Unidad 1:

El software y la ingeniería de software

**La naturaleza del software:**

El software es un:

* **Producto:** Brinda el potencial de cómputo incorporado en el hardware de cómputo. **Es un transformador de información.**
* **Vehículo para entregar un producto:** Actúa como la base para el control de la computadora, para la comunicación de información, y para la creación y control de otros programas

El software distribuye el producto más importante de nuestro tiempo, **información**. Transforma los datos personales de modo que puedan ser más útiles en un contexto local, administra la información de negocios para mejorar la competitividad, provee una vía para las redes mundiales de información, y brinda los medios para obtener información en todas sus formas.

En la actualidad, la enorme industria del software se ha convertido en un factor determinante en las economías del mundo industrializado.

**Definición de software:**

El software son estructuras de datos que permiten que los programas manipulen de forma adecuada la información.

El software es elemento de un sistema lógico y no de uno físico. Por lo tanto, tiene características y difieren considerablemente de las del hardware.

1. El software se desarrolla o modifica con intelecto, no se manufactura en el sentido clásico:
   1. Desarrollo de software:
      1. Los costos se concentran en la ingeniería
      2. No pueden administrarse como si fueran proyectos de manufactura
      3. La calidad se logra a través de un buen diseño (Fabricación de hardware)
      4. Depende de personas, pero la relación entre los individuos dedicados y el trabajo logrado es diferente por completo (Fabricación de hardware)
2. El software no se desgasta:
   1. El software no es susceptible a los problemas ambientales que hacen que el hardware se desgaste, **el software no se desgasta, pero sí se deteriora**. El software sufrirá cambios, es probable que cuando estos cambios se realicen se introduzcan errores que ocasionen que la curva de tasa de fallas tenga aumentos súbitos. Poco a poco, el nivel mínimo de la taza de fallas comienza a aumentar. **El software se está deteriorando como consecuencia del cambio**.
3. Aunque la industria se mueve hacia la construcción basada en componentes, la mayor parte del software se construye para un uso individualizado.

**Dominios de la aplicación del software:**

1. **Software de sistemas:** Conjunto de programas escritos para dar servicio a otros programas. El área de software de sistemas se caracteriza por gran interacción con el hardware de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente que requiere la secuenciación de recursos administrativos y la administración de un proceso sofisticado, estructuras complejas de datos e interfaces externas múltiples.
2. **Software de aplicación:** Programas aislados que resuelven una necesidad específica de negocios. Además de las aplicaciones convencionales de procesamiento de datos, el software de aplicación se usa para controlar funciones de negocios en tiempo real.
3. **Software de ingeniería y ciencias:** Se ha caracterizado por algoritmos “devoradores de números”. Sin embargo, las aplicaciones modernas están abandonando esto. El diseño asistido por computadora, la simulación de sistemas, otras aplicaciones interactivas han comenzado a hacerse en tiempo real e incluso han tomado características de software de sistemas.
4. **Software incrustado:** Reside dentro de un producto o sistema, y se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí.
5. **Software de línea de productos:** Es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes.
6. **Aplicaciones web:** Están evolucionando hacia ambientes de cómputo sofisticados, que no solo proveen características aisladas, funciones de cómputo y contenido para el usuario final, sino que también están integrando con bases de datos corporativas y aplicaciones de negocios.
7. **Software de inteligencia artificial:** Hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo.

**Computación en un mundo abierto:**

* El reto para los ingenieros de software será desarrollar software de sistemas y aplicación que permita a dispositivos móviles, computadoras personales y sistemas empresariales comunicarse a través de redes enormes.

**Construcción de redes:**

* La red mundial se está convirtiendo con rapidez tanto en un motor de computación como en un proveedor de contenido. El desafío para los ingenieros de software es hacer arquitecturas sencillas.

**Fuente abierta:**

* El desafío para los ingenieros de software es elaborar código fuente que sea autodescriptivo y también lo que es más importante, desarrollar técnicas que permitan tanto a los consumidores como a los desarrolladores saber cuáles son los cambios hechos y cómo se manifiestan dentro del software.

**Software heredado:**

* Los sistemas de software heredados fueron desarrollados hace varias décadas, y han sido modificados de manera continua par que satisfagan los cambios en los requerimientos de los negocios y plataformas de computación. La proliferación de tales sistemas es causa de dolores de cabeza para las organizaciones grandes, a las que les resulta costoso mantenerlos y riesgoso hacerlos evolucionar.
* El software heredado se caracteriza por su longevidad e importancia crítica para el negocio, desafortunadamente en ocasiones, el software heredado es de mala calidad.
* La meta de la ingeniería de software moderna es desarrollar metodologías que se basen en el concepto de evolución, es decir el concepto de que los sistemas de software cambian continuamente, que los nuevos sistemas de software se desarrollan a partir de los antiguos y que todo lo debe operar entre sí y cooperar con cada uno de los demás.

**La naturaleza única de las webapps:**

La mayoría de las webapps presenta los siguientes atributos:

1. Uso intensivo de redes: Una webapp reside en una red y debe atender a las necesidades una comunidad diversa de clientes. La red permite acceso y comunicación mundial o tiene acceso y comunicación limitada.
2. Concurrencia.
3. Carga impredecible: El número de usuarios de la webapp cambia en varios órdenes de magnitud de un día para el otro.
4. Rendimiento
5. Disponibilidad
6. Orientadas a los datos: La función principal de muchas webapp es el uso de hipermedios para presentar al usuario final contenido en forma de texto, gráficas, audio y video.
7. Contenido sensible: La calidad y naturaleza estética del contenido constituye un rasgo importante de la calidad de una webapp.
8. Evolución continua.
9. Inmediata: Necesidad apremiante de que el software llegue con rapidez al mercado.
10. Seguridad.
11. Estética.

**Ingeniería de Software:**

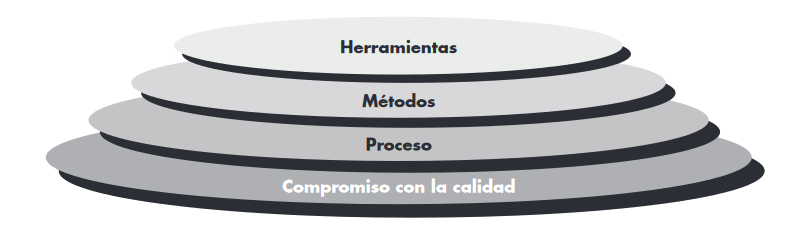
* Debe hacerse un esfuerzo concentrado para entender el problema antes de desarrollar una aplicación de software.
* El diseño se ha vuelto una actividad crucial.
* El Software debe tener alta calidad.
* Debe tener facilidad para recibir mantenimiento.

**Definiciones:**

* Fritz Bauer: “La ingeniería de software es el establecimiento y uso de principios fundamentales de la ingeniería con objeto de desarrollar en forma económica software que sea fiable y que trabaje con eficiencia en máquinas reales”
* IEEE: “La ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, es decir, la aplicación de la ingeniería al software”

La administración total de la calidad, Six Sigma y otras filosofías similares alimentan la cultura de mejora continua, y es esta cultura la que lleva en última instancia al desarrollo de enfoques cada vez más eficaces de la ingeniería de software. El fundamento en el que se apoya la ingeniería de software es el compromiso con la calidad.

El proceso de ingeniería de software es el aglutinante que une las capas de la tecnología y permite el desarrollo racional y oportuno del software de cómputo. Define una estructura que debe establecerse para la obtención eficaz de la tecnología de información de proyectos de software, y establece el contexto en el que se aplican métodos técnicos, se generan productos del trabajo, se establecen puntos de referencia, se asegura la calidad y se administra el cambio de manera apropiada.



Los métodos de la ingeniería de software proporcionan la experiencia técnica para elaborar software, se basan en un conjunto de principios fundamentales que gobiernan cada área de la tecnología e incluyen actividades de modelación y otras técnicas descriptivas.

Las herramientas de la ingeniería de software proporcionan un apoyo automatizado o semiautomatizado para el proceso y los métodos.

**El proceso de software:**

Es un conjunto de:

* Actividades: Buscan lograr un objetivo amplio.
* Acciones: Conjunto de tareas que producen un producto importante del trabajo.
* Tareas: Se centra en un objetivo pequeño, pero bien definido.

Es un enfoque adaptable que permite que las personas que hacen el trabajo busquen y elijan el conjunto apropiado de acciones y tareas para el trabajo.

**Una estructura de proceso general para la ingeniería de software consta de cinco actividades:**

* 1. **Comunicación:** Se busca entender los objetivos de los participantes respecto del proyecto.
  2. **Planificación:** Se crea una guía para el equipo, llamado plan del proyecto de software.
  3. **Modelado:** Un ingeniero de software crea modelos a fin de entender mejor los requerimientos del software y diseño que lo satisfará.
  4. **Construcción:** Combina la generación de código y las pruebas.
  5. **Despliegue:** El software se entrega al consumidor que lo evalúa.

**Son complementadas con actividades sombrillas:**

1. Seguimiento y control del proyecto de software
2. Administración del riesgo
3. Aseguramiento de la calidad del software
4. Revisiones técnicas
5. Medición
6. Administración de la configuración del software
7. Administración de la reutilización
8. Preparación y producción del producto del trabajo

**La práctica de la ingeniería de Software:**

Principios generales:

1. **La razón de que exista todo:** Un sistema de software existe por una razón: Dar valor a sus usuarios.
2. **MSE (Mantenlo sencillo, estúpido):** Todo diseño debe ser tan simple como sea posible, pero no más.
3. **Mantener la visión:** Una visión clara es esencial para que el éxito de un proyecto de software. Sin ella casi infaliblemente el proyecto terminará siendo “un ser con dos o más mentes”.
4. **Otros consumirán lo que usted produce:** Siempre establezca especificaciones, diseñe e implemente con la seguridad de que alguien más tendrá que entender lo que usted haga.
5. **Ábrase al futuro:** Un sistema con larga vida útil tiene más valor.
6. **Planee por anticipado la reutilización**
7. **Piense:** Pensar en todo con claridad antes de emprender la acción casi siempre produce mejores resultados.

Unidad 2

El proceso de software

El proceso es un diálogo en el que el conocimiento que debe convertirse en software se reúne e incorpora en éste. El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes, y entre diseñadores y herramientas en evolución [tecnología].

**¿“Proceso” es sinónimo de “ingeniería de software”?**

* La respuesta es “sí y no”.
* Un proceso del software define el enfoque adoptado mientras se hace ingeniería sobre el software. Pero la ingeniería de software también incluye tecnologías que pueblan el proceso: métodos técnicos y herramientas automatizadas.

**¿Qué es?**

* Cuando se trabaja en la construcción de un producto o sistema, es importante ejecutar una serie de pasos predecibles
* El mapa de carreteras que lo ayuda a obtener a tiempo un resultado de alta calidad.
* El mapa que se sigue se llama “proceso del software”.

**¿Quién lo hace?**

* Los ingenieros de software y sus gerentes adaptan el proceso a sus necesidades y luego lo siguen. Además, las personas que solicitaron el software tienen un papel en el proceso de definición, elaboración y prueba.

**¿Por qué es importante?**

* Porque da estabilidad, control y organización a una actividad que puede volverse caótica si se descontrola.

**UN MODELO GENERAL DE PROCESO**

**Patrones del proceso**

Un patrón de proceso da un formato [Amb98]: un método consistente para describir soluciones del problema en el contexto del proceso del software. Al combinar patrones, un equipo de software resuelve problemas y construye el proceso que mejor satisfaga las necesidades de un proyecto.

**Los patrones se definen en cualquier nivel de abstracción**. En ciertos casos, un patrón puede usarse para describir un problema (y su solución) asociado con un modelo completo del proceso (por ejemplo, hacer prototipos). En otras situaciones, los patrones se utilizan para describir un problema (y su solución) asociado con una actividad estructural (por ejemplo, planeación) o una acción dentro de una actividad estructural (estimación de proyectos).

Ambler [Amb98] ha propuesto un formato para describir un patrón del proceso:

* **Nombre del patrón:** El patrón recibe un nombre significativo que lo describe en el contexto del proceso del software.
* **Fuerzas:** El ambiente en el que se encuentra el patrón y los aspectos que hacen visible el problema y afectan su solución.
* **Tipo:** Se especifica el tipo de patrón. Ambler [Amb98] sugiere tres tipos:
  + 1. **Patrón de etapa:** define un problema asociado con una actividad estructural para el proceso. Como una actividad estructural incluye múltiples acciones y tareas del trabajo, un patrón de la etapa incorpora múltiples patrones de la tarea que son relevantes para la etapa (actividad estructural).
  + 2. **Patrón de tarea:** define un problema asociado con una acción o tarea de trabajo de la ingeniería de software y que es relevante para el éxito de la práctica de ingeniería de software.
  + 3. **Patrón de fase:** define la secuencia de las actividades estructurales que ocurren dentro del proceso, aun cuando el flujo general de las actividades sea de naturaleza iterativa.
* **Contexto inicial:** Describe las condiciones en las que se aplica el patrón.
* **Problema:** El problema específico que debe resolver el patrón.
* **Solución:** Describe cómo implementar con éxito el patrón. Esta sección describe la forma en la que se modifica el estado inicial del proceso (que existe antes de implementar el patrón) como consecuencia de la iniciación del patrón. También describe cómo se transforma la información sobre la ingeniería de software o sobre el proyecto, disponible antes de que inicie el patrón, como consecuencia de la ejecución exitosa del patrón.
* **Contexto resultante:** Describe las condiciones que resultarán una vez que se haya implementado con éxito el patrón.
* **Patrones relacionados:** Proporciona una lista de todos los patrones de proceso directamente relacionados con éste.
* **Usos y ejemplos conocidos:** Indica las instancias específicas en las que es aplicable el patrón.

**MODELOS DE PROCESO PRESCRIPTIVO**

**Modelo de la cascada**

El modelo de la cascada, a veces llamado ciclo de vida clásico sugiere un enfoque sistemático y secuencial para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir con el apoyo del software terminado.

El modelo de la cascada es el paradigma más antiguo de la ingeniería de software. Sin embargo, en las últimas tres décadas, las críticas hechas al modelo han ocasionado que incluso sus defensores más obstinados cuestionen su eficacia.

Hoy en día, el trabajo de software es acelerado y está sujeto a una corriente sin fin de cambios (en las características, funciones y contenido de información). El modelo de la cascada suele ser inapropiado para ese tipo de labor. No obstante, sirve como un modelo de proceso útil en situaciones en las que los requerimientos son fijos y el trabajo avanza en forma lineal hacia el final.

**Modelos de proceso incremental**

El modelo incremental aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce “incrementos” de software susceptibles de entregarse.

Cuando se utiliza un modelo incremental, es frecuente que el primer incremento sea el producto fundamental. Es decir, se abordan los requerimientos básicos, pero no se proporcionan muchas características suplementarias (algunas conocidas y otras no). El cliente usa el producto fundamental (o lo somete a una evaluación detallada). Como resultado del uso y/o evaluación se desarrolla un plan para el incremento que sigue. El plan incluye la modificación del producto fundamental para cumplir mejor las necesidades del cliente, así como la entrega de características adicionales y más funcionalidad. Este proceso se repite después de entregar cada incremento, hasta terminar el producto final.

**Modelos de proceso evolutivo**

El software, como todos los sistemas complejos, evoluciona en el tiempo. Es frecuente que los requerimientos del negocio y del producto cambien conforme avanza el desarrollo, lo que hace que no sea realista trazar una trayectoria rectilínea hacia el producto final.

Los modelos evolutivos son iterativos. Se caracterizan por la manera en la que permiten desarrollar versiones cada vez más completas del software.

Hacer prototipos: Aunque es posible hacer prototipos como un modelo de proceso aislado, es más común usarlo como una técnica que puede implementarse en el contexto de cualquiera de los modelos de proceso. El paradigma de hacer prototipos le ayudará a usted y a otros participantes a mejorar la comprensión de lo que hay que elaborar cuando los requerimientos no están claros.

Éste se centra en la representación de aquellos aspectos del software que serán visibles para los usuarios finales. Éste se entrega y es evaluado por los participantes, que dan retroalimentación para mejorar los requerimientos.

El prototipo sirve como “el primer sistema”. Lo que Brooks recomienda es desecharlo. Pero esto quizá sea un punto de vista idealizado. Aunque algunos prototipos se construyen para ser “desechables”, otros son evolutivos; es decir, poco a poco se transforman en el sistema real.

El modelo espiral. Propuesto en primer lugar por Barry Boehm [Boe88], el modelo espiral es un modelo evolutivo del proceso del software y se acopla con la naturaleza iterativa de hacer prototipos con los aspectos controlados y sistémicos del modelo de cascada. Tiene el potencial para hacer un desarrollo rápido de versiones cada vez más completas.

El modelo de desarrollo espiral es un generador de modelo de proceso impulsado por el riesgo, que se usa para guiar la ingeniería concurrente con participantes múltiples de sistemas intensivos en software. Tiene dos características distintivas principales. La primera es el enfoque cíclico para el crecimiento incremental del grado de definición de un sistema y su implementación, mientras que disminuye su grado de riesgo. La otra es un conjunto de puntos de referencia de anclaje puntual para asegurar el compromiso del participante con soluciones factibles y mutuamente satisfactorias.

El modelo es dividido por el equipo de software en un conjunto de actividades estructurales.

A diferencia de otros modelos del proceso que finalizan cuando se entrega el software, el modelo espiral puede adaptarse para aplicarse a lo largo de toda la vida del software de cómputo.

El modelo espiral es un enfoque realista para el desarrollo de sistemas y de software a gran escala. Como el software evoluciona a medida que el proceso avanza, el desarrollador y cliente comprenden y reaccionan mejor ante los riesgos en cada nivel de evolución. El modelo espiral usa los prototipos como mecanismo de reducción de riesgos, pero, más importante, permite aplicar el enfoque de hacer prototipos en cualquier etapa de la evolución del producto. Mantiene el enfoque de escalón sistemático sugerido por el ciclo de vida clásico, pero lo incorpora en una estructura iterativa que refleja al mundo real en una forma más realista. El modelo espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y, si se aplica de manera apropiada, debe reducir los riesgos antes de que se vuelvan un problema.

**Modelos concurrentes**

* El modelo de desarrollo concurrente, en ocasiones llamado ingeniería concurrente, permite que un equipo de software represente elementos iterativos y concurrentes de cualquiera de los modelos de proceso descritos.
* El modelado concurrente define una serie de eventos que desencadenan transiciones de un estado a otro para cada una de las actividades, acciones o tareas de la ingeniería de software.
* A pesar de los beneficios incuestionables de los procesos evolutivos de software, existen algunas preocupaciones. La primera es que hacer prototipos plantea un problema para la planeación del proyecto debido a la incertidumbre en el número de ciclos que se requieren para elaborar el producto. La mayor parte de técnicas de administración y estimación de proyectos se basa en un planteamiento lineal de las actividades, por lo que no se ajustan por completo.
* En segundo lugar, los procesos evolutivos de software no establecen la velocidad máxima de la evolución. Si las evoluciones ocurren demasiado rápido, sin un periodo de relajamiento, es seguro que el proceso se volverá un caos. Por otro lado, si la velocidad es muy lenta, se verá perjudicada la productividad.
* En tercer lugar, los procesos de software deben centrarse en la flexibilidad y capacidad de extensión en lugar de en la alta calidad. Esto suena preocupante. Sin embargo, debe darse prioridad a la velocidad del desarrollo con el enfoque de cero defectos. Extender el desarrollo a fin de lograr alta calidad podría dar como resultado la entrega tardía del producto, cuando haya desaparecido el nicho de oportunidad. Este cambio de paradigma es impuesto por la competencia al borde del caos.
* El objetivo de los modelos evolutivos es desarrollar software de alta calidad14 en forma iterativa o incremental. Es posible usar un proceso evolutivo para hacer énfasis en la flexibilidad, expansibilidad y velocidad del desarrollo.

**MODELOS DE PROCESO ESPECIALIZADO**

**Desarrollo basado en componentes**

* El modelo de desarrollo basado en componentes incorpora muchas de las características del modelo espiral. Es de naturaleza evolutiva [Nie92] y demanda un enfoque iterativo para la creación de software. Sin embargo, el modelo de desarrollo basado en componentes construye aplicaciones a partir de fragmentos de software prefabricados.
* El modelo del desarrollo basado en componentes lleva a la reutilización del software, y eso da a los ingenieros de software varios beneficios en cuanto a la mensurabilidad. Si la reutilización de componentes se vuelve parte de la cultura, el equipo de ingeniería de software tiene la posibilidad tanto de reducir el ciclo de tiempo del desarrollo como el costo del proyecto.

**El modelo de métodos formales**

* El modelo de métodos formales agrupa actividades que llevan a la especificación matemática formal del software de cómputo. Los métodos formales permiten especificar, desarrollar y verificar un sistema basado en computadora por medio del empleo de una notación matemática rigurosa.

• El desarrollo de modelos formales consume mucho tiempo y es caro.

• Debido a que pocos desarrolladores de software tienen la formación necesaria para aplicar métodos formales, se requiere mucha capacitación.

• Es difícil utilizar los modelos como mecanismo de comunicación para clientes sin complejidad técnica.

**Desarrollo de software orientado a aspectos**

* El desarrollo de software orientado a aspectos (DSOA), conocido también como programación orientada a aspectos (POA), es un paradigma de ingeniería de software relativamente nuevo que proporciona un proceso y enfoque metodológico para definir, especificar, diseñar y construir aspectos: “mecanismos más allá de subrutinas y herencia para localizar la expresión de una preocupación global”.

**EL PROCESO UNIFICADO**

* El proceso unificado es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software. El proceso unificado reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema (el caso de uso).Hace énfasis en la importancia de la arquitectura del software y “ayuda a que el arquitecto se centre en las metas correctas, tales como que sea comprensible, permita cambios futuros y la reutilización”. [Jac99]: Sugiere un flujo del proceso iterativo e incremental, lo que da la sensación evolutiva que resulta esencial en el desarrollo moderno del software.

**Fases del proceso unificado**

* **La fase de concepción del PU** agrupa actividades tanto de comunicación con el cliente como de planeación. Al colaborar con los participantes, se identifican los requerimientos del negocio, se propone una arquitectura aproximada para el sistema y se desarrolla un plan para la naturaleza iterativa e incremental del proyecto en cuestión.
* **La fase de elaboración** incluye las actividades de comunicación y modelado del modelo general del proceso. La elaboración mejora y amplía los casos de uso preliminares desarrollados como parte de la fase de concepción y aumenta la representación de la arquitectura para incluir cinco puntos de vista distintos del software: los modelos del caso de uso, de requerimientos, del diseño, de la implementación y del despliegue.

En ciertos casos, la elaboración crea una “línea de base de la arquitectura ejecutable” [Arl02] que representa un sistema ejecutable de “primer corte”.20 La línea de base de la arquitectura demuestra la viabilidad de ésta, pero no proporciona todas las características y funciones que se requieren para usar el sistema. Es frecuente que en este momento se hagan modificaciones al plan.

* **La fase de construcción del PU** es idéntica a la actividad de construcción definida para el proceso general del software. Con el uso del modelo de arquitectura como entrada, la fase de construcción desarrolla o adquiere los componentes del software que harán que cada caso de uso sea operativo para los usuarios finales. Para lograrlo, se completan los modelos de requerimientos y diseño que se comenzaron durante la fase de elaboración, a fin de que reflejen la versión final del incremento de software.
* **La fase de transición del PU** incluye las últimas etapas de la actividad general de construcción y la primera parte de la actividad de despliegue general (entrega y retroalimentación). Se da el software a los usuarios finales para las pruebas beta, quienes reportan tanto los defectos como los cambios necesarios. Además, el equipo de software genera la información de apoyo necesaria (por ejemplo, manuales de usuario, guías de solución de problemas, procedimientos de instalación, etc.) que se requiere para el lanzamiento.
* **La fase de producción del PU** coincide con la actividad de despliegue del proceso general. Durante esta fase, se vigila el uso que se da al software, se brinda apoyo para el ambiente de operación (infraestructura) y se reportan defectos y solicitudes de cambio para su evaluación.

Es probable que al mismo tiempo que se llevan a cabo las fases de construcción, transición y producción, comience el trabajo sobre el siguiente incremento del software. Esto significa que las cinco fases del PU no ocurren en secuencia, sino que concurren en forma escalonada.

El flujo de trabajo de la ingeniería de software está distribuido a través de todas las fases del PU. En el contexto de éste, un flujo de trabajo es análogo al conjunto de tareas.

**TECNOLOGÍA DEL PROCESO**

Se han desarrollado herramientas de tecnología del proceso que ayudan a las organizaciones de software a analizar su proceso actual, organizar las tareas de trabajo, controlar y vigilar el avance, y administrar la calidad técnica.

Unidad 8

Conceptos de diseño

El diseño de software agrupa el conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. Los principios de diseño establecen una filosofía general que guía el trabajo de diseño que debe ejecutarse.

El diseño es crucial para el éxito de la ingeniería de software.

**Un buen diseño de software debe tener:**

* Resistencia: un programa no debe tener ningún error que impida su funcionamiento.
* Funcionalidad: un programa debe ser apropiado para los fines que persigue.
* Belleza: la experiencia de usar el programa debe ser placentera. Éstos son los comienzos de una teoría del diseño de software.

**Para lograrlo, debe practicarse la diversificación y luego la convergencia**. Belady [Bel81] afirma que “la diversificación es la adquisición de un repertorio de alternativas, materia prima del diseño: componentes, soluciones con los componentes y conocimiento, todo lo cual está contenido en catálogos, libros de texto y en la mente”. Una vez que se reúne este conjunto diversificado de información, deben escogerse aquellos elementos del repertorio que cumplan los requerimientos definidos por la ingeniería y por el modelo de análisis (capítulos 5 a 7). A medida que esto ocurre, se evalúan las alternativas, algunas se rechazan, se converge en “una configuración particular de componentes y, con ello, en la creación del producto final” [Bel81].

La diversificación y la convergencia combinan la intuición y el criterio con base en la experiencia en la construcción de entidades similares, un conjunto de principios heurísticos que guían la forma en la que evoluciona el modelo, un conjunto de criterios que permiten evaluar la calidad y un proceso iterativo que finalmente conduce a una representación del diseño definitivo.

**DISEÑO EN EL CONTEXTO DE LA INGENIERÍA DE SOFTWARE**

* El diseño del software comienza una vez que se han analizado y modelado los requerimientos, es la última acción de la ingeniería de software dentro de la actividad de modelado y prepara la etapa de construcción (generación y prueba de código).
* El trabajo de diseño es alimentado por el modelo de requerimientos, manifestado por elementos basados en el escenario, en la clase, orientados al flujo, y del comportamiento. (Gráfico triángulo)
* El diseño de datos o clases transforma los modelos de clases (capítulo 6) en realizaciones de clases de diseño y en las estructuras de datos que se requieren para implementar el software.
* Parte del diseño de clase puede llevarse a cabo junto con el diseño de la arquitectura del software. Un diseño más detallado de las clases tiene lugar cuando se diseña cada componente del software.
* El diseño de la arquitectura define la relación entre los elementos principales de la estructura del software, los estilos y patrones de diseño de la arquitectura que pueden usarse para alcanzar los requerimientos definidos por el sistema y las restricciones que afectan la forma en la que se implementa la arquitectura.
* El diseño de la interfaz describe la forma en la que el software se comunica con los sistemas que interactúan con él y con los humanos que lo utilizan. Entonces, los modelos de escenarios de uso y de comportamiento dan mucha de la información requerida para diseñar la interfaz.
* El diseño en el nivel de componente transforma los elementos estructurales de la arquitectura del software en una descripción de sus componentes en cuanto a procedimiento. La información obtenida a partir de los modelos basados en clase, flujo y comportamiento sirve como la base para diseñar los componentes.
* Durante el diseño se toman decisiones que en última instancia afectarán al éxito de la construcción del software y, de igual importancia, a la facilidad con la que puede darse mantenimiento al software.
* **La importancia del diseño del software se resume, en una palabra: calidad.** **El diseño es el sitio en el que se introduce calidad en la ingeniería de software.**
* Sin diseño se corre el riesgo de obtener un sistema inestable, que falle cuando se hagan cambios pequeños, o uno que sea difícil de someter a prueba, o en el que no sea posible evaluar la calidad hasta que sea demasiado tarde en el proceso de software, cuando no queda mucho tiempo y ya se ha gastado mucho dinero.

**EL PROCESO DE DISEÑO**

**El diseño de software es un proceso iterativo por medio del cual se traducen los requerimientos en un “plano” para construir el software**. Es decir, el diseño se representa en un nivel alto de abstracción, en el que se rastrea directamente el objetivo específico del sistema y los requerimientos más detallados de datos, funcionamiento y comportamiento. A medida que tienen lugar las iteraciones del diseño, las mejoras posteriores conducen a niveles menores de abstracción.

**Lineamientos y atributos de la calidad del software:**

Características que funcionan como guía para evaluar un buen diseño:

• Debe implementar todos los requerimientos explícitos contenidos en el modelo de requerimientos y dar cabida a todos los requerimientos implícitos que desean los participantes.

• Debe ser una guía legible y comprensible para quienes generan el código y para los que lo prueban y dan el apoyo posterior.

• Debe proporcionar el panorama completo del software, y abordar los dominios de los datos, las funciones y el comportamiento desde el punto de vista de la implementación.

**Lineamientos de la calidad:**

1. **Debe tener una arquitectura que**:
   1. Se haya creado con el empleo de estilos o patrones arquitectónicos reconocibles,
   2. Esté compuesta de componentes con buenas características de diseño y
   3. Se implementen en forma evolutiva, de modo que faciliten la implementación y las pruebas.
2. **Debe ser modular**, es decir, el software debe estar dividido de manera lógica en elementos o subsistemas.
3. **Debe contener distintas representaciones** de datos, arquitectura, interfaces y componentes.
4. **Debe conducir a estructuras de datos apropiadas** para las clases que se van a implementar y que surjan de patrones reconocibles de datos.
5. **Debe llevar a componentes** que tengan características funcionales independientes.
6. **Debe conducir a interfaces** que reduzcan la complejidad de las conexiones entre los componentes y el ambiente externo.
7. **Debe obtenerse con el empleo de un método repetible** motivado por la información obtenida durante el análisis de los requerimientos del software.
8. **Debe representarse** con una notación que comunique con eficacia su significado.

Estos lineamientos de diseño se consiguen con la aplicación de los principios de diseño fundamentales, una metodología sistemática y con revisión.

**Atributos de la calidad.**

**Los atributos de calidad FURPS representan el objetivo de todo diseño de software:**

• **La funcionalidad** se califica de acuerdo con el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad general del sistema.

• **La usabilidad** se evalúa tomando en cuenta factores humanos , la estética general, la consistencia y la documentación.

• **La confiabilidad** se evalúa con la medición de la frecuencia y gravedad de las fallas, la exactitud de los resultados que salen, el tiempo medio para que ocurra una falla (TMPF), la capacidad de recuperación ante ésta y lo predecible del programa.

• **El rendimiento** se mide con base en la velocidad de procesamiento, el tiempo de respuesta, el uso de recursos, el conjunto y la eficiencia.

• **La mantenibilidad** combina la capacidad del programa para ser **ampliable** (extensibilidad), **adaptable** y **servicial** (estos tres atributos se denotan con un término más común: **mantenibilidad**), y además que pueda probarse, ser compatible y configurable y que cuente con la facilidad para instalarse en el sistema y para que se detecten los problemas.

**CONCEPTOS DE DISEÑO**

* M. A. Jackson [Jac75] dijo: “El principio de la sabiduría [para un ingeniero de software] es reconocer la diferencia que hay entre hacer que un programa funcione y lo haga bien”. Los conceptos fundamentales del diseño del software proveen la estructura necesaria para “hacerlo bien”.

**Abstracción**

* Cuando se considera una solución modular para cualquier problema, es posible plantear muchos niveles de abstracción. En el más elevado se enuncia una solución en términos gruesos con el uso del lenguaje del ambiente del problema. En niveles más bajos de abstracción se da la descripción más detallada de la solución.
* Cuando se desarrollan niveles de abstracción distintos, se trabaja para crear abstracciones tanto de procedimiento como de datos. Una abstracción de procedimiento es una secuencia de instrucciones que tienen una función específica y limitada. Una abstracción de datos es un conjunto de éstos con nombre que describe a un objeto de datos.

**Arquitectura**

* La arquitectura del software alude a “la estructura general de éste y a las formas en las que ésta da integridad conceptual a un sistema”.
* Una meta del diseño del software es obtener una aproximación arquitectónica de un sistema.
* Ésta sirve como estructura a partir de la cual se realizan las actividades de diseño más detalladas.
* Propiedades que deben especificarse como parte del diseño de la arquitectura:
  + **Propiedades estructurales:** Este aspecto de la representación del diseño arquitectónico define los componentes de un sistema (módulos, objetos, filtros, etc.) y la manera en la que están agrupados e interactúan unos con otros.
  + **Propiedades extrafuncionales:** La descripción del diseño arquitectónico debe abordar la forma en la que la arquitectura del diseño satisface los requerimientos de desempeño, capacidad, confiabilidad, seguridad y adaptabilidad, así como otras características del sistema.
  + **Familias de sistemas relacionados:** El diseño arquitectónico debe basarse en patrones repetibles que es común encontrar en el diseño de familias de sistemas similares.

Dada la especificación de estas propiedades, el diseño arquitectónico se representa con el uso de uno o más de varios modelos diferentes [Gar95].

* **Los modelos estructurales** representan la arquitectura como un conjunto organizado de componentes del programa.
* **Los modelos de marco** aumentan el nivel de abstracción del diseño, al tratar de identificar patrones de diseño arquitectónico repetibles que se encuentran en tipos similares de aplicaciones.
* **Los modelos dinámicos** abordan los aspectos estructurales de la arquitectura del programa e indican cómo cambia la estructura o la configuración del sistema en función de eventos externos.
* **Los modelos del proceso** se centran en el diseño del negocio o proceso técnico al que debe dar acomodo el sistema.
* **Los modelos funcionales** se usan para representar la jerarquía funcional de un sistema.

**Patrones**

* Brad Appleton define un patrón de diseño de la manera siguiente: “**Es una mezcla con nombre propio de puntos de vista que contienen la esencia de una solución demostrada para un problema recurrente dentro de cierto contexto de necesidades en competencia**”. El objetivo de cada patrón de diseño es proporcionar una descripción que permita a un diseñador determinar

1) Si el patrón es aplicable al trabajo en cuestión

2) Si puede volverse a usar (con lo que se ahorra tiempo de diseño)

3) Si sirve como guía para desarrollar un patrón distinto en funciones o estructura.

**División de problemas**

* La división de problemas es un concepto de diseño que sugiere que cualquier problema complejo puede manejarse con más facilidad si se subdivide en elementos susceptibles de resolverse u optimizarse de manera independiente. Un problema es una característica o comportamiento que se especifica en el modelo de los requerimientos para el software.

**Modularidad**

* La modularidad es la manifestación más común de la división de problemas. El software se divide en componentes con nombres distintos y abordables por separado, en ocasiones llamados módulos, que se integran para satisfacer los requerimientos del problema.
* De acuerdo con la figura 8.2, el esfuerzo (costo) de desarrollar un módulo individual de software disminuye conforme aumenta el número total de módulos. Sin embargo, a medida que se incrementa el número de módulos, el esfuerzo (costo) asociado con su integración también aumenta. figura. Existe un número, M, de módulos que arrojarían el mínimo costo de desarrollo, pero no se dispone de las herramientas necesarias para predecir M con exactitud.
* Debe hacerse un diseño (y el programa resultante) con módulos, de manera que el desarrollo pueda planearse con más facilidad, que sea posible definir y desarrollar los incrementos del software, que los cambios se realicen con más facilidad, que las pruebas y la depuración se efectúen con mayor eficiencia y que el mantenimiento a largo plazo se lleve a cabo sin efectos colaterales de importancia.

**Ocultamiento de información**

* Deben especificarse y diseñarse módulos, de forma que la información (algoritmos y datos) contenida en un módulo sea inaccesible para los que no necesiten de ella.
* La abstracción ayuda a definir las entidades de procedimiento (o informativas) que constituyen el software. El ocultamiento define y hace cumplir las restricciones de acceso tanto a los detalles de procedimiento como a cualquier estructura de datos local que utilice el módulo [Ros75].
* El uso del ocultamiento de información como criterio de diseño para los sistemas modulares proporciona los máximos beneficios cuando se requiere hacer modificaciones durante las pruebas, y más adelante, al dar mantenimiento al software.

**Independencia funcional**

* La independencia funcional se logra desarrollando módulos con funciones “miopes” que tengan “aversión” a la interacción excesiva con otros módulos. Dicho de otro modo, debe diseñarse software de manera que cada módulo resuelva un subconjunto específico de requerimientos y tenga una interfaz sencilla cuando se vea desde otras partes de la estructura del programa.
* La independencia se evalúa con el uso de dos criterios cualitativos: la cohesión y el acoplamiento. La cohesión es un indicador de la fortaleza relativa funcional de un módulo. El acoplamiento lo es de la independencia relativa entre módulos.
* Un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa.
* El acoplamiento es una indicación de la interconexión entre módulos en una estructura de software, y depende de la complejidad de la interfaz entre módulos, del grado en el que se entra o se hace referencia a un módulo y de qué datos pasan a través de la interfaz. En el diseño de software, debe buscarse el mínimo acoplamiento posible.

**Refinamiento**

* Un programa se elabora por medio del refinamiento sucesivo de los detalles del procedimiento. Se desarrolla una jerarquía con la descomposición de un enunciado macroscópico de la función (abstracción del procedimiento) en forma escalonada hasta llegar a los comandos del lenguaje de programación.
* En realidad, el refinamiento es un proceso de elaboración. Se comienza con un enunciado de la función (o descripción de la información), definida en un nivel de abstracción alto.
* **La abstracción y el refinamiento son conceptos complementarios.** La primera permite especificar internamente el procedimiento y los datos, pero elimina la necesidad de que los “extraños” conozcan los detalles de bajo nivel. El refinamiento ayuda a revelar estos detalles a medida que avanza el diseño. Ambos conceptos permiten crear un modelo completo del diseño conforme éste evoluciona.

**Aspectos**

* Idealmente, un modelo de requerimientos se organiza de manera que permita aislar cada preocupación (requerimiento) a fin de considerarla en forma independiente. Sin embargo, en la práctica, algunas de estas preocupaciones abarcan todo el sistema y no es fácil dividirlas en compartimientos.
* Cuando comienza el diseño, los requerimientos son refinados en una representación de diseño modular. Considere dos requerimientos, A y B. El A interfiere con el B “si se ha elegido una descomposición [refinamiento] en la que B no puede satisfacerse sin tomar en cuenta a A” [Ros04].
* Un aspecto es una representación de una preocupación de interferencia. Es importante identificar aspectos, de modo que el diseño les pueda dar acomodo conforme sucede el refinamiento y la división en módulos. En un contexto ideal, un aspecto se implementa como módulo (componente) separado y no como fragmentos de software “dispersos” o “regados” en muchos componentes [Ban06].

**Rediseño**

* Una actividad de diseño importante que se sugiere para muchos métodos ágiles es el rediseño, técnica de reorganización que simplifica el diseño (o código) de un componente sin cambiar su función o comportamiento.
* Cuando se rediseña el software, se examina el diseño existente en busca de redundancias, elementos de diseño no utilizados, algoritmos ineficientes o innecesarios, estructuras de datos mal construidas o inapropiadas y cualquier otra falla del diseño que pueda corregirse para obtener un diseño mejor.

**Conceptos de diseño orientados a objeto**

* El paradigma de la orientación a objeto se utiliza mucho en la ingeniería de software moderna.

**Clases de diseño**

* **El modelo de requerimientos define un conjunto de clases de análisis** (capítulo 6). Cada una describe algún elemento del dominio del problema y se centra en aspectos de éste que son visibles para el usuario. El nivel de abstracción de una clase de análisis es relativamente alto.
* Conforme el diseño evoluciona, se definirá un conjunto de clases de diseño que refinan las clases de análisis, dando detalles del diseño que permitirán que las clases se implementen y generen una infraestructura para el software que apoye la solución de negocios. Pueden desarrollarse cinco tipos diferentes de clases de diseño, cada una de las cuales representa una capa distinta de la arquitectura del diseño [Amb01]:
  + **Clases de usuario de la interfaz:** Definen todas las abstracciones necesarias para la interacción humano-computadora (IHC). En muchos casos, la IHC ocurre dentro del contexto de una metáfora (por ejemplo, cuaderno de notas, formato de orden, máquina de fax, etc.) y las clases del diseño para la interfaz son representaciones visuales de los elementos de la metáfora.
  + **Clases del dominio de negocios:** Es frecuente que sean refinamientos de las clases de análisis definidas antes. Las clases identifican los atributos y servicios (métodos) que se requieren para implementar algunos elementos del dominio de negocios.
  + **Clases de proceso:** Implantan abstracciones de negocios de bajo nivel que se requieren para administrar por completo las clases de dominio de negocios.
  + **Clases persistentes:** Representan almacenamientos de datos (por ejemplo, una base de datos) que persistirán más allá de la ejecución del software.
  + **Clases de sistemas:** Implantan las funciones de administración y control del software que permiten que el sistema opere y se comunique dentro de su ambiente de computación y con el mundo exterior.

**Definen cuatro características de las clases de diseño bien formadas:**

1. **Completa y suficiente:** Una clase de diseño debe ser el encapsulado total de todos los atributos y métodos que sea razonable esperar (con base en una interpretación comprensible del nombre de la clase) y que existan para la clase. La suficiencia asegura que la clase de diseño contiene sólo los métodos que bastan para lograr el objetivo de la clase, ni más ni menos.
2. **Primitivismo:** Los métodos asociados con una clase de diseño deben centrarse en el cumplimiento de un servicio para la clase. Una vez implementado el servicio con un método, la clase no debe proveer otro modo de hacer lo mismo.
3. **Mucha cohesión:** Una clase de diseño cohesiva tiene un conjunto pequeño y centrado de responsabilidades; para implementarlas emplea atributos y métodos de objetivo único.
4. **Poco acoplamiento:** Dentro del modelo de diseño, es necesario que las clases de diseño colaboren una con otra. Sin embargo, la colaboración debe mantenerse en un mínimo aceptable. Si un modelo de diseño está muy acoplado (todas las clases de diseño colaboran con todas las demás), el sistema es difícil de implementar, probar y mantener con el paso del tiempo. En general, las clases de diseño dentro de un subsistema deben tener sólo un conocimiento limitado de otras clases. Esta restricción se llama Ley de Demeter.

**EL MODELO DEL DISEÑO**

* El modelo del diseño puede verse en dos dimensiones distintas, como se ilustra en la figura 8.4. La dimensión del proceso indica la evolución del modelo del diseño conforme se ejecutan las tareas de éste como parte del proceso del software. La dimensión de la abstracción representa el nivel de detalle a medida que cada elemento del modelo de análisis se transforma en un equivalente de diseño y luego se mejora en forma iterativa.
* Los elementos del modelo de diseño usan muchos de los diagramas UML que se utilizaron en el modelo del análisis. La diferencia es que estos diagramas se refinan y elaboran como parte del diseño; se dan más detalles específicos de la implantación y se hace énfasis en la estructura y en el estilo arquitectónico, en los componentes que residen dentro de la arquitectura y en las interfaces entre los componentes y el mundo exterior.

**Elementos del diseño arquitectónico**

* El diseño de la arquitectura del software es el equivalente del plano de una casa. Los elementos del diseño de la arquitectura dan la visión general del software.
* Por lo general, el elemento de diseño arquitectónico se ilustra como un conjunto de sistemas interconectados, con frecuencia obtenidos de paquetes de análisis dentro del modelo de requerimientos. Cada subsistema puede tener su propia arquitectura.

**Elementos de diseño de la interfaz**

* El diseño de la interfaz para el software es análogo al conjunto de trazos (y especificaciones) detalladas para las puertas, ventanas e instalaciones de una casa. Los elementos de diseño de la interfaz del software permiten que la información fluya hacia adentro y afuera del sistema, y cómo están comunicados los componentes que son parte de la arquitectura.
* **Hay tres elementos importantes del diseño de la interfaz**:
  1. La interfaz de usuario (IU)
  2. Las interfaces externas que tienen que ver con otros sistemas, dispositivos, redes y otros productores o consumidores de información
  3. Interfaces internas que involucran a los distintos componentes del diseño. Estos elementos del diseño de la interfaz permiten que el software se comunique externamente y permita la comunicación y colaboración internas entre los componentes que constituyen la arquitectura del software.
* **El diseño de la usabilidad** incorpora elementos estéticos, elementos ergonómicos y elementos técnicos.
* **El diseño de interfaces externas** requiere información definitiva sobre la entidad a la que se envía información o desde la que se recibe. En todo caso, esta información debe recabarse durante la ingeniería de requerimientos y verificarse una vez que comienza el diseño de la interfaz.
* **El diseño de las interfaces internas** se relaciona de cerca con el diseño de componentes. Las realizaciones del diseño de las clases de análisis representan todas las operaciones y esquemas de mensajería que se requieren para permitir la comunicación y colaboración entre las operaciones en distintas clases.
* **En el UML se define interfaz**, en pocas palabras, como un conjunto de operaciones que describen alguna parte del comportamiento de una clase y dan acceso a aquéllas.

**Elementos del diseño en el nivel de los componentes**

* El diseño de componentes para el software describe por completo los detalles internos de cada componente. Para lograrlo, este diseño define estructuras de datos para todos los objetos de datos locales y detalles algorítmicos para todo el procesamiento que tiene lugar dentro de un componente, así como la interfaz que permite el acceso a todas las operaciones de los componentes (comportamientos).
* Los detalles del diseño de un componente se modelan en muchos niveles de abstracción diferentes.

**Elementos del diseño del despliegue**

* Los elementos del diseño del despliegue indican la forma en la que se acomodarán la funcionalidad del software y los subsistemas dentro del ambiente físico de la computación que lo apoyará.

Unidad 13

Diseño de webapps

**¿Qué es?**

* El diseño de webapps incluye actividades técnicas y no técnicas que incluyen: establecer la vista y sensación de la webapp, creando la distribución estética de la interfaz de usuario, definiendo la estructura arquitectónica general, desarrollando el contenido y la funcionalidad que residen en la arquitectura y planeando la navegación que ocurre dentro de la webapp.

**¿Quién lo hace?**

* Intervienen ingenieros web, diseñadores gráficos, desarrolladores de contenido y varios participantes más.

**¿Por qué es importante?**

* El diseño permite crear un modelo que se evalúe respecto de su calidad para mejorarlo antes de la generación de contenido y código, de la realización de las pruebas, **el diseño es el lugar donde se establece la calidad de la webapp.**

**¿Cuáles son los pasos?**

* El diseño de una webapp incluye **seis** etapas principales que son orientadas por la información obtenida durante el modelado de los requerimientos.
  1. Diseño del contenido.
  2. Diseño estético.
  3. Diseño arquitectónico.
  4. Diseño de la interfaz.
  5. Diseño de la navegación.
  6. Diseño de los componentes

**¿Cuál es el producto final?**

* El principal producto durante el diseño es un modelo de diseño que incluye aspectos de las **seis** etapas mencionadas en el punto anterior.

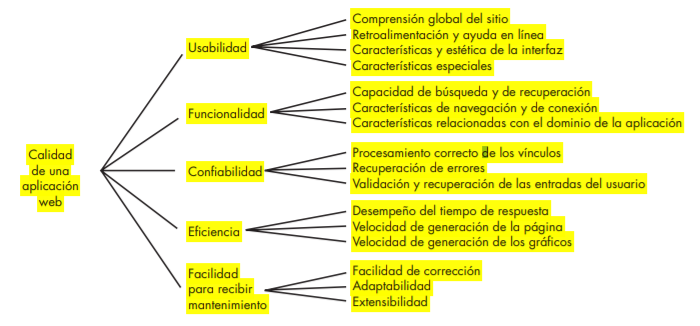
**¿Cómo me aseguro de que lo hice bien?**

* Cada elemento del modelo del diseño se revisa, además se toman en cuenta soluciones alternativas y se evalúa el grado en el que el modelo actual del diseño llevará a una implementación eficaz.

**Calidad del diseño de webapps**

Como ya se sabe, la calidad de un producto se determina en el diseño del mismo, por ende para webapps corren los mismos atributos generales y **cuatro** más que fueron agregados por Offutt, dando un total de **nueve** atributos que determinan la calidad de una webapp.

1. **Usabilidad**
2. **Funcionalidad**
3. **Confiabilidad**
4. **Eficiencia**
5. **Facilidad para recibir mantenimiento**
6. **Seguridad:** La medida clave de la seguridad de una webapp y de su ambiente de servidor es su capacidad para rechazar los accesos no autorizados o para detener un ataque proveniente del exterior.
7. **Disponibilidad:** Es la medida porcentual de tiempo en el cual la webapp puede utilizarse y **también** hay que tener en cuenta que “El empleo de características que solo se encuentren disponibles en un navegador o plataforma” también afectará a la disponibilidad porque el usuario se irá a otro sitio/webapp.
8. **Escalabilidad:** También es importante que pueda asimilar la carga del éxito (muchos más usuarios) y que tenga aún más éxito.
9. **Tiempo para llegar al mercado:** Si bien no es un atributo de calidad en el sentido técnico, la primera webapp que llega a un segmento específico del mercado frecuentemente es la que obtiene un número desproporcionado de usuarios finales.

****

**¿Cómo el usuario evalúa la calidad de una webapp?**

**Lista de comprobación:**

* ¿Es fácil determinar el alcance y la profundidad del contenido a fin de estar seguros de que satisface las necesidades del usuario?
* ¿Puede identificarse fácilmente la formación y la autoridad de los autores del contenido?
* ¿Es posible determinar la actualidad del contenido, la fecha de su última actualización y en qué consistió ésta?
* ¿El contenido y su ubicación son estables (permanecerán en la URL de referencia)?
* Además de estas preguntas relacionadas con el contenido, deben agregarse las siguientes:
  + ¿Es creíble el contenido?
  + ¿El contenido es único?, es decir, ¿la webapp brinda algún beneficio único a quienes la emplean?
  + ¿Es valioso el contenido para la comunidad de usuarios a la que se dirige?
  + ¿Está bien organizado el contenido? ¿Está indizado? ¿Se accede a él con facilidad?

**Metas del diseño:**

1. **Simplicidad:**  El contenido debe ser informativo pero sucinto y debe utilizar un modo de entrega que resulte apropiado para la información que se envíe. La estética debe ser agradable pero no abrumadora. La arquitectura debe lograr los objetivos de la webapp de la manera más sencilla posible. La navegación debe ser directa y sus mecanismos, obvios para la intuición del usuario final. Las funciones deben ser fáciles de utilizar y más fáciles de entender.
2. **Consistencia:** El contenido debe construirse de modo congruente. El diseño gráfico, debe presentar una vista consistente en todas las partes de la webapp. El diseño arquitectónico debe establecer plantillas que genera una estructura de hipermedios constante. El diseño de la interfaz debe definir modos consistentes de interacción, navegación y despliegue del contenido.
3. **Identidad:** El diseño de la estética, la interfaz y la navegación de una webapp debe ser consistente con el dominio de la aplicación para la que se va a elaborar.
4. **Robustez:** El usuario espera contenido y funciones robustas que sean relevantes para sus necesidades. Si no existen o son insuficientes, es probable que la webapp fracase.
5. **Navegación:** El usuario debe comprender como moverse por la webapp sin tener que buscar vínculos o instrucciones para la navegación.
6. **Atractivo visual:** La belleza radica sin lugar a dudas en los ojos del espectador, pero muchas características del diseño aumentan el atractivo visual.
7. **Compatibilidad:** Una webapp se usará en varios ambientes y debe diseñarse para que sea compatible con cada uno

**Diseño de la interfaz de la webapp:**

* Uno de los retos del diseño de la interfaz de las webapps es la naturaleza indeterminada del punto en el que entra el usuario, el diseño debe dar características de navegación en la interfaz que acompañen a todos los objetos de contenido y de las cuales se disponga sin importar el modo en el que le usuario ingrese al sistema.
* Los objetivos de la interfaz de una webapp son:
  1. Establecer una ventana congruente en el contenido y las funciones que brinda.
  2. Guiar al usuario a través de una serie de interacciones con la webapp.
  3. Organizar las opciones de navegación y contenido disponibles para el usuario.
* Se puede seleccionar uno de los siguientes mecanismos para implementar la navegación:
  1. Menús de navegación: Contienen palabras clave que enlistan contenido o funciones claves.
  2. Íconos gráficos: Botones, interruptores y otras imágenes similares que permiten que el usuario seleccione alguna propiedad o que especifique una decisión.
  3. Imágenes: Cierta representación gráfica que el usuario selecciona para establecer un vínculo hacia un objeto de contenido o función de la webapp.

**Diseño de la estética:**

* El diseño estético es una actividad artística que complementa los aspectos técnicos del diseño de las webapps.
* Aspectos de la distribución:
  1. No tema al espacio en blanco
  2. Hacer énfasis en el contenido
  3. Organizar los elementos con una distribución que vaya desde arriba a la izquierda hacia abajo a la derecha
  4. Agrupar la navegación, el contenido y la función en forma geográfica dentro de la página.
  5. No aumente la superficie con la barra de desplazamiento.
  6. Cuando se diseñe la distribución hay que considerar la resolución y tamaño de la ventana del navegador.

**Diseño del contenido:**

* Existen dos tareas diferentes del diseño del contenido:
  1. Se desarrolla una representación del diseño para los objetos del contenido y los mecanismos requeridos para establecer una relación entre ellos.
  2. Se genera el contenido que usará la webapp.

**Diseño arquitectónico:**

* Se divide en dos arquitecturas distintas:
  1. La arquitectura del contenido: Se centra en la manera en que los objetos del contenido se estructuran para la presentación y la navegación, existen **cuatro** arquitecturas de contenido:
     + Estructuras lineales
     + Estructuras de malla
     + Estructuras jerárquicas
     + Estructuras de red
  2. La arquitectura de la webapp: Describe una infraestructura que permite que un sistema o aplicación basados en web alcance sus objetivos empresariales. Se recomienda una arquitectura en tres capas que desacopla la interfaz de la navegación y del comportamiento de la aplicación.
     + El **modelo** contiene todo el contenido y la lógica de procesamiento específicos de la aplicación.
     + La **vista** contiene todas las funciones específicas de la interfaz y permite la presentación de contenido y lógica de procesamiento, incluido todos los objetos de contenido.
     + El **controlador** administra el acceso al modelo y la vista, y coordina el flujo de datos entre ellos.

El **controlador** maneja las solicitudes o datos del usuario. El controlador también selecciona el objeto de vista que sea aplicable con base a la solicitud del usuario, una vez seleccionado se transmite un pedido al **modelo**.

**Diseño de la navegación:**

* Una vez definida la arquitectura de la webapp deben definirse las rutas de navegación que permitan a los usuarios acceder al contenido y a las funciones de la webapp. Para lograr esto se debe hacer lo siguiente:
  1. Identificar la semántica de navegación para los distintos usuarios del sitio
  2. Definir la mecánica (sintaxis) para efectuar la navegación

**Diseño en el nivel de componentes:**

* Las webapps modernas dan funciones de procesamiento cada vez más complejas que:
  1. Realizan un procesamiento localizado para generar contenido y capacidad de navegación de forma dinámica
  2. Proporcionan capacidad de cómputo o de procesamiento de datos que resultan apropiados para el domino del negocio de la webapp
  3. Dan consulta y acceso complejos a bases de datos
  4. Establecen interfaces de datos con sistemas corporativos externos.

Se deben diseñar y construirse dichos componentes del a misma manera que los componentes del software tradicional.

**Diseño de Hipermedios Orientado a Objetos:**

* Es una de varias propuestas para hacer el diseño de webapps. El MDHOO sugiere un proceso que incluye:
  1. El diseño conceptual
  2. Diseño de navegación
  3. Diseño abstracto de la interfaz
  4. La implementación