REQUERIMIENTOS FUNCIONALES Y NO FUNCIONALES

"la Ing. del software me provee las herramientas para q yo pueda hacer un software de calidad"

la Ing. del soft Es la aplicación de un enfoque sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento de software.

Stakeholders es alguien q se ve afectado por el software

Universo de información (universo de discurso o dominio de aplicación): es el dominio del problema, toda la información el vocabulario, el proceso de organización…. Toda la información q se utiliza para realizar el software

Clasificación de requerimientos:

Requerimientos Funcionales (RF): describen la funcionalidad o los servicios que se espera que el sistema proveerá. ◦ Cuando se expresan como requerimientos del usuario, se describen de forma general; mientras que los requerimientos funcionales del sistema describen con detalle la función de éste, sus entradas y salidas, excepciones, etc.

Requerimientos No Funcionales (RNF): ◦ se refieren a las propiedades emergentes del sistema. ◦ definen las restricciones del sistema: capacidad de los dispositivos de entrada/salida, representación de datos que se utiliza en las interfaces, etc.
Calidad: confiabilidad, disponibilidad, robustez...
Factores humanos: facilidad de uso, simplicidad de las interfaces,
Características de rendimiento: tiempos de respuesta, performance, ...
Restricciones de hardware y software: compatibilidad con equipamiento y/o sistemas disponibles, ...
Cambios y/o adaptaciones a nuevos requerimientos: adaptabilidad, reusó de componentes, ... Restricciones de seguridad.

Requerimientos Inversos (RI): definen cómo el software nunca se debe comportar.

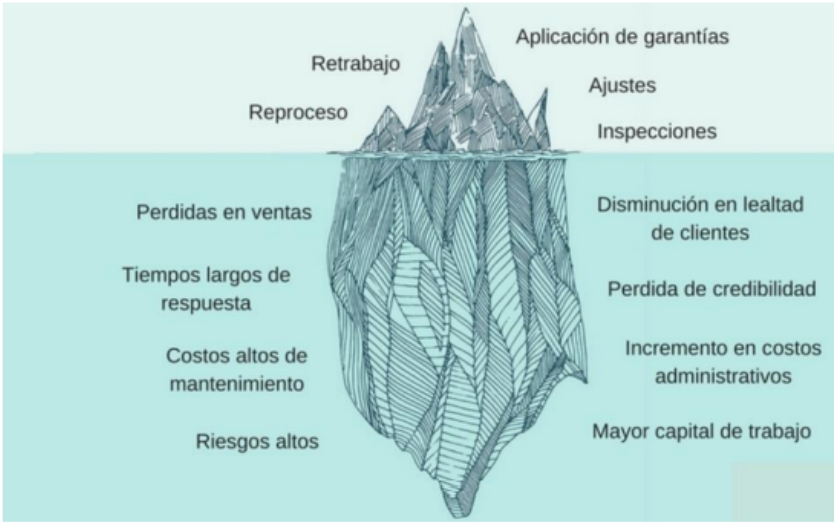
Los requerimientos no funcionales se refieren al sistema como un todo más que a rasgos particulares del mismo; a menudo son más críticos que los requerimientos funcionales particulares. } Mientras que el incumplimiento de un requerimiento funcional degradará al sistema, una falla en un requerimiento no funcional del sistema lo inutiliza.

El software es de calidad? ¿Hace lo que el usuario quiere? ¿Le soluciona el problema que intenta resolver? ¿Lo hace como el cliente quiere? ¿Es factible de construir?, ¿de corregir, de expandir, de mejorar? ¿Se lo puede construir

rápido, barato y en forma segura? Fue modificado, ¿Sigue funcionando bien? ¿Le gusta al cliente?

El software no es de calidad cuando… Cada cambio es difícil de introducir No se entiende el código “Es feo”, lento, difícil de usar, complejo No se sabe cómo probar que funciona bien. Sólo una persona puede arreglarlo Reinventa la rueda El sistema no hace algo que el usuario pidió Hace mal algo Hace algo que el usuario no pidió Es fácil cometer errores con él

Costos visibles e invisibles relativos a la calidad del software



CASOS DE USO

CASO DE USO PRIMARIO: son ejecutados de forma directa o especifica por un actor. Modelan el problema central. Representan alguna función fundamental del sistema

CASO DE USO SECUNDARIO: Son invocados o ejecutados por otro caso de uso. Surgen luego de una “explotación” de los casos de uso primarios. Tienen mayor nivel de detalle mediante relaciones de “inclusión” o “extensión”. Representan alguna variación respecto de uno primario.

CU TEMPORALES: el tiempo dispara estos CU. ej: Backup automatico que se ejecutara al final del dia

COMPONENTE DE LOS CU:

Límite del sistema: Define el alcance del sistema. Nombre.

Actores: Roles desempeñados por personas o elementos que utilizan el sistema. Se representan afuera del límite del sistema. No necesariamente

representa a una persona en particular, sino más bien la labor que realiza frente al sistema.

Casos de Uso: Describen interacciones de los actores con el sistema. Es una operación/tarea específica que se realiza tras una orden de algún agente externo

Relaciones: relaciones significativas entre actores y casos de uso, o entre casos de uso. Representada por una línea entre los casos de uso y/o actores relacionados

Diferencia entre actor y usuario:

Un actor es el rol o papel que juega un objeto externo en su relación con el sistema. Un usuario es una persona que, cuando usa el sistema, asume un rol. Un usuario puede acceder al sistema como distintos actores.

INCLUDE

Relación de inclusión <>: ◦ Los casos de uso pueden contener la funcionalidad de otro caso de uso como parte de su proceso normal. ◦ Los casos de uso pueden ser incluidos por uno o más casos de uso. ◦ Un caso de uso incluido se llamará cada vez que se ejecute una ruta básica. ◦ Ayuda a reducir el nivel de duplicación de la funcionalidad realizando un factoreo del comportamiento común en casos de uso que se usan muchas veces. La flecha va desde el CU que incluye hacia el CU incluido

Relación de extensión <>: ◦ Agrega nuevo comportamiento a un CU existente ◦ Refleja situaciones particulares en un CU que pueden ser tratadas (extendidas) por otro. ◦ En la descripción del caso de uso que es extendido debe haber una forma de indicar en que punto entra en juego el caso de uso que lo extiende (punto de extensión). La flecha va desde el CU que extiende al extendido

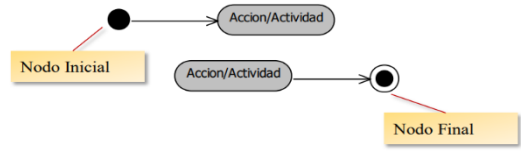
Relación de generalización o herencia: La generalización se usa para indicar herencia. El actor hereda características del origen.

HISTORIA DE USUARIO

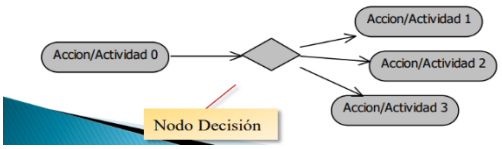
Una historia de usuario (HU) describe una funcionalidad que será valiosa para un usuario o cliente de un sistema o software.

Cuando una historia es demasiado extensa, a veces se la denomina épica. Las épicas pueden dividirse en dos o más historias de menor tamaño

Métodos agiles: Conjunto de métodos en el que las necesidades y soluciones evolucionan a través de la colaboración entre equipos multifunción y autoorganizados.



La HU busca cambiar el enfoque de "escribir" a "fomentar la conversación" sobre requerimientos características. }Se debe describir el rol, la funcionalidad y el resultado de aplicación en una frase corta. } Se incorporan criterios de aceptación, más de 4 por historia, incluyendo contexto, el evento y el comportamiento esperado frente a ese evento.



y la no

Se componen de tres aspectos:

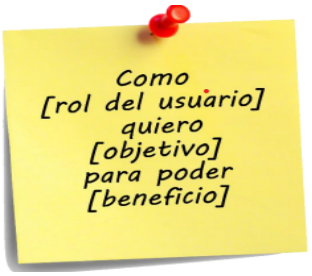
Una descripción escrita de la historia: se utiliza para la planificación y como recordatorio

Conversaciones sobre la historia: Sirven para concretar los detalles de la misma

Pruebas que transmiten y documentan los detalles: Pueden utilizarse para determinar cuándo se ha completado una historia

EJ: **Como** cliente del banco
Quiero retirar dinero de mi cuenta
Para hacer compras con efectivo

PUEDEN SER INCOMPLETAS.



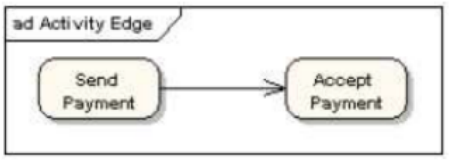
Un **«release plan» o plan de proyecto** es un conjunto de historias de usuario (normalmente épicas) agrupadas por «releases» o versiones del producto que se ponen a los disposición de usuarios incrementando el valor para estos respecto de la anterior

DIAGRAMA DE ACTIVIDADES

Permiten modelar un proceso como una actividad que se compone de una colección de nodos conectados por arcos.

Las actividades pueden tener precondiciones y pos condiciones como los casos de uso: Ej:
Precondición: saber tema de carta
Poscondición: carta enviada a dirección

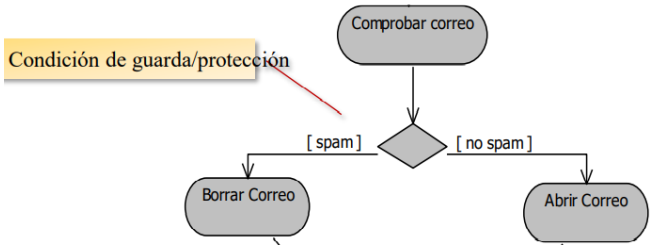
Las transiciones reflejan el paso de un estado a otro, bien sea de actividad o de acción. Se produce como resultado de la finalización del estado del que parte el arco dirigido que marca la transición

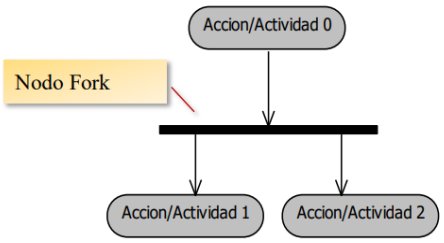


Nodo inicial: Indica dónde empieza el flujo cuando se invoca una actividad

Nodo final: Termina una actividad

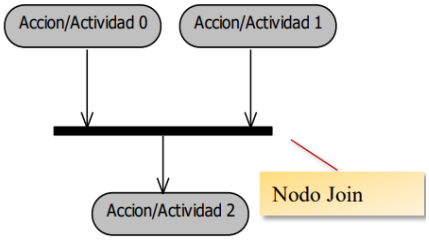
Nodo decisión: Posee un extremo de entrada y dos o más de salida. Un token que llega al extremo de entrada se ofrecerá a todos los extremos de salida, atravesando uno. Cada uno de los extremos condición de protección (guarda), que debe evaluar la verdadera.

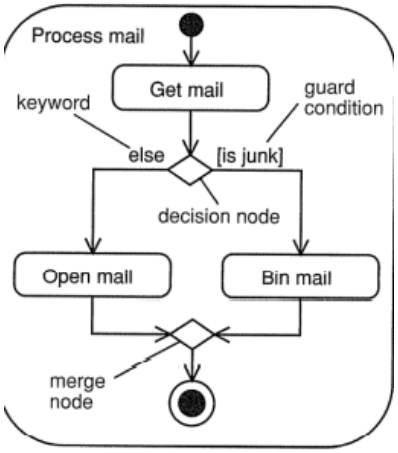




Nodo fork: Divide el flujo en múltiples flujos concurrentes ◦ Tiene un extremo de entrada y dos o más de salida paralelos

Nodo Join: Múltiples extremos entrantes y uno solo de salida ◦ Sincroniza flujos





Nodo Merge: Posee dos o más arcos de entrada y un único arco de salida. Fusionan los flujos de entrada en un flujo de salida simple

diagrama de clases DISEÑO ORIENTADO A OBJETOS

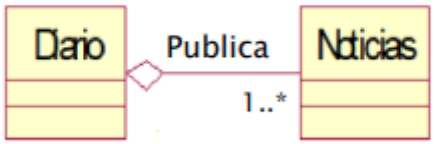
Los principios básicos son:
Modularización: ◦ Módulos fáciles de manejar ◦ Comprenden las estructuras de datos y las operaciones.
Encapsulado: ◦ Distingue entre la interfaz de un objeto (qué es lo que hace), de la implementación (cómo lo hace).
Tipos de datos abstractos: ◦ Agrupa todos los objetos que tienen la misma interfaz y los trata como si fueran del mismo tipo
Herencia: ◦ Reutilización.

Los elementos de un diagrama de clases son: ◦ Clases, atributos, operaciones ◦ Relaciones, cardinalidad, roles
Un objeto es una instancia (u ocurrencia) de una clase
Clase ◦ Una clase es una descripción de un grupo de objetos que tienen:
*Propiedades similares (atributos)
* Comportamiento común (operaciones y diagrama de estado) y semántica común.
*Establece el mismo tipo de relaciones con otros componentes del modelo
Atributo ◦ Un atributo es una propiedad de una clase a la que se le asigna un nombre y que contiene un valor para cada objeto de la clase
Identificación de clases:
Identificar objetos esenciales, las descripciones de casos de uso colaboran en la identificación de objetos y operaciones del sistema.
Análisis gramatical de una descripción en lenguaje natural del sistema a ser construido.
Objetos son sustantivos.
Operaciones son verbos.
Metodos: Una operación es una función o procedimiento que puede ser ejecutada por un objeto o aplicada a un objeto. Cada operación actúa sobre un objeto que es su argumento implícito. Al nombre de la operación se le puede

agregar detalles tales como la lista de argumentos o el tipo de resultado. La implementación de una operación para una clase se denomina método
Relación: Una relación es una conexión semántica (significativa) entre elementos del modelo.
◦ Se le puede dar un nombre, se le puede asignar una cardinalidad y se pueden establecer roles que juegan las clases relacionadas
◦ Existen diferentes tipos de relaciones: Herencia, Asociación Agregación Composición

Herencia: ◦ Las propiedades heredadas pueden reutilizarse o redefinirse en la subclase.
◦ La subclase puede definir nuevas relaciones no presentes en la superclase.
◦ El caso en que una subclase tiene múltiples superclases inmediatas se denomina herencia múltiple
Asociación: ◦ Relación entre clases.
◦ Semántica: una asociación entre clases indica que se puede tener vínculos entre objetos de esas clases.
Asociación reflexiva ◦ Objetos de la clase tienen vínculos con objetos de la misma clase

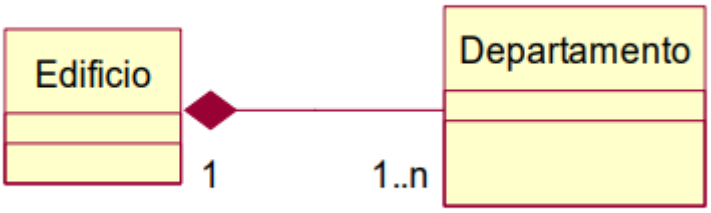
Agregación ◦ Relación no simétrica
◦ Permite modelar una relación parte-todo en la cual un objeto es propietario de otro objeto pero no en exclusividad.
◦ Uno de los extremos cumple un rol predominante respecto del otro extremo.
◦ No realiza ninguna afirmación sobre el ciclo de vida de las partes involucradas
◦ Semántica: El conjunto pude existir independientemente de las partes.
Las partes pueden existir independientemente del conjunto.
El conjunto está incompleto si falta alguna de las partes.
Es posible tener propiedad compartida de las partes por varios conjuntos



Si el diario muere las noticias siguen existiendo

Composición ◦ Relación no simétrica
◦ Permite modelar una relación parte-todo en la cual un objeto es propietario de otro objeto en exclusividad.
◦ Uno de los extremos cumple un rol predominante respecto del otro extremo
Semántica: Las partes con multiplicidad variable pueden ser creadas luego del

todo una vez creadas no podrán persistir si el todo desaparece. Expresa una relación de pertenencia
Asocia los ciclos de vida del todo y la parte.
Las partes pueden solamente pertenecer a un conjunto.
Tiene la responsabilidad para su creación y destrucción
Si se destruye el conjunto, debe destruir todas sus partes.

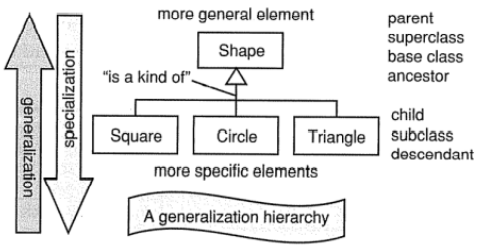


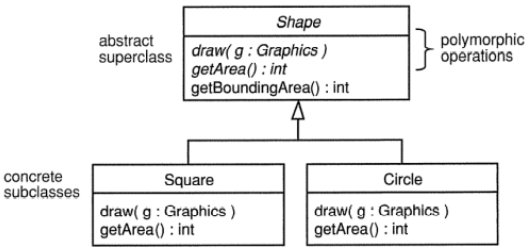
Multiplicidad: ◦ Especifica la cantidad de objetos que pueden participar en una relación en cualquier momento.

Rol ◦ El rol de un objeto indica el papel que juega en su relación con otros objetos. ◦ Los nombres de los roles enriquecen la semántica de una relación. ◦ Son opcionales y aparecen como sustantivos. ◦ Se escribe cerca de la clase a la cual califica

Polimorfismo

- Una operación polimórfica tiene muchas implementaciones.





polimorfismo es la capacidad para hacer que,al invocar al mismo método desde distintos objetos, cada uno de esos objetos pueda responder a ese mensaje de forma distinta.

Patrones arquitectonicos:

Patrones Arquitectonico: Decisiones de diseño efectivas para organizar ciertas clases de sistemas de software. Su propósito es compartir una solución probada, ampliamente aplicable a un problema particular de diseño. Se presenta en una forma estándar y es fácilmente reutilizado

Patrón en capas: Sistema complejo que requiere el desarrollo de partes de manera independiente.

Requieren la separación clara y bien documentada de los aspectos involucrados de manera que los módulos puedan ser desarrollados y mantenidos independientemente.

Divide el software en unidades denominadas capas. Cada capa agrupa a módulos que ofrecen un conjunto cohesivo de servicios.
Ejemplo: Web de datos

Patrón centrado en datos: Varios componentes necesitan compartir y manipular grandes cantidades de datos.

Esos datos no pertenecen exclusivamente a ninguno de los componentes.

La interacción está dominada por el intercambio de datos persistentes entre múltiples entidades que acceden a esos datos y al menos un almacenamiento de datos compartido

Es posible cambiar componentes existentes y agregar nuevos componentes cliente a la arquitectura, los componentes clientes operan en forma independiente.

Patrón modelo-vista-controlador: Software para IU es la porción modificada con mayor frecuencia de una aplicación interactiva. Es recomendable mantener las modificaciones de la IU separada del resto del sistema.
Ejemplo: Los datos de una hoja de cálculo pueden mostrarse en formato tabular, con un gráfico de barras, con uno de torta. Los datos son el modelo. Si cambia el modelo deberán actualizarse las vistas

PATRONES GRASP:

Patrón: es la pareja de Problema/solución con un nombre, que es aplicable a otros contextos, con una sugerencia sobre la manera de usarlo en situaciones nuevas

Patron Experto: ◦ Problema: Cuál es el principio general en la asignación de responsabilidades a objetos?
◦ Solución: Asignar una responsabilidad al experto en información la clase que tiene la información necesaria para cumplir con la responsabilidad. La responsabilidad de realizar una labor es de la clase que tiene o puede tener los datos involucrados (atributos)
◦ Ventajas: Se mantiene el encapsulamiento de la información, ya que los objetos utilizan su propia información para realizar las tareas. Esto suele favorecer el bajo acoplamiento, que conduce a sistemas más robustos y mantenibles. Normalmente se admite una alta cohesión.

Patrón Creador: ◦ Problema: Quién es el responsable de crear una instancia de una clase?
◦ Solución: Asigne a la clase B la responsabilidad de crear una instancia de la clase A si: B contiene a A. B es una agregación o composición de A. B almacena a A. B tiene los datos de inicialización de A (datos que requiere su constructor). B usa a A. La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. Si se asignan de manera correcta, el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulamiento y reutilización.
◦ Ventajas: Se admite un bajo acoplamiento, lo que implica menores dependencias de mantenimiento y mayores oportunidades de reutilización. El acoplamiento no se incrementa dado que la clase creada ya es visible para la clase creadora. Las asociaciones que motivaron su elección como creador ya eran existentes

Bajo Acoplamiento: Poca dependencia entre las clases.
Si todas las clases dependen entre sí no es posible la reutilización.
Mal diseño: Herencia profunda
Se debe considerar las ventajas de la delegación respecto de la herencia.
◦ Ventajas: No se ve afectado por cambios en otros componentes Simple de entender en forma aislada Conveniente para la reutilización

Alta cohesión: ◦ Solución: Asignar la responsabilidad para que la cohesión permanezca alta.
◦ Problema: Cómo mantener la complejidad manejable?

La cohesión es una medida de cuán fuertemente relacionadas están las responsabilidades de un elemento. Un elemento con responsabilidades muy relacionadas, y que no hace una gran cantidad de trabajo, tiene una alta cohesión. Una clase con baja cohesión hace muchas cosas no relacionadas o hace demasiado trabajo: difícil de comprender difícil de reutilizar difícil de mantener constantemente afectada por el cambio

Diseño Arquitectónico:

El estilo y estructura particular elegido para una aplicación dependen fuertemente de los requerimientos no funcionales.

Patron Arquitectonico: colección de decisiones de diseño arquitectónico con nombre específico aplicables a problemas de diseño recurrentes parametrizadas ◦ para diferentes contextos de desarrollo de software.

Patrón En Capas:

◦ Contexto: Sistema complejo que requiere el desarrollo de partes de manera independiente. Requieren la separación clara y bien documentada de los aspectos involucrados de manera que los módulos puedan ser desarrollados y mantenidos independientemente. ◦ Problema: El software requiere ser segmentado en módulos a desarrollar por separado con mínima interacción entre las partes. Soporta portabilidad, modificabilidad y reutilización. ◦ Solución: Divide el software en unidades denominadas capas. Cada capa agrupa a módulos que ofrecen un conjunto cohesivo de servicios. La relación entre las capas debe ser unidireccional (allowed-to-use).
◦ Ventajas Facilita la comprensión Facilita mantenimiento Facilita reutilización Facilita portabilidad ◦ Desventajas No siempre es fácil estructurar en capas ni identificar los niveles de abstracción a partir de los Requerimientos.

Patrón centrado en datos:

Contexto: Varios componentes necesitan compartir y manipular grandes cantidades de datos. Esos datos no pertenecen exclusivamente a ninguno de los componentes. ◦ Problema: ¿Cómo pueden los sistemas almacenar y manipular datos persistentes que son accedidos por múltiples componentes independientes? ◦ Solución: La interacción está dominada por el intercambio de datos persistentes entre múltiples entidades que acceden a esos datos y al menos un almacenamiento de datos compartido. El tipo de conector es lectura y escritura.

◦ Ventajas: Promueve la capacidad de integración. Es posible cambiar componentes existentes y agregar nuevos componentes cliente a la arquitectura, los componentes clientes operan en forma independiente.

Patrón Modelo Vista Controlador:

◦ Contexto: Software para IU es la porción modificada con mayor frecuencia de una aplicación interactiva. Es recomendable mantener las modificaciones de la IU

separada del resto del sistema. ◦ Relaciones: La relación notifies conecta instancias del modelo, vista y controlador notificando elementos de cambios de estado relevantes. ◦ Restricciones: Debe haber por lo menos una instancia de modelo, una de vista y una de controlador. ◦ Debilidad: La complejidad puede no valer la pena para interfaces de usuario simples. ISII FICH U

Ventajas: Se presenta la misma información de distintas formas. Las vistas y comportamiento de una aplicación deben reflejar las manipulaciones de los datos de forma inmediata. Debería ser fácil cambiar la interfaz de usuario (incluso en tiempo de ejecución). Permitir diferentes estándares de interfaz de usuario o portarla a otros entornos no debería afectar al código de la aplicación.

VERIFICACION Y VALIDACION:

Son los procesos que aseguran que el programa satisface su especificación y brinda la funcionalidad esperada por las personas que pagan por el software.

• **Validación:** ¿Estamos construyendo el producto correcto?
Consiste en mostrar que el software hace lo que el usuario requiere.
• **Verificación** ¿Estamos construyendo el producto correctamente?
Consiste en mostrar que se cumple con las especificaciones.

El propósito de V&V es asegurar que el software sea lo suficientemente bueno para su uso pretendido.

Actividades de la gestión de Verificación y Validación
• Utiliza estándares y procedimientos para las inspecciones y pruebas del software.
• Define checklists para las inspecciones
• Define el plan de pruebas (PP)

El Plan de Pruebas contiene:
• Descripción del proceso de pruebas: fases de pruebas a realizar.
• Requerimientos a probar: casos de uso, por ejemplo.
• Productos de software a probar.
• Calendarización de las pruebas
• Procedimientos de registro de las pruebas: registración de defectos, asignación, tipos. estimados de corrección, etc.
• Requerimientos de hardware y software.
• Restricciones: sucesos que afectan al proceso de pruebas.

Inspecciones de software

Desventajas de las pruebas:

• Requieren que el software se desarrolle (prototipo o producto final).
• El sw debe ejecutarse reiteradas veces, y por lo general se encuentra una falla por cada
prueba.

Ventajas de las inspecciones:

• No requieren que el programa se ejecute; incluso pueden realizarse antes que se implemente.
• Son más efectivas y menos costosas que las pruebas
• No reemplazan a las pruebas.
• No pueden validar el comportamiento dinámico del sw. A veces no es práctico inspeccionar
“todo” un sistema completo.
• Se descubren problemas en las mismas inspecciones y ayudar a diseñar formas más
efectivas para probar el sistema.
• Las inspecciones siempre requieren sobre-costos al inicio, y solo conducen a ahorros en los
costos después