ÁREA 3. DESARROLLO DE SOFTWARE DE APLICACIÓN

SUBÁREA 3.1 INGENIERÍA DE SOFTWARE

TEMAS IMPORTANTE:

Área 1. Análisis de Sistemas de Software

**Procesos de Ing. de Requerimientos:**

La ingeniería de requisitos es una disciplina que se enfoca en el proceso de identificar, analizar, documentar y verificar los requisitos de un sistema de software. El proceso de ingeniería de requisitos puede ser dividido en varias etapas, incluyendo las siguientes:

* Identificación de requisitos: En esta etapa, se identifican los requisitos del sistema a partir de la información proporcionada por los interesados en el sistema, los usuarios finales y otros stakeholders relevantes.
* Análisis de requisitos: En esta etapa, se realiza un análisis detallado de los requisitos identificados en la etapa anterior, para determinar su relevancia, prioridad y factibilidad técnica.
* Especificación de requisitos: En esta etapa, se documentan los requisitos identificados y analizados en un formato formal y detallado, incluyendo los casos de uso, requisitos funcionales y no funcionales, y las restricciones del sistema.
* Validación de requisitos: En esta etapa, se verifica que los requisitos especificados en la etapa anterior cumplen con los objetivos y necesidades de los stakeholders, y se asegura que los requisitos sean coherentes y completos.
* Gestión de requisitos: En esta etapa, se realiza la gestión de los requisitos del sistema, incluyendo la identificación de cambios y la gestión de las versiones de los requisitos, para garantizar que el sistema cumpla con las necesidades y expectativas de los stakeholders.

Cada una de estas etapas es importante para el éxito del proceso de ingeniería de requisitos, y se deben realizar de forma cuidadosa y rigurosa para asegurar que el sistema cumpla con los requisitos del negocio y los usuarios finales.

* **Requerimientos funcionales**

Los requisitos funcionales son aquellos que especifican las funciones y características del sistema de software que se está desarrollando. Estos requisitos describen qué es lo que el sistema debe hacer y cómo debe hacerlo para satisfacer las necesidades de los usuarios y cumplir con los objetivos del negocio.

Algunos ejemplos de requisitos funcionales son:

* Registro de usuarios: el sistema debe permitir que los usuarios se registren proporcionando su información personal y de contacto.
* Gestión de pedidos: el sistema debe permitir que los usuarios realicen pedidos y realicen un seguimiento de su estado.
* Generación de reportes: el sistema debe permitir la generación de informes sobre los datos del sistema.
* Procesamiento de pagos: el sistema debe permitir el procesamiento de pagos en línea.
* Gestión de inventario: el sistema debe permitir la gestión del inventario de productos.

Es importante que los requisitos funcionales sean claros, específicos, medibles y verificables, para asegurar que se puedan implementar y probar de manera efectiva. Además, es necesario que estos requisitos estén alineados con las necesidades y expectativas de los usuarios finales y el negocio.

* **Requerimientos no funcionales**

Los requerimientos no funcionales son aquellos que describen las características y atributos que un sistema de software debe poseer en términos de calidad, usabilidad, rendimiento y seguridad, entre otros aspectos. Estos requisitos suelen ser menos específicos y detallados que los requisitos funcionales, pero son igualmente importantes para el éxito del sistema.

Algunos ejemplos de requerimientos no funcionales son:

* Usabilidad: El sistema debe ser fácil de aprender y usar para los usuarios finales, y debe ofrecer una experiencia de usuario agradable y efectiva.
* Rendimiento: El sistema debe ser capaz de responder a las demandas de los usuarios finales y procesar grandes cantidades de datos en tiempos razonables.
* Disponibilidad: El sistema debe estar disponible en todo momento, y cualquier interrupción o caída del sistema debe ser mínima y rápidamente resuelta.
* Seguridad: El sistema debe ser seguro y proteger los datos y la información de los usuarios finales, evitando accesos no autorizados y posibles ataques de hackers.
* Mantenibilidad: El sistema debe ser fácil de mantener, actualizar y mejorar, y los costos asociados con estas actividades deben ser bajos.
* Escalabilidad: El sistema debe ser escalable para poder manejar mayores volúmenes de usuarios y datos, a medida que crece el negocio.
* Interoperabilidad: El sistema debe ser capaz de comunicarse y trabajar con otros sistemas y aplicaciones, garantizando la integración con sistemas externos.
* **Requerimientos del dominio, del usuario y del sistema**

Los requerimientos del dominio, del usuario y del sistema son tipos de requisitos que se identifican durante el proceso de ingeniería de requisitos. A continuación, se describe cada uno de ellos:

* Requerimientos del dominio: Estos requisitos describen las características y restricciones que están directamente relacionadas con el contexto del negocio o industria en el que el sistema se utilizará. Por ejemplo, si se está desarrollando un sistema de gestión de inventarios para una empresa de manufactura, los requerimientos del dominio podrían incluir normas y regulaciones de la industria, especificaciones de los materiales, y procedimientos de producción.
* Requerimientos del usuario: Estos requisitos describen las necesidades, expectativas y preferencias de los usuarios finales del sistema, y se enfocan en cómo se espera que el sistema funcione para satisfacer sus requerimientos. Por ejemplo, los requerimientos del usuario podrían incluir la facilidad de uso, la personalización de la interfaz de usuario, la capacidad de búsqueda y filtrado de datos, y la accesibilidad para usuarios con discapacidades.
* Requerimientos del sistema: Estos requisitos describen las funcionalidades y características específicas que el sistema debe tener para cumplir con los objetivos del negocio y los requerimientos de los usuarios. Por ejemplo, los requerimientos del sistema podrían incluir la capacidad de procesamiento, el rendimiento, la seguridad, la escalabilidad y la interoperabilidad con otros sistemas.
* **Especificación de la Interfaz**

La especificación de la interfaz es un conjunto de documentos que detallan cómo los diferentes componentes del sistema de software interactúan entre sí y con los usuarios. Esta documentación describe los requisitos de la interfaz de usuario, los protocolos de comunicación, los formatos de intercambio de datos, y los procedimientos para la integración de los diferentes módulos del sistema.

La especificación de la interfaz es una parte crítica del proceso de ingeniería de software, ya que una interfaz bien diseñada puede hacer que el sistema sea más fácil de usar, más flexible y escalable, y más fácil de mantener y mejorar con el tiempo. Algunas de las consideraciones clave que se deben tener en cuenta al desarrollar una especificación de la interfaz incluyen:

* Usabilidad: La interfaz debe ser fácil de usar y comprender para los usuarios finales, independientemente de su nivel de experiencia o habilidad técnica.
* Consistencia: La interfaz debe ser consistente en todo el sistema, de modo que los usuarios puedan aplicar las mismas habilidades y conocimientos en diferentes partes del sistema.
* Flexibilidad: La interfaz debe ser lo suficientemente flexible para adaptarse a diferentes tipos de dispositivos y plataformas, así como a las necesidades cambiantes de los usuarios y del negocio.
* Estilo visual: La interfaz debe tener un estilo visual atractivo y coherente que refleje la identidad de la marca o empresa, y que sea agradable a la vista del usuario.
* Documentación: La interfaz debe ser documentada de manera clara y precisa, con instrucciones y guías de usuario detalladas que permitan a los usuarios comprender y utilizar el sistema de manera efectiva.
* **Especificaciones de requerimientos de software (SRS)**

Las especificaciones de requerimientos de software (SRS, por sus siglas en inglés) son documentos que describen de manera detallada los requisitos funcionales y no funcionales de un sistema de software. La SRS es una parte crítica del proceso de ingeniería de requisitos, ya que establece las bases para el diseño, desarrollo, pruebas y mantenimiento del sistema.

La SRS debe incluir información detallada sobre los siguientes aspectos del sistema:

* Descripción del sistema: Esta sección debe proporcionar una descripción general del sistema, incluyendo su propósito, objetivos, funciones y características principales.
* Requerimientos funcionales: Estos son los requisitos que describen lo que el sistema debe hacer. Los requerimientos funcionales deben estar organizados en un conjunto de casos de uso o escenarios que describen cómo el sistema interactúa con los usuarios y otros sistemas.
* Requerimientos no funcionales: Estos son los requisitos que describen cómo el sistema debe funcionar, como la seguridad, el rendimiento, la escalabilidad, la usabilidad, la accesibilidad y la compatibilidad.
* Requerimientos de interfaz: Estos son los requisitos que describen cómo el sistema se conectará con otros sistemas y con el hardware y el software del usuario final.
* Restricciones y supuestos: Esta sección debe identificar cualquier restricción o suposición que pueda afectar el diseño, desarrollo o implementación del sistema.
* Plan de pruebas: Esta sección debe describir cómo se probará el sistema para asegurarse de que cumpla con los requisitos establecidos en la SRS.

**Modelos de análisis del sistema:**

Se describe cada uno de estos modelos de análisis del sistema:

* Modelo de contexto: Este modelo se utiliza para identificar los límites del sistema y su entorno, es decir, aquellos elementos que interactúan con el sistema. El modelo de contexto puede representarse mediante un diagrama de contexto, que muestra los elementos externos al sistema y sus interacciones con el mismo.
* Modelo de comportamiento: Este modelo describe cómo el sistema se comporta y cómo se llevan a cabo las interacciones entre los diferentes componentes del sistema. Se puede representar mediante un diagrama de flujo de datos, que muestra los procesos y flujos de información dentro del sistema.
* Modelo de datos: Este modelo se utiliza para describir la estructura de los datos que se utilizan en el sistema. El modelo de datos se puede representar mediante un diagrama de entidad-relación, que muestra las entidades y las relaciones entre ellas.
* Modelo de objetos: Este modelo describe la estructura de los objetos que se utilizan en el sistema y las interacciones entre ellos. El modelo de objetos se puede representar mediante un diagrama de clases, que muestra las clases y las relaciones entre ellas.

**Diagramas de UML (Casos de Uso):**

**Riesgos y su manejo:**

Los riesgos son eventos o situaciones que pueden afectar negativamente el éxito de un proyecto. El manejo de riesgos se refiere a la identificación, evaluación y control de estos riesgos para minimizar sus efectos negativos en el proyecto. A continuación se describen los pasos comunes en el manejo de riesgos:

* Identificación de riesgos: El primer paso es identificar los riesgos potenciales para el proyecto. Esto se puede hacer mediante la revisión de documentos y registros del proyecto, la consulta con expertos y la realización de sesiones de lluvia de ideas.
* Evaluación de riesgos: Una vez que se han identificado los riesgos, se debe evaluar su probabilidad de ocurrencia y su impacto en el proyecto. Esto se puede hacer mediante la creación de una matriz de riesgos, que clasifica los riesgos según su probabilidad e impacto.
* Desarrollo de estrategias de manejo de riesgos: Después de evaluar los riesgos, se deben desarrollar estrategias para manejarlos. Estas estrategias pueden incluir la prevención, la mitigación, la transferencia o la aceptación de los riesgos.
* Implementación de las estrategias de manejo de riesgos: Una vez que se han desarrollado las estrategias de manejo de riesgos, se deben implementar y seguir de manera constante. Esto implica la asignación de responsabilidades para la implementación y el seguimiento de las estrategias de manejo de riesgos.
* Monitoreo y control: Finalmente, se debe monitorear y controlar continuamente los riesgos identificados y las estrategias de manejo de riesgos implementadas. Esto incluye la revisión periódica de la matriz de riesgos

Área 2. Diseño de Sistemas de Software:

**Diseño Arquitectónico**

* **Arquitecturas básicas de software**

La arquitectura básica de software se refiere a la estructura general y el diseño del software. Es el marco en el que se construye una aplicación, y establece la forma en que los diferentes componentes del software interactúan entre sí. La arquitectura básica de software es importante porque afecta directamente a la calidad, la seguridad, la escalabilidad y la capacidad de mantenimiento del software. Las arquitecturas básicas de software comunes incluyen la modelo vista controlador (MVC), la arquitectura de tres capas, la arquitectura cliente-servidor, la arquitectura orientada a servicios (SOA) y la arquitectura basada en microservicios. Cada arquitectura tiene sus propias ventajas y desventajas, y la elección de una arquitectura específica depende de los requisitos y objetivos de cada proyecto de software.

* **Patrones básicos de arquitectura de software (MVC)**
* Modelo Vista Controlador (MVC): divide una aplicación en tres partes principales: el modelo, la vista y el controlador. El modelo representa los datos y la lógica de negocio, la vista es la interfaz de usuario y el controlador maneja la interacción entre el modelo y la vista.
* Capas: separa una aplicación en capas lógicas que se comunican entre sí. La capa de presentación es la interfaz de usuario, la capa de negocio maneja la lógica de negocio y la capa de datos se encarga del acceso a los datos.
* Cliente-Servidor: divide una aplicación en dos partes principales: el cliente, que es la interfaz de usuario, y el servidor, que maneja la lógica de negocio y el acceso a los datos.
* Microservicios: divide una aplicación en componentes pequeños y autónomos que se comunican entre sí a través de una API. Cada microservicio se centra en una tarea específica.
* Arquitectura basada en eventos: se basa en la emisión y recepción de eventos. Los componentes de la aplicación se comunican a través de eventos, que son desencadenados por acciones en la aplicación.
* Arquitectura Orientada a Servicios (SOA): se basa en la creación de servicios reutilizables y bien definidos que se comunican a través de una red. Cada servicio realiza una tarea específica y se puede utilizar en diferentes aplicaciones.

**Diseño de Software:**

* **Diagramas de UML (Clases, Objetos, Secuencia, Comunicación, Actividad y Estados)**

UML (Unified Modeling Language) es un lenguaje gráfico de modelado utilizado para visualizar, especificar, construir y documentar sistemas de software. Hay varios tipos de diagramas UML, algunos de los más comunes son:

* Diagrama de clases: muestra las clases, interfaces, atributos y métodos de un sistema, y cómo se relacionan entre sí.
* Diagrama de objetos: muestra las instancias de las clases y sus relaciones en un momento dado.
* Diagrama de secuencia: muestra la secuencia de interacciones entre los objetos en un escenario específico.
* Diagrama de comunicación: similar al diagrama de secuencia, pero muestra las interacciones en un nivel más alto de abstracción.
* Diagrama de actividades: muestra el flujo de trabajo de una actividad o proceso, y cómo los diferentes elementos interactúan entre sí.
* Diagrama de estados: muestra los diferentes estados de un objeto y cómo cambia de un estado a otro en respuesta a eventos.
* **Patrones básicos de diseño de software.** 
  + **Creacionales (Sigleton, Factory Method, Abstract factory)**

Los patrones de diseño creacionales son patrones que se utilizan para crear objetos y clases de manera más eficiente y flexible. Algunos de los patrones creacionales más comunes son:

1. Singleton: asegura que solo exista una instancia de una clase en todo el sistema. Esto se logra mediante la creación de un método que siempre devuelve la misma instancia de la clase.
2. Factory Method: permite la creación de objetos sin especificar la clase exacta de objeto que se creará. En su lugar, se utiliza una interfaz común y una clase de fábrica para crear el objeto deseado.
3. Abstract Factory: proporciona una interfaz para crear familias de objetos relacionados o dependientes, sin especificar las clases concretas de objetos. Este patrón se utiliza para proporcionar una manera fácil de crear diferentes versiones de un sistema de software sin tener que cambiar el código en varias partes del sistema.
   * **Structural (Facade, Composite, Bridge)**

Los patrones de diseño estructurales se utilizan para organizar los objetos y las clases en una estructura más clara y fácil de entender. Algunos de los patrones estructurales más comunes son:

1. Facade: proporciona una interfaz unificada para un conjunto de interfaces más complejas de un subsistema. Este patrón se utiliza para simplificar el uso de un sistema complejo, proporcionando una única interfaz fácil de usar.
2. Composite: permite tratar un grupo de objetos similares como si fueran un solo objeto. Este patrón se utiliza para simplificar la estructura de un sistema, permitiendo la creación de objetos compuestos que se comportan como un solo objeto.
3. Bridge: separa la abstracción de una clase de su implementación, permitiendo que ambas evolucionen de forma independiente. Este patrón se utiliza para desacoplar la abstracción de la implementación, lo que permite que los cambios en una no afecten a la otra.
   * **Conductual (Iterator, visitor, Strategy)**

Los patrones de diseño conductuales se utilizan para gestionar el comportamiento y las interacciones entre objetos y clases en un sistema de software. Algunos de los patrones conductuales más comunes son:

1. Iterator: permite recorrer los elementos de una colección de objetos sin exponer su estructura interna. Este patrón se utiliza para acceder a los elementos de una colección de manera secuencial, sin tener que conocer su estructura interna.
2. Visitor: permite agregar nuevas operaciones a una clase sin modificar su estructura. Este patrón se utiliza para agregar operaciones a una clase existente, sin tener que modificar la clase en sí.
3. Strategy: define una familia de algoritmos, encapsulándolos y haciendo que sean intercambiables. Este patrón se utiliza para permitir que diferentes algoritmos sean intercambiables en una aplicación, sin tener que modificar la lógica principal de la aplicación.

**Diseño de Interfaces de usuario:**

El diseño de interfaces de usuario es una parte crítica del desarrollo de software, ya que es lo que los usuarios finales ven y con lo que interactúan. A continuación, se presentan algunos aspectos clave a considerar en el diseño de interfaces de usuario efectivas:

Diseño centrado en el usuario: es importante tener en cuenta las necesidades y expectativas de los usuarios al diseñar una interfaz de usuario. Esto implica hacer una investigación de usuarios y pruebas de usabilidad para asegurarse de que la interfaz sea intuitiva y fácil de usar para los usuarios finales.

Consistencia: mantener una interfaz de usuario consistente en todo el sistema ayuda a los usuarios a comprender y usar la aplicación de manera efectiva. Esto se puede lograr mediante la estandarización de elementos de diseño, como la ubicación de botones y la presentación de información.

* Diseño visual atractivo: el diseño visual de la interfaz de usuario es importante para mantener el interés y la atención de los usuarios. El uso de colores apropiados, una disposición clara y el uso de gráficos e iconos pueden hacer que la interfaz sea más atractiva y fácil de usar.
* Facilidad de navegación: los usuarios deben poder navegar por la aplicación de manera intuitiva y sin problemas. Esto implica una navegación clara y bien organizada, así como la disponibilidad de botones y herramientas de búsqueda.
* Feedback claro: es importante proporcionar una retroalimentación clara a los usuarios sobre las acciones que han realizado. Esto puede incluir mensajes de confirmación y la indicación visual de que se ha realizado una acción.
* Adaptabilidad: las interfaces de usuario deben ser adaptables a diferentes dispositivos y tamaños de pantalla, para que los usuarios puedan acceder a la aplicación desde diferentes dispositivos y entornos.

**Área 3. Desarrollo de Sistemas de Software:**

**Todo lo que saben de programación +**

* **Lógica, modularidad, paso de parámetros, ámbito de variables.**
* **Estilo y buenas prácticas de programación**
* **Métricas de código (Cohesión, Acoplamiento)**

**Área 4. Gestión de Proyectos de Software:**

**Modelos de Proceso de Software**

Los modelos de proceso de software son marcos de trabajo utilizados para organizar, planificar y controlar el desarrollo de software. A continuación, se presentan los modelos clásicos y ágiles más comunes:

Modelos clásicos:

Los modelos clásicos son enfoques secuenciales y predecibles para el desarrollo de software. Los modelos clásicos incluyen:

1. Modelo en cascada: es un modelo secuencial en el que cada fase del ciclo de vida del software se completa antes de pasar a la siguiente fase.
2. Modelo en V: es una extensión del modelo en cascada que enfatiza la verificación y validación tempranas para reducir errores.
3. Modelo iterativo: es un modelo en el que se realizan varias iteraciones de las fases de desarrollo del software para mejorar continuamente el producto final.

Modelos ágiles:

Los modelos ágiles se enfocan en la entrega rápida y continua de software y la adaptación a los cambios en los requisitos del cliente. Los modelos ágiles incluyen:

1. Scrum: es un marco de trabajo ágil que se enfoca en la colaboración y la entrega continua de software en iteraciones cortas llamadas "sprints".
2. Extreme Programming (XP): es un modelo ágil que se enfoca en la calidad del software y la entrega rápida mediante el uso de prácticas como programación en parejas y pruebas automatizadas.
3. Kanban: es un modelo ágil que se enfoca en la entrega continua de software mediante la visualización del flujo de trabajo y la limitación del trabajo en progreso.

**Estimación de costos de software**

La estimación de costos de software es un proceso crítico en la planificación del proyecto de software. A continuación se describen tres métodos comunes para la estimación de costos de software:

* Modelo algorítmico: Este método utiliza una fórmula matemática para estimar los costos del proyecto. Se basa en factores como el tamaño del código, la complejidad del proyecto, la experiencia del equipo de desarrollo y la disponibilidad de herramientas de desarrollo. La estimación de costos utilizando modelos algorítmicos puede ser más precisa en proyectos bien definidos y estructurados, pero puede ser menos precisa en proyectos más complejos y cambiantes.
* Puntos de función: Este método mide el tamaño funcional del software, es decir, el número de funciones y características que el software proporcionará. Los puntos de función se calculan utilizando una fórmula que considera factores como el número de entradas y salidas, el número de archivos y tablas, y la complejidad de las operaciones. La estimación de costos utilizando puntos de función es más precisa en proyectos que se enfocan en la funcionalidad y la calidad del software.
* COCOMO: El modelo COCOMO (COnstructive COst MOdel) es un método de estimación de costos de software desarrollado por Barry Boehm. El modelo COCOMO se basa en la complejidad del proyecto, la experiencia del equipo de desarrollo, el tamaño del proyecto y otros factores. El modelo COCOMO se divide en tres niveles: básico, intermedio y detallado. El nivel básico utiliza un conjunto limitado de factores para estimar el costo del proyecto, mientras que el nivel detallado considera una amplia gama de factores.

**Gestión de Calidad**

La gestión de calidad en el desarrollo de software es fundamental para asegurar que los productos desarrollados cumplan con los requerimientos y estándares establecidos. A continuación se describen algunos aspectos importantes de la gestión de calidad en el desarrollo de software:

* Calidad de proceso y de producto: La calidad de proceso se refiere a la capacidad de seguir un proceso estructurado y definido para el desarrollo de software. La calidad de producto se refiere a la capacidad del software para cumplir con los requerimientos y estándares establecidos. Ambos aspectos son importantes para asegurar que el software entregado cumpla con las expectativas de los clientes y usuarios.
* Métricas básicas de producto: Las métricas son medidas utilizadas para evaluar la calidad del software. Las métricas básicas de producto pueden incluir aspectos como la cantidad de errores encontrados, el tiempo de respuesta, el rendimiento y la capacidad de mantenimiento del software. Estas métricas pueden ayudar a identificar áreas de mejora en el software y en el proceso de desarrollo.
* Modelo de capacidad y madurez de procesos (CMM-I): El modelo de capacidad y madurez de procesos (CMM-I) es un modelo de evaluación de procesos de software desarrollado por el Instituto de Ingeniería de Software de Carnegie Mellon. El modelo CMM-I se basa en cinco niveles de madurez: inicial, repetible, definido, gestionado y optimizado. Cada nivel representa una mejora en la capacidad de gestión de los procesos de software. La adopción del modelo CMM-I puede ayudar a las organizaciones a mejorar la calidad del proceso de desarrollo y del producto entregado.

**Pruebas de Software**

Las pruebas de software son una actividad fundamental en el proceso de desarrollo de software, cuyo objetivo es verificar que el software cumple con los requisitos y especificaciones establecidos. A continuación, se describen algunos tipos de pruebas de software:

Pruebas del sistema: Las pruebas del sistema se realizan para verificar que el software cumple con los requisitos funcionales y no funcionales establecidos. Estas pruebas se realizan en el sistema completo y se pueden llevar a cabo mediante diferentes técnicas de pruebas, como pruebas de integración, pruebas de aceptación y pruebas de carga.

Pruebas de componentes: Las pruebas de componentes se realizan para verificar el funcionamiento individual de cada uno de los componentes del software. Estas pruebas se realizan en los componentes de forma aislada y se pueden llevar a cabo mediante diferentes técnicas de pruebas, como pruebas unitarias, pruebas de integración y pruebas de regresión.

Casos de prueba: Los casos de prueba son una serie de pruebas diseñadas para verificar que el software cumple con los requisitos establecidos. Estos casos de prueba pueden ser diseñados manualmente o mediante herramientas automáticas y deben cubrir tanto los casos normales como los casos extremos.

Pruebas de caja blanca, caja negra y caja gris: Las pruebas de caja blanca, caja negra y caja gris son diferentes técnicas de pruebas que se utilizan para verificar diferentes aspectos del software. Las pruebas de caja blanca se realizan conociendo el código fuente del software y se utilizan para verificar su estructura y lógica interna. Las pruebas de caja negra se realizan sin conocer el código fuente del software y se utilizan para verificar su comportamiento externo. Las pruebas de caja gris son una combinación de las anteriores y se realizan conociendo parcialmente el código fuente del software.

**Verificación y Validación**

La verificación y validación son dos actividades importantes en el proceso de desarrollo de software que se utilizan para asegurar que el software cumple con los requisitos y especificaciones establecidos. A continuación, se definen estos dos conceptos básicos:

* Verificación: La verificación es el proceso de evaluar si un producto de software cumple con las especificaciones y requisitos establecidos. Se realiza para asegurarse de que el software se ha construido correctamente, es decir, que se ha implementado de acuerdo con el diseño y las especificaciones.
* Validación: La validación es el proceso de evaluar si el software cumple con las necesidades y expectativas del cliente y con los requisitos del usuario final. Se realiza para asegurarse de que el software se ha construido correctamente, es decir, que cumple con las necesidades y expectativas del cliente y se ajusta a los requisitos del usuario final.