora, consolas de música, WhatsApp)

5– Prevención de Errores

– **Prevenir es mejor que curar**
– Se debe realizar un diseño cuidadoso que **prevenga la ocurrencia de errores**
• **Resaltar un campo que falta llenar**
• Pedir doble confirmación de clave o email
• Realizar comprobaciones en tiempo real
• Opción de autocompletar
• Mascaras al pedir información

6 – Reconocimiento en lugar de recuerdo

– Siempre es **mejor reconocer antes que obligar al usuario a memorizar** acciones u objetos
– El usuario siempre **debe tener la información a mano**
• ¿Dónde está el producto que ya vi y me gustó?
• ¿Cómo vuelvo para atrás?
• ¿Qué objetos, acciones y opciones elegí?
– **Cuanto más un usuario tiene que recordar, más propenso a errores será** la interacción con el sistema
• Usar ComboBox, no TextBox

7 – Flexibilidad y Eficiencia de Uso

– Debemos conseguir que nuestros **sistemas estén preparados para usuarios nuevos y experimentados**
• Si podemos hacer que los **nuevos usuarios naveguen** en nuestra web, logramos **flexibilidad**
• Si **tenemos opciones** para los más **experimentados**, logramos **eficacia**
• Ej. El buscador de Google.
– Es intuitivo para los nuevos usuarios, solo escriben en la barra de texto
– Los usuarios experimentados tienen opciones de búsqueda más específicas

8 – Estética y Diseño Minimalista

– Las páginas web **no deben contener información innecesaria**, si no hace falta, no lo pongas.
– Cada palabra de más está oscureciendo las palabras que son realmente importantes

9 – Ayuda a los Usuarios con los Errores

– **Los errores deben de ser entendibles por el usuario**
• Error 404
• Pueden **sugerir una solución** o un camino alternativo

10 – Ayuda y Documentación

– Con estos principios se pretende que los usuarios no deben utilizar documentación
– Igualmente, se le debe **brindar al usuario un manual de funcionamiento**: FAQs y/o Mini tours

Rating de Severidad

• **Frecuencia**: ¿Es común o raro? – Nro. De usuarios que encuentra el problema dividido el número total de usuarios
• **Impacto**: ¿Es **fácil o difícil superar** este problema?
• **Persistencia**: ¿El problema **es conocido y el usuario puede solucionarlo o no**?

0= No es un problema de usabilidad
1= problemas **cosméticos**: no necesita ser arreglado a no ser que se cuente con tiempo extra
2= problema **menor de usabilidad**: baja prioridad
3= problema **mayor de usabilidad**: importante de arreglar
4= **catástrofe de usabilidad**: imperativo de arreglar

16 Principios de Tognazzini

1 – Anticipación

Las aplicaciones deberían intentar **anticiparse a las necesidades del usuario** y no esperar a que el usuario tenga que buscar la información, recopilarla o invocar las herramientas que va a utilizar.

2 – Autonomía y Control

El usuario debe tener el control y poder moverse con autonomía por el sitio web.

3 – Precaución usando colores

El uso del color no debe ser la única forma de presentar la información, se deben usar otros elementos complementarios, pensando en usuarios que no distinguen los colores.

4 – Consistencia

Hay que ser **consistente con los conocimientos previos y las expectativas del usuario**. Uso de valores por defecto cuando tenga sentido, permitiendo cambiar su configuración con facilidad.

5 – Uso de valores por defecto

Cuando tenga sentido, **permitiendo cambiar su configuración con facilidad**. <> en inputs de texto.

6 – Eficacia del usuario

Hay que **centrarse en la productividad del usuario**. Generalmente al navegar escaneamos más que leemos y tenemos que **promover una rápida comprensión de qué está pasando y de dónde hacer clic**.

7 – Interfaces explorables que den libertad al usuario.

Se debe **permitir que el usuario deshaga las acciones realizadas**.

8 – Ley de Fitts

Cuanta menos distancia haya que recorrer y mayor tamaño tenga un elemento, más fácil será interactuar con él. Mide el **tiempo (estimado)** necesario para moverse rápidamente desde una posición inicial hasta una zona destino final (en base a la distancia y tamaño del objetivo final).

Lo que se conoce como la ley de Fitts es un concepto intuitivo simple.
– **Cuanto más lejos está el objetivo, más tiempo lleva alcanzarlo con el mouse**.
– **El objetivo más chico, más tiempo lleva alcanzarlo con el mouse**.

9 – Uso de estándares

Elementos familiares en la interfaz.

10 – Reducción de demoras

Se debe **minimizar el tiempo de espera del usuario** y mantenerlo informado del tiempo que falta.

11 - Minimizar el aprendizaje

El aprendizaje necesario debe ser mínimo y el sitio web debe poder usarse desde el primer momento.

12 - Uso adecuado de metáforas

Con su uso, siempre que sean apropiadas, **se mejora la comprensión**. "Papelera" para eliminar, "Disquete" para guardar

13 - Protección del trabajo de los usuarios.

Hay que **asegurar que el trabajo de los usuarios no se pierda a consecuencia de un error** (del sistema o del usuario u otros problemas).

14 – Legibilidad

Hay que **favorecer la legibilidad mediante el tamaño de fuente adecuado y suficiente contraste** entre texto y fondo.

15 - Seguimiento de las acciones de usuario (registro de estado)

Hay que **guardar información sobre los usuarios** para posteriormente permitir que las acciones que realiza con más frecuencia se puedan realizar más rápido.

16 - Navegación visible

Hay que **evitar, o reducir al máximo, los elementos de navegación invisibles** y presentarlos de forma clara.

Patrones de Diseño

Patrón

• **Una solución probada y reutilizable a un problema común dentro de un contexto específico en el diseño de software**.

Los patrones de software **NO son fragmentos de código que pueden ser copiados y pegados en un programa**, sino más bien guías generales que describen cómo abordar ciertos problemas y situaciones.

• Los patrones de software pueden ayudar a:
1. **Resolver problemas comunes**: Al utilizar soluciones que han sido probadas en múltiples proyectos, los desarrolladores pueden evitar reinventar la rueda.

2. **Mejorar la comunicación entre desarrolladores**: Los patrones de software tienen nombres estándar. Cuando los desarrolladores usan estos nombres en sus discusiones, otros desarrolladores familiarizados con esos patrones entienden rápidamente la solución propuesta.
3. **Hacer que el código sea más mantenible y flexible**: Los patrones a menudo enfatizan la creación de código que es fácil de modificar y extender, lo que es beneficioso a largo plazo.

1. Patrones de Creación: Se centran en la creación de objetos o clases.
2. Patrones Estructurales: Tratan de cómo se componen los objetos para formar estructuras más grandes.
3. Patrones de Comportamiento: Se centran en la interacción y responsabilidades entre objetos y clases.
4. Patrones de Arquitectura: Estos patrones están a un nivel más alto que los patrones de diseño y tratan con la arquitectura global de una aplicación. Un ejemplo común es el patrón **MVC (Modelo-Vista-Controlador)**, que separa la lógica de negocio, la interfaz de usuario y la entrada del usuario en componentes independientes.
5. Patrones de Managment: Prácticas, estrategias y técnicas que ayudan a gestionar de manera efectiva un equipo, un proyecto o una organización. Estos patrones se pueden aplicar en diversos ámbitos, incluido el desarrollo de software.

– Es importante notar que aunque los patrones de software ofrecen muchas ventajas, **no siempre son la solución adecuada para cada problema**, y su uso inadecuado puede resultar en complicaciones innecesarias. Por lo tanto, **es esencial entender bien los patrones y aplicarlos de manera juiciosa**.

Principios SOLID

SOLID es un **conjunto de principios fundamentales en ingeniería de software**. El acrónimo representa los cinco principios básicos de la programación orientada a objetos y diseño, que incluyen:
• **SRP** Single Responsibility (Responsabilidad Única),
• **OCP** Open-Closed (Abierto-Cerrado),
• **LSP** Liskov Substitution (Sustitución de Liskov),
• **ISP** Interface Segregation (Segregación de Interfaces) y
• **DIP** Dependency Inversion (Inversión de Dependencias).

Cuando se aplican en conjunto, **estos principios ayudan a los desarrolladores a crear sistemas que sean más fáciles de mantener y ampliar con el tiempo**. SOLID proporciona directrices para evitar malos diseños en el desarrollo de software, permitiendo que el código fuente sea refactorizado hasta que sea legible y extensible.

Single Responsibility Principl (SRP)

• Establece que **una clase debe tener solo una razón para cambiar**, ya que cada responsabilidad es un posible cambio.
• Cuando una clase tiene más de una responsabilidad, **las responsabilidades se acoplan, lo que puede conducir a diseños frágiles** que pueden romperse de formas inesperadas cuando cambian.
• La solución es **dividir las responsabilidades en clases distintas** para evitar problemas.
• En el contexto del SRP, una responsabilidad se define como "una razón para cambiar".

• En conclusión, el SRP es uno de los principios más sencillos y difíciles de aplicar correctamente. La esencia del diseño de software se basa en **identificar y separar las responsabilidades**.

Open-Closed Principle

• Establece que **las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación**.
• OCP puede abordar problemas comunes asociados con los "malos" diseños de software,
– fragilidad
– rigidez
– imprevisibilidad
– falta de reutilización.

• En otras palabras, un diseño que sigue el OCP puede soportar cambios en los requisitos a través de la extensión de módulos (agregando nuevo código), en lugar de cambiar el código existente que ya funciona.

• Para que un módulo cumpla con el OCP, debe tener dos atributos principales:
1. **"Abierto para extensión"**: esto significa que el comportamiento del módulo puede extenderse para adaptarse a los cambios en los requisitos de la aplicación o para satisfacer las necesidades de nuevas aplicaciones.
2. **"Cerrado para modificación"**: esto significa que el código fuente de un módulo es inviolable y nadie puede hacer cambios en él.

• **Las variables globales/públicas, violan el principio de abierto-cerrado**. Ningún módulo que dependa de una variable global puede estar cerrado contra cualquier otro módulo que pueda escribir en esa variable.
• **La identificación de tipos en tiempo de ejecución (RTTI) es peligrosa**, ya que a menudo puede violar el principio de abierto-cerrado.
• La diferencia entre un mal y un buen uso de RTTI radica en si el código necesita ser cambiado cada vez que se deriva un nuevo tipo de objeto.

Liskov Substitution Principle

• “Subclasses should be substitutable for their base classes.”
• Los **principales mecanismos** detrás del principio de abierto-cerrado son la **abstracción y el polimorfismo**. En lenguajes como C# uno de los mecanismos clave que soporta la abstracción y el polimorfismo es la herencia.
• Una clase B es una subclase de la clase A, entonces debe ser posible usar B donde sea que A se espere sin cambiar el comportamiento del programa.

Interface Segregation Principle

• **Los clientes no deben ser forzados a depender de tipos que no usan**. En otras palabras, es mejor tener muchas interfaces específicas que una sola interface general.
• Este principio tiene como objetivo reducir los problemas que pueden surgir debido a los cambios en las clases que dependen de las interfaces de gran tamaño. **Los clientes de una interface que contiene métodos que no necesitan aún se ven afectados cuando la interface cambia**.
• Supongamos que tienes una interface IAve que tiene métodos para volar, nadar y comer. Ahora bien, no todas las aves pueden volar o nadar, por lo que si tuvieras una clase Pinguino que implementa la interface IAve, tendrías que implementar un método de volar que realmente no tiene sentido para un pingüino.
• Según el ISP, sería mejor tener tres interfaces separadas: IAveVoladora, IAveNadadora e IAveComedora. De esta forma, Pinguino podría implementar solo IAveNadadora e IAveComedora, y no estaría forzado a implementar IAveVoladora. Esto hace que el diseño sea más flexible y coherent

Dependency Inversion Principle

• **Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.**
• **Las abstracciones no deben depender de los detalles Los detalles deben depender de las abstracciones.**
• El propósito de este principio es ayudar a desacoplar el software. La idea es que cuando los módulos de alto nivel están fuertemente acoplados a los módulos de bajo nivel, se dificulta la reutilización y las modificaciones en los módulos de bajo nivel.

Antipatrones

Antipatrón
• Es un término utilizado en ingeniería de software que hace referencia a **una solución comúnmente utilizada para resolver un problema en el desarrollo de software, pero que produce resultados contraproducentes o negativos**. Los antipatrones son a menudo el resultado de la experiencia limitada, la falta de conocimiento de mejores prácticas, plazos apresurados o soluciones a corto plazo sin considerar las consecuencias a largo plazo.
• Es importante destacar que lo que hace que una práctica sea un antipatrón no es simplemente que sea una mala solución, sino que es una solución que **parece ser beneficiosa en la superficie o a corto plazo, pero que conduce a problemas en el futuro**. Son errores que se cometen comúnmente en la industria.
• Características:
1. **Ineficiencia**: Pueden hacer que el software sea más lento o consuma más recursos de los necesarios.
2. **Dificultad de mantenimiento**: Hacen que el código sea más difícil de entender, modificar o extender.
3. **Introducción de defectos**: Pueden introducir errores o hacer que sea más probable que ocurran problemas.
4. **Complejidad innecesaria**: Añaden complejidad al diseño o al código sin un beneficio claro

The Blob

• Definición
• **Una sola clase se encarga de la mayoría de las responsabilidades** mientras que otras clases son simples contenedores de datos.
• Es un diseño similar a la programación procedural en lugar de ser verdaderamente orientado a objetos.

• Cómo Reconocer The Blob
• Una clase con **un número extremadamente alto de atributos y métodos**.
• Falta de cohesión en los atributos y operaciones de la clase.
• **Las demás clases sirven principalmente para almacenar datos** y tienen poca o ninguna lógica.

• Consecuencias de Usar The Blob
• **Dificulta la mantenibilidad y comprensión** del código.
• **Reduce la reusabilidad** de las clases.
• Puede **afectar el rendimiento** debido a la carga excesiva en una sola clase.

• Cómo Evitar o Resolver The Blob
• **Refactorizar**: Dividir la clase gigante en clases más pequeñas y especializadas.
• **Redistribuir responsabilidades entre clases** basándose en principios de diseño orientado a objetos, como el Principio de Responsabilidad Única.
• Asegurar que cada clase tenga un propósito claro y bien definido.

Lava flow

• Definición: Refiere a **piezas de código que quedan en el sistema, aunque ya no son útiles, y se vuelven difíciles de eliminar**.
• Características:
– **Código que se mantiene a pesar de que ya no es necesario**
– Código que se ha vuelto **obsoleto** debido a cambios en el diseño o requisitos
– Código que **se evita modificar por miedo a romper funcionalidades** existentes

• Cómo se forma:
– Cambios rápidos en los requerimientos del proyecto
– Falta de entendimiento o documentación del código existente
– Temor a modificar código antiguo por posibles consecuencias en el funcionamiento del sistema

• Consecuencias:
• **Aumento de la complejidad** del código
• **Reducción de la mantenibilidad**
• **Inflación del tamaño del código**, haciendo que el software sea más difícil de entender y modificar

• Cómo prevenir y solucionar Lava Flow
• **Documentar bien el código y los cambios realizados**
• **Implementar revisiones** de código periódicas
• **Refactorizar** el código obsoleto
• **Eliminar código innecesario** después de pruebas rigurosas y revisiones de código

Golden Hammer

• Definición: **cuando un equipo de desarrollo o individuo depende demasiado de una herramienta o tecnología familiar**, a expensas de posiblemente mejores alternativas.

• Características:
• **Sobreutilización de una herramienta** o tecnología en particular.
• Reluctancia para considerar o adoptar alternativas.
• Creencia de que la herramienta familiar puede resolver todos los problemas.

• Cómo se forma:
• **Falta de conocimiento de otras herramientas** y tecnologías.
• **Comodidad y familiaridad** con una herramienta específica.
• **Resistencia al cambio** o a aprender nuevas habilidades.

• Consecuencias:
– **Soluciones ineficientes o subóptimas.**
– **Mayor costo y tiempo de desarrollo.**
– **Limitación en la innovación y adaptabilidad del proyecto**.

• Cómo prevenir y solucionar Golden Hammer
– **Educar al equipo** sobre diferentes herramientas y tecnologías.
– **Realizar evaluaciones objetivas** antes de seleccionar herramientas o tecnologías.
– Fomentar un **entorno abierto a la experimentación** y aprendizaje continuo.

Spaghetti Code
• Definición: el código fuente de un programa tiene una **estructura compleja y enredada, parecida a un plato de espaguetis, lo que lo hace difícil de leer, mantener y depurar**.

• Características:
– **Falta de estructura y organización.**
– Uso excesivo de saltos incondicionales (GOTOs).
– **Difícil de seguir el flujo de control**.
– Ausencia de modularidad y encapsulación.

• Cómo se forma:
– **Desarrollo apresurado** sin una arquitectura adecuada.
– **Falta de conocimiento** de buenas prácticas de programación.
– Modificaciones y parches sucesivos sin reconsiderar la estructura general.

• Consecuencias:
• **Dificultad en mantenimiento y depuración**.
• **Incremento en el riesgo de introducir errores**.
• Mayor costo y tiempo de desarrollo.

• Recomendaciones para evitar el Spaghetti Code:
• **Planificar y diseñar la arquitectura antes de escribir** el código.
• **Seguir buenas prácticas de programación**.
• **Realizar revisiones de código**.

• Recomendaciones para refactorizar el Spaghetti Code:
• **Dividir el código en funciones y clases más pequeñas y coherentes**.
• **Reemplazar saltos incondicionales con estructuras de control más legibles**.
• **Mejorar la documentación y comentarios**

Cut−and−Paste P
• Definición: se refiere a la **práctica de copiar y pegar bloques de código dentro de una aplicación**, en lugar de crear funciones o módulos reutilizables.

• Características:
• **Duplicación de código.**
• **Falta de modularidad y abstracción.**
• **Dificultad en la implementación de cambios en lógicas duplicadas**.

• Cómo se forma:
• **Desarrollo apresurado o bajo presión de tiempo.**
• **Falta de conocimiento de buenas prácticas** de programación.
• **Tratar de evitar "reinventar la rueda" sin tener una adecuada reutilización d**e código.

• Consecuencias
– **Dificultad en el mantenimiento y la corrección** de errores.
– **Incremento en el riesgo de inconsistencias y errores lógicos**.
– Código **menos legible y comprensible**.

• Recomendaciones para evitar Cut-and-Paste Programming:
• **Pensar en modularidad y reutilización de códig**o desde el inicio.
• Crear funciones y clases para lógicas comunes.

• Recomendaciones para solucionar Cut-and-Paste Programming:
• **Identificar y eliminar la duplicación..**
• **Refactor**

Tester Driven Development

• O "Bug Driven Development", se refiere a un antipatrón en el que **las pruebas de software o los informes de errores dirigen el desarrollo de software, en lugar de las necesidades del usuario o los requerimientos de las funcionalidades**. Este enfoque puede dar lugar a una baja calidad del código y retrasos en la entrega del software.

• Esto puede suceder si:
– Las **pruebas comienzan demasiado pronto**
– Los **requerimientos no están completos**
– Los testers o desarrolladores son inexpertos
– La gestión del proyecto es deficiente.

• En estos casos, los **testers pueden terminar dictando cómo debería ser el software**, lo que puede desviarse de lo que los usuarios realmente necesitan o quieren.

Para evitar caer en este antipatrón, es importante asegurarse de que:
1. **Las pruebas de software comienzan en el momento adecuado**: no demasiado pronto que no hay suficiente para probar, y no demasiado tarde que los errores no son detectados hasta que es muy costoso arreglarlos.
2. **Los requerimientos están completos y bien definidos**: los desarrolladores y testers deben tener una clara comprensión de lo que se supone que debe hacer el software.
3. **Tanto los testers como los desarrolladores están adecuadamente capacitados**: deben entender no sólo cómo realizar sus tareas, sino también cómo encajan en el proceso general de desarrollo de software.
4. **La gestión del proyecto es eficaz**: esto incluye planificar adecuadamente, comunicarse de manera eficaz, y asegurarse de que todos los miembros del equipo entienden sus roles y responsabilidades. Mantener un enfoque en el valor del usuario y los requerimientos de las funcionalidades. **Las pruebas de software son una herramienta importante para asegurar la calidad del software, pero no deben ser la única fuerza impulsora detrás del desarrollo del software**.