Ciclo de vida del Software

- **Porque necesitamos una metodología?**
  Antes el programador realizaba un relevamiento de las solicitudes de quien necesitaba cierto programa o producto de software, y con los requerimientos bajo el brazo empezaba a programar. Esta tarea no estaba administrada, supervisada o gestionada de ningún modo. Los errores se iban corrigiendo a medida que surgían. Los programas crecieron en complejidad y esta vieja técnica de *code & fix* quedo obsoleta. Al no seguir normas para el proyecto, el cliente solo impartía especificaciones muy generales del producto que necesitaba. Se programa y corregía sobre la marcha..

- **Que es una metodología?**
  La metodología para el desarrollo del software es un modo sistemático de realizar, gestionar y administrar un proyecto para llevarlo a cabo con altas probabilidades de éxito. Son los procesos a seguir sistemáticamente para idear, implementar y mantener un software desde que nace la necesidad del producto hasta que cumplimos el objetivo para el cual fue creado

- **Que etapas tiene un ciclo de vida?**
  - Expresión de necesidades: Armado de un documento donde se reflejan los requerimientos y funcionalidades que tendrá el producto.
  - Especificaciones: Formalizamos los requerimientos obtenidos en el paso anterior
  - Análisis: Ya determinamos que elementos van a intervenir en el sistema que vamos a desarrollar. Su estructura, relaciones, funciones. Sabemos QUE producto vamos a construir
  - Diseño: Ya sabemos qué hacer. Ahora en esta etapa determinamos como hacerlo. Definimos en detalle entidades y relaciones de la BD, el lenguaje que usaremos y demás
  - Implementación: Empezamos a codificar los algoritmos y estructuras definidos anteriormente.
  - Debbugging/Prueba: El objetivo de esta etapa es garantizar que nuestro software no tiene errores de diseño o codificación. Deseamos encontrar la mayor cantidad de errores.
  - Validación: Verificamos si el software desarrollado cumple con los requerimientos expresados inicialmente por el cliente
  - Evolución y Mantenimiento: Se contempla en esta etapa la corrección de errores que pudiesen surgir (mantenimiento) como así también la incorporación de nuevas funcionalidades a nuestro software (evolución)

- **Tipos de ciclo de vida que conocemos:**
  - Ciclo de vida lineal: Es el más simple de los ciclos de vida. Consiste en etapas separadas de manera lineal. Cada etapa se realiza una sola vez, y se pasa a la etapa siguiente. Es rigido, y necesitamos contar con todos los requerimientos para empezar, ya que no podemos volver atrás, o es demasiado costoso hacerlo. Es apropiado para proyectos muy pequeños de ABM
  - Ciclo de vida en cascada puro: Es un ciclo de vida que admite iteraciones. De todas formas es un modelo rígido, con muchas restricciones. Como ventaja podemos decir que se puede entregar un software de calidad sin contar con personal altamente calificado, ya que antes de pasar a la etapa siguiente se realizar una o varias revisiones. Inconvenientes: contar con todos (o la mayoría) de los requisitos antes de empezar. Si se comete un error y no se detecta en la etapa siguiente es muy difícil y costoso volver atrás
  - Ciclo de vida en V : Las mismas etapas que el ciclo de vida en V, pero se le agrega dos subetapas de retroalimentación entre el análisis y mantenimiento y entre el diseño y el debugging que son la Validación, y la Verificación. Se usa en aplicaciones simples, pero que necesitan alta confiabilidad
  - Ciclo de vida tipo Sashimi: También parecido al ciclo de vida en cascada puro, con la diferencia que en este las etapas están solapadas. Aumenta su eficiencia porque las etapas se retroalimentan implícitamente en el modelo, lo que también ayuda a realizar un software de calidad. Las desventajas son producto del solapamiento también: es muy difícil gestionar el inicio

y el fin de cada etapa. Es una opción válida para desarrollar una aplicación que compartirá recursos con otras aplicaciones.

- Ciclo de vida en cascada con subproyectos: Cada una de las cascadas se divide en subproyectos o subetapas independientes que se pueden desarrollar en paralelo, lo que permite tener más gente trabajando al mismo tiempo. Ojo, pueden surgir dependencias entre las distintas subetapas o subproyectos. Se debe administrar muy bien.
- Ciclo de vida Iterativo: Modelo que buscar reducir el riesgo producido por malos entendimientos durante la etapa de solicitud de requerimientos. Se iteran varios ciclos en cascada. Al final de cada iteración se le entrega al cliente una versión del producto mejorada o con mas funcionalidades. Luego de cada iteración el cliente propone mejoras o correcciones. Se usa en proyectos donde los requerimientos no están claros de entrada. Lo utilizamos en aplicaciones medianas a grandes donde el cliente no necesita todas las funcionalidades desde principio. Ejemplo: migración de arquitectura de aplicaciones de una empresa (puede hacerlo paulatinamente)
- Ciclo de vida evolutivo: Hace una iteración de ciclos requerimientos-desarrollo-evaluación. Se adapta a requerimientos que van cambiando a través del tiempo, y en proyectos donde los mismos no están completos al comienzo. Ejemplo: Sistemas de stock-ventas-facturación
- Ciclo de vida incremental: Construye incrementando funcionalidades del sistema. Se realiza construyendo por módulos que cumplen las diferentes funciones del sistema, lo que permite ir aumentando gradualmente las capacidades del programa. Se aplica el ciclo de vida en cascada en cada funcionalidad del programa a construir. Se entrega al final de cada cascada (ciclo), una versión al cliente que contiene una nueva funcionalidad. Es una táctica de divide y conquistaras, donde si producimos pequeños sistemas es menos riesgoso y más simple que hacer uno grande. Además, no necesitamos disponer de todos los requerimientos al inicio del proyecto. Se usa cuando el usuario necesita entregas rápidas aunque sean parciales.
- Ciclo de vida en espiral: Se basa en una serie de ciclos repetitivos para ir ganando madurez con el producto final. Se tiene en cuenta el riesgo (debido a la incertidumbre sobre los requerimientos). A medida que el ciclo se cumple (el espiral avanza) se van obteniendo prototipos sucesivos que van ganando la satisfacción del cliente. La ventaja de este modelo es que puede comenzarse un proyecto con un alto grado de incertidumbre, y que hay bajo riesgo en el retraso de la detección de errores (ya que deberían solucionarse en la próxima rama del espiral). Desventajas: costo temporal que da cada vuelta del espiral, dificultad para evaluar riesgos y necesidad de una comunicación continua con el cliente. Se usa cuando necesitamos relevamientos exhaustivos en sistemas realmente grandes que utilizara mucha gente. Ejemplo: sistema que administre reclamos, pedidos e incidentes

Conclusión: Como elegir un ciclo? Va a depender de la complejidad del problema, el tiempo que tengamos para hacer una entrega final, si el cliente quiere o no entregas parciales, la comunicación que tenemos con el cliente, y la certeza o incertidumbre que tenemos sobre los requerimientos (son completos y correctos?)

Ingeniería de Requerimientos

- **Que es la ingeniería de requerimientos?**
  Es el proceso de recopilar, analizar y verificar las necesidades del cliente o el usuario. También se documentan todos los requisitos

- **Porque es tan importante la ingeniería de requerimientos?**
  Porque nos va a permitir gestionar el proyecto de forma estructurada (ya que consiste en una serie de pasos bien definidos), nos va a permitir plantear cronogramas de proyectos confiables (estimar costos, recursos, tiempos), también nos va a ayudar a disminuir los costos y retrasos de proyectos (errores encontrados a tiempos ahorran costos y tiempo) y por supuesto a construir un software de mejor calidad

- **Que son los requerimientos?**
  Un requerimiento es una condición o capacidad requerida que debe tener el sistema para resolver un problema, para satisfacer al cliente.

- **Como debe ser un requerimiento?**
  - Necesario: Su falta provoca deficiencia en el sistema a construir. No puede ser reemplazado
  - Conciso: Es fácil de leer y entender, su redacción es simple
  - Completo: No necesita ampliar detalles en su redacción. Proporciona la información suficiente
  - Consistente: No es contradictorio con otro requerimiento
  - No ambiguo: Tiene una sola interpretación. No debe causar confusiones su interpretación
  - Verificable: Se puede verificar mediante un proceso que el software cumple con ese requisito. Si no lo puedo verificar... como se que se cumplió con él o no?

- **Qué tipo de requerimientos conocemos?**
  - Funcionales: Definen las interacciones entre el sistema y su ambiente. Definen lo que el sistema va a ser capaz de realizar. Describen que hace el sistema, las transformaciones que produce sobre las entradas para producir salidas. Es importante que describa el QUE y no él COMO
  - No funcionales: Incluyen restricciones como el tiempo de respuesta o precisión. Tienen que ver con características que de una u otra forma puede limitar el sistema, como por ejemplo el rendimiento, la interfaces de usuario, etc.

- **Porque es tan difícil extraer los requerimientos?**
  Los requisitos son difíciles de extraer por varios motivos:
  - Provienen de fuentes diversas
  - Existen muchos, de diferentes tipos y niveles de detalle
  - Son difíciles de expresar en palabras
  - Muchas veces el cliente recuerda lo extraordinario, y no lo rutinario (req que nos van a faltar luego)
  - Un requerimiento puede cambiar a lo largo del ciclo de desarrollo

- **Entonces, cuales son las principales actividades que realiza la Ingeniería de Requerimientos?**
  - Extracción: Es donde descubrimos los requerimientos del sistema. Se debe trabajar junto al cliente para descubrir el problema a recibir, los servicios que debe prestar el sistema, las restricciones que debe presentar, etc.
  - Análisis: Se enfoca en descubrir problemas con los requerimientos hallados en la etapa anterior. Se leen los requerimientos, se conceptúan, investigan. Se resaltan los problemas
  - Especificación: Ahora si se documentar los requerimientos acordados con el cliente. Se puede decir que la especificación es "pasar en limpio" el análisis realizado previamente, aplicando técnicas y estándares.
  - Validación: La validación es la actividad de la IR que permite demostrar que los requerimientos definidos en el sistema son los que realmente quiere el cliente. Además tenemos que garantizar que todos los requerimientos especificados sigan los estándares de calidad

- **Como extraemos los requerimientos? Que métodos existen?**
  - Entrevistas y Cuestionarios: Se emplean para reunir información proveniente de personas o grupos. Las preguntas deber ser de alto nivel, para obtener información sobre aspectos globales del problema del usuario y soluciones potenciales. Debemos usar preguntas abiertas, para descubrir sentimientos, opiniones, experiencias o explorar un proceso o problema. Con este método podemos extraer una gran cantidad de requerimientos.
  - Lluvia de Ideas: Se usa para generar ideas. La intención es la de generar la máxima cantidad posible de requerimientos para el sistema. Los participantes deben pertenecer a muchas

disciplinas y preferentemente deben tener mucha experiencia. Cuantas más ideas se sugieren, habrá mejor resultados. En un principio toda idea es válida y ninguna debe ser rechazada. Se debe hacer en un ambiente relajado

- Prototipos: Es una técnica usada para validar requerimientos hallados. Son simulaciones de un posible producto que nos permiten una importante retroalimentación en cuanto si los requerimientos que hemos identificado son los que le permiten al cliente realizar su trabajo eficazmente. Puede ser usado como un medio para explorar nuevos requerimientos
- Casos de Uso: Son una táctica para especificar el comportamiento de un sistema. Permiten entonces, describir la posible secuencia de interacciones entre el sistema y uno o varios actores (usuario), en respuesta a un estimulo inicial proveniente del actor. La mayoría de los **requisitos funcionales** pueden expresarse mediante casos de uso. Describen lo que hace el actor, y lo que hace el sistema cuando interactúa con él. Como están expresados desde el punto de vista del actor, nos va a permitir obtener requisitos desde el punto de vista del usuario. Están acotados a una determinada funcionalidad del sistema, por lo que puede ser complicado si queremos hacer casos de uso para todas las funcionalidades de un sistema mediano/grande.

- **Especificación de Requisitos de software (SRS)**

  Es un documento que contiene una descripción completa de las necesidades y funcionalidades del sistema que será desarrollado. Describe el alcance del sistema, las funciones que realizará, definiendo requisitos funcionales y no funcionales.

  Es el resultado final del análisis y validación (o evaluación) de requerimientos. Los clientes y usuarios usan la SRS para comparar si lo que se está proponiendo coincide con las necesidades de la empresa. Los analistas y programadores utilizan el SRS para determinar el producto que debe desarrollarse. Las pruebas se van a elaborar también en base a este documento. La SRS, al igual que los requerimientos, debe ser: completa, precisa, consistente, no ambigua, verificable, concisa.

  Las características de la SRS deben ser validadas.

Diseño Estructurado

- **Que es el diseño? Y el diseño estructurado?**

  Diseñar significa planear la forma y el método de una solución. Es el proceso que determina las características principales del sistema final, establece los límites en performance y calidad. Se caracteriza por un gran número de decisiones técnicas.
  Diseño estructurado es el proceso de definir que componentes y la interconexión entre los mismos para solucionar un problema bien especificado

- **Porque hay que diseñar?**

  Porque en el diseño es donde se toman decisiones que afectaran finalmente al éxito de la implementación del programa, al igual que la facilidad de mantenimiento que tendrá el mismo (corrección de errores, evolución, etc.). El diseño es el proceso en el que se asienta la calidad del desarrollo del software. Sin diseño nos arriesgamos a un sistema inestable, que falle ante pequeños cambios, que pueda ser difícil de probar, de mantener.

- **Diseño y Calidad**

  Cuando diseñamos tendremos que buscar diseños que hagan uso inteligente de los recursos. Se dice que un sistema es más eficiente cuando hace un mejor uso no solo de memoria, sino de procesador, almacenamiento secundario, tiempo de periféricos y demás. Además trataremos de lograr un sistema confiable, libre de errores, que permita que se le introduzcan modificaciones (extensiones o correcciones) con facilidad, que se adapte al cliente y que sea fácil de usar para él.

Por lo tanto, apuntamos a diseñar un sistema de calidad. Esto es: confiable, mantenible, libre de error, modificable, amigable, útil.

- **Restricciones y Diseño**

  Las restricciones sobre el proceso de diseño de un sistema caen en 2 categorías:
  - Restricciones de Desarrollo: son las limitaciones al consumo de recursos. Estas pueden ser detalladas en horas hombre, tiempo de máquina, etc. Aquí también entran las restricciones de planificación y tiempo. Ejemplo: el modulo X debe estar para febrero
  - Restricciones Operacionales: Son restricciones técnicas, como ser máximo tamaño de memoria disponible, máximo tiempo de respuesta aceptable, etc.

  Generalmente estas restricciones no fijan límites rígidos, sino un intervalo de tolerancia, mediante el cual el diseñador puede moverse

- **Estrategias del diseño estructurado. Como deben ser los módulos o partes del sistema?**

  El diseño exitoso se baja en el viejo principio divide y conquistaras.
  El costo de implementación de un sistema, se minimizara cuando pueda separarse en partes manejablemente pequeñas, y solucionables separadamente.
  El mantenimiento de un sistema, también se verá minimizado cuando sus partes son: fácilmente relacionables con la aplicación, manejablemente pequeñas y corregibles separadamente.
  Para minimizar los costos de mantenimiento debemos lograr que cada pieza sea independiente de otra (bajamente acopladas)

  Podemos afirmar entonces que los costos de implementación, mantenimiento y modificación serán minimizados, cuando cada pieza del sistema corresponda a exactamente una pequeña, bien definida pieza del dominio del problema, y cada relación entre las piezas del sistema corresponde a relaciones de piezas en el dominio del problema

- **Complejidad. Como manejarla**

  El costo de resolver un problema es directamente proporcional a la complejidad y tamaño del mismo. Es preferible crear dos piezas pequeñas que una sola grande, si ambas solucionan el problema. A medida que los módulos sean más pequeños, podemos esperar que su complejidad y costo disminuyan, pero también tendremos mayor posibilidad de errores debidos a la conectividad intermodulos.

- **La estructura de un programa. Que es un modulo? Interface de modulo**

  Un modulo es una secuencia lexicográficamente continua de sentencias encerrada entre elementos de frontera y que posee un identificador del conjunto de dichas sentencias. Una interface define un límite de un modulo, se lo activa a través de ella y por allí fluyen datos y control.
  Un modulo puede ser identificado y activado por medio de una interfaz simple.

- **Conexiones normales y patológicas**

  Entre 2 módulos existe una conexión normal cuando la conexión se produce a través de alguna interfaz definida formalmente. Si la conexión intermodular se realiza a un identificador de un elemento interno del modulo invocado sin pasar a través de una interfaz, diremos que esta es una conexión patológica.

- **Acoplamiento. Qué es?**

  Dos módulos son totalmente independientes si puede funcionar uno sin el otro. A mayor número de interconexiones entre módulos tendremos una menor independencia
  El acoplamiento nos indica el grado de dependencia intermodular .

- **Qué factores influyen en el acoplamiento?**
    - Tipo de conexión entre módulos: Las referencias intermodulares deben realizarse a través de sus interfaces. Si un sistema esta mínima o normalmente conectado, va a estar menos acoplado que un sistema que tiene módulos patológicamente conectados
    - Complejidad de la interface: Cuanto mas compleja es una conexión, mayor acoplamiento se tiene. La interface puede ser compleja por la cantidad de información (numero de argumentos o parámetros que son pasados en la llamada al modulo). La interfaz es menos compleja si la información es presentada localmente dentro de la misma sentencia de llamada que si la información necesaria es remota. Además, tenemos que tener en cuenta que la estructura de la información también puede ser un punto calve en la complejidad (la información es menos compleja si se presenta en forma lineal que si se presenta en forma anidada)
    - Tipo de flujo de la información en la conexión: Esto tiene que ver con el tipo de información que se transmite entre el modulo superior y el subordinado.
        - Acoplamiento por datos: Se presenta si la salida de datos del modulo superior es usada como entrada de datos del subordinado. Es básico y está en todo sistema
        - Acoplamiento de control: En este caso el modulo superior comunica al subordinado información que controlara la ejecución del mismo. Puede pasarse como datos utilizados como banderas o como direcc de memoria para salto condicional, por ejemplo
        - Acoplamiento hibrido: En este caso para el modulo de destino el acoplamiento es visto como de control, mientras que para el modulo llamador es considerado como de datos. El grado de interdependencia entre 2 módulos híbridamente acoplados es muy fuerte
    - Momento en que se produce el ligado: Cuando un valor de una variable dentro de una pieza de código son fijados más tarde, el sistema es mas fácilmente modificable y adaptable a cambios. Una referencia intermodular fijada en tiempo de definición tendrá un acoplamiento mayor que una referencia fijada en tiempo de traslación. La posibilidad de compilación independiente de un modulo va a facilitar el mantenimiento y la modificación del sistema.
    - Acoplamiento por contenido: Un modulo esta incluido dentro de otro; es decir que un modulo no puede funcionar sin otro

- **Cohesión. Qué es?**

La cohesión es la medida cualitativa de cuan estrechamente están relacionados los elementos internos de un módulo. Tenemos que tener en cuenta que un sistema modularmente mas efectivo será aquel cuya relación funcional entre pares de elementos que pertenezcan a diferentes módulos sea mínima; ya que tiene a minimizar el número de conexiones intermodulares y el acoplamiento intermodular.
Cohesión y acoplamiento están íntimamente relacionados. Un mayor grado de cohesión implica un menor grado de acoplamiento

- **Niveles o tipos de cohesión que existen**
    - Cohesión casual: existe poca o ninguna relación entre los elementos de un modulo. Puede aparecer como consecuencia de la modularización de un programa ya escrito.
    - Cohesión lógica: Los elementos de un módulos están lógicamente asociados si pueden pensarse en ellos como pertenecientes a la misma clase lógica de funciones. Ejemplo: Se pueden combinar en un modulo simple todos los elementos de procesamiento que caen en la clase "entradas", que abarca todas las operaciones de entrada. Un modulo lógicamente cohesivo no realiza una función específica, sino que realiza una serie de funciones
    - Cohesión temporal: todos los elementos de procesamiento de una colección ocurren en el mismo periodo de tiempo durante la ejecución del sistema. Un ejemplo común son las rutinas de inicialización comúnmente encontradas en la mayoría de los programas
    - Cohesión de procedimiento: los elementos de procesamiento están relacionados proceduralmente en una unidad común. Un ejemplo puede ser un proceso de iteración (loop) y de decisión o una secuencia lineal de pasos. Se tiene este nivel de cohesión cuando se deriva una estructura modular a partir, por ejemplo, de DFDs

- Cohesión de comunicación: todos los elementos de procesamiento operan sobre el mismo conjunto de datos de entrada o de salida. Los DFD son un medio objetivo para determinar si los elementos de un modulo están asociados por comunicación. Este tipo de cohesión es aceptable. Ejemplo: Modulo que imprima o grabe un archivo de transacciones

- Cohesión secuencial: En ella los datos de salida de un elemento sirven como datos de entrada al siguiente elemento de procesamiento. Este claramente es un principio asociativo relacionado con el dominio del problema
- Cohesión funcional: Cada elemento en un modulo funcional es parte integral de y esencial para la realización de un principio asociativo relacionado con el dominio del problema.
  Este último tipo de cohesión es la mejor. Módulos altamente cohesivos funcionalmente y poco acoplados es lo más recomendable.

- **Heurísticas del diseño. Tamaño de los módulos. Fan-in y Fan-out**

Para la mayoría de los propósitos módulos de mucho mas de 100 sentencias están fuera del tamaño optimo en lo que respecta a la economía en detección y corrección de errores.
De todas formas no siempre módulos muy grandes o muy pequeños son necesariamente malos.
Lo importante es que los módulos reflejen la estructura del problema lo mas fielmente posible.
Descomponer un modulo simplemente para llevarlo a un tamaño optimo, aun perdiendo su significado respecto a la estructura del problema es algo que no tiene mucho sentido.

Módulos muy grandes pueden deberse a dos razones principales:

- Factorización incompleta en módulos subordinados apropiados
- Dos o más funciones han sido combinadas (frecuentemente con cohesión lógica) en un mismo modulo

Para el tratamiento de módulos muy pequeños debemos distinguir si se trata de un modulo atómico o no atómico. Módulos atómicos pueden ser comprimidos en módulos superiores (siempre y cuando el grado de entrada a ese modulo no sea muy elevado, ya que esto puede ser muy peligroso si el fan-in es grande). Cuando el modulo es no atómico el análisis es más complicado ya que deberemos ver si podemos comprimir el modulo hacia arriba (en su superior) o hacia abajo (en algún subordinado por el)

La amplitud de control es el numero de subordinados inmediatos de un modulo (fan-out). Amplitudes de control muy altas o muy bajas indican pobre diseño. Deben verificarse anchos de salida mayores de 10 y menores que 2. Por supuesto, una amplitud de control alta es más peligrosa que una baja.

El ancho de entrada (fan-in), siempre que sea posible, desearemos maximizarlo. Esto es una señal de que hemos utilizado bien este modulo (o función) y podemos evitar también asi duplicidad de código. Cada vez que vamos a dibujar un nuevo modulo en el diagrama de la estructura, primero debemos ver si ya otro modulo no realizar la función requerida. La especificación del fan-in es una tarea del diseñador y no del implementador

- **Alcance de efecto. Alcance de control**

El alcance de efecto de una decisión es la colección de todos los módulos que contienen algún procesamiento que está condicionado por dicha decisión
El alcance de control de un modulo es el modulo mismo y todos sus subordinados.

Para una decisión dada, el alcance de efecto deber ser un subconjunto del alcance de control en el cual se encuentra la decisión. Es decir, todos los módulos que son afectados o influenciados por una determinada decisión, deben estar subordinados por ese modulo que tomo la decisión
En el caso ideal el alcance de efecto debe estar limitado al modulo en el cual se realiza la decisión y sus módulos subordinados inmediatos.

Estrategias de Prueba del Software

- **En qué consiste una estrategia de prueba de un software?**
  La prueba es un conjunto de actividades que se planean con anticipación y se realizan de manera sistemática. Cualquier estrategia de prueba debe incorporar la planeación de pruebas, el diseño de caso de pruebas, la ejecución de las pruebas y la recolección y evaluación de los datos resultantes.

- **Verificación y Validación**
  Verificación es el conjunto de actividades que aseguran que el software implemente correctamente una función específica.
  Validación es un conjunto diferente de actividades que aseguran que el software construido corresponde con los requisitos del cliente

  Las pruebas son el último bastión para la evaluación de la calidad, y para el descubrimiento y corrección de errores. Pero la calidad debe incorporarse a lo largo de todo el proceso de ingeniería, a través de la aplicación apropiada de métodos y herramientas, revisiones técnicas, y mediciones solidas.

- **Organización para las pruebas de software. Quien prueba?**
  El desarrollador de software siempre será el responsable de probar las unidades individuales del programa y asegurar que cada una realice su función para la cual se la diseño. En muchos casos el desarrollador también aplica la prueba de integración. Solo después de que la arquitectura de software esta completa participara un grupo independiente de prueba. El papel de este grupo consiste en eliminar los problemas propios de dejar que el constructor pruebe lo que el mismo ha construido; es decir, elimina el conflicto de intereses que pudiera haber

  Error común: Pensar que el equipo de pruebas es responsable de asegurar calidad. Si no hubo calidad a lo largo de todo el proceso de ingeniería, no aparecerá mágicamente en la prueba

- **Estrategia de prueba para arquitecturas convencionales de software. Como debemos probar?**
  Al principio la prueba se concentra en cada componente individual, asegurando que funciona de manera apropiada como unidad. Enseguida deben ensamblarse o integrarse los componentes para formar el paquete de software completo. La prueba de integración atiende los aspectos asociados con el doble problema de verificación y construcción del programa. Después de que se construido el software se ejecutan pruebas de alto nivel. Se deben evaluar los procesos de validación establecidos durante el análisis de requisitos, para saber si el software cumple con todos los requisitos funcionales, de comportamiento y desempeño

  Orden de las pruebas:

  - Prueba de Unidad: se concentran en la lógica del procesamiento interno y en las estructuras de datos dentro de los límites de un componente. Debemos probar la interfaz para asegurar que la información fluye apropiadamente hacia dentro y hacia afuera del modulo.
  - Prueba de Integración: es una técnica sistemática para construir la arquitectura del software al mismo tiempo que ese aplican pruebas para descubrir errores asociados a la interfaz. EL objetivo es tomar unidades ya probadas e integrarlas para construir una estructura de programa determinada en el diseño
    - Descendente: se integran los módulos al descender por la jerarquía de control, empezando con el modulo principal. Ventaja: verifica los ppales puntos de control al inicio (en un sistema bien diseñado). Desventaja: muchas veces se requiere procesamiento en los niveles inferiores de jerarquía para probar niveles superiores adecuadamente
    - Ascendente: Empieza la construcción y prueba por los módulos atómicos. Ventaja: siempre esta disponible el procesamiento requerido para los componentes subordinados a

un determinado nivel. Desventaja: necesidad de controladores (programa de control para pruebas), con el fin de coordinar entradas y salidas durante las pruebas

- *Prueba de Humo:* Es un enfoque de prueba de Integración. Esta diseñado como mecanismo para marcar el ritmo en proyectos en los cuales el tiempo es crítico, ya que permite que se evalúe el software frecuentemente. Como funciona? Los componentes ya codificados se integran a una construcción. Se prueba esa construcción y luego se integra con otras construcciones. Diariamente se aplica una prueba de humo a todo el producto (en su forma actual)

- **Documentación de la prueba de Integración. *Especificación de la prueba*.**
  Un plan integral para la integración del software y una descripción de pruebas especificas se documentan en la *Especificación de la prueba*
  El plan de prueba describe la estrategia general de integración, el orden de integración y las pruebas correspondientes en casa paso de integración. Además se incluyen una lista de todos los casos de prueba y los resultados esperados.
  Una historia de resultados, problemas o peculiaridades de las pruebas reales se registra en el Informe de Prueba.

- **Prueba de validación**: empiezan luego de las pruebas de integración. La validación se alcanza cuando el software funciona de tal manera que satisface las expectativas razonables del cliente (expectativas redactadas como requisitos en la SRS)
  - Pruebas alfa y beta: Son pruebas en las que participa el usuario. En las pruebas alfa el software se utiliza en un entorno natural mientras el desarrollador observa y registra los errores y problemas de uso. En las pruebas beta el software se utiliza en el lugar de trabajo de los usuarios finales, y son ellos mismos quienes anotan y registran todos los problemas que pudieran surgir

- **Prueba del sistema:** Al final el software se incorpora a otros elementos del sistema como hardware y personas, y se realiza una serie de pruebas de integración del sistema y validación. Se trabaja para verificar que se hayan integrado adecuadamente todos los elementos del sistema y que se realizan las funciones apropiadas

- **Depuración. El arte de depurar. Porque es tan difícil depurar?**
  La depuración no es una prueba, pero siempre ocurre como consecuencia de una. Se evalúan los resultados y se encuentra una falta entre el desempeño esperado y el real. La depuración trata de corregir este error.
  Es difícil depurar por varias razones:
    - El síntoma y la causa pueden estar en diferentes parte del programa
    - Podría deberse a un error humano difícil de localizar
    - Podría deberse a un problema de tiempo y no de procesamiento

Gestión de la Calidad

- **Que es la gestión de la calidad de un software?**
  La gestión de la calidad de un software es una actividad protectora que incorpora tanto aseguramiento como control de la calidad. Se aplica a cada paso de la ingeniería del software

- **Tipos de calidad que conocemos:**
  - Calidad de diseño: la calidad de diseño se refiere a las características que los diseñadores especifican para un elemento. Incluye requisitos, especificaciones y el diseño del sistema.

- Calidad de concordancia: Tiene que ver con el grado en el que las especificaciones de diseño se aplican durante la implementación. Si esta sigue el diseño y el sistema satisface los requisitos y metas, la calidad de concordancia es alta.

- **La SQA. Que tareas realiza?**
  La SQA abarca procedimientos para la aplicación eficaz de métodos y herramientas, revisiones técnicas formales, estrategias, métodos de prueba, procedimientos para garantizar la concordancia con los estándares y mecanismos de medición y reporte

- **Revisiones del Software:** Son una de las actividades de control de calidad más importantes. Sirven como filtros, que eliminan errores mientras son relativamente poco costosos de encontrar y corregir.
  El objetivo principal de las revisiones técnicas formales es descubrir los errores durante el proceso, de modo que no se conviertan en defectos después de liberar el software

Diccionario de Datos y DFD:

El diagrama de flujo de datos (DFD) nos permite modelizar el sistema, es decir, vamos a poder ver como la información fluye a través del sistema. También se observa en un DFD las transformaciones que aplica el sistema.

En un DFD tenemos:

- Entidades externas: Son quienes producen o consumen los datos. Representan una fuente de destino o información.
- Procesos: Son los componentes funcionales del sistema. Transforman los datos. Se pude ver como una función del sistema
- Almacenes de datos: Representan información almacenada. Si son estructuras simples, luego seguramente la almacenaremos en registros. Si son estructuras complejas, resultara en algún diagrama E/R
- Flujo de datos: Representan datos en movimientos

En el DFD los datos juegan un rol importante; por eso debemos representar sus características de forma organizada, en un **Diccionario de Datos**

El Diccionario de datos es:

- Una lista organizada de todos los elementos de datos que representamos en el DFD
- Debe tener definiciones precisas y rigurosas
- Detalla también las relaciones entre los datos
- Representa datos sobre datos (metadatos)

Para que sirve el Diccionario de datos?

- Describe el significado de los flujos y almacenamientos del DFD
- Describe la composición de los paquetes de datos
- Especifica los valores y unidades relevantes de piezas elementales de información, tanto de los flujos de datos como de los almacenamientos

El diccionario de datos es un complemento del DFD, en el cual debemos incluir la estructura de los datos, información sobre los flujos, sobre los datos elementales y los procesos.