UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA

MAESTRÍA Y DOCTORADO EN CIENCIAS E INGENIERÍA

"MODELO PARA LA CONSERVACIÓN DE LOS ATRIBUTOS DE CALIDAD DE MANTENIBILIDAD Y FLEXIBILIDAD DE LA ARQUITECTURA DEL SOFTWARE CON LA INTEGRACIÓN DEL MODELO DE USUARIO"

TESIS

QUE PARA OBTENER EL GRADO DE:

MAESTRÍA EN CIENCIAS

PRESENTA:

ANDRÉS MEJÍA FIGUEROA

TIJUANA, BAJA CALIFORNIA ENERO 2013

Dedicatoria

A mi amorcito y mi familia por haberme apoyado ante todo y por aguantar mis largas ausencias…

Agradecimientos

A mis padres por ayudado y apoyado en cumplir mis objetivos, por haberse sacrificado para mi bienestar para que tuviera un buen futuro.

A mi querida novia, Nancy, por tener paciencia y comprensión para soportar tantas ausencias y descuidos durante mis estudios de posgrado. Por haber estado conmigo en las altas y bajas, y apoyarme en seguir adelante con mis objetivos.

A mis grupos de estudiantes de Ing. De Requerimientos, Diseño de Interacciones y Programación Orientada a Objetos Avanzada, por usarlos como conejillos de indias y por su apoyo en diversos proyectos e investigaciones.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT), por el apoyo económico brindado durante la duración de los 2 años de mis estudios de maestría.

A la Universidad Autónoma de Baja California, Facultad de Ciencias Químicas e Ingeniería y la Coordinación de Ingeniería en Computación, por todo el apoyo que me brindaron para realizarlas investigaciones y participaciones en Congresos.

A mi profesor de tesis, Dr. Reyes Juárez, por su apoyo y ayuda en todas mis actividades de investigación y de la maestría. Extrañaré sus mensajes a las 3 am para que me presente temprano el siguiente día…

INDICE

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE FÓRMULAS

RESUMEN

Enfoques de desarrollo de software centrados en el usuario hacen especial hincapié en las características del usuario y las tareas que debe de realizar, ya que afecta en gran medida la

forma en que el usuario interactúa con el sistema. La mayoría de trabajos de investigación proponen algunas características de los usuarios, sin embargo, no existe un modelo que integra aspectos cognitivos, tales como psicológico, y físico.

Este trabajo propone un modelo de usuario integrar todas las características del usuario, con el propósito de lograr una interfaz de software adaptiva para mejorar la facilidad de uso general de cualquier sistema y una manera de integrar en la arquitectura de software, que afecta de un modo mínimo, la capacidad de mantenimiento de la arquitectura. Se tomó en consideración los factores de la arquitectura que tienen un efecto considerable sobre los atributos de calidad del software, ya que al integrar el modelo del usuario se añade complejidad a la arquitectura afectando de alguna forma a algunos atributos de calidad, como son la cohesión y el acoplamiento.

Se ha integrado este modelo de usuario mediante la recolección de información de diferentes disciplinas como la medicina, la sociología y sus practicantes. Con el fin de validar nuestro modelo, hemos implementado las interfaces de usuario para diferentes proyectos reales, teniendo en cuenta las características de los usuarios reales y su impacto en la arquitectura del software mediante un análisis por medio de métricas de cohesión y acoplamiento. También presentamos aquí un ejemplo de cómo una interfaz de usuario se puede adaptar para un conjunto específico de capacidades de los usuarios.

De igual forma, se presenta un intento de cambiar la perspectiva de los alumnos en cuanto a la importancia de la usabilidad desde las fases iniciales de diseño por medio de proyectos de carácter social, interactuando con el usuario, en este caso niños con autismo, de tal forma que vean el impacto sobre el usuario sobre las decisiones que diseño y de arquitectura que toman los desarrolladores.

ABSTRACT

Current user-centered software development approaches make special emphasis on the characteristics of the user and the tasks that must be accomplished because it affects greatly the way the user interacts with the system. Most research works propose some user characteristics; however, there is not a model that integrates aspects such as psychological, cognitive, and physical.

This work proposes a user model integrating all the user's characteristics, with the purpose of achieving an adaptive software interface improving the overall usability of any system and a way to integrate it into the software architecture, affecting in a minimal way, the maintainability of the architecture. The factors of architecture that have a significant effect on the quality attributes of the software were taken into account, and that by integrating the user model complexity is added to the architecture somehow affecting some attributes of the architecture such as cohesion and coupling levels.

We have integrated this model gathering information from different disciplines such as medicine, sociology and their practitioners. In order to validate our model, we have implemented user interfaces for various real projects taking into account characteristics from real users and measuring the impact of the integration of the user model to the architecture by doing an analysis with coupling and cohesion metrics. Also we present here an example of how a user interface can be adapted for a specific set of user capabilities.

We also attempt to change the perspective of the students about the importance of usability from the early stages of design through social projects, interacting with the user, in this case children with autism, so they see the user impact on design decisions taken by the developers.

I

INTRODUCCIÓN

I. INTRODUCCIÓN

I.1. La Usabilidad en el Software

El software, como todo producto, debe de tratar de cumplir con ciertos atributos o criterios de calidad los cuales deben ser medibles para determinar si se llegaron a satisfacer. Los atributos de calidad de software son definidos por Bardacci, *et al* (1995) como los parámetros que describen el comportamiento previsto del sistema dentro de un contexto y proporcionan los medios para medir la aptitud e idoneidad de un producto. Existen varios modelos para los atributos de calidad del producto siendo los principales los de McCall (1977) y Boehm (1978). Ejemplos de algunos atributos son: mantenibilidad, escalabilidad, flexibilidad, rendimiento, usabilidad, etc. Ferre *et al* (2011) denotan que la usabilidad es uno de los más importantes debido a que tiene que ver con la experiencia que tiene el usuario con el manejo del software, pero desgraciadamente es uno de los que se le da menos prioridad.

Tradicionalmente hay una total desconexión entre la usabilidad y la funcionalidad en el desarrollo de sistemas de software (Ferre *et al*, 2006; Folmer *et al*, 2003), al no pensar en el usuario desde las fases del diseño, enfocándose completamente en los requerimientos funcionales del sistema sin considerar los posibles impactos que pueda tener los requerimientos de usabilidad en el ciclo de vida del desarrollo del software.

Con el fin de encontrar una solución para mejorar la usabilidad del software, se han propuesto varias metodologías centradas en el usuario donde se integran los pasos básicos de la ingeniería de la usabilidad en el ciclo de vida del software, el análisis del usuario y el análisis de tarea, como se indica en Zhang (2004).

I.2. El diseño de la Arquitectura del Software

La arquitectura es el esqueleto del sistema. Como una definición más formal se puede decir que, la arquitectura del software es el conjunto de componentes (clases, bases de datos, etc.), sus interrelaciones y el conjunto de reglas que rigen el software a desarrollar en general (Shaw *et al*, 1996; Bass *et al*, 1996).

La necesidad de diseñar una arquitectura de software nace por la creciente complejidad y tamaño del nuevo software a desarrollar para poder lidiar de mejor manera con los retos de diseño, cumplir con los requerimientos estipulados y mantener los atributos de calidad de un software cada vez más complejo.

La identificación de las cualidades deseadas del sistema antes de que esté siendo construido permite al diseñador modelar una solución para que coincida con las necesidades deseadas en el contexto de las restricciones, ya que con cada elemento que se le añada a la arquitectura, está crece complicando así el desarrollo del software. Cuando un diseñador comprende las características deseadas del sistema antes de ser diseñado, se podrá diseñar una arquitectura mucho más apta para el sistema a desarrollar.

I.3. El Problema del efecto de la usabilidad a la Arquitectura del Software

Actualmente, la usabilidad normalmente no es considerada en la arquitectura del software en las etapas tempranas del proceso de desarrollo del software (Ferre *et al*, 2003), posiblemente por la concepción errónea que solo tiene que ver con la parte “visual” del sistema, o sea la interfaz de usuario (Folmer *et al*, 2005). En realidad la usabilidad es parte del sistema y debe de ser considerado cuando se está diseñando el resto del mismo debido a los grandes impactos que puede tener sobre la arquitectura y los demás atributos del sistema (Ferre *et al*, 2004).

La interfaz de usuario es el medio de comunicación entre la computadora y el usuario (Stephanidis, 2001), comúnmente viene siendo una interfaz gráfica por donde el usuario interactúa e intercambia información con el sistema, aunque también puede ser por medio de otros medios, como por ejemplo, dispositivos receptores de audio donde se analizan los comandos de voz para ejecutar alguna acción en particular, o alguna combinación de software y hardware para usuarios con discapacidades.

Para el diseño de interfaces gráficas se utilizan diversos componentes gráficos para desplegar e ingresar información (como son las cajas de texto, botones, entre otros). En base a estos componentes se han propuestos diversos patrones de usabilidad. Un patrón de usabilidad es una solución a un problema recurrente de usabilidad para problemas comunes en el diseño de la interfaz (Bass *et al*, 2001). Algunos ejemplos son: el deshacer la última acción, el cancelar la acción, validación de datos, entre otros.

Los patrones de usabilidad son compuestos por varios componentes gráficos, pero cabe mencionar que no siempre son los más óptimos para usuarios con distintas características. Al diseñar un software, normalmente se toman en consideración algunas características en particular del usuario en el que se está basando el diseño del sistema, como la edad y los estudios/conocimientos que pueda tener, omitiendo otros como las características cognitivas, afectivas, e incluso físicas que pueden tener un gran impacto en la satisfacción final que tendrá el usuario del sistema (Zhang *et al*, 2004). Además la integración de algunos patrones de usabilidad pueden tener un impacto sobre el diseño de la arquitectura de software, haciendo que sea de gran importancia el tomar en cuenta estos desde las etapas de especificaciones de requerimientos.

La arquitectura de software tiene un profundo efecto en la mayoría de las cualidades de una forma u otra, y afecta a los atributos de calidad del software ya que encapsula las decisiones de negocio así como también los atributos de calidad que se espera del software a desarrollar (Shaw *et al*, 1996).

El desarrollo de software de alta calidad es difícil de lograr, sobre todo cuando la interpretación del término "calidad" es desigual en función del entorno en el que se utiliza. Con el fin de saber si la calidad se ha logrado, o degradado, tiene que ser medido, pero determinar qué medir y cómo es la parte difícil.

En este trabajo de tesis se propone un modelo integral del usuario con el propósito de mejorar la usabilidad para una amplia gama de usuarios con diferentes características, así como también una forma de integrarlo a la arquitectura del software de manera fácil conservando los atributos de mantenibilidad y flexibilidad de la arquitectura, promoviendo así la consideración de la usabilidad desde las fases iniciales del desarrollo tomando en cuenta las características de los usuarios.

I.4 Objetivos de la Tesis

Los objetivos planteados para esta tesis están relacionados con la sección anterior y la problemática descrita ahí sobre los efectos de la usabilidad y la arquitectura de software. En las siguientes secciones se describen de igual forma las metas relacionas con los objetivos presentados.

I.4.1 Objetivos principales

- O1: Definir un modelo de estructuración de la arquitectura que garantice la mantenibilidad y flexibilidad de la misma con la integración del modelo integral del usuario.
- O2: Asegurar la conservación de los atributos de calidad (mantenibilidad, flexibilidad) después de la incorporación del modelo del usuario a la arquitectura

I.4.2 Metas

- M1: Desarrollar un conjunto de conceptos (modelo del usuario, módulos o elementos de usabilidad) a considerar para el diseño de la arquitectura de la usabilidad.

- M2: Evangelizar a los programadores nuevos a implementar la usabilidad a la par que la funcionalidad en la arquitectura del software por medio de proyectos con impacto social donde interactúan con los usuarios, encuestas a los programadores involucrados y observaciones posteriores.

I.4.3. Hipótesis

- H1: Si el modelo de usuario se integra conforme al patrón arquitectónico de capas, es posible integrarlo a la arquitectura del sistema sin impactos negativos, ya que no altera las propiedades mantenibilidad y flexibilidad.

- H2: Los diseñadores novatos pueden modelar de mejor manera el perfil de usuario dentro de la arquitectura si interactúan directamente con los usuarios finales durante la fase de análisis.

II

Fundamentos Teóricos

II. FUNDAMENTOS TEÓRICOS

La arquitectura del software ha surgido como una parte crucial del proceso de diseño del software ya que es formada a partir de las estructuras de los sistemas de software (Shaw *et al*, 1996). El diseño de la arquitectura de un software es el primer paso para diseñar un sistema con los atributos deseados. En este caso, para lograr entender el impacto del atributo de la usabilidad en la arquitectura y las cualidades de la arquitectura que se deben de cuidar primero se tiene que definir los conceptos esenciales a continuación.

II.1. ¿Qué es la arquitectura de software?

La arquitectura de software de puede definir como los componentes o elementos del software, las propiedades de éstos y las relaciones entre ellos (Bass *et al*, 1996; Shaw *et al*, 1996). Aparte de especificar la estructura y topología del sistema, la arquitectura muestra la correspondencia entre los requerimientos del sistema y los elementos que van a conformar el sistema a construirse, justificando así las decisiones técnicas y de negocio para el diseño (Bass *et al*, 1996; Shaw *et al*, 1996).

Para tener un sistema aceptable es necesario que tenga ciertas propiedades, como es el rendimiento, soportabilidad, mantenibilidad, escalabilidad, compatibilidad, entre otros, que generalmente son denotados en los requerimientos no funcionales, siendo éstos los que impactarán de mayor forma a las decisiones tomadas para el diseño de la arquitectura del software a desarrollar (Bass *et al*, 1996; Shaw *et al*, 1996).

A continuación se presentan los principios fundamentales en el que se basa el diseño de la arquitectura del software.

II.2. Divide y vencerás

La estrategia del divide y vencerás se aplica al modularizar el diseño y el código conforme se avanza en el desarrollo del sistema bajo diferentes criterios, como por funcionalidad, entidades o cualidades (Parnas, 1972).

Esta estrategia se puede lograr en el software por medio del diseño modular. Se basa en la descomposición de comportamientos del software en unidades encapsuladas del software.

II.3. Atributos de la Arquitectura de Software

III.3.1. Diseño Modular

La modularidad se puede adquirir al agrupar elementos del software lógicamente relacionados, algunos de los elementos son: procedimientos, variables, atributos de objetos, entre otros (Parnas, 1972).

El objetivo principal del uso de la modularidad es adquirir una alta cohesión y un bajo acoplamiento (Parnas, 1972). Con respecto a unidades de código, la cohesión representa la conectividad de los elementos dentro de un módulo y acoplamiento representa conectividad entre módulos. Como se puede ver en la Figura 1, los recuadros grandes representan módulos, cada uno relacionado con otro módulo, cada módulo teniendo varios elementos internos interrelacionados entre sí.

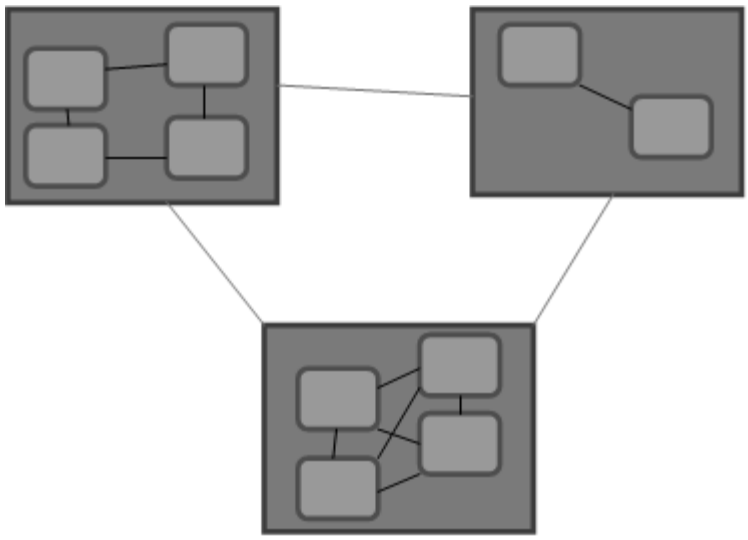


Figura 1. Módulos altamente cohesivos, poco acoplados.

II.3.1.1. Cohesión

Cohesión tiene que ver con la medida de qué tan relacionados son las responsabilidades de un solo módulo (Chidamber *et al*, 1994). Existen siete niveles de cohesión:

- **Coincidental.** Partes del módulo no están relacionados, simplemente se encuentran en el mismo módulo.
- **Lógico.** Partes que hacen funciones similares son puestos en un módulo.
- **Temporal.** Acciones que se ejecutan dentro de un mismo rango de tiempo se juntan.
- **Procedural.** Los elementos del módulo hacen una sola secuencia de control.
- **Comunicacional.** Todos los elementos de un módulo actúan de la misma manera que una estructura de datos.
- **Secuencial.** La salida de una parte del módulo sirve como entrada para otra parte del módulo.
- **Funcional.** Cada parte del módulo es necesaria para la ejecución de una sola función.

Alta cohesión implica que cada módulo representa una sola parte de la solución de un problema. Por lo tanto si un sistema necesita alguna modificación, entonces la parte que

necesita ser modificada existe en una sola parte, permitiendo así su cambio de una manera más sencilla, es uno de los atributos de la arquitectura que afecta directamente a la mantenibilidad (Kafura *et al*, 1987; Rombach, 1987) y será una de las características que se tienen que cuidar cuando se integre el modelo de usuario.

II.3.1.2. Acoplamiento

Se refiere al grado en que cada módulo del software depende en uno de los otros módulos (Cecil, 2002). Existe un gran beneficio al reducir el acoplamiento, permite que cambios hechos a una unidad de código no se propaguen a otros, por lo tanto se encuentran ocultos. El principio de "ocultar información", también conocido como el particionamiento de Parnas (Parnas, 1972), es la base del diseño de software.

El acoplamiento bajo limita los efectos de errores en un módulo y reduce la posibilidad de problemas de integridad de los datos. En algunos casos es necesario tener acoplamiento alto debido a estructuras de control. Por ejemplo, la mayoría de las interfaces de usuario se maneja un nivel de acoplamiento de control inevitable por la misma naturaleza de las mismas.

Los tipos de acoplamiento son los siguientes (Cecil, 2002):

- **Sin acoplamiento**. Todos los módulos son completamente independientes.
- **De Información.** Cuando un componente o módulo pasa información a otro en forma de un argumento.
- **De sello.** Cuando un módulo pasa una estructura de datos a otro, pero el módulo que la recibe solo usa algunos de los elementos de datos de la estructura.
- **Control.** Cuando un módulo pasa un elemento de control a otro, o sea, que un módulo controla la lógica de otro de manera explícita.
- **Común.** Si dos módulos tienen acceso a la misma información global.
- **Contenido.** Un módulo directamente hace referencia al contenido de otro.

Al igual que la cohesión, es uno de los atributos de la arquitectura que afecta directamente a la mantenibilidad (Kafura *et al*, 1987; Rombach, 1987) y será una de las características que se tienen que cuidar cuando se integre el modelo de usuario.

II.4. Particionamiento de Parnas

Particionar el software en unidades con bajo acoplamiento y alta cohesión se puede lograr con el principio de ocultar información (Parnas 1972). En esta técnica, se denotan las decisiones de diseño que es muy probable que vayan a cambiar. Unidades de código son designadas para ocultar la implementación de cada decisión o funcionalidad del resto del sistema. Por lo tanto, solo la función de las unidades de código es visible a otros módulos, no la forma en que implementó. Cambios en esas unidades de código es poco probable que afecten al resto del sistema.

Esta forma de descomposición funcional es basado en la noción que algunos aspectos de un sistema son fundamentales, mientras que otras pueden cambiar. Y esos aspectos que pueden cambiar, contienen datos y son fuente de complejidad.

El particionamiento oculta los detalle de implementación de funcionalidades del sistema, decisiones de diseño, entre otros, para limitar el impacto de futuros cambios o correcciones. Al particionar cosas que pueden cambiar, solo ese módulo se necesitará modificar cuando algún cambio es necesario de implementa sin tocar algún código de otro módulo.

En base de los fundamentos teóricos de la arquitectura de software surgen del trabajo de Parnas (1972), en particular sus nociones de la separación de preocupaciones y el de ocultar información de funcionalidad de módulos de código.

II.5. Estilos o patrones arquitecturales

Con el tiempo los diseñadores fueron logrando soluciones arquitectónicas a ciertos problemas de desarrollo de software, principalmente los concernientes a ciertos atributos de calidad los cuales se les pedía en los requerimientos del sistema, lo cual dio paso a los estilos o patrones arquitecturales (Bass *et al*, 1996).

Un estilo o patrón arquitectural define un conjunto de elementos en forma de un patrón u estructura organizacional. Determina el vocabulario de componentes y conectores que pueden ser usados en instancias de ese estilo, junto con una serie de reglas en como pueden ser combinados (Bass *et al*, 1996).

Conocer las características de los diferentes patrones es importante ya que cada uno de ellos es más apto para ciertos atributos de calidad que otros, teniéndolos que tomar en cuenta a la hora de diseñar un sistema dependiendo de las especificaciones de los requerimientos (Harrison *et al*, 2007).

En el Apéndice A se explican algunos de los estilos arquitecturales más comunes según Shaw *et al* (1996), y de los que generalmente se derivan otros estilos.

II.5.1. Arquitectura por capas

Un sistema basado en capas es estructurado jerárquicamente, sus componentes distribuidos en varias capas dependiendo de su funcionalidad. Cada una de las capas provee un servicio a la capa superior, así como también actuado como cliente de la capa inferior, como se muestra en la Figura 2.

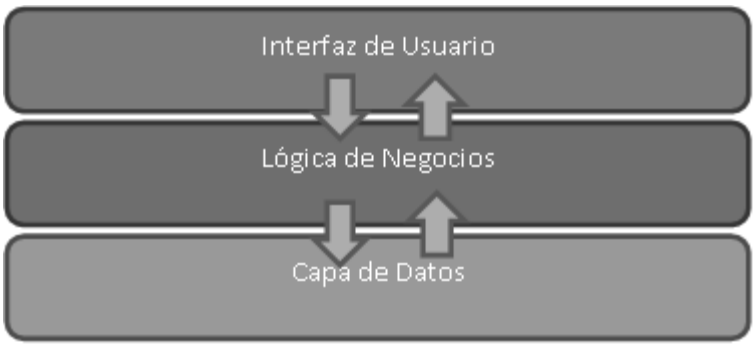


Figura 2. Arquitectura de capas. En este caso la arquitectura tradicional de tres capas.

Los sistemas basados en la arquitectura de capas tienen varias ventajas. El diseño por naturaleza es perfecto para la abstracción, permitiendo la descomposición de un problema en pasos incrementales. De igual manera fácilmente puede mejorar e incrementar los componentes de una capa ya que cualquier cambio solamente puede afectar con las que tenga comunicación, o sea las capas superior e inferior correspondientes. También tienden a ser altamente reusables, ya que cualquier capa puede ser cambiada por otra implementación sin problema alguno mientras se comunique de la misma forma con las otras capas con las cuales interactúa.

El estilo por capas también tiene sus desventajas. No todos los sistemas son aptos para un diseño por capas. Aún si el sistema puede construirse de manera lógica en capas, ciertas consideraciones de rendimiento pueden requerir un acoplamiento alto entre componentes de alto nivel y sus implementaciones de bajo nivel.

Este estilo es el que se utilizará como base para el modelo de integración debido a la combinación de características que tiene que favorece a la mantenibilidad, mientras mantiene un buen grado de usabilidad.

II.6. Vistas Arquitecturales

Para poder describir una arquitectura de software, se utilizan modelos compuestos de múltiples vistas o perspectivas (Kruchten, 1995). Estas vistas se pueden comparar a los planos de un edificio, permiten visualizar de diferentes maneras la arquitectura de lo que se va a construir, en este caso un software.

Existen varios modelos para la representación de la arquitectura de software, uno de los principales y de los más usados modelos es el framework 4+1 (Kruchten, 1995) el cual describe la arquitectura de sistemas de software basado en múltiples vistas concurrentes.

II.6.1. 4+1 Framework

Las vistas usadas describen al sistema desde el punto de vista de los usuarios finales del sistema, así como también los desarrolladores del mismo. Las cuatro vistas son: lógica, de desarrollo, de proceso y física. Además de una extra llamada escenarios o casos de uso para ilustrar las funcionalidades a representar del sistema (véase Figura 3).

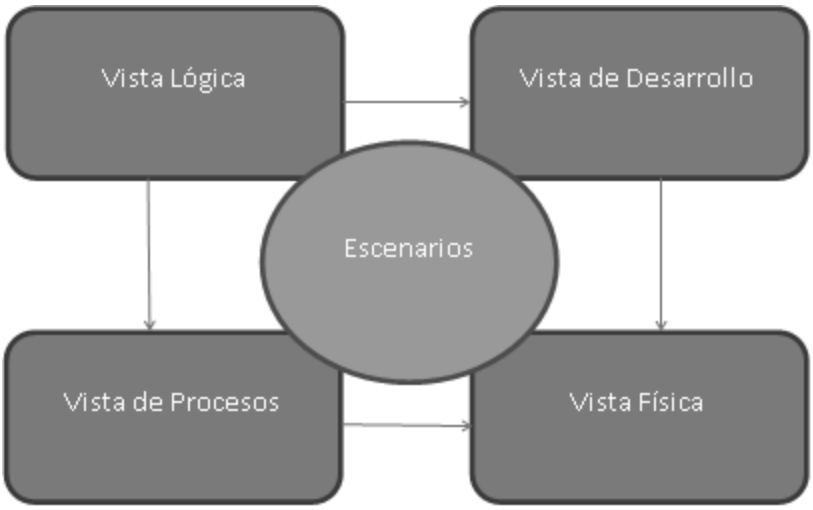


Figura 3. El Framework 4+1.

III.6.1.1. Vista Lógica

La vista lógica soporta los requerimientos funcionales. El sistema se descompone en una serie de abstracciones en la forma de objetos u clases de objetos. Esta descomposición no es solamente para hacer el análisis funcional, sino también para identificar mecanismos comunes e los elementos de diseño entre varias partes del sistema.

Para representar esta vista se utilizan los diagramas de clase, diagramas de comunicación y los diagramas de secuencia.

III.6.1.2. Vista de proceso

Toma en consideración los requerimientos no funcionales del sistema. El enfoque de esta vista es en los distintos atributos de calidad que se desean que tenga el software y cómo las abstracciones de la vista lógica serán distribuidas dentro del proceso del software.

Los diagramas UML utilizados para representar esta vista son los diagramas de actividad.

III.6.1.3. Vista de desarrollo

Esta vista se centra en la organización del módulo de software en su entorno de desarrollo. El software regularmente es compuesto de módulos, como librerías y subsistemas, que pueden ser usados por los desarrolladores. La vista de desarrollo es definido por diagramas que representan los módulos y subsistemas y las relaciones entre estos.

La vista de desarrollo toma en consideración los requerimientos internos relacionados con la facilidad de desarrollo, gestión del software, reusabilidad, entre otros, y las restricciones impuestas por el entorno de desarrollo o el lenguaje de programación.

Los diagramas UML utilizados para representar esta vista son los diagramas de componentes y los diagramas de paquetes.

III.6.1.4. Vista Física

La vista física toma en consideración principalmente los requerimientos no funcionales del sistema, como la disponibilidad, fiabilidad, rendimiento y escalabilidad. El software es ejecutado en diferentes sistemas físicos, como servidores, nodos de procesamiento, etc. Normalmente se utilizan diversas configuraciones físicas, algunas durante el desarrollo y fase pruebas otras durante el despliegue cuando los usuarios ya utilizan el sistema.

El diagrama UML utilizado para la vista física es el diagrama de despliegue.

III.6.1.5. Escenarios

Es la manera de representar como las cuatro vistas previamente mencionadas trabajan entre sí mediante una serie de escenarios o casos de uso, en el cual se describen las secuencias de interacciones entre los objetos y los procesos. Los escenarios o casos de uso son abstracciones de los requerimientos más importantes.

Esta vista tiene dos funciones principales:

- Sirve para descubrir elementos arquitecturales durante el diseño.
- Validar e ilustrar después de que el diseño ha sido completado, en papel y como punto de partida para pruebas del prototipo arquitectural.

Por medio de las vistas arquitecturales nos podemos dar una idea de las implicaciones que el diseño pueda tener, como por ejemplo los atributos de calidad que van a ser afectados por las decisiones arquitecturales que se toman, por lo tanto es importante tener bien documentada la arquitectura conforme se va diseñando por medio de vistas de los diferentes aspectos que la componen.

En esta tesis se usarán algunas vistas (la vista lógica y de desarrollo) para la ilustrar los experimentos realizados.

II.7. Calidad de Software

Como todo producto, ciertos atributos o características son esperados para poder decir que es un software de calidad. Pero la definición de un software de calidad puede variar dependiendo de las especificaciones del cliente para el software.

Como definición formal la norma ISO 9000:2001 define calidad como:

"Grado en que un conjunto de características inherentes cumple con los requisitos."

Existen varios modelos especificando los distintos atributos que se tiene que tomar en consideración para producir software de calidad. Los predecesores de los modelos de calidad de software actuales son el modelo de McCall (1977) y Boehm (1978).

II.7.1. Modelo de Calidad de McCall

Presentado por McCall *et al* (1977), es enfocado a los desarrolladores de sistemas y al desarrollo de procesos de software. En este modelo se intenta unir a los usuarios y los desarrolladores al enfocarse en un número de factores de calidad que reflejan la visión de los usuarios con las prioridades de los desarrolladores.

El modelo consta de tres partes para definir la calidad de un software: Revisión del producto, transición del producto y operaciones del producto (véase Figura 4).

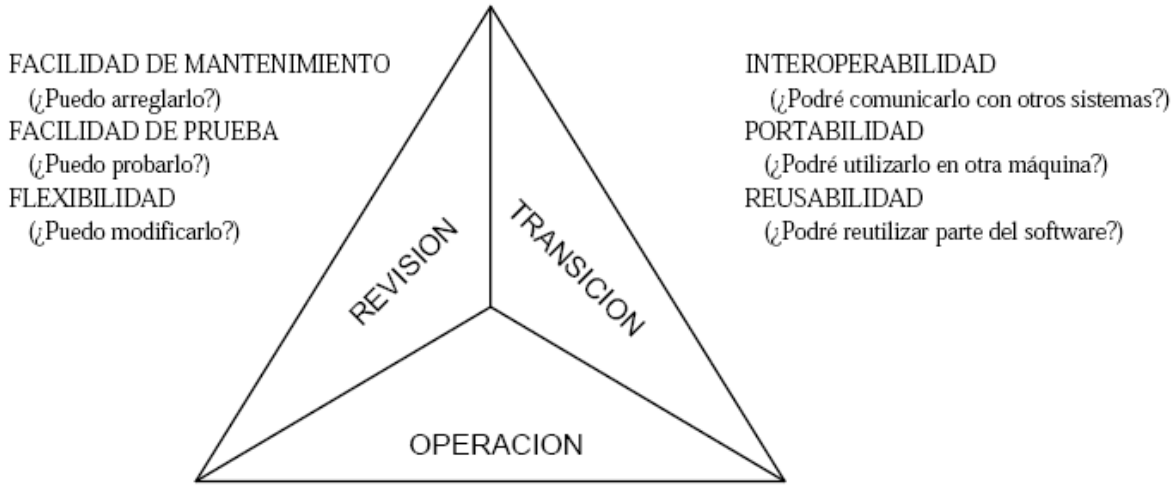


Figura 4. Modelo de McCall.

La revisión del producto incluye los siguientes atributos:

- Mantenibilidad. Facilidad para encontrar y arreglar una falla en el programa.
- Flexibilidad. Facilidad para hacer cambios en el sistema.
- Verificabilidad. Facilidad para hacer pruebas al sistema y comprobar que está conforme a las especificaciones.

La transición del producto contiene los siguientes atributos:

- Portabilidad. Esfuerzo requerido para transferir de un entorno a otro, por ejemplo, diferentes sistemas operativos.
- Reusabilidad. La facilidad de reusar el software en un contexto diferente.
- Interoperabilidad. El esfuerzo requerido para hacer que un sistema se relacione o funcione con otro sistema.

Las operaciones del producto dependen de:

- Correctitud. Grado en el que el software cumple con las especificaciones.
- Fiabilidad. Habilidad del sistema para no fallar.
- Eficiencia. Manejo adecuado de los recursos del entorno del sistema.
- Integridad. Protección del sistema por accesos no autorizados,
- Usabilidad. Facilidad de uso del sistema.

El modelo también describe los tres tipos de características de calidad en una jerarquía de factores, criterios y métricas (véase Tabla 1):

- 11 Factores para especificar. Describen la vista externa del software, vista por los usuarios.
- 23 criterios de calidad para construir o desarrollar. Describen la vista interna del software, vista por los desarrolladores.
- Métricas para controlar. Se usan para proveer una manera de medir ciertas cualidades.

Tabla 1. Factores de calidad relacionados a sus criterios de calidad

Métrica/Factor	Correctitud	Fiabilidad	Eficiencia	Integridad	Mantenibilidad	Flexibilidad	Testabilidad	Portabilidad	Reusabilidad	Interoperabilidad	Usabilidad
Facilidad de Auditoría				x			x				
Exactitud		x								x	
Normalización de las comunicaciones											
Completitud	x										
Consición			X		x	x					
Consistencia	x	x			x	x					
Estandares en los datos										x	
Tolerancia de errores		x									
Eficiencia de ejecución			X								
Expansibilidad						x					
Generalidad						x		x	x	x	
Independencia de harware								x	x		
Instrumentación				x	x		x				
Modularidad		x			x	x	x	x	x	x	
Operabilidad			X								x
Seguridad				x							
Documentación					x	x	x	x	x		
Simplicidad		x			x	x	x				
Independencia de Sistema								x	x		
Rastreabilidad	x										
Entrenamiento											x

Los factores de calidad describen diferentes comportamientos del sistema, y los criterios de calidad son atributos de uno o más factores de calidad. La métrica de calidad es para capturar o medir algún aspecto de un criterio de calidad.

II.7.2. Modelo de Calidad de Boehm

El segundo en los cuales se basan los modelos actuales de calidad de software es el presentado por Boehm (1978). En su modelo intenta definir cuantitativamente la calidad del software por medio de una serie de atributos y sus respectivas métricas (véase Figura 5).

Es similar al modelo de McCall en el sentido que también está estructurado de manera jerárquica estructurado en torno a características de alto nivel, de nivel intermedio y de bajo nivel o primitivas que contribuyen a las calidad en general del software.

Las características generales representan los requerimientos en los que el software puede ser evaluado. Las características de alto nivel son:

- Utilidad percibida: Nivel en que se puede usar el sistema tal cual como está.
- Mantenibilidad: Facilidad para entender, modificar y probar el software.
- Portabilidad: Facilidad para cambiar el entorno del desarrollo del software.

Las características de nivel intermedio representan las características que se esperan del software:

- Confiabilidad: Característica del software en el cual cumple sus funciones de manera satisfactoria.
- Portabilidad: Si el software puede ser operado en otros sistemas con diferentes configuraciones.
- Eficiencia: Capacidad del sistema para realizar sus funciones con la menor cantidad de recursos posibles.
- Capacidad de prueba: Característica del código de un software para poder ser verificado y evaluado.

- Flexibilidad: Capacidad del software para ser modificado e incorporar cambios.
- Usabilidad: Característica del software para poder ser usable.
- Comprensibilidad: Característica del código del software para poder ser entendido por algún inspector fuera del equipo de desarrollo.

El nivel más bajo del modelo son las características primitivas. Las características primitivas son la base para poder definir métricas de calidad.

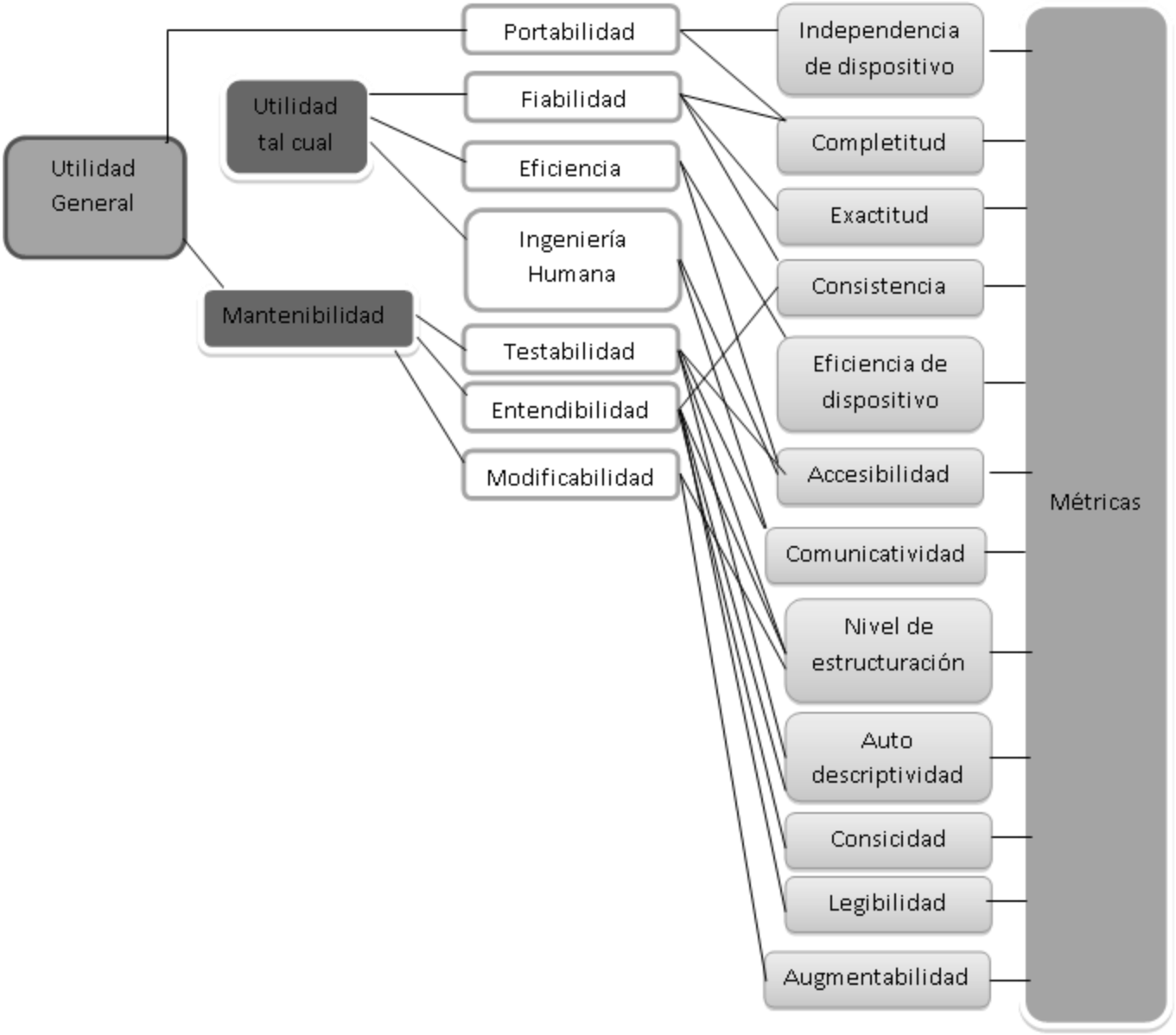


Figura 5. Modelo de Calidad de Boehm.

La diferencia entre los modelos de McCall y Boehm es que el modelo de McCall se enfoca en la medida de las características del alto nivel del software mientras que el de Boehm es basado en una gama más amplia de características con un enfoque extendido y detallado en la mantenibilidad.

II.7.3. Los Atributos de Calidad y la Arquitectura del Software

Según Bass *et al* (2003) la funcionalidad del software es independiente de los atributos de calidad. Si no lo fuera los requerimientos de funcionalidad dictarían el nivel de rendimiento, o de confiabilidad o cualquier otro atributo de calidad. Pero cada funcionalidad que se vaya a implementar puede tener diversos niveles de diferentes atributos de calidad.

La funcionalidad es la habilidad del sistema para poder llevar a cabo una determinada tarea. Una tarea requiere que muchos de los elementos del sistema trabajen de una manera coordinada para poder completar la tarea. Por lo tanto si esos elementos no se le han sido asignados las responsabilidades correctas, el sistema no podrá cumplir con la funcionalidad deseada.

Si la funcionalidad fuera el único requerimiento, el sistema puede ser un gran módulo monolítico sin estructura interna. Pero vemos que no se hacen así las cosas. Al ir desarrollando el sistema, la funcionalidad se va descomponiendo en módulos para hacerlo más entendible y que también cumpla con otros propósitos; de tal modo se puede concluir que la funcionalidad es independiente de la estructura. La estructura o arquitectura del software se va componiendo cuando otros atributos de calidad son importantes, por lo tanto, la arquitectura de software está basada en los requerimientos no funcionales.

Lograr los niveles deseados de los atributos de calidad debe de ser considerado desde las etapas de diseño, desarrollo y despliegue del sistema. Ningún atributo de calidad es enteramente dependiente del diseño o de la implementación. Para lograr un nivel satisfactorio

es necesario tener en cuenta panorama general del sistema, o sea la arquitectura, y los detalles particulares, o sea la implementación.

Bass *et al* (2003) concluye dos puntos importantes sobre la arquitectura y su relación con los atributos de calidad: Primero, que la arquitectura es crítica para la realización de los atributos de calidad deseados, y que deben de ser diseñados en y deben de ser evaluados en el nivel arquitectural; y segundo, que la arquitectura, por si sola, no puede llegar a tener los atributos de calidad que se desea del sistema, provee de las bases para poder obtenerlos, pero no será posible si no le presta atención a los detalles.

II.8. Usabilidad

La usabilidad, como muchos otros términos en la ingeniería de software, tiene diferentes definiciones. El ISO 9126:2001 expresa la siguiente definición:

"La usabilidad se refiere a la capacidad de un software de ser comprendido, aprendido, usado y ser atractivo para el usuario, en condiciones específicas de uso".

En términos más simples, la usabilidad, como un atributo de calidad del software, es el grado de la facilidad de uso de un software.

II.8.1. Atributos de la usabilidad

La usabilidad consta de los siguientes atributos (Nielsen, 1993):

- **Aprendibilidad.** Facilidad para que los usuarios puedan llevar a cabo las funcionalidades básicas la primera vez que utilizan el sistema.
- **Eficiencia.** Una vez que los usuarios han aprendido a usar el sistema, la rapidez con la cual los usuarios logran realizar una tarea.
- **Memorabilidad.** Facilidad de los usuarios para reaprender a usar el sistema después de no usarlo por un periodo de tiempo.

- **Tasa de errores.** Cantidad de errores que el usuario comete al intentar realizar una tarea, también toma en consideración la severidad de los errores, así como también si el usuario es posible de recuperarse de dichos errores.
- **Satisfacción.** Atributo subjetivo de la usabilidad en el cual influyen factores del usuario como el humor, gustos, preferencias, entre otros. Determina que tan placentera fue la experiencia con el software.

En base a estos atributos se ha desarrollado un proceso enfocado a la usabilidad el cual ayuda a los actuales procesos de desarrollo de software a hacer un desarrollo centrado en el usuario asegurando un software mucho más usable.

II.8.2. Ingeniería de la Usabilidad

La ingeniería de usabilidad se puede definir como un proceso iterativo sistematizado para la implementación de la usabilidad en el software, con el fin de lograr software más usable (Nielsen, 1993; Priece *et al*, 1994).

La objetivo de aplicar la ingeniería de usabilidad en el ciclo de vida de desarrollo de software es que al aumentar la usabilidad, el usuario tiende a ser más productivo, logrando que sea más fácil lograr sus tareas aumentando la satisfacción al usarlo (Ferre *et al*, 2001).

La usabilidad se debe de tomar en cuenta desde la etapas del diseño del software, incluyendo sus atributos, ya que pueden ser la clave para que un software sea exitoso; si el software es demasiado difícil de usar o poco usable no va a importarle toda la funcionalidades que pueda tener, el usuario común se desesperara y no usará el software si existe una alternativa más fácil de usar. De igual forma, se pueden detectar problemas de usabilidad desde etapas de desarrollo tempranas y evitar estos tipos de problemas más adelante ahorrando tiempo y dinero, logrando así mejores oportunidades a que el software sea exitoso en el mercado.

II.8.3. Ciclo de vida de la ingeniería de usabilidad

Con las tendencias actuales, principalmente en las aplicaciones móviles, se desea una alta usabilidad. Para asegurar la usabilidad de los sistemas interactivos, se debe de tomar en consideración los detalles de usabilidad en el proceso de desarrollo de software.

Para lograr los niveles deseados de usabilidad, se han propuestos varios modelos del proceso de ingeniería de software, siendo los dos principales el propuesto por Jacob Nielsen en 1992 y Deborah Mayhew en 1999. A partir de esos modelos se han propuesto varias metodologías para el desarrollo de software que integran varios de las actividades de la ingeniería de usabilidad.

Dos de los factores que más impactan a la usabilidad son las características individuales del usuario y las tareas que deben hacer (Nielsen, 1993), por ello la ingeniería de usabilidad se centra en dos técnicas: el análisis de tareas y el análisis de usuario.

II.8.3.1. Análisis de tareas

Un definición simple del análisis de tareas es la descripción de una serie de técnicas que las personas utilizan para lograr una tarea (Ferre *et al*, 2001). Durante el análisis de tareas en la ingeniería de usabilidad es necesario describir todas las acciones para lograr el cometido de una cierta tarea.

Durante esta etapa, las metas del usuario son analizadas, así como también todas las acciones, la información requerida y todas las posibles consecuencias de esas acciones. Las tareas son descompuestas en subtareas, y subsecuentemente en acciones que el usuario va a hacer. Con las acciones identificadas, los componentes de la interfaz de usuario para llevar a cabo dicha acción pueden ser determinados.

II.8.3.2. Análisis de usuario

El segundo paso del ciclo de vida de la ingeniería de usabilidad es el estudiar a los usuarios para determinar la clase de personas que usarán el sistema (Nielsen, 1993; Kim *et al*, 2002; Bellotti *et al*, 2009).

Esto significa conocer de manera profunda las características del usuario para poder visualizar su perfil. El análisis de usuario identifica a los usuarios que van a usar el sistema y sus características.

II.8.4. El proceso de la ingeniería de usabilidad según Nielsen

Propone el siguiente listado de elementos para obtener un modelo dela ingeniería de usabilidad (Nielsen, 1993):

0. Considera el contexto
1. Conoce al usuario
    a. Características individuales del usuario
    b. Las tareas del usuario
    c. Análisis funcional
    d. Evolución del usuario
2. Análisis competitivo
3. Definir metas de usabilidad
4. Diseño participatorio
5. Diseño total coordinado de la interface
    a. Estándares
    b. Identidad del producto
6. Guías y análisis heurístico
7. Prototipado
8. Pruebas empíricas

9. Diseño iterativo
    a. Capturar el raciocinio del diseño
10. Recolectar retroalimentación de las pruebas en el campo

Los elementos más básicos del modelo son las pruebas empíricas y el prototipado combinado con el diseño iterativo. Porque es casi imposible diseñar la interfaz del usuario adecuada en el primer intento, se ocupa probar, prototipar y planear las modificaciones a ésta al usar un proceso iterativo para el desarrollo del software.

II.8.4.1. En el desarrollo del producto

La siguiente sección se presentan actividades de usabilidad para tres de las fases principales de un proyecto de software: antes, durante y después del diseño e implementación.

II.8.4.1.1 Antes del diseño

La primera fase del ciclo de vida de la ingeniería de usabilidad es el entender a nuestros usuarios y las tareas que van a hacer con nuestro software. Podemos hacer decisiones sobre el diseño solo cuando entendamos a esos factores, entonces el adquirir esa información es vital y de primera prioridad. De igual forma, no se debe de apresurar demasiado al diseño hasta no obtener toda la información necesaria. Es más barato el hacer lo más posible antes de hacer el diseño a tener que hacer cambios constantes más adelante.

Varias de las actividades de usabilidad que se hacen antes del diseño se pueden considerar parte del estudio del mercado o del proceso de planeación del proyecto y se que se pueden llevar a cabo por un equipo de mercadotecnia. Pero, un estudio de mercado no usa todos los métodos de diseño de usabilidad, lo que ocasiona que haya problemas de comunicación de los resultados entre los de mercadotecnia y los desarrolladores. Pero no

debería de haber problema si los directivos integran de buena manera las actividades de usabilidad con las del estudio de mercado.

II.8.4.1.1.1. Conoce a tu usuario

El primer paso en el proceso de la ingeniería de usabilidad es estudiar a nuestros usuarios. Los desarrolladores deberán visitar el entorno en donde se va a hacer uso del software para tener una mejor idea de cómo va a ser usado. Diferencias individuales del usuario y diferencias en las tareas a realizar son los dos factores que tienen mayor impacto en la usabilidad, por lo tanto es necesario estudiar profundamente.

II.8.4.1.1.1.1. Características individuales del usuario

Es necesario saber el tipo de personas que utilizarán el sistema. En algunas situaciones es posible identificar los usuarios como un tipo de usuario específico, o en caso de que vaya a ser usado por una amplia gama de usuarios, un segmento representativo de los usuarios.

Al conocer características de los usuarios como experiencia laboral, nivel educativo, entre otros podemos diseñar una mejor interfaz y anticipar en que partes pudieran tener problemas al usar nuestro software.

II.8.4.1.1.1.2. La actual tarea a realizar del usuario

El análisis de tarea es extremadamente importante para el primer diseño del sistema. Las metas del usuario deben de ser estudiadas, así como también, los pasos para lograr dichas metas. De tal manera que puedan servir para la identificación de los componentes gráficos necesarios, así como también, las estrategias necesarias para lograr dichas metas.

II.8.4.1.1.1.3. Análisis funcional

Se analizan las tareas y acciones en conjunto con el análisis de tareas para determinar si se pueden ser mejoras u optimizaciones en los procesos que se llevan a cabo para cumplir con las tareas.

II.8.4.1.1.2. Análisis competitivo

Se analizan productos similares de competidores para determinar las ventajas y desventajas de su propio diseño, tomando lo bueno para nuevas ideas y guías. La mejor forma de hacer el análisis es haciendo pruebas de usabilidad con el sistema de la competencia que incluyan al usuario.

II.8.4.1.1.3. Definiendo metas de usabilidad

Se deben de considerar los atributos de la usabilidad para obtener el mejor nivel de usabilidad posible, siendo estos: aprendibilidad, eficiencia, memorabilidad, tasa de errores y satisfacción.

No es posible obtener los mejores resultados para cada uno de los atributos, así que se debe de establecer prioridades entre los atributos dependiendo de los resultados obtenidos del análisis de tareas y de usuario.

II.8.4.1.2. En la etapa del diseño

El objetivo principal en esta etapa es llegar a una implementación usable que pueda lanzarse. Para eso se ocupan otros dos objetivos, uno siendo el diseñar un prototipo que sigue

con los principios de usabilidad establecidos y el otro el de verificar el diseño con los usuarios reales para determinar si cumple con sus necesidades.

II.8.4.1.2.1. Diseño participatorio

Aunque se intente conocer al usuario lo más posible, nunca se va a poder conocerlo completamente de tal forma que se conozcan todas las respuestas a todos los problemas o preguntas que se puedan presentar. Los diseñadores deben de tener acceso a un grupo representativo de usuarios durante la etapa del diseño, para cualquier pregunta o problema que se pueda presentar. De igual manera, los usuarios pueden hacer preguntas que pueden alterar la manera en que se está haciendo el diseño, en especial con respecto a la manera en que se llevan a cabo las tareas del usuario.

Para ayudar al usuario entender cómo se está diseñando el sistema, es importante presentarle la información pertinente de manera sumamente simple, por ello se recomienda el uso de prototipos del software. Si no es posible presentar un prototipo funcional, en especial durante las primeras instancias de la etapa del diseño, se pueden presentar bosquejos o dibujos de algunas interfaces pueden dar como resultado opiniones y discusiones de los usuarios en algunos aspectos y por ende, modificaciones a requerimientos o nuevos requerimientos.

II.8.4.1.2.2. Diseño coordinado

Consistencia del diseño de la interfaz es una de las características de la usabilidad más importantes. Debe de aplicar a través de toda la interfaz del software y documentación del mismo. De igual forma debe también aplicar en subsecuentes lanzamientos del software aunque entre en conflicto con alguna otra característica. A veces se pierde algo de flexibilidad del sistema al forzar un mal diseño al usuario por intentar mantener consistencia.

Para lograr una mejor consistencia se recomienda que haya un comité o una persona encargada para coordinar los diversos aspectos de la interfaz, además de tener una cultura compartida entre todos los desarrolladores de cómo debe de ser la interfaz.

En esta etapa de nuevo es indispensable el prototipado y la observación del diseño en software similar ya que pueden dar una mejor idea del camino a seguir en el diseño de la interfaz.

II.8.4.1.2.1. Estándares

Un estándar de diseño puede ser uno promovido por el un sistema en particular, en muchos casos el del sistema operativo, o puede ser un estándar interno de la empresa que desarrolla el software. El uso de estándares de diseño asegura que el producto sea consistente con otras series de productos hechos por otras empresas. La ventaja de estándares internos es que puede ser adaptado para las necesidades específicas de un software en particular.

II.8.4.1.2.2. Identidad del producto

Es una descripción de que se trata el producto, o software en este caso. Especifica las metas del proyecto: el problema que el producto va a resolver, quien lo va a usar y en conjunto con que otros productos se va a usar. Con esto en mente, puede ayudar a coordinar el diseño ya que es un documento pequeño que todos los miembros del equipo de desarrollo deben de conocer.

II.8.4.1.3. Guías y análisis heurístico

Guías son una lista de principios o recomendaciones para el diseño de la interfaz que deben de ser seguidas por todos en el desarrollo. Existen varias tipos de guías:

- Guías generales. Aplicable a todas las interfaces de usuario.

- Específicas a cierta categoría. Aplicables solo a unos tipos de sistemas que se desarrollan.
- Guías específicas al producto. Afectan a un producto en particular.

Guías generales pueden ser publicadas en journals técnicos o en otras literaturas académicas. Una serie de guías cortas, como las heurísticas de usabilidad de la Tabla 2 pueden ser usados como una lista de elementos a considerar en una evaluación por heurísticas, mientras que una gran serie de reglas guías pueden servir como referencia para responder preguntas específicas de diseño.

Tabla 2. Heurísticas propuestas por Nielsen en su publicación del ciclo de vida de la Ing. De Usabilidad.

Heurísticas de usabilidad
Usa diálogos simples y naturales
Usar el lenguaje del usuario
Minimizar la carga de memoria del usuario
Proveer de retroalimentación
Ser consistente
Proveer salidas claramente marcadas
Proveer buenos mensajes de errores
Prevenir errors

Guías de usabilidad pueden contener contradicciones que pueden ser difíciles de resolver. Para hacer las propias compensaciones entre las guías se necesita entender el porqué de las guías para hacer las decisiones apropiadas. Las guías que contienen listas de ventajas y desventajas de varias formas de diseño también nos pueden ayudar a hacer las debidas compensaciones, pero son difíciles de aplicar para alguien que no es experto en usabilidad.

En general se recomienda que exista en el proyecto al menos un experto de usabilidad para ayudar a resolver guías posiblemente contradictorias y ayudar con la evaluación por medio de las heurísticas. Puede ser que haya diferencias entre los consejos de varios expertos de usabilidad, esto no implica que por lo menos uno de ellos esté equivocado. Existe mucha libertad en el diseño de interfaz de usuario que más de una solución puede ser razonable.

II.8.4.1.4. Prototipado

Es recomendable la presentación de prototipos ya que es frecuente cuando un cliente no sabe lo que quiere, pero cuando lo prueban saben bien que es lo que no quieren.

Un prototipo es una representación limitada de nuestro software, generalmente muestra solamente algunas funcionalidades muy particulares en cada versión. Es especialmente importante para el correcto diseño de la interfaz, ya que es muy rara la ocasión en la que se acierte a la primera vez un diseño óptimo para nuestros usuarios. Inclusive no es necesaria una implementación ejecutable para hacer prototipos al inicio del diseño, simples dibujos en hoja de la interfaz pueden ser suficientes para causar discusión entre usuarios y desarrolladores y proponer posibles modificaciones.

En los modelos de desarrollo tradicionales de software se obtiene un software ejecutable en las últimas etapas de desarrollo. Un problema con esta forma de desarrollo es que no existe una interfaz para que los usuarios finales prueben hasta tener la versión casi final del software, lo cual dificulta cualquier posible modificación si desean alguna los usuarios.

II.8.4.1.5. Pruebas empíricas

Hay dos tipos de pruebas empíricas:

- Pruebas con una interfaz terminada para determinar si se han obtenido las metas de usabilidad impuestas. Estas pruebas ocupan alguna forma de medición cuantitativa.
- Una evaluación formativa del sistema mientras está siendo diseñado para ver qué aspectos de la interfaz de usuario causan problemas. Este tipo de pruebas

regularmente usa métodos cualitativos ya que es más importante saber el por qué está mal algún aspecto de la interfaz que saber que tan mal esta.

En ambos tipos de pruebas los usuarios efectúan tareas obtenidas del análisis de tareas. Algunos de los métodos para la evaluación son:

- Pensar en voz alta.
- Interacción constructiva. Donde dos usuarios trabajan en conjunto para hacer la prueba. Facilita la interacción e intercambio de ideas.
- Cuestionarios de actitud. Donde se califica el sistema en base a una escala.
- Probar el nivel de conocimiento del usuario antes y después de usar el sistema.
- Registro automático en la computadora de acciones del usuario. Análisis posterior puede ayudar a encontrar algún problema con la interfaz.
- Observación de los usuarios en su entorno laboral haciendo sus propias tareas.
- Observación de los usuarios haciendo ciertas tareas y actividades representativas.

De los resultados de las pruebas se obtienen una lista de problemas de usabilidad detectados en la versión de prueba, y posibles mejoras o adiciones a la interfaz en subsecuentes versiones. Debido a que generalmente no es posible arreglar todos los problemas de usabilidad se deben de dar prioridad a los que más afectan a las metas de usabilidad establecidas.

En ocasiones al intentar arreglar un problema de usabilidad, se crea un problema para otros usuarios que no tuvieron ese problema. Se tiene que hacer un análisis del impacto para determinar si se mantiene la interfaz como está o se hacen las modificaciones para corregir el problema. El análisis es basado en la frecuencia de usuarios que se tendrán el problema comparado con el número de usuarios que les ocasionará problemas la implementación de la solución. De igual manera se deben de tomar en cuenta el tiempo y los costos asociados con el determinar las prioridades de corregir los problemas de usabilidad.

II.8.4.1.6. Diseño iterativo

La misma naturaleza del uso de prototipos y las pruebas empíricas pide un diseño iterativo, ya que cada nueva versión de la interfaz tiene que ser también sujeta de nuevo a los procesos de pruebas para la detección de problemas usabilidad y la implementación de soluciones de esos problemas para el siguiente prototipo. De igual forma, este tipo de diseño, ayuda al análisis necesario para el análisis que se tiene que hacer si la solución a algún problema de usabilidad afecta de manera negativa a algún otro usuario que no presentaba este problema.

II.8.4.2. En la etapa después del diseño

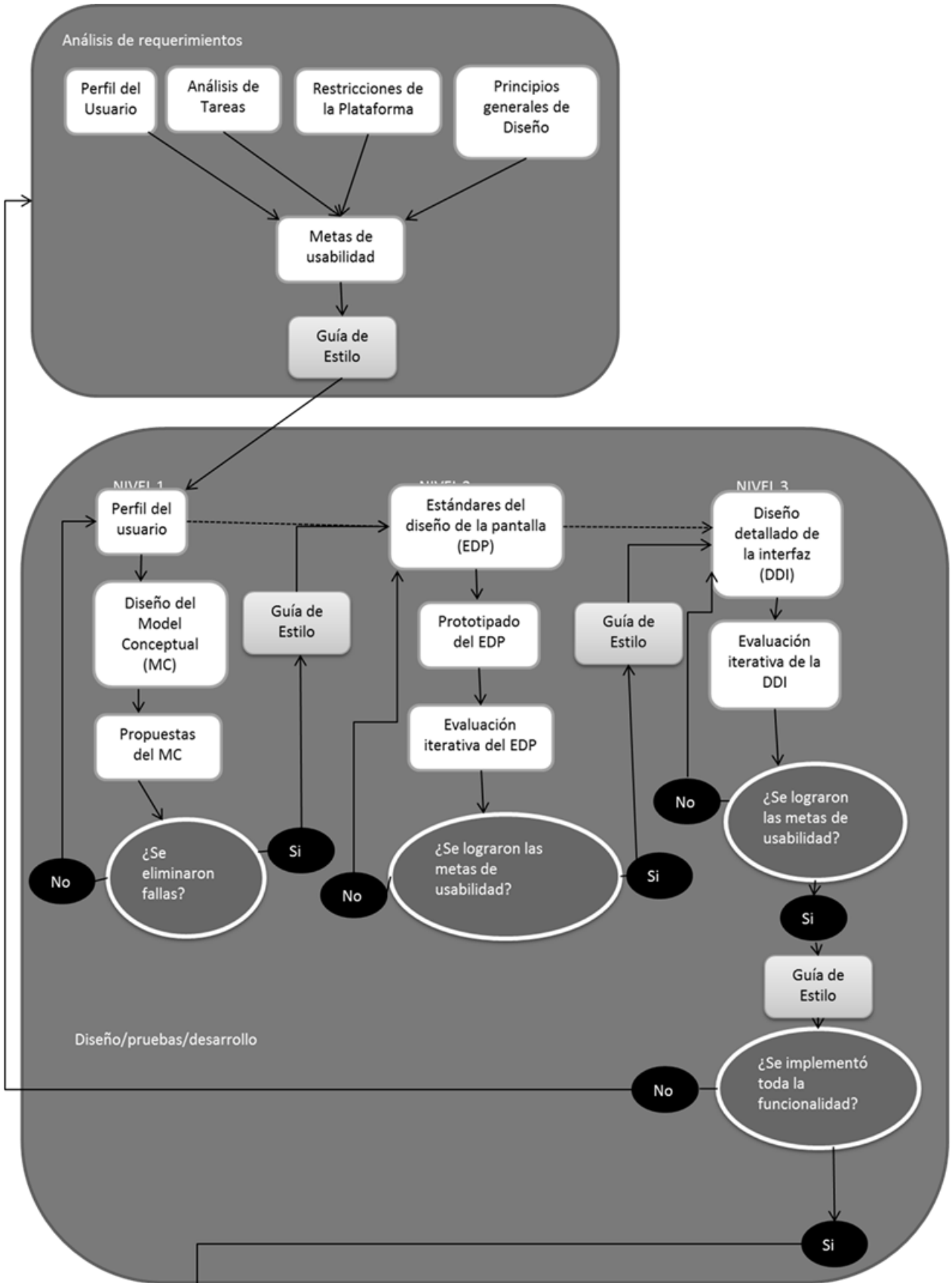
El objetivo principal es recabar datos para los siguientes productos o versiones del software, de tal forma que un producto recién lanzado puede verse como un prototipo para futuros productos. Por lo tanto se requieren estudios del producto lanzado siendo usado por los usuarios. Dichos estudios son basados en el uso del software por los usuarios en el ambiente en que se requieren de las tareas para que el software fuera diseñado.

Una forma de obtener retroalimentación de los usuarios es el guardar información de los usuarios cuando llaman para pedir ayuda con el producto u otras estructuras de ayuda que ofrezca la empresa que diseño el software. De igual forma se tiene que conocer también los aspectos positivos del sistema visitando a los usuarios y aplicando encuestas. También guardando logs de como el usuario usa el sistema nos da información importante que puede usarse para mejorar subsecuentes versiones.

II.8.5. Ciclo de vida de la ingeniería de usabilidad según Deborah Mayhew

Fue propuesto en 1999 (Mayhew, 1999). Provee un ciclo de vida con una visión integral de la ingeniería de usabilidad y una descripción detallada de cómo llevar a cabo las actividades de usabilidad, de igual forma especifica cómo pueden ser integrados esas actividades en los ciclos de vida tradicionales de desarrollo de software.

El ciclo de vida consta de tres tareas o actividades: análisis de requerimientos, diseño/pruebas/desarrollo, e instalación, siendo la etapa de en medio la más grande y formada de varias subtareas (Figura 6).


Análisis de requerimientos
Perfil del Usuario
Análisis de Tareas
Restricciones de la Plataforma
Principios generales de Diseño
Metas de usabilidad
Guía de Estilo
NIVEL 1
NIVEL 2
NIVEL 3
Perfil del usuario
Estándares del diseño de la pantalla (EDP)
Diseño detallado de la interfaz (DDI)
Diseño del Model Conceptual (MC)
Guía de Estilo
Prototipado del EDP
Guía de Estilo
Evaluación iterativa de la DDI
Propuestas del MC
Evaluación iterativa del EDP
No
¿Se lograron las metas de usabilidad?
Si
No
¿Se eliminaron fallas?
Si
No
¿Se lograron las metas de usabilidad?
Si
Si
Guía de Estilo
Diseño/pruebas/desarrollo
No
¿Se implementó toda la funcionalidad?
Si
Instalación

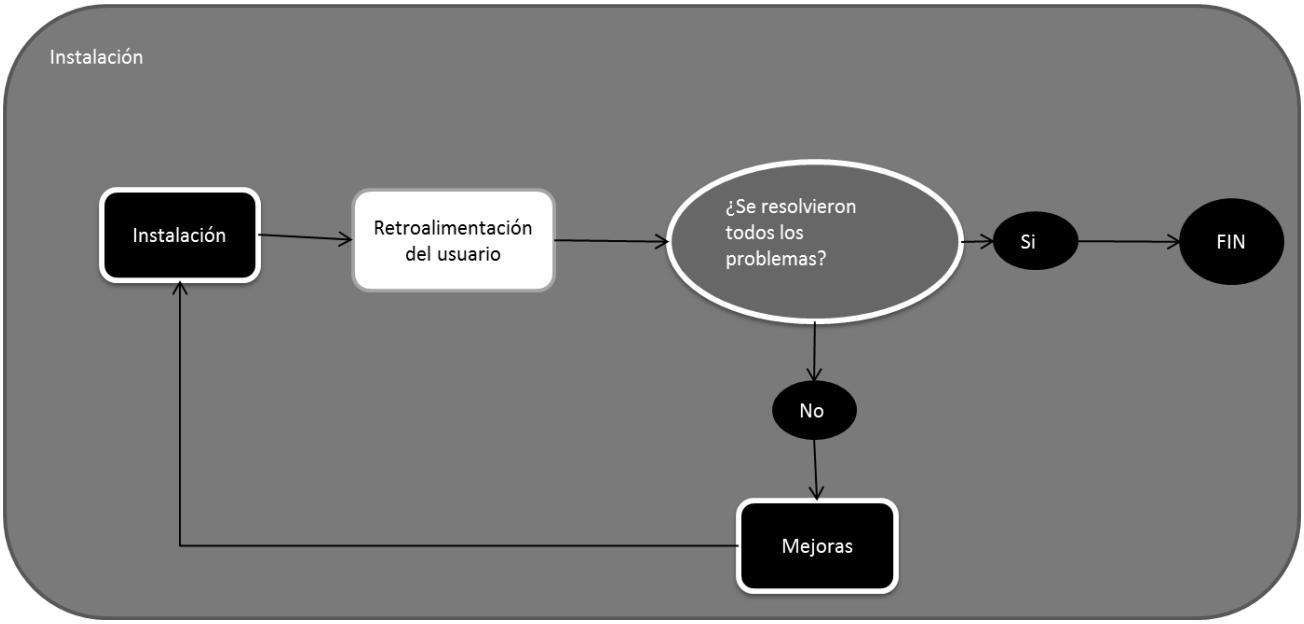


Figura 6. Ciclo de vida de la Ingeniería de Usabilidad de Deborah Mayhew

Incluye etapas de identificación de requerimientos, diseño, evaluación y creación de prototipos de igual forma que el modelo de Nielsen. También incluye la guía de estilos como un mecanismo para capturar y diseminar las metas de usabilidad del proyecto.

Consiste de diversos tipos de tareas:

- Un análisis estructurado de los requerimientos de usabilidad.
- Estipular metas de usabilidad, en base a lo obtenido del análisis de los requerimientos.
- Tareas soportando un diseño de interfaz de usuario de arriba hacia abajo dirigido directamente por las metas de usabilidad y otra información obtenida de los requerimientos.
- Tareas objetivas de la evaluación de la usabilidad en un proceso iterativo para lograr las metas de usabilidad.

II.8.5.1. Fase 1: Análisis de requerimientos

II.8.5.1.1. Perfil del usuario

El perfil de usuario es una descripción de las características relevantes del usuario al diseño de interfaz. El perfil influenciará de manera importante las decisiones de diseño que se tomen con respecto a la interfaz y ayuda a identificar categorías de usuario para el análisis de tareas contextualizado.

II.8.5.1.2. Análisis de tarea contextual

Un estudio detallado de las tareas que el usuario debe de realizar, resultando en una descripción de las metas del usuario. Estas metas se usarán para determinar metas de usabilidad e impulsar el diseño de la interfaz de usuario y el trabajo de reingeniería.

II.8.5.1.3. Determinar metas de usabilidad

Se determinan metas cualitativas determinados de los requerimientos de usabilidad obtenidos del perfil de usuario y del análisis de tareas, y metas cuantitativas, definiendo valores mínimos aceptables de la eficiencia y satisfacción del usuario basándose en un criterio de metas cualitativas de alta prioridad. Esas metas de usabilidad enfocan los esfuerzos de diseño y forman la base de subsecuentes evaluaciones iterativas de la usabilidad.

II.8.5.1.4. Cualidades de la plataforma y sus limitaciones

Las cualidades inherentes del Sistema Operativo en que se desarrolla para la interfaz del usuario son determinadas y documentadas. Estos determinarán las posibilidades para el diseño de la interfaz.

Estas cuatro tareas en la etapa de análisis de requerimientos se documentan en un documento llamado la Guía de estilo del producto.

II.8.5.1.5. Principios generales de diseño

Información general relevante sobre los principios y guías de diseño de la interfaz son recopilados y analizados. Se aplicarán durante el proceso del diseño que seguirá, junto con otra información específica del proyecto reunida de las cuatro tareas anteriores.

II.8.5.2. Fase Dos: Diseño, Pruebas y Desarrollo

Esta fase del ciclo de vida se divide en tres niveles.

II.8.5.2.1. Nivel 1 de Diseño

Contiene cuatro tareas que tratan decisiones de diseño de alto nivel.

II.8.5.2.1. Trabajo de Reingeniería

Es basado en toda la información obtenida del análisis de requerimientos y las metas de usabilidad extraídas de ahí, tareas del usuario son rediseñadas al nivel de organización y flujos de trabajo para automatizar las tareas. No se involucra el diseño de interfaz aún, solo una organización abstracta de la funcionalidad y diseño del flujo de trabajo.

II.8.5.2.2. Diseño del Modelo Conceptual

Basado en las tareas previas, diseños alternativos de alto nivel son generados. A este nivel, la navegación e interfaces principales son identificadas, y reglas para una presentación consistente de los productos de trabajo, procesos y acciones son establecidas. Aún no se detalla diseños de las pantallas.

II.8.5.2.3. Diseños Conceptuales Preliminares

Diseños a lápiz y papel o diseños prototipos son generados a partir de las tareas previas, representando ideas organizacionales funcionales y del diseño del modelo conceptual.

II.8.5.2.4. Evaluación del modelo conceptual iterativo

Los diseños preliminares propuestos son evaluados y modificados de manera iterativa por medio de métodos formales de evaluación de usabilidad, en donde usuarios representativos intentan llevar a cabo tareas representativas con un mínima ayuda e intervención, imaginando que las interfaces prototipo son una interfaz final del producto. Esta tarea y las dos previas son llevadas a cabo en ciclos iterativos hasta que todos los problemas principales de usabilidad detectados son corregidos del modelo conceptual. Una vez que el modelo conceptual es relativamente estable, diseño de la arquitectura del sistema puede comenzar.

II.8.5.2.2. Nivel 2 de Diseño

También con 4 tareas, se encarga de imponer estándares.

II.8.5.*2.2.1. Estándares de Diseño de Interfaz*

Una serie de estándares específicos al producto para todos los aspectos concernientes al diseño de la interfaz son desarrollados, basados en algún estándar impuesto por la industria o empresa, la información obtenida de la fase de análisis de requerimientos y el modelo conceptual de diseño obtenido durante el nivel 1.

El manejo de estándares de interfaz asegurará un nivel de coherencia y consistencia, los cuales forman parte de la base de la usabilidad.

II.8.5.*2.2.2. Prototipado de los Estándares de Interfaz*

Los estándares de interfaz y el diseño conceptual de interfaz son aplicados al diseño de varias de las interfaces de usuario de funcionalidades seleccionadas. Este diseño es implementado en forma de un prototipo ejecutable.

II.8.5.*2.2.3. Evaluación Iterativa de los Estándares de Diseño de Interfaz*

Evaluaciones son diseñadas para obtener retroalimentación de la usabilidad de los prototipos. Pueden ser aplicadas antes del diseño detallado y del desarrollo de código. Por lo tanto, cambios se pueden hacer a la interfaz propuesta fácilmente.

II.8.5.*2.2.4. Desarrollo de la Guía de Diseño o Estilo*

Aquí se completa la Guía de Interfaz del producto de la fase anterior, y junta todos los documentos individuales relacionados. Contiene los principios, guías y estándares de diseño que se aplicarán al producto.

II.8.5.2.3. Nivel 3 del Diseño

II.8.5.*2.3.1. Diseño Detallado de la Interfaz de Usuario.*

Diseño y documentación detallada de la interfaz de usuario de completan.

II.8.5.*2.3.2.* *Evaluación Iterativa Detallada del Diseño de la Interfaz de Usuario*

Aplicando alguna de las técnicas de evaluación de la usabilidad para evaluar de manera iterativa y refinar los detalles del Diseño de la Interfaz de Usuario.

II.8.5.*3.* *Fase Tres: Instalación*

II.8.5.*3.1.* *Retroalimentación del Usuario*

Reunir retroalimentación de los usuarios después de que un producto es instalado, con el fin de mejorar los diseños y nuevos productos relacionados.

Como se puede observar, en ambos ciclos de la ingeniería de usabilidad enfatizan la importancia de las características del usuario, así como también las tareas de deben de realizar ya que son los dos factores que afectan de mayor medida a la usabilidad. De este modo se puede tener una mejor idea de las implicaciones arquitecturales que pueden tener la implementación de diversos patrones de usabilidad óptimos para el usuario teniéndolos en cuenta desde las etapas iniciales de requerimientos y diseño de la arquitectura.

II.8.6. Modelos de Usuario

La usabilidad se puede mejorar con la aplicación de modelos de usuario, que se definen como modelos de los usuarios que residen dentro de un entorno computacional (Elain, 1976), en otras palabras, una representación abstracta de las características del usuario.

Los modelos de usuario se utilizan como una forma de sistemas de software para adaptar la interfaz de usuario para mostrar información relevante de una manera apropiada y el tiempo (Gerhard, 2001).

Hay varios tipos de modelos de usuario, siendo los más utilizados:

- **Modelo de usuario estático:** Información sobre el usuario es recogida y almacenada, el sistema no controla ni se entera de los cambios en la forma en que el usuario de interactuar con el sistema (Johnson *et al*, 2005).
- **Modelo de usuario dinámico:** El sistema controla la forma en que el usuario interactúa con el sistema y el modelo se actualiza como resultado, lo que resulta en una actualización del modelo del usuario. (Hothi *et al*, 1998).
- **Modelos en base a un Estereotipo**: Basado en información estadística de un grupo demográfico específico de usuario (Elain, 1979).

La principal diferencia entre los modelos es el tipo de información almacenada acerca de los usuarios y la forma en que se pone al día, por esas razones es por lo general utiliza una mezcla de los modelos mencionados en el desarrollo de un sistema modelo de usuario basado en (Hothi *et al*, 1998).

El problema para asistir a la usabilidad en un sentido amplio surge cuando el usuario del software tiene un rango distinto y amplio de características, haciendo que el software utilizable para algunos usuarios más que a otros, en algunos casos, no utilizable en absoluto, tal es el caso para los usuarios con discapacidades físicas o mentales, o con o sin los conocimientos necesarios para utilizar la aplicación de software. El problema se agrava cuando los usuarios tienen diferentes niveles de experiencia en el uso de sistemas informáticos, la cultura y las características demográficas.

Aparte del problema de considerar las características de los usuarios y las tareas que realizan, la integración de estos aspectos a la arquitectura aumenta su complexidad, afectando de alguna manera algunos de los atributos de calidad del sistema, como es la mantenibilidad, enfoque principal de esta tesis.

II.9 La Arquitectura de Software y la Usabilidad

Previamente, la usabilidad se asumía que solamente tenía que ver con la presentación gráfica de un sistema, por lo tanto simplemente se mantenía separado del procesado de información del sistema para su fácil modificación y mantenimiento. Esa práctica, aunque necesaria, no es suficiente para lograr un sistema altamente usable (Bass *et al*, 2001). Como el ejemplo de la arquitectura Modelo-Vista-Controlador (Figura 7).

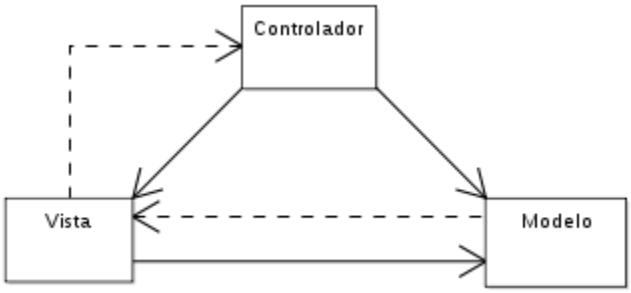


Figura 7. Modelo Vista-Controlador

Donde el modelo es donde se encuentra la funcionalidad; la Vista es la interfaz donde se visualiza la información; y el controlador donde se encarga de establecer la forma en que los usuarios interactúan con el sistema.

La tendencia actual es hacer el análisis de la usabilidad desde la fase de requerimientos para su debido análisis e implementación en la arquitectura debido a los grandes impactos arquitecturales, tiempo necesario para hacer los cambios y costos que puede tener hacer la modificación.

Como menciona Bass *et al* en (2001), la idea es obtener mejor usabilidad a través de decisiones de diseño ejemplificadas en la arquitectura de software. Esas decisiones son las más difíciles de cambiar conforme se avanza en el ciclo de vida de desarrollo del software. Por lo tanto es importante conocer las relaciones entre la arquitectura del software para que el sistema pueda llegar a las metas de usabilidad establecidas. De igual forma los ingenieros de software deben de determinar los aspectos de usabilidad dentro del contexto de la arquitectura

e implementarlos dentro de otros factores limitantes, como son otros atributos de calidad que pueden ser afectados.

El obtener un mejor nivel de usabilidad a través de la arquitectura del software no es algo nuevo. Como menciona Bass *et al* (2001), durante muchos años se han intentado implementar técnicas de desarrollo necesarias para la implementación de la usabilidad, las cuales se han enfocado en seleccionar la arquitectura general del sistema correcta o más óptima mientras soportando eficiencia y mantenibilidad junto con otra funcionalidad.

No solo es cuestión de tomar en cuenta los requerimientos de usabilidad desde fases tempranas del diseño, sino también de escoger un patrón arquitectural adecuado para los atributos que se quieren lograr. A continuación se muestra el resultado de un análisis de varios patrones arquitecturales tomado de Harrison *et al* (2007) para determinar cuál atributo de calidad satisfacen mejor (Tabla 3):

Tabla 3. Relación de algunos estilos arquitecturales y algunos atributos de calidad.

	Usabilidad	Seguridad	Mantenibilidad	Eficiencia
Capas	*Neutral*	*Fortaleza clave* Soporta capas de acceso	*Fortaleza clave* Pruebas y modificaciones a capas hechas por separado y soporta reusabilidad	*Debilidad* Propagación de llamadas a través de las capas puede ser inneficiente
Pipas y filtros	*Debilidad* Generalmente no interactivo	*Debilidad* Cada filtro necesita su propia seguridad	*Fortaleza* Se pueden agregar o modificar filtros por separado	*Fortaleza* Si uno puede usar procesamiento en paralelo *Debilidad* Tiempo y espacio para copiar datos
Blackboard	*Neutral*	*Debilidad* Agentes indispensables pueden ser vulnerables	*Fortaleza clave* Funcionalidad extendible *Debilidad clave* Difícil de probar	*Debilidad* Difícil de soportar paralelismo
Modelo Vista-Controlador	*Fortaleza clave* Múltiples vistas sincronizadas	*Neutral*	*Debilidad* Acoplamiento alto entre las vistas y los controladores al modelo	*Debilidad* Ineficiencia de acceso de datos en la vista
Microkernel	*Neutral*	*Neutral*	*Fortaleza clave* Muy flexible, extensible	*Debilida clave* Mucho gasto de recursos
Reflección	*Neutral*	*Neutral*	*Fortaleza clave* No modificación explicita al código fuente	*Debilidad* Protocolos de meta-objetos son generalmente ineficientes

Como se puede observar cada estilo o patrón arquitectural favorece a ciertos atributos de calidad sobre otros debido a sus características (forma de distribución de sus elementos, organización, etc.). En nuestro caso en particular que estamos buscando encontrar una forma de integrar elementos de usabilidad conservando niveles altos de mantenibilidad, el estilo por capas resulta ser el más óptimo, ya que por su naturaleza se es alta la mantenibilidad siendo neutra hacia la usabilidad.

Existen varios escenarios en donde los requerimientos de usabilidad tienen un profundo impacto arquitectural en el diseño del software, por eso el énfasis de tener en consideración la usabilidad desde la etapa de requerimientos y análisis para hacer las decisiones apropiadas para su incorporación. A continuación se presenta una lista de escenarios de usabilidad con impacto arquitectural, extraído de Bass *et al* (2001):

- **Agregación de datos**.

  El usuario puede querer hacer una o más acciones sobre un mismo objeto.

- **Agregación de comandos.**

  Un usuario puede querer completar un proceso tardado, de pasos múltiples de una sola vez. Por lo tanto debe de poder de ingresar toda la información necesaria para el proceso desde del inicio.

- **Cancelación de comandos.**

  El usuario puede querer que una operación se lleve a cabo y debe de poder detenerla en vez de esperar a que sea completada, sin importar por qué la operación fue iniciada.

- **Usar aplicaciones de manera concurrente.**

  El usuario puede querer usar diferentes aplicaciones al mismo tiempo. Se tiene que asegurar que el software no interfiera con otras aplicaciones para su correcto funcionamiento.

- **Revisar por correctitud.**

  El usuario puede cometer errores que no se da cuenta. El error humano puede ser previsto por la estructura del sistema de tal manera que pueda ser corregido ya sea por el usuario o de manera automática por el sistema.

- **Manteniendo Independencia del dispositivo.**

  Si el usuario instala un nuevo dispositivo, el dispositivo puede interferir con otros ya presentes en el sistema.

- **Evaluando el sistema.**

  El diseñador o un administrador puede que no pueda probar el sistema por alguno de sus atributos de una manera sistemática.

- **Recuperarse de errores.**

  El sistema puede de manera repentina dejar de funcionar mientras el usuario esta trabajando. En caso de un error, el sistema debe de proveer al usuario de maneras para reducir la cantidad de trabajo perdido por fallas del sistema.

- **Recobrar contraseñas.**

  Un usuario puede olvidar su contraseña, por lo tanto el sistema debe de proveer de maneras para que los usuarios puedan acceder de manera segura el sistema.

- **Proveer buena ayuda.**

  Cuando el usuario ocupa ayuda, puede ser que los procedimientos del sistema para proveer la ayuda no se adapten adecuadamente al contexto.

- **Reusando información.**

  El usuario puede querer mover información de una parte del sistema a otro sin tener que volverla a ingresar manualmente.

- **Soportando el uso internacional.**

  El usuario puede querer configurar una aplicación a comunicarse en su propio idioma o normas culturales.

- **Utilizar el conocimiento humano.**

  La gente usa lo que ya saben cuándo se encuentran en nuevas situaciones. Este tipo de situaciones puede incluir el uso de nuevas aplicaciones en una plataforma familiar, una nueva versión de una aplicación familiar, o un nuevo producto en una línea de productos establecida. Nuevos enfoques suelen traer una nueva funcionalidad, sin embargo, si los usuarios no pueden de aplicar lo que ya saben, un costo correspondiente en el tiempo de la productividad y entrenamiento son necesarios.

- **Modificando interfaces.**

  El Diseño iterativo es el elemento vital de la práctica actual de desarrollo de software, sin embargo, un desarrollador de sistemas le puede resultar difícil cambiar la interfaz de usuario de una aplicación para reflejar nuevas funciones y / o deseos nuevas presentaciones. Los diseñadores de sistemas deben asegurarse de que sus interfaces pueden ser fácilmente modificadas.

- **Soportando múltiples actividades.**

  Los usuarios por lo regular necesitan hacer múltiples actividades de manera simultánea, por lo tanto, un sistema o sus aplicaciones deben de permitir al usuario poder cambiar rápidamente entre esas actividades.

- **Navegando dentro de una sola vista.**

  Un usuario puede que quiera navegar de información actualmente desplegada en la pantalla a información no desplegada en ese momento. El sistema debe de poder permitir al usuario navegar entre distinta información de manera rápida y sencilla.

- **Observando el estado del sistema.**

  Puede ser que el usuario no sea presentado con la información del estado actual del sistema para poder operar el sistema, o presentado de manera incorrecta poco clara. El diseñador debe de tomar en cuenta las necesidades y cualidades humanas al decidir que aspectos del sistema debe de presentar y cómo presentarlas al usuario.

- **Trabajando a la velocidad del usuario.**

  El sistema puede ser que no se ajuste adecuadamente a la velocidad del usuario de hacer alguna tarea, haciendo que el usuario se desespere al emplearse sonidos o despliegues de información demasiado rápidos que el usuario no alcanza a leer, etc.

- **Predecir la duración de las tareas.**

  El usuario puede querer hacer alguna otra tarea mientras se completa alguna operación que toma tiempo en el sistema. Por ello el sistema debería de informar al usuario de cuánto tiempo aproximadamente tomaría para completar dicha operación.

- **Soportar búsquedas.**

  El usuario puede querer buscar archivos o algún aspecto e información contenido en el archivo. Los sistemas deberían de tener la función de buscar información de una manera sencilla y consistente.

- **Soportar la función de deshacer.**

  Si el usuario hace alguna operación, y se equivoca o ya no quiere el efecto de esa operación, debe de poder regresar al estado anterior del sistema antes de haber hecho dicha operación.

- **Trabajando en un contexto desconocido.**

  A veces el usuario necesita trabajar en un problema en un entorno diferente. Puede existir discrepancias entre este nuevo contexto o entorno y al que estaba acostumbrado, afectando su desempeño. El sistema debe de proveer una interfaz para inexpertos para guiar a los usuarios operando en un contexto no conocido.

- **Verificando recursos**.
  Una aplicación puede que no verifique si están disponibles los recursos necesarios para poder llevar a cabo alguna operación.

- **Operar con consistencia a través de distintas vistas.**
  El usuario puede confundirse por cambios de las funcionalidades entre distintas vistas. Comandos que estaban disponibles en una vista pueden no estarlo en otras y requieran otra manera de accesarlas.

- **Tener las vistas accesibles.**
  El sistema debe de proveer diferentes formas de ver la misma información.

Como se puede ver, diversos elementos de usabilidad pueden tener un gran impacto en el diseño de la arquitectura, por lo que es necesario tomar en consideración desde las etapas de diseño. Sumándole a esto el considerar las características del usuario y hacer las necesarias modificaciones a los patrones de usabilidad, le puede añadir mucha complejidad a la arquitectura solamente considerando el atributo de usabilidad, descuidando los demás atributos de calidad.

II.2. Métricas de Software

Para saber si se ha logrado las metas con respecto a los atributos de calidad que se necesitan del software que se está desarrollando es necesario contar con una forma de medirlas de forma cuantitativa.

Esencialmente, una métrica de software es una medida de una propiedad del software o del proceso del desarrollo del software (Mills, 1998; Stavrinoudis *et al*, 1999). Estas mediciones se pueden aplicar durante el proceso de software con el propósito de mejorarlo de manera continua y ayudar durante las tomas de decisiones en diferentes etapas del proceso de desarrollo.

II.2.1. Clasificación de las métricas de software

Las métricas de software se pueden clasificar en dos tipos (Mills, 1998):

- **Métricas de producto.** Mediciones de alguna propiedad del producto de software en cualquier etapa de su desarrollo. Pueden medir cualidades del código del producto, como la complejidad y el tamaño, así como también la documentación del producto.
- **Métricas de proceso.** Son mediciones del proceso de desarrollo del software, como el tiempo de desarrollo, tipo de metodología empleada, etc.

También existe otra forma en que pueden categorizarse las métricas (Mills, 1998) como métricas primitivas o métricas computadas. Las métricas primitivas son las que se pueden observar directamente, como es el tamaño del programa (Líneas de Código o LOC), número de defectos por unidad de código, entre otros. Métricas computadas son las que no pueden ser directamente observadas y son derivadas a partir de otras métricas. Un ejemplo son las métricas usadas para medir la productividad, como las LOC producidas por persona-mes (LOC/persona-mes), o para medir la calidad del producto, como el número de defectos por cada mil líneas de código (defectos/KLOC).

II.2.1. Ejemplos de métricas

Existen métricas para diversos tipos de mediciones, a continuación se presentan algunas de las métricas más comúnmente utilizadas:

II.2.2.2.1. Volumen de Halelstad

Una de las métricas propuestas por Halstead (1977) para intentar establecer como una ciencia empírica el desarrollo de software. El Volumen da una idea de la cantidad de código escrito.

- Vocabulario
  - $(n) = n1 + n2$ (1)
- Largo del programa
  - $(N) = N1 + N2$ (2)
- Volumen
  - $(V) = N \log_2 n$ (3)

Donde:

- n1= Número de operadores distintos.
- n2 = Número de operandos distintos.
- N1 = Número total de operadores.
- N2 = Número total de operandos.

II.2.2.2.2. Complejidad Ciclomática

Fue propuesta por Thomas J. McCabe (1976) para determinar el número de caminos independientes que un programa puede tomar al ser ejecutado. Se basa en que la complejidad de un programa aumenta de manera proporcional a la complejidad de su árbol de decisiones, en otras palabras, entre más caminos pueda tomar debido a condiciones, mayor es la cantidad de pruebas necesarias.

Para calcular la complejidad ciclomática se representa el código en forma de grafo, donde cada sentencia de programa representa un nodo. La complejidad ciclomática se representa de la siguiente manera:

$$v(G) = e - n + p \qquad (3)$$

Donde:

- e es el número de aristas del grafo.
- p es el número de nodos correspondientes a sentencias del código.
- p es el número de componentes conectados.

II.2.2.2.3. Líneas de Código (LOC)

LOC es una de las métricas más utilizadas para medir el tamaño de un programa. Pero existen diferentes versiones de lo que constituye una línea de código de un programa en particular. Esas diferencias tratan sobre cómo se manejan las líneas en blanco, los comentarios, comandos no ejecutables, múltiples declaraciones por línea de código, y múltiples líneas por declaración, así como también el cómo contra líneas de código reusadas.

La definición más simple cuenta cualquier línea que no esté en blanco o sea un comentario sin importar el número de declaraciones por línea (Mills *et al*, 1998).

II.2.2.2.4. Índice de mantenibilidad (MI)

Propuestas por Paul W. Oman, *et al* (1992), define una serie de funciones en base a diferentes métricas de complejidad: Complejidad ciclométrica (CC), volumen de Halstead (VH) y LOC.

La ecuación del índice de mantenibilidad (MI) es la siguiente:

$$MI = 171 - 3.42 * \ln(VH) - 0.23(CC) - 16.2 * \ln(LOC) \qquad (4)$$

Nos da como resultado un valor entre 0 y 100, donde un valor menor de 65 representa una mantenibilidad pobre; un valor de entre 65 y 85 una mantenibilidad decente y de 85 en adelante una mantenibilidad excelente.

III

Propuesta de Agregación del Modelo Integral del Usuario a la Arquitectura de Software

III. PROPUESTA DE AGREGACIÓN DEL MODELO INTEGRAL DEL USUARIO A LA ARQUITECTURA DEL SOFTWARE

III.1. Introducción al Modelo Integral del Usuario

Como fue mencionado en el capítulo anterior, para poder lograr un sistema usable, es importante conocer qué tipo de usuario va a usar el sistema. Esto significa conocer en profundidad las características del usuario con el fin de visualizar su perfil. Para ello utilizamos la técnica de análisis de usuario con el fin de determinar dichas características.

En base a esto, hemos propuesto un modelo de usuario (Mejía *et al*, 2012) (mostrado en la Figura 8) basándose en el modelado de las características generales del usuario, que se integran a partir de trabajos anteriores (Zhang *et al*, 2004; Biswas *et al*, 2005; Johnson *et al*, 2005; Weinschenk, 2011; Stuart *et al*, 1983):

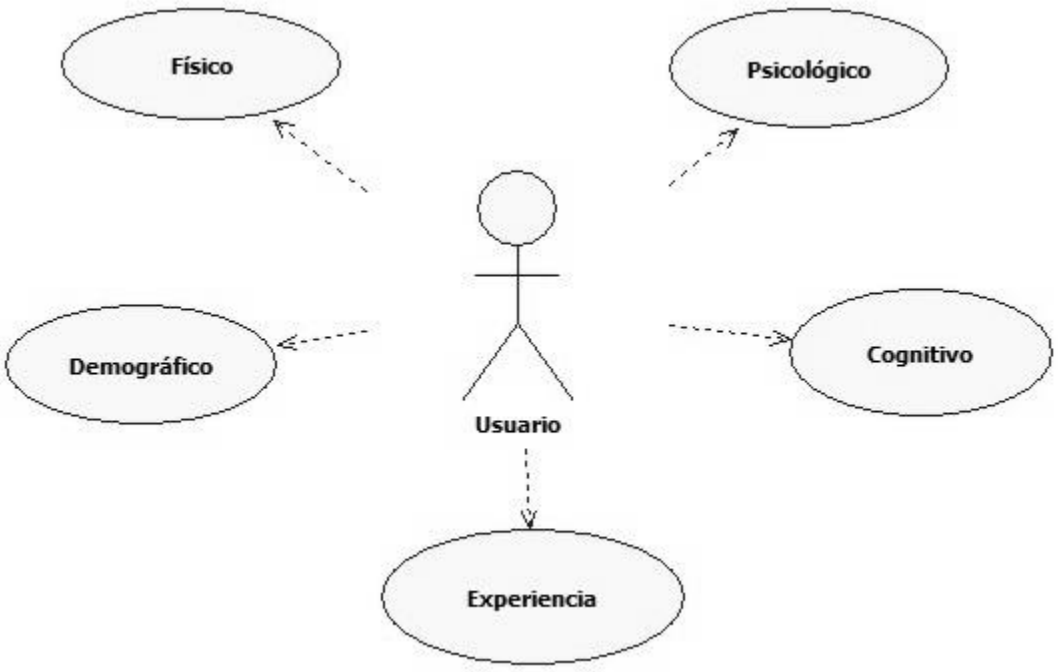


Figura 8. Factores que integran al modelo integral del usuario.

III.1.1. Físico

El aspecto físico del usuario implica las partes del cuerpo, que son utilizados por el usuario para interactuar con el sistema. Algunas partes del cuerpo se toman en consideración con su sentido correspondiente en trabajos anteriores (Folmer *et al*, 2004; Bass *et al*, 2003; Seffah *et al*, 2008).

Como puede verse en la Figura 9, se modelaron algunas de las partes del cuerpo del usuario y algunas de sus características y sus sentidos correspondientes. Para modelar las distintas clases de las partes del cuerpo, tomamos como atributos tales como las partes que el usuario tiene (Presencia), si puede moverlos (Movimiento), y si el usuario tiene el control de las mismas (control). Dado que las partes del cuerpo, como las manos, que son la forma más común de interactuar con cualquier sistema de software, tiene que existir (Presencia), tiene que ser capaz de moverse (Movimiento) y tiene que poder ser controlado por el usuario (Control) con el fin de interactuar con el sistema de una manera eficaz. Todas las partes del cuerpo comparten estos atributos básicos.

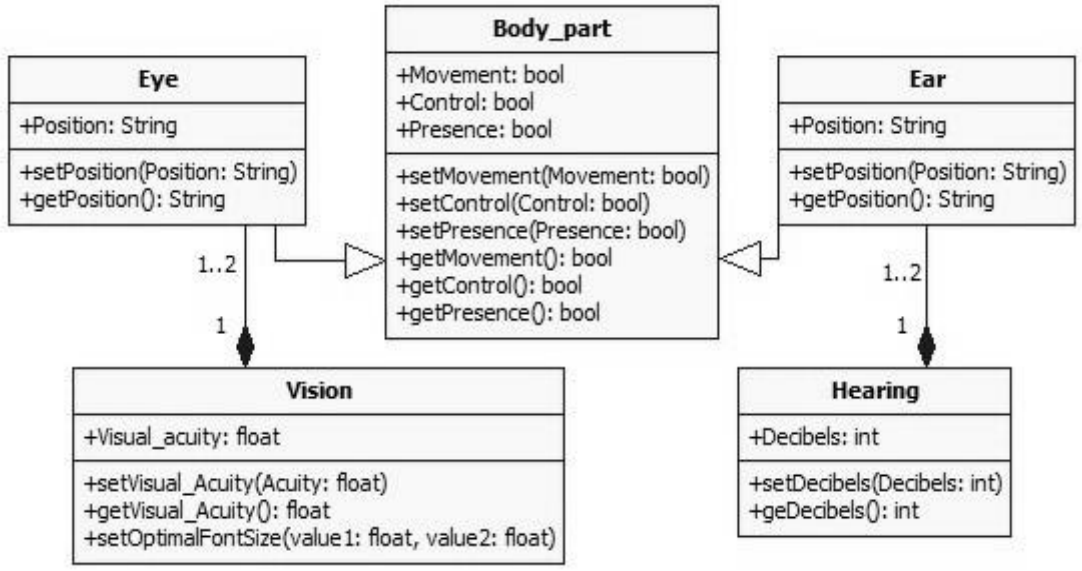


Figura 9. Aspectos físicos tomados en cuenta.

Hemos modelado el cuerpo del usuario de esta manera para que, como nuestros avances en la investigación con la ayuda de especialistas, podemos añadir alguna funcionalidad a cada clase que puede ayudar al usuario a tener una mejor experiencia con el sistema, como se puede ver en el ejemplo siguiente clase.

La clase Visión puede calcular el tamaño de fuente óptima basada en la calificación del usuario de la agudeza visual (que se representa como un valor de tipo de flotador en la clase Visión). La agudeza visual es la claridad de la visión y es la medida que los oftalmólogos y uso oculista a determinar ampliación del objetivo de gafas. La agudeza visual se determina por el resultado de la división de la distancia en metros entre la persona y la tabla de Snellen (la gráfica de letras de distintos tamaños) dividido por el tamaño de las letras que la persona está tratando de leer, de modo que si una persona tiene una calificación de agudeza visual de 20/20 (visión perfecta), el resultado representado en decimales es de 1.

En nuestra implementación, el tamaño de la fuente se calcula en el método setOptimalFontSize () de la clase Vision. Se utilizó como puntos de referencia los tamaños de fuente óptima: tamaño 12 para una persona con una agudeza visual de 20/20, o en decimal 1,0, y el tamaño 140 para una persona con una agudeza visual de 20/60 o 0,33 (visión débil). Utilizando los datos visuales del usuario agudeza lo interpolados con los puntos de referencia para obtener una aproximación al tamaño de fuente más apropiada para ese usuario en particular. Tenga en cuenta que esta no es la versión final del proceso, sino que lo hizo como una forma de probar el modelo y obtener un tamaño de fuente óptima. Seguiremos consultando especialistas para que podamos obtener mejores puntos de referencia y los datos para mejorar el procedimiento para obtener el tamaño ideal de la fuente.

III.1.2. Cognitivo

El sistema cognitivo tiene en cuenta las habilidades y los rasgos de inteligencia, como la capacidad de memoria y la atención, que puede afectar a uno de los muchos atributos de usabilidad (véase Figura 10).

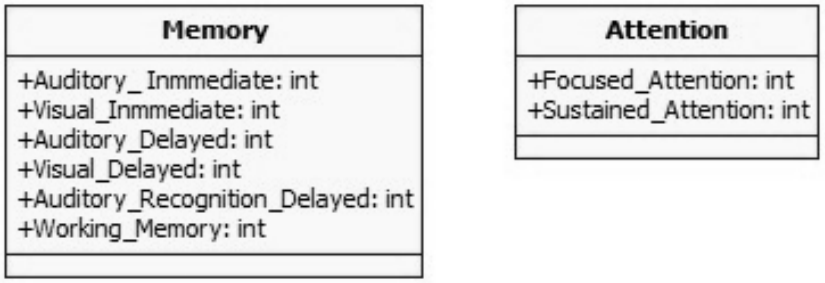


Figura 10. Aspectos cognitivos tomados en consideración.

III.1.3. Demografía

Se refiere a las características estadísticas de una población, o en este caso el usuario, como la edad, el género, la raza, la educación, la religión y otros (Figura 11). Algunas de estas características pueden tener un impacto notable en la usabilidad como la edad, ya que el usuario vaya creciendo, los problemas de usabilidad tienden a aumentar.

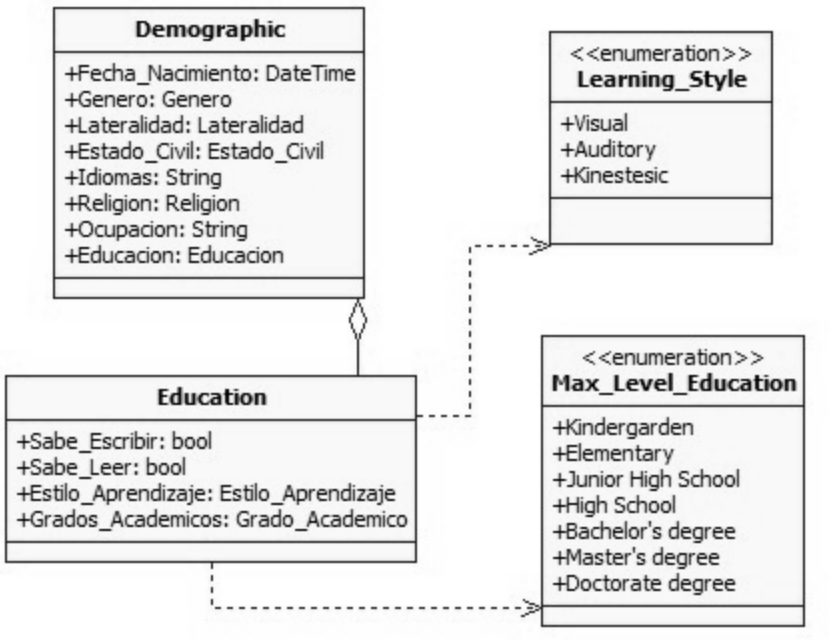


Figura 11. Aspectos demográficos tomados en consideración.

III.1.4. Experiencia

Este aspecto se refiere a la experiencia del usuario con el ordenador o dispositivos similares, experiencia de trabajo, el conocimiento del dominio de aplicación, paquetes de software utilizados, etc. pueden tener un efecto sobre la capacidad de uso debido a la familiaridad del usuario con el sistema, lo que mejora la capacidad de aprendizaje, y la eficiencia, así como el recordar la interfaz de usuario (Figura 12).

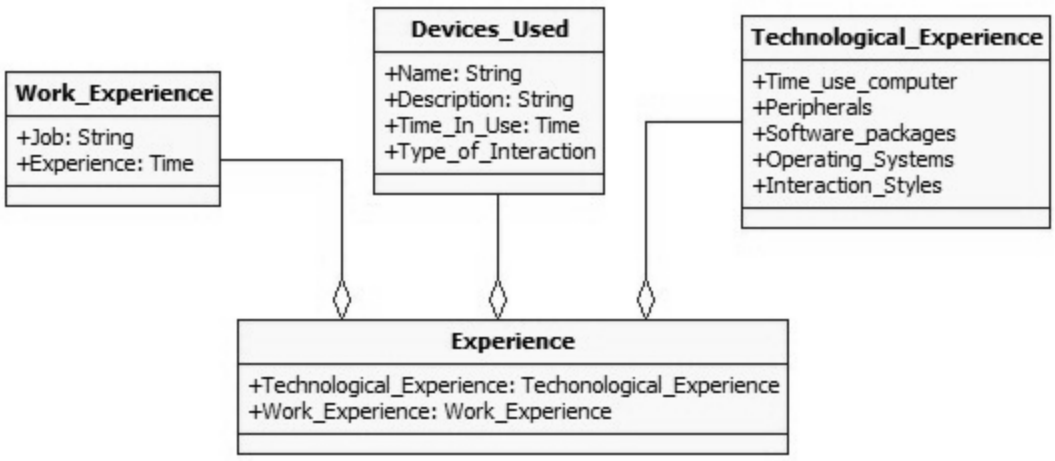


Figura 12. Aspectos de la experiencia tomados en consideración.

III.1.5. Psicológico

Este aspecto se refiere a tener en cuenta los comportamientos de los usuarios que puedan afectar a los atributos de usabilidad del software. La motivación es un ejemplo, si el usuario no está motivado para usar o aprender el software, que será difícil de usar (Figura 13).

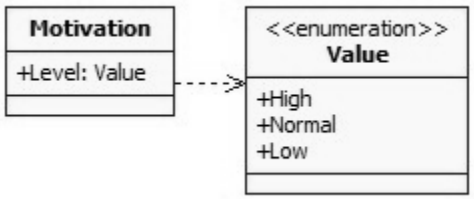


Figura 13. Aspectos psicológicos.

Teniendo en cuenta las características de los usuarios antes mencionados podemos cubrir la mayor parte de los aspectos que podrían tener un efecto sobre la forma en que el usuario interactúa con el sistema, por lo que es posible que las tenga en cuenta para el desarrollo de la interfaz de usuario, teniendo como resultado una mejor experiencia de usuario en general.

III.2.2. Métricas de software y los atributos de mantenibilidad y flexibilidad

La mantenibilidad, como fue mencionado en el capítulo II, es el grado en que el software puede ser entendido, corregido, adaptado o mejorado; y la flexibilidad es el grado en que se puede modificar el software (McCall, 1977; Boehm, 1978). Ambos atributos están estrechamente relacionados ya que les afectan esencialmente los mismos factores (McCall, 1977; Boehm, 1978).

Aunque la mantenibilidad es uno de los factores que afecta de manera indirecta a los usuarios, se estima que el 75% del presupuesto para el desarrollo de un software es gastado en el mantenimiento y modificaciones, por lo que resulta ser la fase más costosa del ciclo de vida del desarrollo del software. De igual forma, al haber un cambio en algún requerimiento, el costo del impacto puede ser más de 10 veces de lo que hubiera costado si el cambio se hubiera hecho durante la fase de diseño (Stavrinoudis *et al*, 1999).

Como resultado, se deben de emplear varias formas para medir la facilidad del mantenimiento y flexibilidad del sistema para poder reducir costos y determinar si el mantenimiento y soporte de ciertos sistemas es viable o no.

La mantenibilidad y la flexibilidad son factores difíciles de cuantificar, pero pueden ser medidos de manera indirecta considerando la estructura del diseño y algunas métricas de software. La complejidad lógica, así como la estructura del programa tienen una fuerte relación sobre el atributo de mantenibilidad (Kafura *et al*, 1987; Rombach, 1987), por lo tanto para poder medir la mantenibilidad se utilizan diversas métricas que tiene que ver con la complejidad del código y la estructura del software.

III.2.2.1. Métricas del diseño

Son las métricas relacionadas con la estructura del software, en este caso dos de las principales cualidades de una arquitectura de software, el acoplamiento y la cohesión, las cuales forman la base para cualquier arquitectura de software.

III.2.2.1.1. De Acoplamiento o dependencia

Como se mencionó en el capítulo III, se basa en el principio de modularidad, es el grado en que un módulo del software depende de otros módulos.

Las métricas más usadas para medir el acoplamiento entre módulo o elementos del software son las propuestas por Robert Cecil Martin en (Cecil, 2003):

- **Acoplamiento aferente (Ca):** Número de otros paquetes dependientes de las clases de un paquete.
- **Acoplamiento eferente (Ce):** Número de otros paquetes de los cuales las clases de un paquete dependen.
- **Instabilidad:**
  - $I = Ce/(Ce + Ca)$ (5)

    Esta métrica es un indicador de la resistencia al cambio de un módulo o paquete. Tiene un rango de 1 a 0, donde 0 indica un paquete completamente estable, y 1 representa un paquete inestable.

III.2.2.1.2. Métricas de cohesión

Se refiere a la medida de que tan relacionados son las responsabilidades de un módulo. Existe una variedad de métricas para medir la cohesión, en los actuales diseños orientados en objetos se utilizan las propuestas Chidamber y Kerember (Chidamber *et al*, 1994) llamada Lack of Cohesión Metric (LCOM) o Métrica de Falta de Cohesión, pero en particular una versión modificada de su métrica por Henderson-Sellers (1996) (LCOM2), presentada a continuación:

$$LCOM2 = \frac{1 - sum(mA)}{(m*a)} \qquad (6)$$

Donde:

m = # de métodos de la clase.

a = # de atributos de la clase.

ma= # de métodos con acceso al atributo a.

sum(ma) = Suma de todas las ma por todos los atributos de la clase.

III.3 Integración del Modelo Integral del Usuario a la Arquitectura del Software

El modelo para la integración del modelo de usuario se basa en primera instancia en el principio de modularidad, explicado en el capítulo III, que se basa en el agrupar elementos del software lógicamente relacionados, algunos de los elementos son: procedimientos, variables, atributos de objetos, entre otros y en la separación de preocupaciones. Con esto se está

tomando en cuenta uno de los principales atributos de la arquitectura que afecta a la mantenibilidad, como fue mencionado en el capítulo II.

De igual forma es de suma importancia la selección de un patrón o estilo arquitectónico que permita la fácil integración del módulo del modelo de usuario, permitiendo un mejor nivel de usabilidad, así como también un buen nivel de mantenibilidad.

En base a la investigación de Harrison *et al* (2007) mencionado en el capítulo V, donde se establece la relación de los atributos de calidad del software con los estilos o patrones arquitecturales, se selecciona un estilo para tomarlo como base para la integración del módulo del modelo del usuario.

El estilo seleccionado fue el de capas, por ser el que tiene las mejores características combinadas de usabilidad y mantenibilidad, que es lo que se busca, el tener el nivel más alto de usabilidad, así como el de mantenibilidad posible.

Con estos puntos en mente, el siguiente modelo es propuesto para la integración del modelo del usuario en la arquitectura de un software (Figura 14):

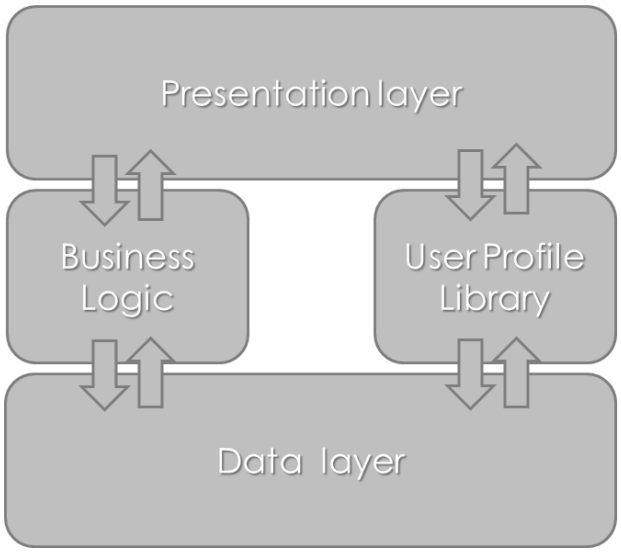


Figura 14. Modelo propuesto integrando el modelo integral del usuario.

Como se puede observar el modelo integral del usuario es integrado entre la capa de presentación y la de datos. Obtiene la información del usuario de la capa de datos, la procesa en sus clases internas, y modifica las clases o formas correspondientes de la interfaz, adaptándola hasta cierto grado conforme a las características del usuario.

III

Trabajos relacionados

III. TRABAJOS RELACIONADOS

La mayoría de los estudios se han hecho con el fin de incluir y evaluar la facilidad de uso durante el proceso de desarrollo de software. Fundamentos y cuestiones prácticas han sido tratados en estos estudios como los siguientes.

Seffah *et al* (2009) remarca la importancia de la usabilidad como un atributo de calidad del software, haciendo hincapié en criterios, factores y parámetros específicos para medir la usabilidad. Asimismo, destaca la importancia y los fundamentos de la ingeniería de usabilidad y su integración en el ciclo de vida de desarrollo. Por otra parte, en Seffah *et al* (2009) proponen el modelado de las relaciones entre los atributos internos del software y los factores de usabilidad externamente visibles. Un enfoque basado en patrones es indicado para tratar con estas relaciones. En términos prácticos, se proponen los componentes de interfaz de usuario para asistir a las necesidades de uso tales como la presentación de información al usuario y que indica el estado de la computación.

Engel y Martin (2009) proponen el uso de un lenguaje de patrones para el modelado estructurado tarea trata de código, que indica los componentes de interfaz de usuario y sus características. También se introducen PaMGIS, un marco que combina el modelo y el desarrollo basado en patrones para generar sistemas interactivos utilizando lenguajes de patrones de HCI para automatizar la generación de la interfaz.

Komogortsev (2009) presenta un modelo que se basa en la noción de que la facilidad de uso es una función inversa a esfuerzo. También postula una métrica objetiva obtenida y deducir de registro de la actividad física y mental y seguimiento de los ojos. Los experimentos apoyan fuertemente la correlación entre el esfuerzo y la facilidad de uso y que los atributos de usabilidad se pueden medir usando como métrica el esfuerzo.

En Zhang *et al* (2004) se describen algunas de las características del usuario que afectan la manera en la que el usuario interactúa con el sistema, como se muestra en la Tabla 1 (columnas primera y segunda). La tercera columna es una descripción de cada factor tratando

de aspectos concretos relacionados con el diseño de la interfaz de usuario (Juárez-Ramírez *et al*, 2011).

Tabla 4. Características del perfil de usuario.

Características	**Factor**	**Descripción**
Información demográfica:	Edad.	Adecuación del tema de la aplicación a la edad.
	Género.	Para el tema de la aplicación.
	Educación.	Conocimiento adquirido. - Habilidades de razonamiento.
	Ocupación.	Actividades hechas. - Habilidades adquiridas o prácticas en el trabajo.
	Antecedentes culturales.	Costumbres o hábitos relacionados con el contenido de la aplicación: el vocabulario, los signos (símbolos).
	Necesidades especiales.	Capacidades y limitaciones para operar el software.
	Conocimiento y entrenamiento sobre computadoras.	Habilidades para operar la computadora o dispositivos similares.
	Experiencia con productos	Conocimiento de la funcionalidad del sistema.

	similares.	- Habilidades para manipular el sistema.
Características de inteligencia:	Estilos cognitivos.	Estilo y habilidades de aprendizaje.
	Caracterísicas afectivas.	Capacidad de gesturas del cuerpo humano.
	Habilidades.	Habilidades poseidas para el aprendizaje.
Factores relacionados a la tarea:	Características del trabajo.	Tipo de tarea, capacidades físicas y cognitivas necesarias.
	Conocimiento del dominio de la aplicación Y características del trabajo.	Pericia en el dominio de la aplicación. - Experiencia en el trabajo.
	Uso de la computadora.	Frecuencia del uso de la computadora en el trabajo.

De manera similar, en Biswas *et al* (2005) proponen un modelo de usuario para tener en cuenta para el desarrollo de software para usuarios con discapacidades, como se ve en la Tabla 5, y también proponer algunos valores posibles.

Tabla 5. Clasificación de las características del usuario

Características de alto nivel	**Características de bajo nivel**	**Valores**
Experiencia	Con software	1. Principiante 2. Intermedio 3. Experto
	Con software similar	1. Principiante 2. Intermedio 3. Experto
Edad	Edad actual	1. Debajo de los 5 2. (6-15) 3. (16-25) 4. (25-40) 5. (40-65) 6. Mayor de 65
Características oculomotoras	Visión	1. Superior 2. Medio 3. Inferior
Sexo	Sexo	1. Masculino 2. Femenino
Nivel de lenguaje	Medio de lenguaje	1. Solo imagenes 2. Una palabra 3. Dos palabras 4. Tres palabras 5. Frases 6. Enunciado 7. Normal

	Lenguaje	1. Ingles 2. Francés 3. Hindú, etc.
Nivel de Educación	Nivel de Educación	1. Debajo de primaria 2. Primaria 3. Secundaria 4. Preparatoria 5. Arriba de preparatoria
Personalidad	Motivación	1. Alta 2. Normal 3. Baja

Como podemos ver, los fundamentos, las métricas, los idiomas y los marcos se han propuesto y la usabilidad del sistema se ha mejorado en un cierto nivel. Además, las propuestas diferentes para las características del usuario que se han propuesto. Sin embargo, es necesario tener en cuenta las características específicas de los usuarios humanos con el fin de tener mejores niveles de usabilidad, que abarca los aspectos más humanos, como físico, motor, cognitivo, psicológico y demográfico. Además de que ninguno hace mención sobre la manera de integrar dichos modelos en la arquitectura del software de manera adecuada, limitando las posibles repercusiones en los otros elementos de la arquitectura, y por ende, el los demás atributos de calidad del software. Hasta donde sabemos no existe una arquitectura de software que integre estos aspectos de usabilidad.

IV

Experimentos realizados

IV. EXPERIMENTOS REALIZADOS

Los experimentos realizados en la presente tesis tienen como objetivo de demostrar que un modelo que garantice la mantenibilidad de un sistema con la integración del modelo integral del usuario es posible, así como determinar el impacto de dicha arquitectura.

De igual forma se pretende evangelizar a los estudiantes mediante un proyecto en colaboración con instituciones de educación para niños con autismo, con el fin de que aprendan a integrar prácticas de ingeniería de usabilidad en el ciclo de vida del desarrollo de software y vean la importancia de la usabilidad para el usuario.

Para poder probar nuestro modelo de usuario propuesto, así como su integración en la arquitectura de un software, hemos implementado una librería con la parte funcional hasta el momento de nuestro modelo, siendo la clase Visión, que puede ser introducida durante el proceso de desarrollo del software para ayudar a mejorar la implementación de la usabilidad.

Llevamos a cabo tres experimentos. En el primer experimento probamos el efecto de la educación del usuario y su experiencia tecnológica en las preferencias de interfaz de usuario. En el segundo experimento probamos el atributo de la vista, calculando el tamaño de la fuente de la letra para diferentes usuarios y se mide el impacto sobre la arquitectura. Y finalmente en el tercero, intentamos crear un cambio en la actitud de los alumnos de licenciatura sobre la usabilidad al involucrarlos en un proyecto con usuarios autistas, para que puedan observar el impacto de dedicarle el tiempo y esfuerzo a la debida implementación de la usabilidad en la arquitectura del software.

IV.1 Desarrollando prototipos de Interfaz para una aplicación de CRM implementando el modelo de usuario

Desarrollamos una aplicación CRM en ASP.NET con C# en el Framework 4.0 y MSSQL, usando una arquitectura de tres capas, donde los usuarios tienen diferentes especialidades y experiencia tecnológica, sus profesiones variaron desde contadores, ingenieros eléctricos, de mercadotecnia, entre otros.

V.1.1 Experimento Uno: Diseño de Interfaz

Usando parte de nuestro modelo, en específico la sección de experiencia y demografía, intentamos diseñar una interfaz de usuario que sea la más apropiada para nuestros usuarios.

Para lograr esto, aplicamos a nuestros usuarios una serie de preguntas que tienen que ver con su nivel de educación (Apéndice B), su puesto de trabajo actual y su experiencia en ese puesto, experiencia usando una PC y las aplicaciones de software, entre otros.

Usando la información adquirida, identificamos aplicaciones usadas en común por nuestros usuarios y determinamos los patrones de usabilidad requeridos en base a esas aplicaciones para poder determinar la mejor manera de presentar la información.

Los resultados de los cuestionarios mostraron que todos tenían mucha experiencia usando Office de Microsoft, en particular Excel. Como resultado diseñamos nuestra interfaz principalmente con tablas, obteniendo así satisfacción del usuario (Figura 15).

Figura 15. Diseño de interfaz obtenida en base a los cuestionarios

IV.1.2 Experimento Dos: Integrando los aspectos físicos del modelo de usuario como una librería.

Los aspectos físicos del usuario que tienen implementación, en este caso el sentido de la Vista en forma de una clase, fueron incorporados en una librería programada en C# y fue añadida al proyecto.

Para poder mostrar cómo fueron incluidas las características del usuario en la arquitectura del software, presentamos un diagrama de paquetes, indicando la posición de la nuestra librería (UserModel en el diagrama)(Figura 16).

El software fue desarrollado siguiendo un modelo tradicional de capas, como se puede apreciar en la figura, tiene su capa de presentación, su capa de lógica de negocios y su capa de datos. Nuestro fragmento del modelo integral del usuario fue agregada a la arquitectura conforme al modelo descrito anteriormente en el capítulo III, entre la capa de presentación y de datos.

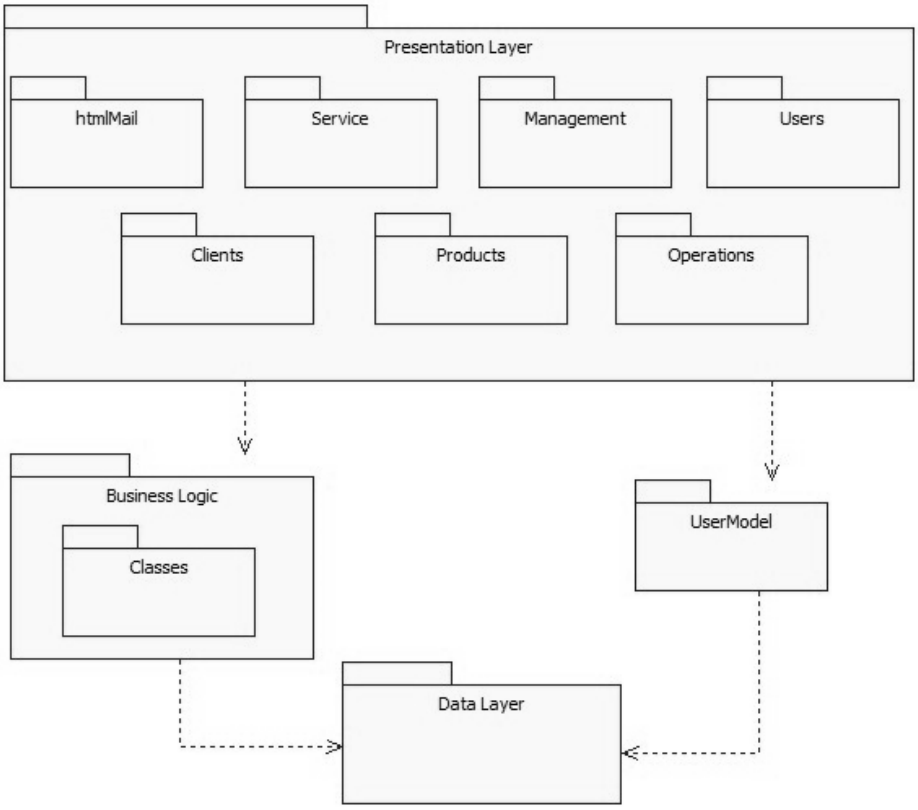


Figura 16. Diagrama de paquetes de la aplicación.

Implementamos las formas de Interfaz de Usuario de tal manera que tomen en consideración la agudeza visual de los usuarios para algunos casos de uso, tomando en cuenta la información de sus cuentas de usuario. En este ejemplo, para cada etiqueta o componente que tiene texto, se adecua el tamaño de la fuente al llamar al método *setOptimalFontSize()* de la librería en vez de usar un valor fijo.

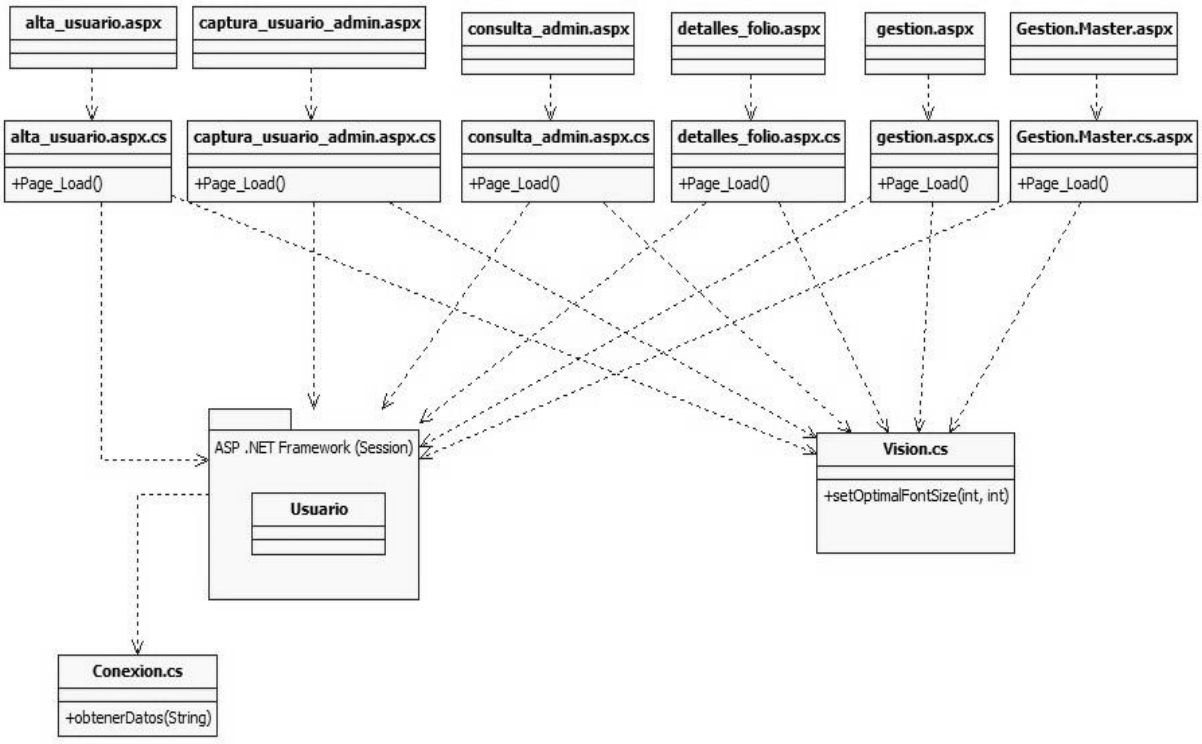


Figura 17. Diagrama parcial de clases de la arquitectura de la aplicación.

En la figura 17, se muestra un esquema de la arquitectura de clases de la aplicación. En la parte superior se encuentras las formas y su clase correspondiente pertenecientes a la capa de presentación. También se ve la clase Usuario, perteneciente a la capa de lógica de negocios; y a la clase Conexión, que corresponde a la capa de datos.

Una vez que el usuario entra al sistema, ingresando su usuario y su contraseña, los datos pertinentes al usuario son obtenidos por medio de la clase Conexión, los cuales son almacenados en una variable de sesión. Acto seguido esos datos son usados en las formas de presentación con la clase Visión, al cual usa los datos de agudeza visual del usuario, para poder calcular el tamaño de fuente antes de que la siguiente página se cargue en el método Page_Load() correspondiente al archivo .aspx.cs correspondiente de la forma (Figura 17).

Como ejemplo, mostramos una de las interfaces o formas (gestión.aspx) de uno de los usuarios con vista excelente 20/20, sin tener ningún problema el mouse y el teclado como dispositivos de entrada. La siguiente interfaz fue el resultado (Figura 18).

Figura 18. Interfaz con visión 20/20.

La forma gestión.aspx es modificada por medio de su clase *gestios.aspx.cs* donde se hace el llamado al método *setOptimalFontsize()* pasándole como parámetros la agudeza visual del usuario, guardada en una variable de sesión, de la clase Visión del modelo de usuario en el método *Page_Load()*, donde se ejecuta código justo antes de cargar la página, obteniendo así una interfaz con el tamaño de letra adecuado a ese usuario en particular

Ahora cambiamos el usuario a uno con agudeza visual de 20/40. La siguiente es la nueva interfaz (Figura 19).

Figura 19. Interfaz con visión 20/40.

De igual forma aunque los resultados soportan la idea de que el modelo ofrece una solución integral posible, puede ser que no sea la solución total a los problemas de usabilidad; puede ser que a un usuario no le guste la interfaz sugerida producida por el modelo, o puede ser que la interfaz no sea la óptima para él. Esperamos poder incluir en el modelo una manera de también poder hacerla adaptable, en donde si el usuario no le gusta la actual interfaz, él pueda hacer modificaciones a ella para mejor adaptarla a sus necesidades.

IV.1.2.1 Midiendo el impacto de la integración del modelo de usuario a la arquitectura del software

Para medir el impacto a la arquitectura del software al introducir la parte del modelo del usuario correspondiente a la vista, primeramente se hicieron mediciones de las métricas de cohesión y acoplamiento eferente y aferente, previamente explicadas en el capítulo III. Paso seguido, se integra el modelo, como es representado en la Figura 16, como se muestra en el experimento dos.

IV.1.2.2 Resultados de la medición del impacto en la arquitectura

Para poder medir el impacto en la arquitectura del software de la integración del modelo del usuario en la interfaz se utilizó una herramienta llamada *NDepend*. Esta clase es un analizador estático de software, o sea que analiza el código de la aplicación sin ser ejecutado.

NDepend calcula una variedad de métricas para diferentes aspectos como es la complejidad, cohesión, acoplamiento, entre otros. Con la herramienta se hizo un cálculo de las métricas necesarias antes y después de integrar el módulo de usuario. Cabe mencionar que el *NDepend* calcula por default las dependencias de los elementos del software contando las llamadas a otros elementos del .Net Framework, por lo que se hizo un conteo detallado para ver la diferencia tomando en cuenta y sin tomar en cuenta el .Net Framework. Las métricas que se tomaron en cuenta son las que fueron mencionadas en el capítulo anterior, ya que son las que afectan de manera directa la flexibilidad y mantenibilidad del sistema.

Los resultados del análisis tomando en cuenta las dependencias del .Net Framework, antes y después de la integración del modelo del usuario son las siguientes (Tabla 6 y 7):

Tabla 6. Resultados del análisis tomando en cuenta el .NET Framework.

Nombre	Ca Antes	Ca Después	Ce Antes	Ce Después	LCOM2 Antes	LCOM2 Después
alta_usuario	0	0	24	29	1.08	1.06
captura_usuario_admin	0	0	36	41	1.08	1.08
consulta_admins	0	0	33	38	0.99	1
Gestion	0	0	16	21	1.5	1.57
gestion	0	0	13	20	2	1.31
Reactivar_Usuario	0	0	35	40	1.04	1.03

Tabla 7. Resultados del análisis tomando sin tomar en cuenta el .NET Framework.

Nombre	Ca Antes	Ca Después	Ce Antes	Ce Después	LCOM2 Antes	LCOM2 Después
alta_usuario	0	0	2	3	1.08	1.06
captura_usuario_admin	0	0	4	5	1.08	1.08
consulta_admins	0	0	1	2	0.99	1
Gestion	0	0	1	2	1.5	1.57
gestion	0	0	0	2	2	1.31
Reactivar_Usuario	0	0	1	2	1.04	1.03

No existe un valor óptimo acordado para los valores de acoplamiento, ya que es dependiendo de la naturaleza del sistema siendo desarrollado, puede ser que algunos componentes necesiten tener un acoplamiento alto mientras que en otros no. *NDepend*, en su documentación[1] estipula que lo más recomendado, tomando en cuenta las dependencias con el framework, es que no pase de 50.

Los resultados confirman que al integrar el modelo en forma de un módulo en la arquitectura entre la capa de presentación y la lógica de negocios de un sistema, afecta de manera mínima a la arquitectura de software, conservando su mantenibilidad ya que no altera los valores de cohesión y aumenta de manera muy mínima los valores de acoplamiento.

1. Tomado de http://www.ndepend.com/metrics.aspx (13 de enero del 2013)

IV.2. Experimento Tres: Desarrollando una aplicación para usuarios autistas.

En el semestre 2012-1 se estableció una colaboración con "Voz del Autismo" y CADEE, una Asociación Civil y una escuela para niños con autismo. CADEE nos visitó para desarrollar aplicaciones de software para ayudar a los procesos de aprendizaje y el tratamiento de los niños con autismo.

CADEE cuenta con un grupo de profesionales en psicología, educación especial y fisioterapia. También las madres de los niños están asistiendo a las actividades de enseñanza y terapia.

Durante los períodos académicos 2012-1 y 2012-2 visitamos CADEE con frecuencia. Nuestros estudiantes de ingeniería visitaron sus instalaciones, dando una buena visión de los grupos de niños con autismo. Los estudiantes de licenciatura analizaron el comportamiento de los niños durante las clases de terapia y la enseñanza. Como resultado de estas visitas, los estudiantes se reunieron los requisitos de especificación para construir aplicaciones de software.

Por otro lado, el personal de CADEE visitó nuestras instalaciones en la universidad, donde los estudiantes llevaron a cabo entrevistas para obtener la información necesaria para definir el modelo de usuario autista. La interacción de CADEE contribuyó en gran medida en la instrucción de los estudiantes en temas tales como la recolección de las necesidades reales de los usuarios. El modelo de usuario autista obtenido por los alumnos de licenciatura es el siguiente (Tabla 8):

Tabla 8. Modelo de usuario autista.

Atributo	Valor
Lenguaje	Si; No
Respuesta a sonidos	Si; No
Contacto visual	Si; No
Identidad	Si; No
Interés en el ambiente	Si; No
Entiende instrucciones	Si; No
Hace decisiones	Si; No
Tolerancia a sonidos	Alto; Medio; Bajo
Actividad	Alto; Normal; Pasivo
Tolerancia al cambio	Si; No
Temperamento	Si; No
Estilo de aprendizaje	Visual; Kinestésico

En nuestra colaboración con las organizaciones de autismo, CADEE nos pidió una aplicación de software móvil con la esperanza de ayudar a sus niños a desarrollar habilidades sociales y de comunicación ya que la terapia toma mucho tiempo, teniendo en cuenta que las nuevas tecnologías, tales como tabletas PC, parece que logran un gran avance en el desarrollo de esas habilidades, ya que permiten que los niños se comuniquen con poco tiempo de uso de esos dispositivos.

Habían utilizado un programa llamado Proloquo2Go (Figura 20) para el iPad, que tiene un gran éxito en lograr a que los niños se comuniquen. Proloquo2Go es una aplicación de software para la comunicación, fácil de usar, que permite al usuario construir frases completas mediante la selección de diferentes elementos clasificados en categorías diferentes, todas con ilustraciones asociados con el elemento o categoría. Asimismo, el software es capaz de leer la frase y permitir que el usuario del habla para comunicar sus deseos.

Figura 20. Proloquo2Go

Nuestros estudiantes se dieron a la tarea de desarrollar su propio software de comunicación similar en español para el sistema operativo Android tratando de lograr un software más usable en general, mediante la aplicación de un proceso de diseño centrado en el usuario y prácticas de la ingeniería de usabilidad.

Para que nuestros alumnos desarrollen de forma adecuada nuestra versión del software, separamos las tareas necesarias para el análisis, diseño y desarrollo del software según el curso que estaban tomando en ese momento.

IV.1.2 Organizando el desarrollo de la aplicación

Los alumnos del curso de Ingeniería de Requerimientos (IR) fueron los encargados de llevar a cabo las técnicas necesarias (entrevistas, observación, casos de uso, entre otros) con el fin de obtener los requisitos, y el desarrollo de casos de uso, haciendo especial énfasis en el uso de técnicas de usabilidad de ingeniería, tales como análisis de tareas y análisis usuario.

En el caso de los alumnos del curso de Diseño de Interacciones (DI), y los que estaban tomando el curso de Análisis y Diseño de Software (ADS), estaban a cargo de: obtener el perfil de usuario (conjuntamente con los estudiantes de IR), el desarrollo de los prototipos basados en los casos de uso obtenidos por los estudiantes de IR, proponiendo diferentes interfaces de usuario para el software y realizar algunas pruebas de usabilidad con los niños para mejorar la interfaz.

El perfil de usuario se obtuvo con la ayuda de los padres, profesores y psicólogos de los niños, dando una visión especial sobre su comportamiento y características a través de entrevistas y visitas a las instalaciones CADEE.

Teniendo el modelo de usuario como guía, los estudiantes fueron capaces de desarrollar mejores casos de uso para la aplicación del sistema.

IV.1.2 Estableciendo la relación entre el prototipo y el modelo de usuario autista

La relación entre las características del usuario y los casos de uso desarrollados son los siguientes (Tabla 9):

Tabla 9. Relación entre las características del usuario y los casos de uso.

Atributo	Caso de uso afectado
Lenguaje	Crear enunciado; Leer enunciado
Respuesta a sonidos	Crear enunciado; Leer enunciado
Contacto visual con otra persona	Ninguno
Identidad	Ninguno
Interes en el ambiente	Crear enunciado; Leer enunciado
Entiende instrucciones	Ninguno
Toma decisiones	Ninguno
Tolerancia a sonidos	Crear enunciado; Leer enunciado
Actividad	Ninguno
Tolerancia al cambio	Crear enunciado; Leer enunciado; Borrar enunciado; Navegar entre grupos y elementos
Temperamento	Ninguno
Estilo de aprendizaje	Crear enunciado; Leer enunciado; Borrar enunciado; Navegar entre grupos y elementos

Como resultado se obtuvo un prototipo del software (Figura 21). En esencia, tiene la misma funcionalidad básica que Proloquo2Go, pero se diferencia en que tiene una interfaz diferente con mejor la navegabilidad por tener todos los elementos necesarios para el usuario con una pantalla con barra de desplazamiento, con cada categoría que se muestra en la pantalla y cada uno con su propio mini barra de desplazamiento mientras en Proloquo2Go hay varias pantallas que contienen los contenidos de las diferentes categorías, lo que obliga al usuario a ir y venir entre varias pantallas para formar una frase. Además del aspecto de navegación del software, los estudiantes también implementaron una estructura gramatical

básica de las oraciones, para que el software pueda corregir el usuario en caso de que ingrese varios elementos en un orden incorrecto.

Figura 21. Interfaz de Prototipo

Con el modelo del usuario autista en mano, los desarrolladores incorporaron elementos de usabilidad en su arquitectura del sistema, como se ve en la Figura 22.

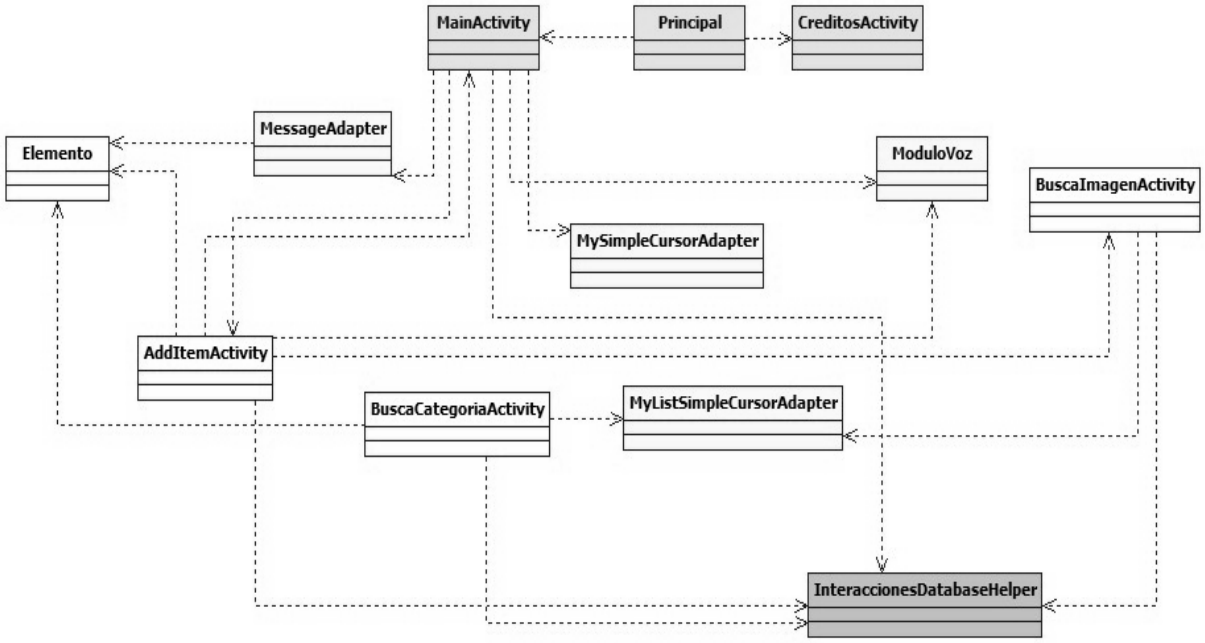


Figura 22. Interfaz de Prototipo

La figura muestra un diagrama de clases de la arquitectura del prototipo. Las clases de *MainActivity, Principal y CreditosActivity* corresponden a la capa de presentación, mientras que la clase *InteraccionesDatabaseHelper* a la capa de datos. El resto de las clases corresponden a la lógica de negocios.

Las clases de *MessageAdapter, MySipleCursorAdapter y ModuloVoz*, aunque estén en la capa de lógica de negocios, son clases que se encargan exclusivamente de cuestiones de usabilidad para las interfaces. Fueron desarrolladas en base al modelo de usuario del autista obtenido a inicios del desarrollo del sistema, de tal forma que integraron elementos de usabilidad en la arquitectura desde las fases iniciales. A continuación se presenta una relación entre las características del modelo del usuario autista y las clases afectadas por ellas (Tabla 10).

Tabla 10. Relación entre las características del usuario y las clases de la arquitectura.

Atributo	Clases afectadas
Lenguaje	MainActivity; Principal;CreditosActivity
Respuesta a sonidos	MainActivity; MessageAdapter; MySimpleCursorAdapter; ModuloVoz.
Contacto visual con otra persona	Ninguno
Identidad	Ninguno
Interes en el ambiente	MainActivity; MessageAdapter; MySimpleCursorAdapter; ModuloVoz.
Entiende instrucciones	Ninguno
Toma decisiones	Ninguno
Tolerancia a sonidos	MainActivity; MessageAdapter; MyAimpleCursorAdapter; ModuloVoz.
Actividad	Ninguno
Tolerancia al cambio	MainActivity; MessageAdapter; MySimpleCursorAdapter; ModuloVoz; MyListCursorAdapter
Temperamento	Ninguno
Estilo de aprendizaje	MainActivity; MessageAdapter; MySimpleCursorAdapter; ModuloVoz; MyListCursorAdapter

IV.1.2 Pruebas de usabilidad del prototipo

El equipo de desarrollo principal compuesto por cinco estudiantes, y se encargaron de llevar a cabo una evaluación de la usabilidad mediante las heurísticas de usabilidad de Nielsen usabilidad (Tabla 10) para ambos Proloquo2Go y el software se desarrolló con el fin de determinar si se logra un sistema global más útil en este prototipo actual. Se aplicó la evaluación heurística de los casos de uso que se implementaron y se compararon los resultados entre Proloquo2Go y el software desarrollado por ellos.

El consenso entre los evaluadores fue que la interfaz de Proloquo2Go todavía tenía mejor usabilidad debido a más elementos de usabilidad, como una guía de ayuda que nuestra versión del software aún carece, y más opciones para personalizar. Las adiciones están siendo consideradas para la siguiente iteración del ciclo de desarrollo. Los resultados heurísticos son los siguientes, como se muestra en las tablas 4 y 5, donde "Sí" se utiliza cuando el software se rige por la heurística y "No" en las que no lo hicieron:

Tabla 11. Heurísticas de Usabilidad de Nielsen.

Etiqueta	Heurística
H1	Visibilidad del estado del sistema
H2	Concordancia entre el sistema y el mundo real
H3	Control y libertad del usuario
H4	Consistencia y estandares
H5	Prevención de errores
H6	Recognición en vez de recuerdo
H7	Flexibilidad y efficiendia de uso
H8	Diseño estético y minimalista
H9	Ayudar a reconocer, diagnosticar y recuperar de errores
H10	Ayuda y documentación

Tabla 12. Evaluación con heurísticas de Proloquo2Go.

Caso de Uso	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
Crear enunciado	Si	No	Si	Si	Si	No	Si	Si	Si	Si
Borrar elemento	Si	No	Si	Si	Si	Si	Si	Si	Si	Si
Leer enunciado	Si	No	Si	Si	Si	No	Si	Si	Si	Si
Agregar elemento	Si	No	Si	Si	Si	Si	Si	Si	Si	Si

Tabla 13. Evaluación por heurísticas del prototipo implementado.

Use case	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
Crear enunciado	No	Si	No	No	No	No	Si	Si	No	No
Borrar elemento	Si	Si	Si	Si	Si	Si	Si	Si	No	No
Leer enunciado	Si	Si	Si	Si	Si	No	Si	Si	No	No
Agregar elemento	No	Si	Si	Si	Si	Si	Si	Si	No	No

Además, actualmente estamos tratando de hacer pruebas de usabilidad con el usuario para determinar si las mejoras en la navegación y la función gramatical ofrecer mayor facilidad de uso general para los niños.

V.1.2 Respuesta de los estudiantes

Después del diseño y desarrollo del primer prototipo, una pequeña encuesta de tres preguntas fue dada a los estudiantes para tratar de determinar si su percepción de la importancia de la HCI y prácticas de usabilidad había cambiado. Una respuesta positiva (Sí), podría significar un cambio de actitud hacia las prácticas de HCI. El cuestionario es como sigue (Tabla 14):

Tabla 14. Cuestionario para los alumnos.

Etiqueta	Pregunta	Posibles respuestas
Q1	Cuando se les encomendó la tarea de desarrollar la aplicación para niños autistas, ¿le hizo despertar su interés?	Si/No
Q2	Durante las visitas a nuestro usuario (niños autistas) y ser testigo de primera mano de sus necesidades, ¿le hizo cambiar su perspectiva sobre la importancia del desarrollo centrado en el usuario de software?	Si/No
Q3	¿Usted se encontró motivado en hacer una contribución duradera al usuario durante el desarrollo del proyecto?	Si/No

V.1.2 Resultados

Un total de 30 estudiantes respondieron a la encuesta. Los resultados fueron como se muestra en la Tabla 14:

Tabla 15. Respuestas de los alumnos.

Pregunta	Respuesta	Número de estudiantes	Porcentaje
Q1	Si	25	83
	No	5	17
Q2	Si	25	83
	No	4	13
Q3	Si	29	97
	No	1	3

Con este proyecto conjunto con alumnos de diferentes cursos centrados en técnicas de Interacción Humano-Computadora, los estudiantes sintieron el impacto de estas prácticas en todas las etapas del ciclo de desarrollo de software logrando diseñar una arquitectura con elementos de usabilidad desde el inicio del desarrollo.

En las últimas experiencias, los estudiantes mostraron profundo interés en la fabricación de software más fácil de usar especialmente cuando la observación directa e interacción con sus usuarios, en este caso los niños autistas, teniendo en cuenta todas las características del usuario a todo lo largo del ciclo de desarrollo, lo que confirma nuestra hipótesis que desarrollando software con enfoque social ayuda a que cambien de actitud hacia prácticas de Interacción Humano-Computadora. Incluso varios estudiantes fueron conmovidos por el enfoque que se le dio al curso y la herramienta que se estaban desarrollando, ya que, como se vio después, varios de los estudiantes tenían un amigo o miembro de la familia con algún grado de autismo.

Este nuevo interés en los estudiantes fue más allá de su actual curso, los profesores de otras asignaturas informaron que los estudiantes estaban aplicando las técnicas que se utilizan en sus cursos sin que el maestro les pidiera o sugiriera que los utilicen.

V

Publicaciones realizadas

V. PUBLICACIONES REALIZADAS

- S. Inzunza, A. Mejia, R. Juarez-Ramirez, M. Gomez-Ruelas. Implementing user-oriented interfaces: From user analysis to framework's components. En *Proceedings of the 2011 International Conference of Uncertainty Reasoning and Knowledge Engineering (URKE),* Bali, Indonesia, IEEE 2011, vol. 1, pp. 107-110.

- A.Mejía, R. Juárez-Ramírez. Modelo para la Conservación de los Atributos de Mantenibilidad y Flexibilidad de la Arquitectura de Software con la Incorporación de Elementos de Usabilidad y el Perfil del Usuario. CONISOFT 2012, Guadalajara, México.

- A. Mejía, R. Juárez-Ramírez, S. Inzunza, R. Valenzuela. Implementing adaptive interfaces: a user model for the development of usability in interactive systems. In *Proceedings of the CUBE International Information Technology Conference*, Pune, India, ACM 2012, pp. 598-604.

- A. Mejía, R. Juárez-Ramírez. Teaching Human-Computer Interaction through developing Applications in Collaboration between Academy and Autism Organizations. A ser publicado en *el International Journal of Engineering Education (IJEE).*

- A. Mejía, R. Juárez-Ramírez, S. Inzunza, R. Valenzuela. Implementing adaptive interfaces: a user model for the development of usability in interactive systems (Versión extendida). Esperando decisión si es publicado en el *International Journal of Computer Systems Science and Engineering (IJCSSE).*

VI

Conclusiones

VI. CONCLUSIONES Y TRABAJO FUTURO

En esta tesis se presenta una propuesta inicial del modelo integral de usuario, el cual considera la gran mayoría de las características del usuario. Hemos descrito partes del modelo usando diagramas UML y se expusieron dos ejemplos de cómo el modelo puede usarse.

Podemos concluir que un modelo integral del usuario es posible de implementar, y que puede ser fácilmente integrado al software considerando que se integre de preferencia desde las fases iniciales del proyecto, al integrarlo de la manera que se propuso, ayudando en la mejora en general de los niveles de usabilidad, al mismo tiempo que conservando la mantenibilidad del software.

Se demostró que se pueden conservar los atributos de calidad de mantenibilidad y flexibilidad de la arquitectura, aún con la integración del modelo, al tomar en consideración ciertas características al diseñar la arquitectura debidamente bajo los principios de modularidad, aplicando un patrón o combinación de patrones arquitecturales óptimos para los atributos de calidad deseados, en este caso la arquitectura por capas, y manteniendo niveles óptimos de cohesión y acoplamiento.

Planeamos en seguir completando el modelo de usuario y reuniendo valores para cada atributo, como es el caso de lo psicológico, cognitivo y experiencia con la ayuda de especialistas como son doctores y psicólogos.

Ya hemos establecido contacto con varios especialistas e investigadores de nuestra propia facultad de medicina y psicología dispuestos a ayudarnos desarrollar partes específicas del modelo.

Por otro lado, esperamos el poder combinar elementos adaptativos y adaptivos para ciertas características, como la visión; por ejemplo, generar un primer prototipo de la interfaz de usuario, pero si al usuario no le gusta el tamaño de letra presentado para el, el usuario pueda hacer zoom o seleccionar un tamaño diferente.

De igual forma sobre el tratar de cambiar a actitud de los alumnos sobre la usabilidad y las prácticas de la Interacción Humano-Computadora, resultó que la cooperación entre los diferentes cursos con un objetivo en mente resultó ser una excelente manera para que el alumno adopte un enfoque más centrado en el usuario en el desarrollo de software, especialmente cuando se vio directamente el impacto de las técnicas que han aprendido al usuario. Ellos muestran una actitud más dispuestos a utilizar esas técnicas en otros cursos y parecen desarrollar un enfoque más humanista de los cursos de desarrollo de software relacionados.

Los alumnos tomaron en consideración las características del usuario, crearon el modelo del usuario autista y diseñaron las clases en base a las características de los usuarios para diferentes elementos de usabilidad desde las etapas tempranas del diseño de la aplicación, de tal modo reforzando las diversas prácticas de la Interacción Humano-Computadora e ingeniería de usabilidad.

Algunos de los estudiantes, incluso, siguen desarrollando la aplicación en su tiempo libre con la esperanza de distribuir el software de forma gratuita a cualquier persona que pudiera ser útil, mientras que otros participan en otros proyectos de software con un enfoque social.

Esperamos que sigan a ver la importancia de la HCI mucho tiempo después de graduarse y comenzar un cambio en nuestra industria local de software.

VII

Referencias

VII. REFERENCIAS

Ahmed S., Edward M. (2009). *On usability and usability engineering, Adoption – Centric Usability Engineering*, Part I, 3-13, doi: 10.1007/978-1-84800-019-3_1, Springer.

Ahmed S., Taleb M, Halima H., Alain A. (2008). *Reconciling usability and interactive system architecture using patterns.* Journal of Systems and Software, Volume 81, pp. 1845-1852. Elsevier.

Barbacci M., Klein M.H, Longstaff T.A., Weinstock C.B. (1995). *Quality Attributes. Technical Report CMU/SEI-95-TR-021 ESC-TR-95-02*.

Bass L., Clemens P., Kazman R. (2003). *Software Architecture in Practice. Second Edition.* Addison-Wesley. Carnegie Mellon, Software Engineering Institute (SEI). The SEI series in Software Engineering.

Bass, L., John, B., & Kates, J. (2001). *Achieving Usability Through Software Architecture*(CMU/SEI-2001-TR-005).

Bellotti V., Fukuzumi S., Asahi T., Suzuki S. (2009). *User-centered design and evaluation - the big picture*. Proc. HCI International 2009, Springer, pp. 214-223, doi: 10.1007/978-3-642-02574-7_24.

Biswas P., Bhattacharya S., Samantha D, (2005). *User Model to Design Adaptable Interfaces for Motor-Impared Users.*

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., Merritt, M. (1978), *Characteristics of Software Quality*, North Holland.

Cecil R.M. (2003). *Agile Software Development: Principles, Patterns and Practices.* Pearson Education.

Chidamber S.R., Kemerer C.F. (1994). *A metric suite for Object Oriented Design.*

Elain R., (1979). *User Modeling via Stereotypes*. COGNITIVE SCIENCE 3, pp. 329-354.

Ferre X., Juristo N., Windl H., Constantine L. (2001) *Usability Basics for Software Developers*, IEEE Software, vol 18, pp. 22-29.

Ferre X., Juristo N., Moreno A.M., Sánchez M.I. (2003), *A Software Architectural View of Usability Patterns*, doi: 10.1.1.59.6084.

Fischer G. (2001). *User Modeling in Human-Computer Interaction*. User Modeling and User-Adapted Interaction, Vollume 11, Issue 1-2, pp. 65-86.

Folmer E., Gurp J.V., Bosch J. (2004). *Software Architecture Analysis of Usability*, Proc. The IFIP Working on Engineering for Human-Computer Interaction, doi: 10.1.1.5.3037.

Halstead M. H. (1977), Elements of Software Science, Operating, and Programming Systems Series Volume 7, Elsevier.

Henderson-Sellers B. (1996), *Object-oriented metrics : measures of complexity*, Prentice-Hall, pp.142-147.

Hothi J., Hall W. (1998). *An Evaluation of Adapted Hypermedia Techniques Using Static User Modelling*. Proceedings of the 2nd Workshop on Adaptive Hypertext and Hypermedia.

ISO/IEC 9126-1:2001 Software engineering: Product quality, Part 1: Quality model.

Johnson A, Taatgen N (2005), *User Modeling, Handbook of human factors in Web design.* Lawrence Erlbaum Associates.

Johnson J. (2010), *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. San Francisco, CA, USA: Morgan Kaufmann.

Juarez-Ramirez, R.; Gomez-Ruelas, M.; Gutierrez, A.A.; Negrete, P. (2011); , "Towards improving user interfaces: A proposal for integrating functionality and usability since early phases," *Uncertainty Reasoning and Knowledge Engineering (URKE), 2011 International Conference on* , vol.1, no., pp.119-123.

Jurgen E., Christian M. (2009 . *PaMGIS: A Framework for Pattern-Based Modeling and Generation of Interactive Systems.* Human Computer Interaction: New Trends. Lecture Notes on Computer Science, Volume 5610/2009, 826-83. Springer.

Kafura D., Reddy R. (1987), *The Use of Software Complexity Metrics in Software Maintenance*. IEEE Trans. Software Engineering, vol. SE-13, no. 3, pp. 335-343.

Kim S. K., Carrington D. (2002), *Integrating Use-Case Analysis and Task Analysis for Interactive Systems.* Asia Pacific Software Engineering Conference (APSEC'02)", EEE Computer Society: Washington, pp. 12-21.

Komogortsev V., Mueller J., Dan T., Liam F. (2009). *An Effort Based Model of Software Usability.* Proceedings of the International Conference on Software Engineering Theory and Practice.

Kruchten P. (1995). *Architectural Blueprints – The 4+1 View Model of Software Architecture*, IEEE Software, pp. 42-50.

Land R., *Measurements of Software Maintainability*. Mälardalen University, Department of Computer Engineering, Box 883, SE-721 23 Vasteras, Sweden.

Mayhew D. (1999) *The Usability Engineering Lifecycle*. Morgan Kauffman, United States.

McCabe T. J. *A Complexity Measure. (1976).*, IEEE Transactions on Software Engineering, Vol. SE-2, No.4.

McCall, J. A., Richards, P. K., and Walters, G. F. (1977). *Factors in Software Quality*, Nat'l Tech.Information Service, no. Vol. 1, 2 and 3.

Mejía A., Juárez-Ramírez R., Inzunza S., Valenzuela R. (2012) *Implementing adaptive interfaces: a user model for the development of usability in interactive systems.* In Proceedings of the CUBE International Information Technology Conference, Pune, India, ACM, pp. 598-604.

Mills E. (1998). *Software Metrics*. SEI Curriculum Module SEI-CM-12-1.1, Carnegie Mellon University, Software Engineering Institute.

Neil B. Harrison, Avgeriou P. (2007). *Leveraging Architecture Patterns to Satisfy Quality Attributes*. Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherland.

Nielsen J., (1993). *Usability Engineering*, Academic Press (Elsevier), United States of America.

Oman P., Hagemeister J. (1992). *Metrics for Assessing a Software System's Maintainability*. pp. 337-344. Conference on Software Maintenance 1992. Orlando, FL, November 9-12, 1992. Los Alamitos, CA: IEEE Computer Society Press, 1992.

Parnas, D. L. (1972). *On the criteria to be used in decomposing systems into modules*, Commun. ACM, 15(12) , pp. 1053–1058.

Preece J., Rogers Y., Sharp H., Benyon D., Holland S., Carey T. (1994). *Human-Computer Interaction.* Addison Wesley.

Rombach D. (1987), *A Controlled Experiment on the Impact of Software*

Shaw M., Garlan D. (1996). *Software Architecture: Perspectives on an emerging discipline.* Prentice Hall.

Stavrinoudis D., Xenos M., Christodoulakis D (1999). *Relationship between software metrics and maintainability*. Proceedings of the FESMA99 International Conference, Federation of European Software Measurement Associations, Amsterdam, The Netherlands, pp. 465-476.

Stephanidis C. (2001), *User Interfaces for All: Concepts Methods and Tools*. Mahwah, NJ: Lawrence Erlbaum Associates.

Structure on Maintainability, IEEE Trans. Software Engineering, vol. SE-13, no. 3, pp. 344-354.

Stuart T. P. M., Newell A., (1983) *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Weinschenk S. (2011), *100 Things Every Designer Needs to Know About People.* New Riders.

Zhang P., Carey J., Te'eni D., Tremaine M. (2004). *Integrating Human-Computer Interaction Development into SDLC: A Methodology.* Proceedings of the Americas Conference on Information Systems, New York, New York.

APÉNDICES

Apéndice A. Patrones arquitecturales más comunes.

A.1. Pipa y filtros

En un estilo pipa y filtros cada componente o filtro tiene una serie de entradas y una serie de salidas. Un componente lee flujos de datos en sus entradas y produce otro flujo de datos en sus salidas, enviando un resultado por medio de conectores o pipas a otros filtros (véase Figura 23).

El filtro transforma o filtra la información que recibe por medio de las pipas al que está conectado, pude tener *n* número de pipas de entrada y pipas de salida.

La pipa es el conector que pasa la información de un filtro al próximo. El flujo de información va en una sola dirección enviándola al siguiente filtro.

Entre las características principales de este estilo, los filtros deben de ser entidades totalmente independientes de uno a otro. Otra característica importante es que los filtros no saben la identidad de los filtros siguientes o los anteriores.

Existen algunos estilos especializados de este patrón, como es el pipeline o tubería, que restringe la topología de la arquitectura en secuencias lineales de filtros; tubería limitada, la cual restringe la cantidad de información que puede haber en una pipa; y tuberías de un tipo, las cuales requieren que la información que pasa entre dos filtros sea de un cierto tipo bien definido.

Figura 23. Estilo arquitectural Pipa y Filtros.

Sistemas basados en la arquitectura de pipas y filtros ayudan al diseñador a entender el comportamiento de entrada/salida de un sistema como una composición de los comportamientos individuales de los filtros. También son altamente reusables: cualquier dos filtros pueden ser conectados uno a otro mientras la información que estén intercambiando sea la adecuada para las operaciones del filtro. También los sistemas basados en esta arquitectura pueden ser muy mantenibles y expandibles ya que nuevos filtros pueden ser agregados a sistemas existentes y filtros viejos pueden ser remplazados por nuevos. Otra ventaja es que permite ejecución concurrente al poder ejecutar varios filtros de manera paralela.

Una de las desventajas de esta arquitectura es que los sistemas pipa y filtro generalmente tienden a tener un procesamiento por medio de lotes o batch, pasando la información por cada filtro independiente transformando la información de entrada de manera incremental a la información de salida. Debido a esta naturaleza de proceso incremental, un sistema basado en la arquitectura de pipa y filtros no es bueno para aplicaciones interactivas o que requieran alto rendimiento.

A.2. Abstracción de Datos y Organización Orientada a Objetos

Representaciones de los datos y sus operaciones son encapsuladas en un tipo de dato abstracto u objeto (véase Figura 24). Los componentes de este estilo son llamados objetos, en otras palabras, instancias abstractas de los tipos de datos. Objetos interactúan por medio de invocaciones a funciones y procedimientos.

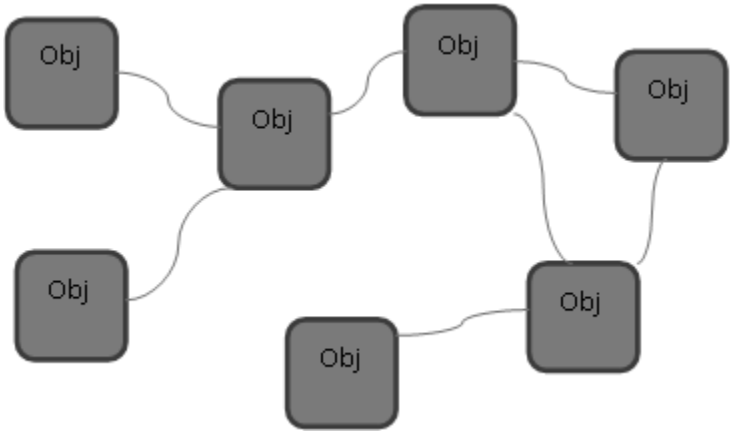


Figura 24. Representación por medio de objetos y tipos de datos abstractos. Donde cada relación entre los objetos es una invocación a alguna funcionalidad de otro objeto.

Los sistemas orientados a objetos tienes varias propiedades ventajosas. Una de ellas es la cualidad de que como un objeto esconde su representación de otros objetos, es posible cambiar la implementación sin afectar otros objetos. Adicionalmente, al agregar procedimientos de acceso a la información que manipulan los objetos permite descomponer problemas en colecciones de elementos o componentes que interactúan.

Una de las desventajas de este estilo es que para que un objeto pueda interactuar con otro debe de conocer la identidad del otro objeto, a diferencia de los sistemas en base a la arquitectura pipa y filtro, que no necesitan saber que otros filtros hay en el sistema para poder interactuar con ellos. El efecto de este detalle, es que cuando la identidad de n objeto cambia, es necesario modificar todos los otros objetos que lo invocan.

A.3. Sistemas basados en eventos e Invocación Implícita.

La noción detrás de la invocación implícita es de que en vez de invocar un procedimiento directamente, un componente puede anunciar uno o más eventos. Otros componentes pueden registrar un interés en un evento al asociar un procedimiento a un evento. Cuando un evento es anunciado, el sistema invoca todos los procedimientos que han sido registrados para ese evento. Entonces el anuncio de un evento implícitamente causa la invocación de procedimientos en otros módulos.

Arquitecturalmente los componentes de un estilo de invocación explícita son módulos que tiene interfaces que proveen de una colección de procedimientos y una serie de eventos. Procedimientos pueden ser llamados normalmente, pero además, un componente puede registrar algunos de sus componentes con eventos del sistema. Esto causará que esos procedimientos sean invocados cuando esos eventos son anunciados a la hora de ejecución.

Un importante beneficio de la invocación implícita es que provee gran reusabilidad. Cualquier componente puede ser introducido al sistema simplemente al registrarlo a los eventos de ese sistema. Un segundo beneficio es que facilita la evolución del sistema. Componentes pueden ser remplazados por otros sin afectar las interfaces de otros componentes en el sistema.

La desventaja principal de la invocación implícita es que los componentes no tienen control sobre las operaciones hechas por el sistema. Cuando un componente anuncia un evento, no tiene idea como los otros componentes van a reaccionar. Aún si supiera no puede basarse en el orden en el cual son invocados, ni saber cuándo terminan. Otro problema es el manejo de los datos, a veces datos pueden ser pasados con el evento, pero a veces los datos son compartidos en un repositorio para interacción. En esos casos el rendimiento y el manejo de recursos puede ser un problema.

A.4. Repositorios

En este estilo, existen dos tipos de componentes: una estructura de datos central, y una colección de componentes independientes que operan la base de datos central. La forma en que los componentes independientes y la estructura de datos interactúan determina el tipo de repositorio. Si los datos de entrada generan procesos de selección a ejecutar, puede ser una clásica base de datos. Si el estado actual de la estructura de datos central es el que se toma en cuenta para la selección del proceso a ejecutar, el repositorio puede ser un blackboard (véase Figura 25).

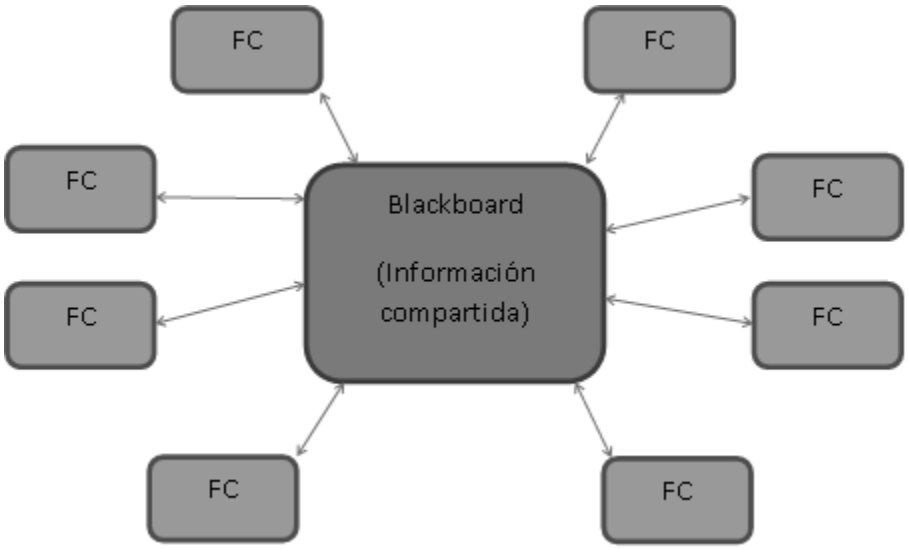


Figura 25. Arquitectura Blackboard.

La arquitectura del blackboard consta de tres partes:

- Las fuentes de conocimientos o módulos especialistas (FC). Componentes independientes que contienen conocimiento específico que el blackboard necesita para la resolución de problemas.

- La estructura de datos blackboard. Un repositorio compartido de problemas organizado jerárquicamente. Las fuentes de conocimiento hacen contribuciones hasta lograr solucionar el problema.
- Control. Responde dependiendo del estado del blackboard permitiendo a las fuentes de conocimiento hacer contribuciones solo cuando les es permitido.

Algunas implementaciones de esta arquitectura incluyen sistemas que requieren complejas interpretaciones para el procesamiento de señales, como el reconocimiento de patrones y el procesamiento de lenguaje.

A.5. Intérpretes

En un intérprete una máquina virtual es creada en el software. El intérprete incluye un pseudo-programa siendo interpretado y el motor de interpretación en sí. El pseudo-programa incluye el programa en sí y el estado de ejecución del intérprete. El motor del intérprete incluye la definición del intérprete y el estado actual de su ejecución. Por lo tanto un intérprete generalmente consta de cuatro componentes: Un motor de interpretación el cual hace el trabajo, una memoria que contiene el pseudo-código siendo interpretado, una representación del control del estado dl motor de interpretación y una representación del estado actual del programa siendo simulado (Figura 26).

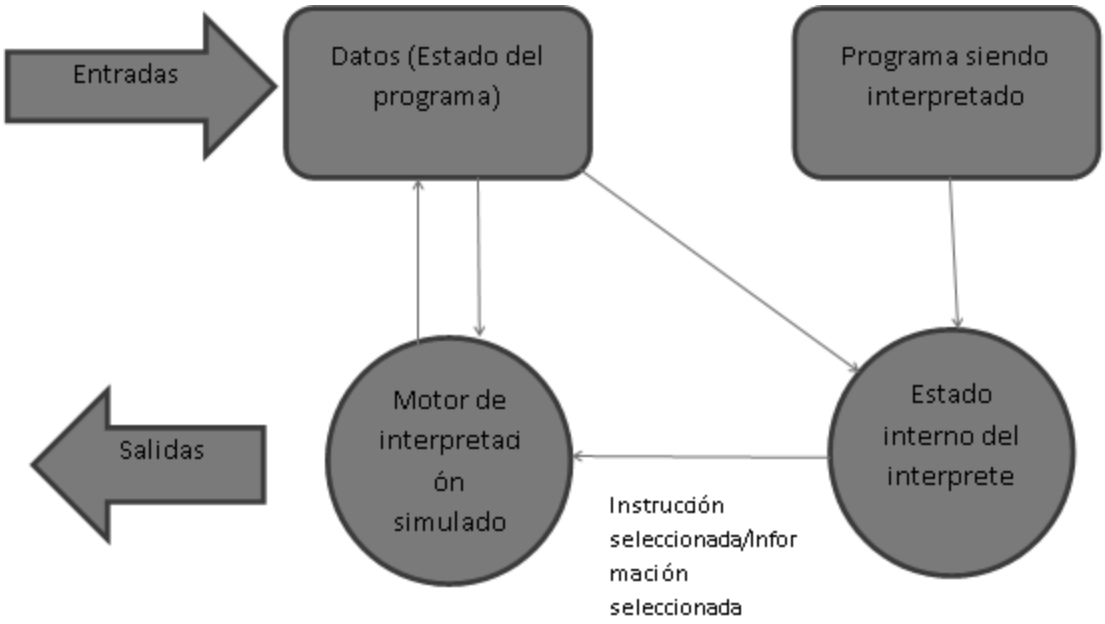


Figura 26. Estructura de un intérprete.

El uso más común de este estilo arquitectural es para la creación de máquinas virtuales cuando no se encuentra con el lenguaje o equipo de software apropiado para un cierto problema.

A.6 Otros estilos arquitecturales

Existen numerosos estilos arquitectónicos. Cada uno de ellos para la resolución de problemas específicos para lo que fueron creados. Anteriormente se mencionaron los estilos más comunes y de los que se basan para muchos otros estilos. A continuación se presentan algunas categorías de estilos o patrones arquitecturales:

- **Procesos distribuidos.** Sistemas distribuidos son caracterizados principalmente por sus características topológicas, como por ejemplo, topología de anillo o estrella. Otros son

mejor caracterizados en términos de los protocolos de comunicación usados para sus procesos. Una de las formas más común de este estilo es la organización cliente-servidor. El servidor representa un proceso que provee servicios a otros procesos (clientes). El servidor regularmente no conoce las identidades o los números de clientes de los cuales tendrá acceso a la hora de ejecución. Por otra mano, los clientes si conocen la identidad del servidor y acensarlo de manera remota.

- **Programa principal/organización por subrutinas.** Los programas que se desarrollan tienden a ser organizados de manera similar al lenguaje de programación en que fueron desarrollados. Para lenguajes en los que no se puede modularizar el código, se organiza en torno al programa principal o main y una serie de subrutinas.
- **Arquitecturas de software para dominios específicos.** Esas arquitecturas proseen una organización estructural específica para una familia de aplicaciones, como es el caso de sistemas de aviación y el manejo de sistemas de vehículos.
- **Sistemas de transición de estados.** Son sistemas definidos en término de una serie de estados y una serie de transiciones que transformas o mueven al sistema de un estado a otro.
- **Sistemas de control de procesos.** Sistemas que proveen el control dinámico de un ambiente físicos regularmente son organizados con este estilo. Se caracterizan por tener un ciclo de retroalimentación en donde un flujo de entrada por medio de sensores son utilizados por el sistema de control de procesos para determinar una serie de flujo de datos de salida que producirá un nuevo estado en el ambiente.

A.7 Arquitecturas Heterogéneas

La mayoría de los sistemas combinan varios estilos arquitecturales dependiendo de las necesidades del sistema. Hay varias formas en las que se pueden combinar, una de las maneras es por medio de una jerarquía. Un componente de un sistema puede estar organizado bajo un cierto estilo arquitectural puede tener una estructura interna que fue desarrollado con otro estilo arquitectural, incluso los conectores pueden ser descompuestos jerárquicamente.

Otra forma de combinar estilos es el de permitir que un componente utilice una mezcla de conectores arquitecturales. Por ejemplo, un componente puede accesar un repositorio a través de parte de su interfaz, pero interactuar por pipas con otros componentes del sistema y aceptar información de control por otra parte de su interfaz.

Otro ejemplo es el de una “base de datos activa”. Es un repositorio en el que activa componentes externos por medio de invocación explicita. En esta organización componentes externos registran interés en porciones de la base de datos. La base de datos automáticamente invoca las herramientas apropiadas basadas en esta asociación.

Apéndice B. Cuestionario aplicado para el diseño de interfaz del CRM.

NOMBRE: _____ PUESTO:_____

1. Grado de estudios:

Primaria	Secundaria	Preparatoria	Licenciatura	Posgrado

2. Experiencia en el puesto

Meses	1	3	6	9	12	12+

3. Experiencia en el uso de la PC.

Meses	1	3	6	9	12	12+

4. Experiencia con diferentes Sistemas Operativos.

S.O's	1 mes	3	6	9	12	12+
Windows						
Mac						
Linux						
Solaris						
Otro:_____						

5. Aplicaciones que ha utilizado:

   _____

   _____

6. Experiencia en cada aplicación

Aplicación	1 mes	3	6	9	12	12+