Resumen Técnicas de Documentación y Validación

Proceso de Validación, Verificación y Documentación

- Es el proceso de todo un ciclo vital: La V & D debe aplicarse en cada etapa de un proyecto.
- Independiente de la magnitud.

Documentación:

- Preguntas esenciales para hacerse:
  - **¿Qué y por qué documentamos?**
  - **¿Cuándo documentamos?**
  - **¿Quiénes son los destinatarios de la documentación?**
  - **¿La documentación, es redundante?**
- **¿Cuándo y Qué documentamos?**
  - **Análisis**:
    - Arquitectura tecnológica
    - Catálogo de requerimientos
    - Propuesta de solución
  - **Diseño**:
    - Diagramas UML
    - Diccionario de Datos
    - Documentación del código
  - **Pruebas**:
    - Plan de pruebas
    - Resultados
    - Modificaciones realizadas
  - **Implementación**:
    - Plan de implementación
    - Plan de capacitación

Metas de la Validación y Verificación:

- La verificación y la validación deberían establecer la confianza de que el software es adecuado al propósito.
  - **Validación**: ¿Estamos construyendo el producto correcto? Se ocupa de controlar si el producto satisface los requerimientos del usuario
  - **Verificación**: ¿Estamos construyendo correctamente el producto? Implica controlar que el producto conforma su especificación inicial.
- Esto NO significa que esté completamente libre de defectos.
- Debe ser lo suficientemente bueno para su uso previsto y el tipo de uso determinará el grado de confianza que se necesita.

Validación:

- Preguntas esenciales para hacerse:
  - **¿Qué y por qué validamos/testeamos?**
  - **¿Cuándo validamos/testeamos?**
  - **¿Quiénes son los destinatarios de la validación/testeo?**
  - **¿Cuánto debemos dedicarle a la validación/testeo?**

Confianza de la Validación y Verificación:

- **Función del software**
  - El nivel de confianza depende de lo crítico que es el sistema para una organización.
- **Expectativas del usuario**
  - Los usuarios pueden tener bajas expectativas para ciertas clases de software.
- **Entorno de marketing**
  - Introducir un producto en el mercado pronto puede ser más importante que encontrar defectos en el programa

Verificación Dinámica y Estática:

- **Inspecciones de software.**
  - Se ocupa del análisis de representaciones estáticas del sistema para describir problemas (**verificación estática**)
  - Pueden ser complementadas por documentos basados en herramientas y análisis del código
- **Pruebas del software.**
  - Se ocupa de la ejercitación y la observación del comportamiento del producto (**verificación dinámica**)
  - El sistema se ejecuta con datos de pruebas y se observa su comportamiento operativo.

Planificación de la Validación y Verificación:

- Se requiere una cuidadosa planificación para sacar el máximo de los procesos de inspección y pruebas. La planificación debería comenzar pronto en el proceso de desarrollo.
- El plan debería identificar el balance entre la verificación estática y las pruebas.
- La planificación trata de definir estándares para el proceso de prueba en lugar de describir pruebas de productos.

Estructura de un plan de pruebas:

- Proceso de pruebas
- Trazabilidad de requerimientos.
- Elementos probados.
- Calendario de pruebas.
- Procedimientos de registro de las pruebas.
- Requerimientos hardware y software.
- Restricciones.

Requerimientos:

- **Objetivo**:
  - Definir la necesidad de la organización
  - Comunicación entre clientes, usuarios y desarrolladores
  - Controlar la evolución del sistema
- Escribir requerimientos nos permite mostrarle al cliente todo el funcionamiento del sistema
- **¿Qué contiene?**
  - Información del Problema
  - Como contribuye el nuevo sistema para que mejore la empresa
- **¿Qué no contiene?**
  - Planificación y costos
- **DRU vs ERS**
  - **ERS (Estándar IEEE 830)**:
    - **Especificación de Requerimientos de Software**
    - Punto de vista del desarrollador
    - Está dirigida tanto al cliente como al equipo de desarrollo. El lenguaje utilizado para su redacción debe ser informal, de forma que sea fácilmente comprensible para todas las partes involucradas en el desarrollo.
  - **DRU**:
    - **Documentación de Requerimiento de Usuario**
    - Punto de vista del cliente interesado

Inspección de Software:

- Implican que las personas examinen la representación de la fuente con el propósito de descubrir anomalías y defectos
- Las inspecciones no requieren la ejecución de un sistema por lo que debe utilizarse antes de la implementación.
- Pueden estar aplicados a cualquier representación del sistema (requerimientos, diseño, configuración, datos, pruebas de datos, etc).
- Éxito de la inspección:
  - Al probar, un defecto puede enmascarar a otro así que se requieren varias ejecuciones.

Inspecciones y Pruebas:

- Ambas deben utilizarse durante el proceso de V & V.
- Las inspecciones pueden comprobar el ajuste con una especificación pero no la conformancia con los requerimientos reales del cliente.
- Las inspecciones no pueden comprobar características no funcionales como rendimiento, usabilidad, etc.

Inspecciones de Programas:

- Es la aproximación formalizada a las revisiones del documento
- Está pensado explícitamente para la detección de defectos (no su corrección)
- Los defectos pueden ser errores lógicos, anomalías en el código que pueden indicar una condición errónea o no conformidad con los estándares.

Validación y Documentación de Diseño

Arquitectura de Software:

- Estructura o estructuras de un sistema que involucra elementos de software, propiedades externamente visibles y su relación
- Las estructuras exactas para considerar y las formas de representarlas varían según los objetivos de ingeniería.
- Cada subsistema puede ser descripto por una sub arquitectura.

¿Por qué documentar una Arquitectura?:

- **Durante el análisis del sistema**:
  - Permite estimar de manera más precisa el costo y el tiempo de desarrollo.
  - Mediante abstracción permite facilitar la comprensión e implementación de sistemas complejos.
  - Estimación de atributos de calidad.
- **Durante el desarrollo del sistema**:
  - Provee una vista general del sistema.
  - Exhibe relación de puntos de diseño a tratar.
  - Permite visualizar dependencias entre componentes.
  - Facilita el desarrollo simultáneo de componentes.
  - Expone errores de diseño en fases tempranas.

¿Cuándo se diseña (escoge) una Arquitectura?:

- Posterior al análisis del problema
- **En la etapa de Diseño de la arquitectura**
- **Antes del Diseño detallado**
- El diseño de la arquitectura evolucionará-mutará a lo largo del desarrollo

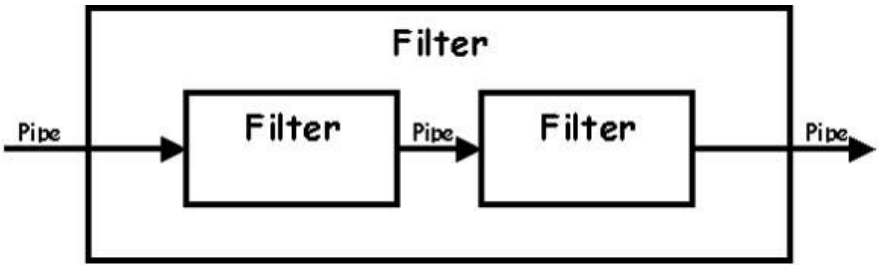
Evaluación de Arquitecturas:

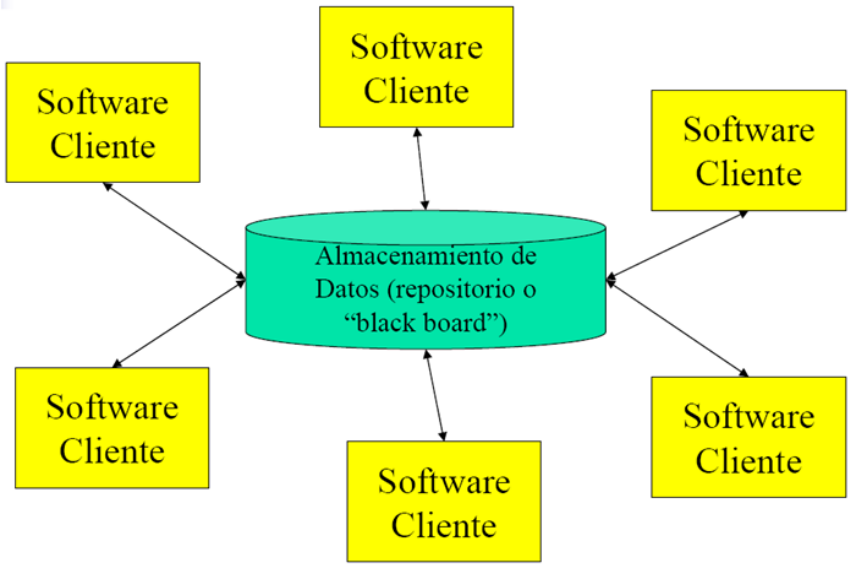
- **¿Por qué?**
  - Antes es mejor
  - Agenda del proyecto
- **¿Cuándo?**
  - Evaluación temprana
  - Evaluación tardía
- **¿Quiénes?**
  - Equipo de evaluación
  - Stakeholders (Personas o negocios que han invertido)
- **¿Qué resultados se obtiene?**
  - Es o no adecuada
  - La más adecuada

Estilos y Patrones:

- **Diseño arquitectónico**:
  - Arquitectura de un sistema concreto.
- **Estilo arquitectónico**:
  - Restricciones sobre la arquitectura de una familia de diseños arquitectónicos.
- **Patrón arquitectónico**:
  - Solución general a un problema común
- **Pipes & Filters**:
  - Provee una estructura para sistemas que procesan datos en serie.
  - Cada paso es encapsulado en un componente filter.
  - Los datos pasan por pipes entre filtros adyacentes.
  - La recombinación de filtros permite construir familias de sistemas relacionados.
- **Blackboard**:
  - Es utilizado para problemas en los cuales no existen o se conocen estrategias determinísticas. Posee varios subsistemas que ensamblan su conocimiento para construir una solución parcial o aproximada.
- **2 Tier (2 capas)**:
  - Describe la interacción en un entorno client/server, en el cual la interfaz de usuario se almacena en el cliente y los datos en el servidor.
  - La lógica de la aplicación puede estar tanto en el cliente como en el servidor.
- **3 Tier (3 capas):**
  - Existe un nivel intermediario. Esto significa que la arquitectura generalmente está compartida por:
    - **Un cliente**, es decir, el equipo que solicita los recursos, equipado con una interfaz de usuario (generalmente un navegador Web) para la presentación

        - **El servidor de aplicaciones** (también denominado software intermedio), cuya tarea es proporcionar los recursos solicitados, pero que requiere de otro servidor para hacerlo
        - **El servidor de datos**, que proporciona al servidor de aplicaciones los datos que requiere
- **MVC (Model View Controller):**
    - Es una forma de partir una aplicación o una parte de su interfaz, en tres partes: Modelo, vista y controlador
    - MVC fue desarrollado originalmente para mapear el tradicional entrada – procesamiento – salida





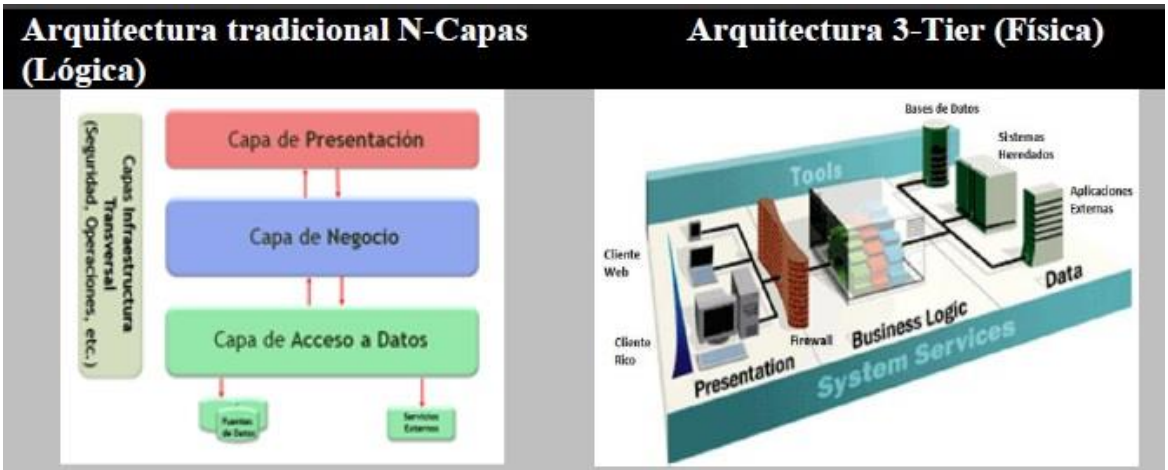
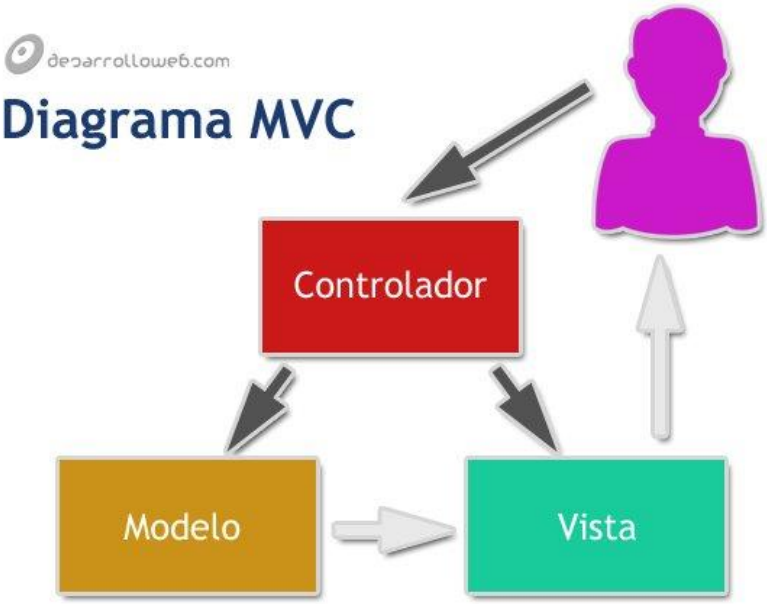


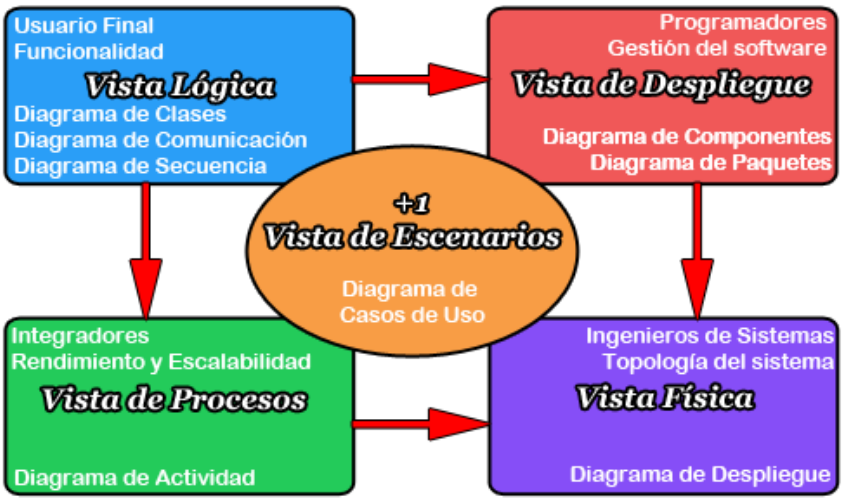
Figura 1.- Arquitectura tradicional N-Layer (Lógica)

Figura 2.- Arquitectura 3-Tier (Física)

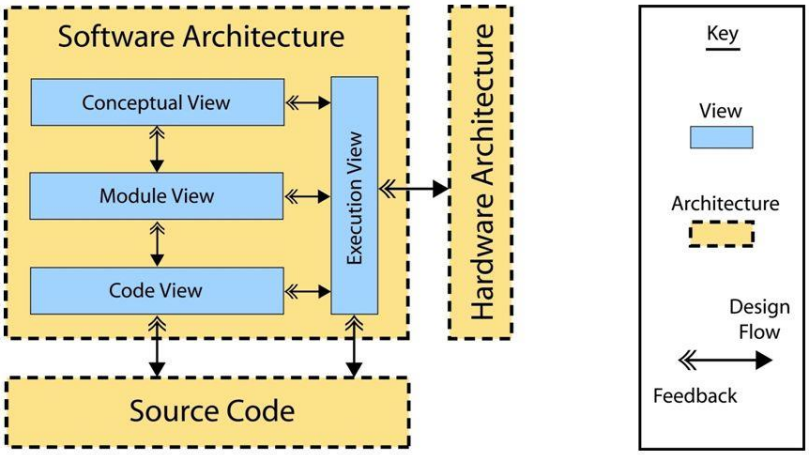


Vistas de una Arquitectura:

- **System stakeholder**:
  - Un individuo, equipo, u organización con intereses o preocupaciones relativas al sistema.
- **View**:
  - Una representación de todo el sistema desde la perspectiva de un conjunto de preocupaciones.
- **Viewpoint**:
  - Una especificación de las convenciones para construir y usar una vista.
- Para planificar un conjunto de vistas es necesario comprender las necesidades de los stakeholders y los recursos disponibles
- **Vista 4+1 Kruchten**:
  - Un sistema software se ha de documentar y mostrar con 4 vistas bien diferenciadas y estas 4 vistas se han de relacionar entre sí con una vista más, que es la denominada vista "+1"
  - Estas 4 vista las denominó Kruchten como: **vista lógica**, **vista de procesos**, **vista de despliegue** y **vista física** y la vista "+1" la denominó **vista de escenario**.
- **Vista Siemens Four View**:
  - **Vista conceptual**
  - **Vista de interconexión**
  - **Vista de ejecución**
  - **Vista de código**



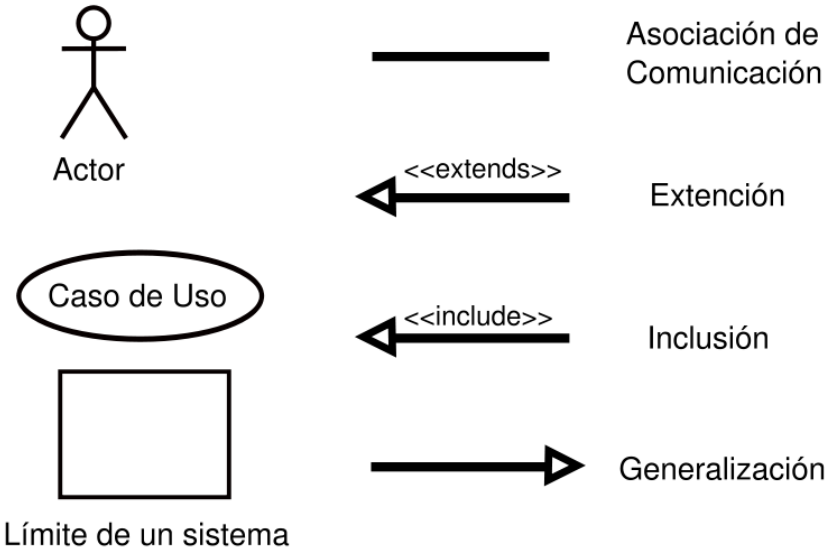
Siemens 4 Views



¿Qué vistas deben elegirse?:

- Depende de:
  - Quienes son los stakeholders
  - Como usarán la documentación.
- Usos de la documentación de la arquitectura:
  - **Educación**: presentación del proyecto
  - **Comunicación**: entre stakeholders
  - **Análisis**: Aseguramiento de la calidad

Diagrama de Casos de Uso:



- Cada Actor tiene que realizar al menos un caso de uso
- Cada caso de uso tiene que ser realizado por un solo actor
- Los casos de uso no se relacionan con los otros casos de uso de manera directa
- Las personas que otorgan la información son los actores
- No existe comunicación entre actores
- Los casos de uso son las acciones más importantes para los actores

Ejemplo Diagrama CU:

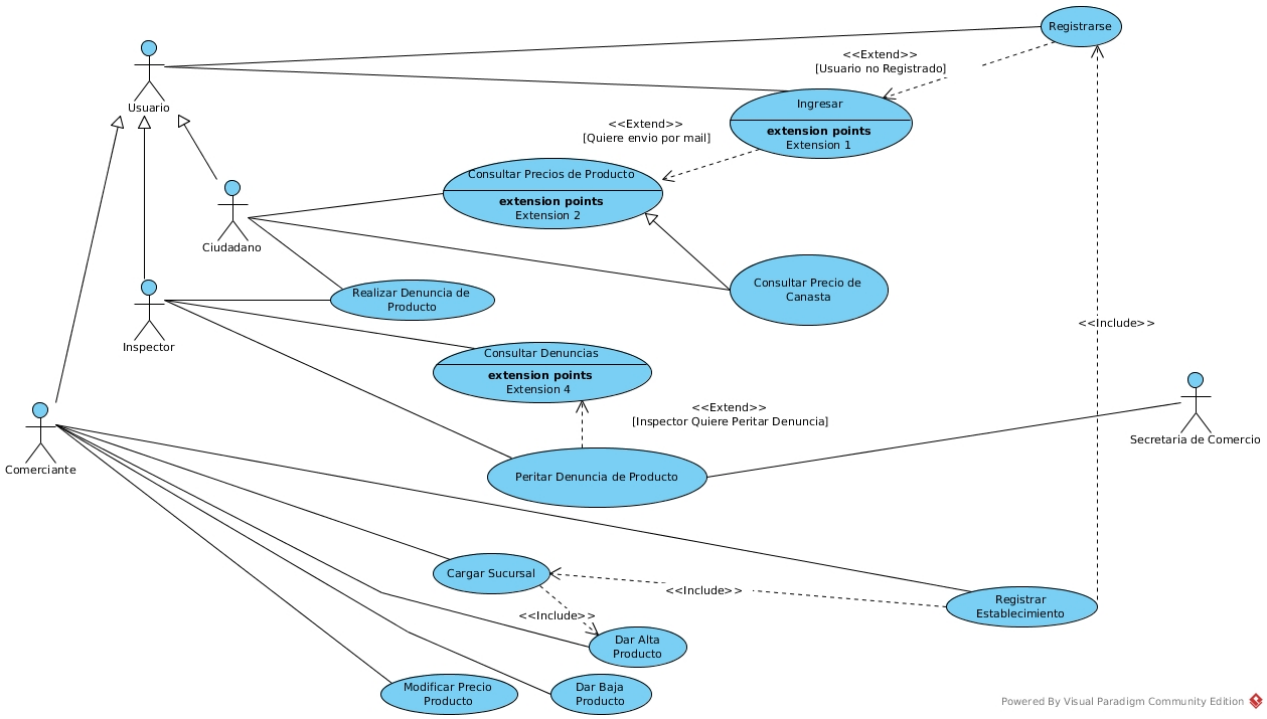
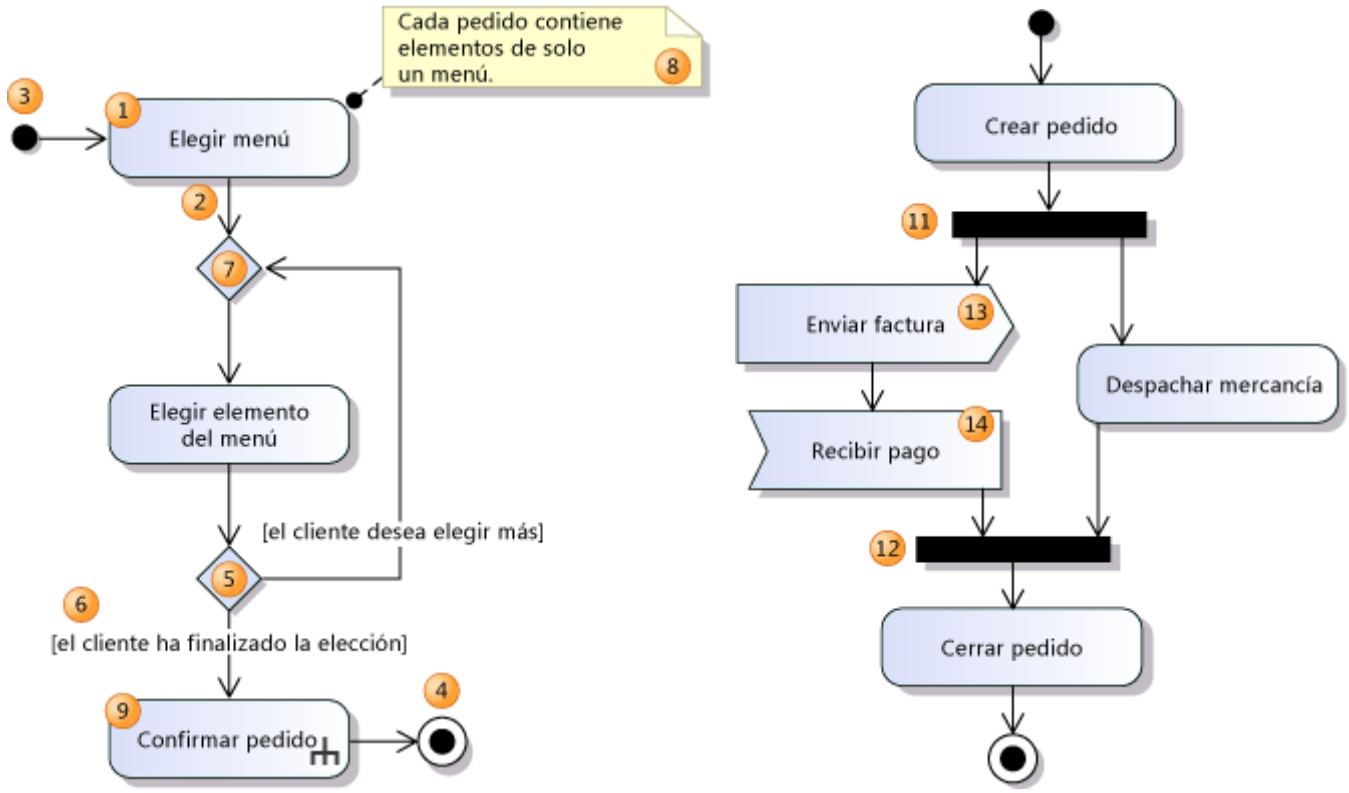


Diagrama de Actividades:

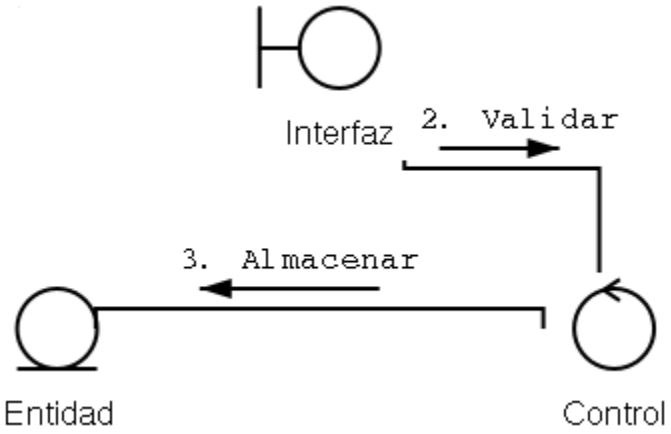


1. **Acción**: Paso de la actividad en el que los usuarios o el software realizan alguna tarea.
2. **Flujo de Control**: Conector que muestra el flujo de control entre las acciones.
3. **Initial Node:** Indica la primera acción o las primeras acciones de la actividad.
4. **Activity Final Node:** Fin de la actividad.
5. **Branch:** Bifurcación condicional de un flujo. Tiene una entrada y dos o más salidas.
6. **Guarda:** Condición que especifica si un token puede fluir por un conector. Se usa con más frecuencia en los flujos salientes de un nodo branch.
7. **Merge/Unbranch:** Necesario para combinar los flujos que se dividieron mediante un nodo de decisión. Tiene dos o más entradas y una salida.

11. **Fork Node:** Divide un único flujo en flujos simultáneos.
14. **Join Node:** Combina flujos simultáneos en un único flujo.
15. **Swim Lanes**: Limita las actividades que realiza cada actor.

Se puede usar este diagrama en la etapa de INCEPTION pero primordialmente se utiliza en la etapa de ELABORACION para complementar los CU.

Diagrama de Clases de Análisis:

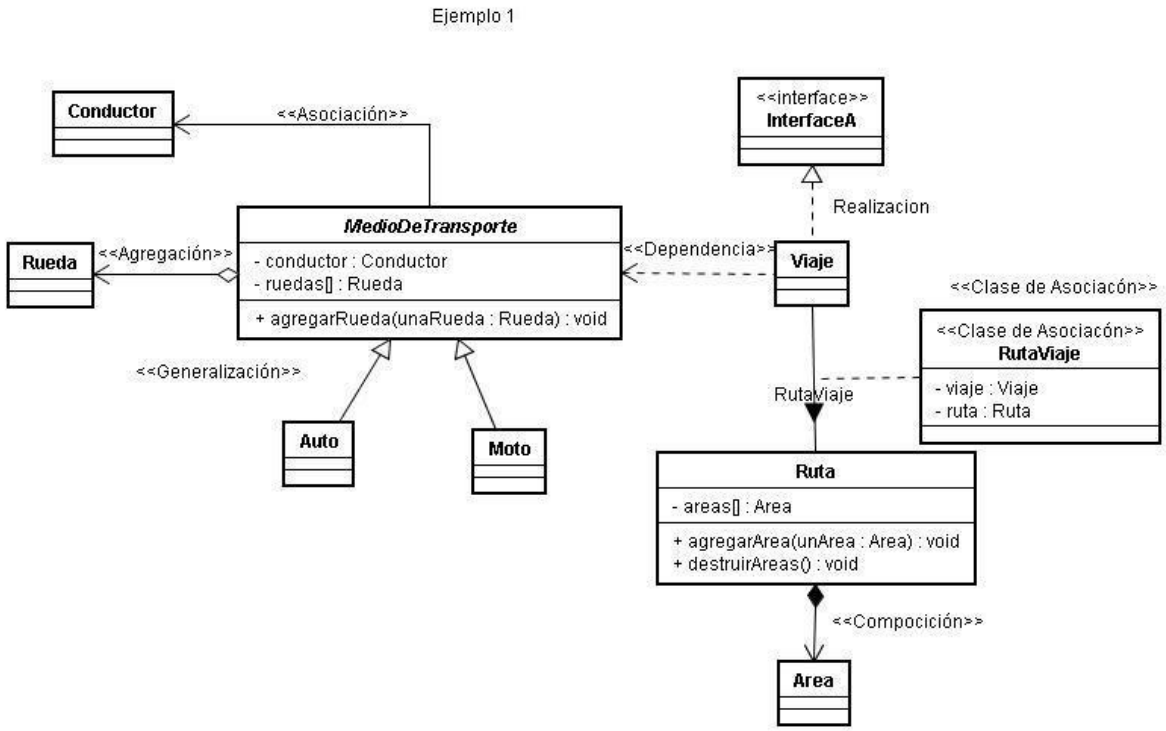


- Control Modela la coordinación, secuencia, transacciones y control de otros objetos y representa el flujo de uno o más casos de uso.

- Entidad Es una clase que es usada para modelar información y comportamiento asociado que debe tener una vida larga y a menudo persistente.

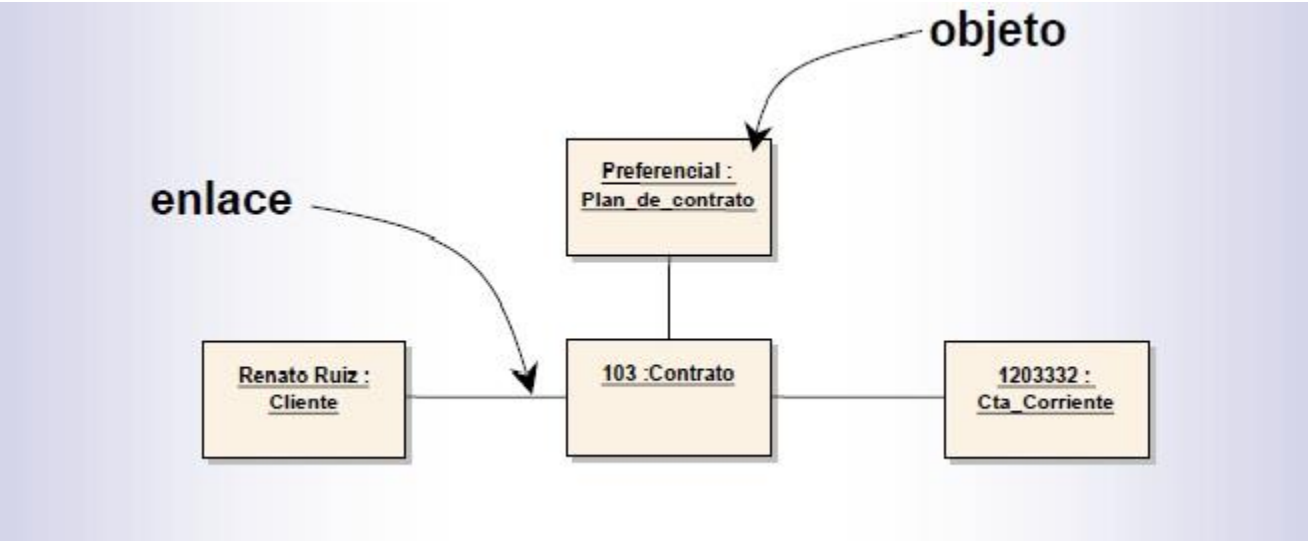
- Interfaz Interactúa con un usuario o un programa externo, ya sea un GUI o un API.

Diagrama de Clases:



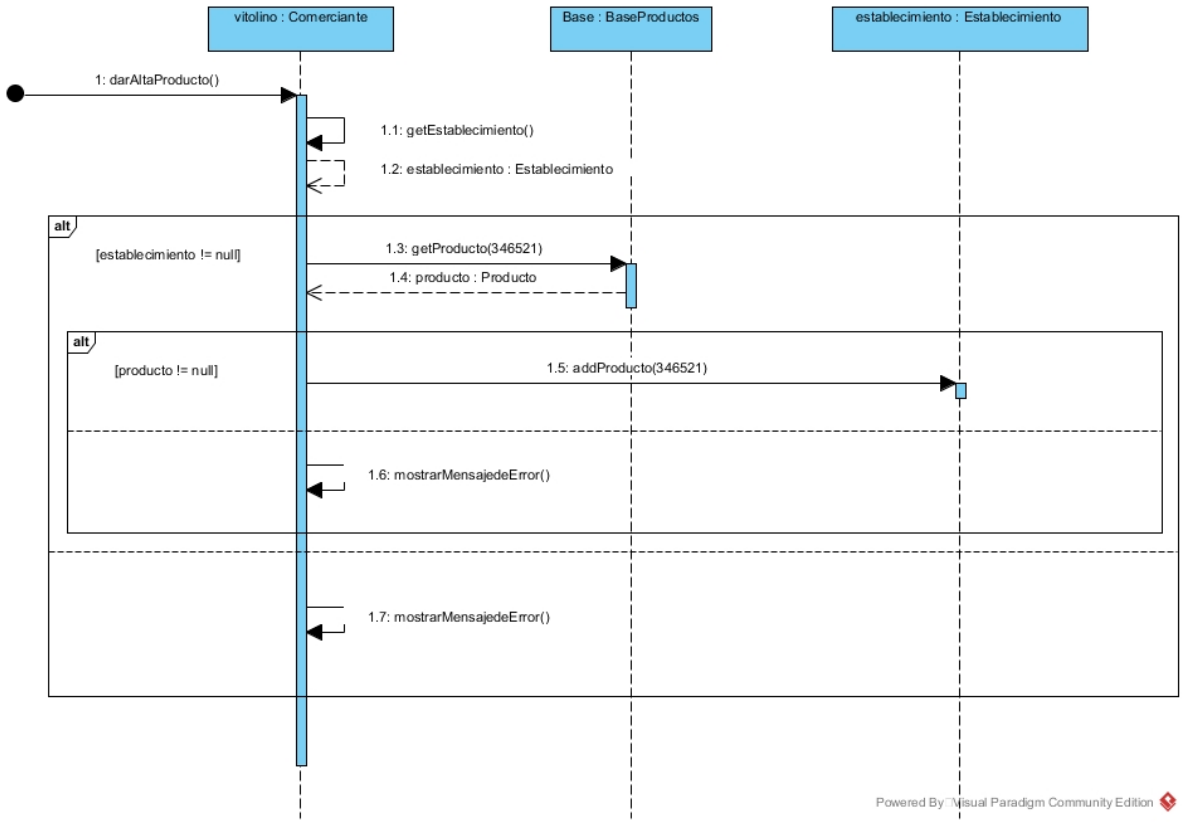
1. **Dependencia**: Cuando una clase "usa" a otra clase
    a. Retorno de un método
    b. Variable interna de una operación
    c. Un parámetro de un método
    d. Import
2. **Asociación:** Relación "estructural" entre dos clases. Una clase tiene un objeto de otra
    a. Atributo de Instancia
    b. Si hay asociación, no se dibuja dependencia
3. **Generalización (Herencia)**:
    a. El nombre de la clase abstracta se coloca en cursiva
4. **Agregación:** Clase A "está formada por" clase B. Cuando se destruye A no se destruyen los elementos de B
5. **Composición:** Mismo que agregación pero, cuando se destruye A se destruyen los elementos de B relacionados.
6. **Clase de Asociación:** Similar a una relación N:N en bases. Cuando A y B están relacionadas se crea C que contendrá elementos de la relación.
7. **Realización:** Cuando una clase utiliza una interfaz.

Diagrama de Objetos:



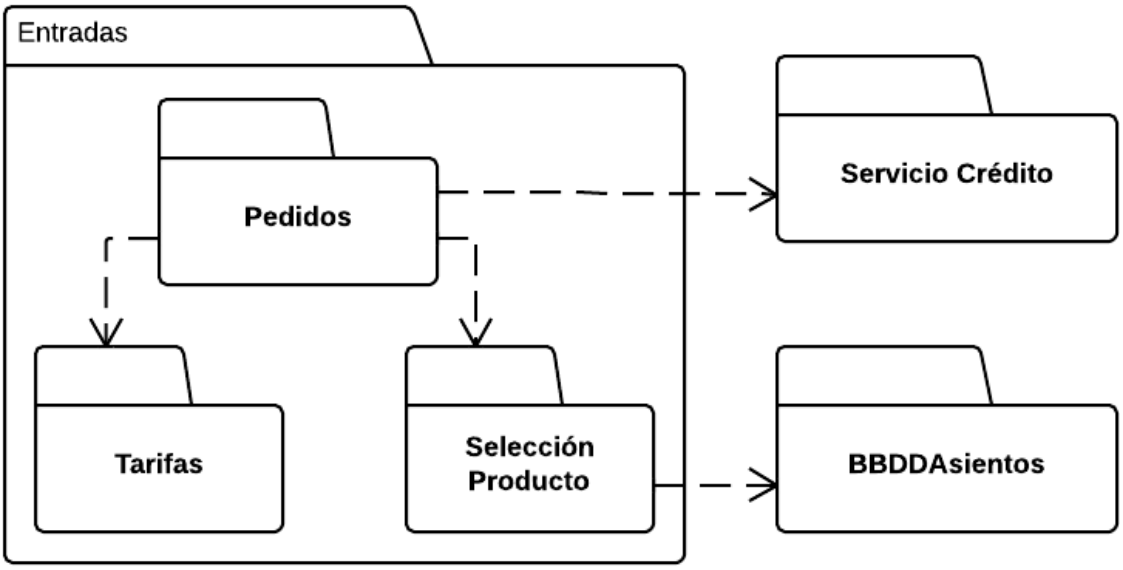
Modela las instancias de las clases en un determinado momento. Nunca olvidar que los nombres de cada objeto van subrayados y de la siguiente manera: Instancia: Clase

Diagrama de Secuencia:



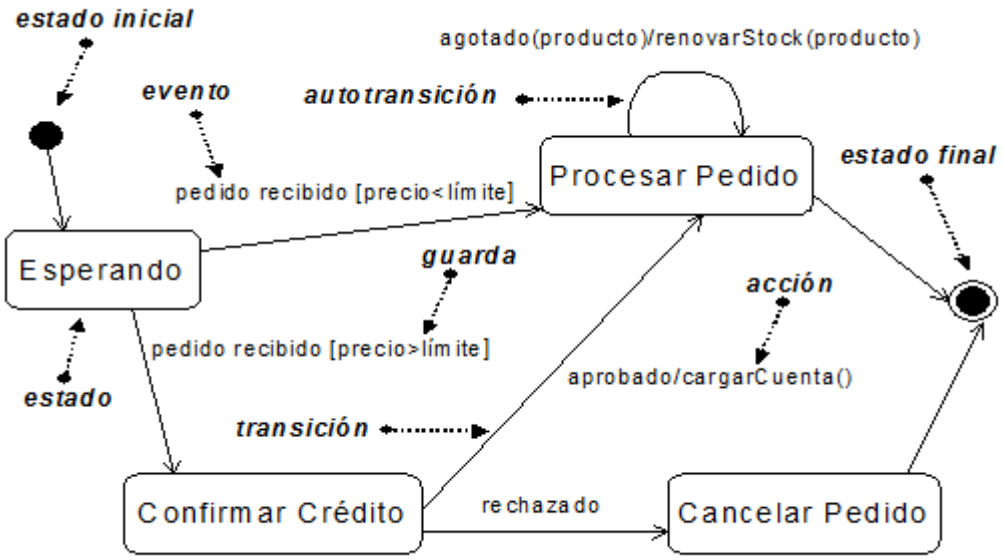
Se utiliza para modelar el flujo de control de una operación.

Diagrama de Paquetes:



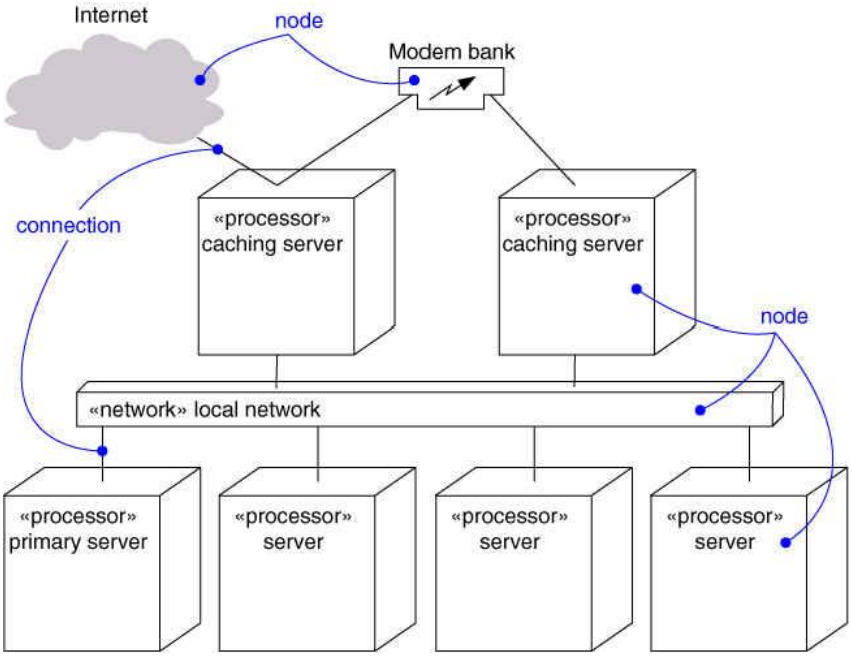
- Permite organizar los elementos modelados con UML, facilitando de ésta forma el manejo de los modelos de un sistema complejo.
- Define un espacio de nombres: Dos elementos de UML pueden tener el mismo nombre, con tal y estén en paquetes distintos.
- En este sentido, son similares a los paquetes en Java.
- Los paquetes pueden ser simples estructuras conceptuales o pueden estar reflejados en la implementación
- Permiten dividir un modelo para agrupar y encapsular sus elementos en unidades lógicas individuales
- Pueden tener una interfaz (métodos de clases e interfaces exportadas) y una realización de éstas interfaces (clases internas que implementan dichas interfaces)
- Los paquetes pueden estar anidados unos dentro de otros, y unos paquetes pueden depender de otros paquetes

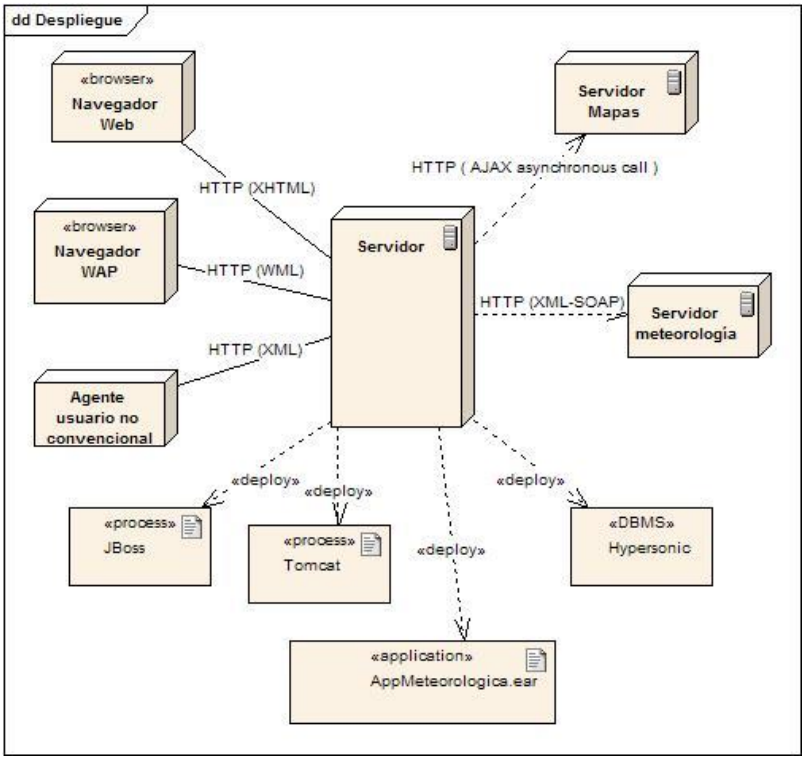
Diagrama de Transición de Estados:



- Muestran una Máquina de Estado
- Son útiles para modelar la vida de un objeto
- Muestra el flujo de control entre estados (en qué estados posibles puede estar "cierto algo" y como se producen los cambios entre dichos estados)
- Una máquina de estados es un comportamiento que especifica las secuencias de estados por las que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus respuestas a esos eventos
- **Estado**: es una condición o situación en la vida de un objeto durante la cual satisface una condición, realiza alguna actividad o espera algún evento
- **Evento**: es la especificación de un acontecimiento significativo que ocupa un lugar en el tiempo y en el espacio. Es la aparición de un estímulo que puede (o no) activar una transición de estado
- **Transición**: es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones especificadas
- **Acción**: Se ejecuta cuando se dispara la transición
- **Guarda**: Condición por la que se realiza el evento

Diagrama de Deployment:





- Muestra las relaciones físicas entre los componentes hardware y software en el sistema final, es decir, la configuración de los elementos de procesamiento en tiempo de ejecución y los componentes software (procesos y objetos que se ejecutan en ellos)
- Describen la arquitectura física del sistema durante la ejecución
- Describen la topología del sistema: la estructura de los elementos de hardware y el software que ejecuta cada uno de ellos
- Nodos: son objetos físicos que existen en tiempo de ejecución, y que representan algún tipo de recurso computacional (capacidad de memoria y procesamiento)
- Las asociaciones poseen tipos de comunicaciones que se identifican con un estereotipo que indica el protocolo de comunicación o la red
- Poseen un archivo ejecutable.

Metodología Scrum

Manifiesto Ágil:

- Los **individuos y su interacción**, por encima de los procesos y las herramientas.
- El **software que funciona**, por encima de la documentación exhaustiva.
- La **colaboración con el cliente**, por encima de la negociación contractual.
- La **respuesta al cambio**, por encima del seguimiento de un plan.

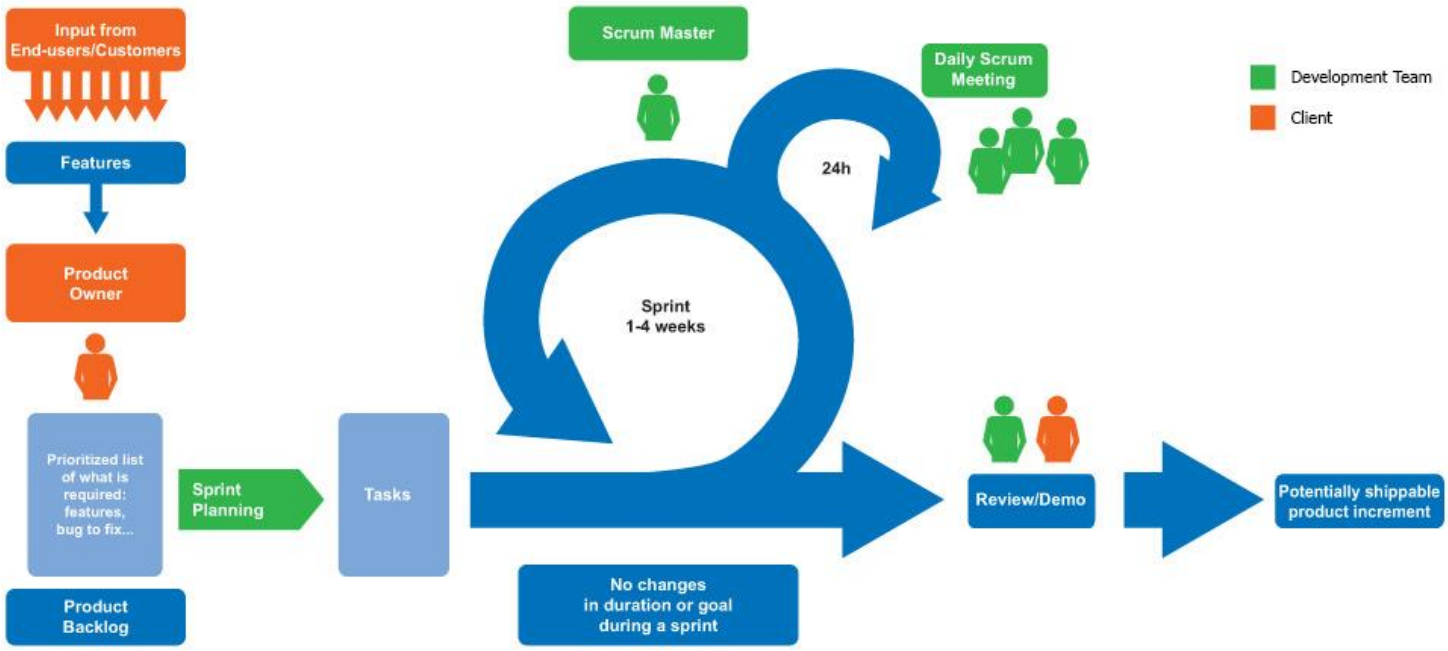
Características Scrum:

- Conformada por equipos de personas multifuncionales
- Necesita de personas con experiencia
- Funciona sobre un paradigma horizontal (No hay jefes en el equipo)
- Tiene herramientas de estrategias para resolución de proyectos

Roles:

- **Product Owner**:
  - Conoce que se desea y porque se desea
- **Scrum Master**:
  - Es un facilitador del proceso de Scrum
- **Scrum Team**:
  - Conoce como y cuán rápido se puede realizar el producto

Framework:



1ra Parte del Sprint:

User Stories:

- Son recordatorios de los requerimientos
- Pueden cambiar, confirmarse y eliminarse durante el Proyecto
- Lo más importante es la conversación con el cliente
- Describe un requerimiento funcional
- No es la única documentación de los requerimientos
- Su objetivo es determinar:
  - **Quién**
  - **Qué**
  - **Para qué**
- Se pueden complementar con documentos adicionales enlazados a las notas de las User Stories
- No pueden ser modificadas hasta que no termine el ciclo (Sprint)
- Ej: "**Como** inquilino del edificio **quiero** ver el calendario de pago de expensas **para** organizar mi agenda de pagos"
- Reverso de User Stories:
  - Especifica las salidas de la funcionalidad
  - Verifica la funcionalidad
  - Está directamente relacionado con el testing
  - Describen requerimientos no funcionales
  - Solo se puede desechar cuando se cumplen todas las condiciones
  - Su objetivo es determinar:
    - **Que**
    - **Cuando**
    - **Entonces**
  - Ej: "Dado que <**condición**> Cuando <**causa**> Entonces <**efecto**>"

Requerimientos en Profundidad:

- **THEMES**:
  - Tema de nuestro sistema ("Incrementar tráfico del sitio")
- **EPIC**:
  - Abstracto y grande ("Agrega nueva sección de video")
- **STORIES**
- **TASKS**:
  - Tarea atómica, desprende de las Stories ("Validar con BD", etc.)

Splitting:

- Dividir las **Epic** en **Stories** y las **Stories** en **Tasks**
- Las user stories deben ser estimables y capaces de construirse en una iteración
- Se buscan user stories efectivas
- En general, no tiene sentido dividir una user story:
  - Por debajo de un tiempo estimado de 2 o 5 días
  - En tareas: Hay que dividir por funcionalidad, no por tareas!

2da Parte del Sprint:

Backlog:

- **Product Backlog**:
  - Todas las user Stories validadas y preparadas con sus Tasks
  - Se charla con el cliente para decidir la prioridad de ejecución
  - No se puede quitar o agregar nada
- **Sprint Backlog**:
  - Se ordenan las primeras Tasks a realizar y se orden de las demás
  - Se realiza una estimación de la duración de las Tasks en el Product Backlog poniéndoles un peso estimado a cada una (Una técnica muy utilizada es la de Planning Poker)

Release Plan:

- Se ubican User Stories en Iteraciones de acuerdo a los **Story Points**, la velocidad y el objetivo del Sprint

3ra Parte del Sprint:

Tasks Board:

- Se reparten las Tasks de cada User Story de forma auto personal.
- Todas las Tasks tienen que estar realizadas para pasar la User Story a terminada y también tiene que ser validada

Daily Scrum:

- Asegurarse de que todos los miembros se auto-asignen tareas
- Documentar cuanto sea necesario
- No perder de vista el Sprint Goal
- Siempre se busca un incremento de producto

Daily Meeting:

- Cada día del Sprint, el equipo se reúne en una **daily scrum meeting**
- El time-box es estricto a **15 minutos**
- Se plantean los problemas, no se resuelven
- La reunión es obligatoria para todos los miembros
- Preguntas que siempre se deben hacer:
  - **¿Qué hiciste ayer?**
  - **¿Qué vas a hacer hoy?**
  - **¿Tenes algún impedimento?**

4ta Parte del Sprint:

Sprint Review:

- Se prepara un **DEMO** para validar las **User Stories** con el cliente
- Se asegura que el producto funciona bajo condiciones aceptables
- Se recibe un **feedback** acerca del incremento del producto

Sprint Retrospective:

- Cada equipo mantiene una reunión privada
- Se reflexiona acerca de lo que sucedió durante el sprint
- Cada miembro expone su opinión respecto a:
    - Algo para empezar a realizar (**UNA MEJORA**)
    - Algo para finalizar de hacer (**LO MALO**)
    - Algo que se debe continuar realizando (**LO BUENO**)

Manejo de Código

Subversión:

- Sistema de control de versiones.
- Sistema centralizado para compartir información.
- Gestiona archivos y directorios, y sus cambios a través del tiempo.
- Se puede recrear un proyecto desde cualquier momento en su historia
- **Ventajas**:
    - Similar a un sistema de archivos
    - Modificaciones atómicas.
    - Se guardan/obtienen sólo diferencias
    - Acceso mediante Apache, sobre WebDAV/DeltaV.
    - Eficiente manejo de binarios
    - Trabajo colaborativo.
- **Desventajas**
    - Renombrado de archivos (rename) incompleto.
    - Ineficiente manejo de parches.
    - No implementa algunas operaciones administrativas
    - NO es exactamente un sistema de archivos (para copiar un proyecto se debe exportar)
- **Estructura TTB**
    - **Trunk**: Rama donde se está desarrollando.
    - **Tags**: Versiones etiquetadas y cerradas.
    - **Branches**: Evoluciones paralelas al Trunk.

Git:

- Software de control de versiones
- Orientado a la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.
- **Características**:
    - Código distribuido
    - Pensado para desarrollo no lineal
    - Eficiente gestión de ramas y merges de diferentes versiones.
    - Basado en el principio de que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.

    - Cada programador obtiene una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local.
- **Working dir**: Carpeta donde se trabaja
- **Index**: Registro de archivos modificados/creados
- **Head**: Último commit realizado

Documentación de Código Fuente:

- Artefactos vitales de un sistema informático
- La documentación de programas sirve para **dejar registro de** :
    - Decisiones de implementación
    - Estructuras de artefactos
    - Cambios (informales)
- **Documentación visible**:
    - Proyecto (Formal)
    - Local al código fuente (Comentarios. Informal)
- **Ventajas**
    - Se documenta utilizando lenguaje natural
    - Especificación local al código
    - Permite comprender razonamientos y decisiones del programador
- **Desventajas**
    - La documentación debe ser realizada por el programador

Buenas prácticas:

- **Primera oración**
    - Resumen/descripción concreta pero precisa del ítem.
    - Aplica a miembro, clase, interface o paquete.
- **Dependencia de la implementación**
    - No debe indicar detalles de la implementación (solo si fuese necesario)
    - Definir que es requerido y que variar cuando se cambia de plataforma o implementación.
    - Información para poder crear casos de tests sin leer el código fuente. (información de condiciones extremas, rangos de parámetros)
- Si se va a describir un comportamiento específico de la implementación o plataforma, hacerlo en un párrafo separado con una frase que indique que es específico para esa implementación o plataforma
- Utilizar frases en vez de oraciones completes
- **Utilizar tercera persona** (declarativa) y no segunda persona (imperativa)
- Un método implementa una operación, entonces, empezar frases con verbos:
    - **Obtiene la etiqueta de este botón (recomendado)**
    - **Este método obtiene la etiqueta de este botón. (evitar)**
- Cuando se refiere a instancias de objetos creadas por la clase utilizar “este” en vez de “el”.

JavaDoc:

- Utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java.
- Es el estándar de la industria para documentar clases de Java.

Doxygen:

- Basado en bloques de comentarios
  - Brief
  - Detailed
  - In body (para métodos y funciones)
- Sirve para Documentar: C/C++, C#, Objective-C, PHP, Java, Python, VHDL, Fortran, Tcl

Validación de Código:

- Es posible embeber código que facilite la validación del código fuente
  - **Aserciones**
- Algunos errores pueden surgir de la incorrecta codificación
  - Herramientas de **Linting**
- En algunos casos el código puede implementar lo que indican los requerimientos
  - Especificaciones sobre cómo programar (**estándares**)

Aserciones:

- Es una condición lógica insertada en el código fuente que se asume cierta. El sistema se encarga de comprobarlas y disparar excepciones en caso de no cumplirse
- **Útil en implementación y testeo** – Sobrecarga en producción
- **Motivación**:
  - **Diseño por contrato**
    - **Precondición**: Que asumimos como cierto al comenzar el programa
    - **Postcondición**: Que asumimos como cierto al terminar el programa
    - **Invariante** (de bucle): Que se asume como cierto durante la ejecución del programa
- **Si usarlas**:
  - ... a la entrada de métodos privados
  - ... a la salida de métodos
  - ... para corroborar las variables y estructuras de datos internas
  - ... en estructuras de control else y switch cuando todos los casos correctos están explícitos (es decir, cuando la rama else no debería tomarse jamás)
  - ... en bucles largos
- **No usarlas**:
  - ...a la entrada de métodos públicos
  - ...para detectar errores en los datos de entrada al programa

Linting (PMD, JSLint, etc):

- Es una técnica de **análisis estático**.
- Busca estructuras sospechosas.
- **Ejemplos**:
  - Variables que pueden exceder su rango
  - Índices de arreglos fuera de límites.
- Utilizado por los compiladores para realizar optimizaciones

Profiling (Atom, Ruby, Shark, etc):

- **Análisis dinámico**
- **Aplica**:
  - Complejidad
  - Uso de memoria
  - Activaciones de unidades
  - Frecuencia/Duración de llamados
- **Granularidad**
  - Basado en eventos
  - Estadístico
  - Instrumentado

Testing de Unidad

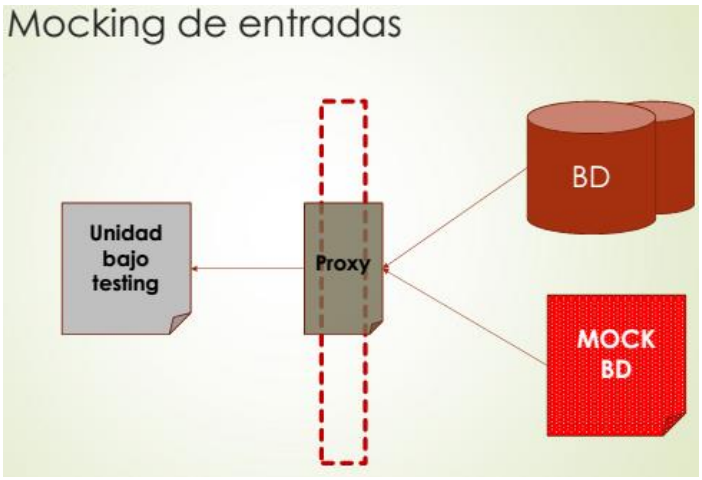
JUnit:

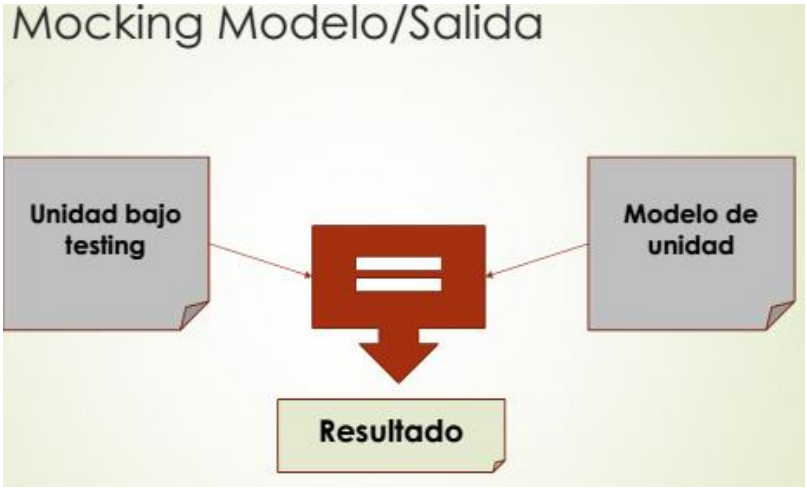
- Es un **framewok** que permite realizar la ejecución de clases Java de manera controlada, y de esta forma evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.
- **Ventajas**:
  - Simple.
  - Código Java.
  - Herramienta gratuita.
  - Fácil de aplicar en un proyecto.
  - Tests implementados por el desarrollador.
  - Autoevaluación de resultados y feedback inmediato
  - Jerarquía de tests en árbol.
  - Software más estable.
- **Diseño**: Patrones **Command** y **Composite**
- **TestCase**: La clase que posee métodos para testear debe extender de TestCase y utilizar la anotación **@Test**.
- **TestSuite**: Composición en forma de árbol de TestCase
- **Pasos**:
  - Escribir la clase con los **Tests**
  - Implementar el método **setUp()**.
  - Implementar el método **tearDown().**
  - Definir métodos **TEST_XXX()** cómo públicos para probar y verificar el resultado.
  - Implementar una clase **suite** (sin métodos)

    - Implementar un **Runner** para la Suite con un método **main()**
- **Asserts**:
    - **assertTrue(expresión):** Comprueba que expresión evalúe a true
    - **assertFalse(expresión):** comprueba que expresión evalúe a false
    - **assertEquals(esperado,real):** comprueba que esperado sea igual a real
    - **assertNull(objeto):** comprueba que objeto sea null
    - **assertNotNull(objeto):** comprueba que objeto no sea null
    - **assertSame(objeto_esperado,objeto_real):** comprueba que objeto_esperado y objeto_real sean el mismo objeto
    - **assertNotSame(objeto_esperado,objeto_real):** comprueba que objeto_esperado no sea el mismo objeto que objeto_real

Mocking:

- Técnica que permite reemplazar el comportamiento complejo (o impráctico) de un objeto por otro que lo simule.
- Usado en testeo de unidades para eliminar dependencias.
- **Aplicaciones**:
    - Generación de entradas
    - Comprobación de salidas





TestNG:

- **Framework** inspirado en Junit y Nunit
- Aplicable desde testeo de unidad hasta testeo de integración.
- Provee soporte para anotaciones
- Test dirigido por datos
- Tests parametrizables
- **Pasos**:
    - Escribir la lógica del test e insertar anotaciones
    - Agregar información de los tests en un archivo XML (testng.xml o build.xml)
    - **Tags**
        - **<suite>** Puede contener uno o más tests
        - **<test>** Puede contener uno o más clases

        - **<class>** Una clase TestNG que puede contener uno o más métodos **(@Test**)
    - **Anotaciones**:
        - **@BeforeSuite** El método se ejecutará antes de los tests
        - **@AfterSuite** El método se ejecutará después de los tests
        - **@BeforeTest** El método se ejecutará antes de cualquier test dentro **<test>**
        - **@AfterTest** El método se ejecutará después de cualquier test dentro **<test>**
        - **@BeforeGroups** El método se ejecutará una vez antes del primer test del grupo
        - **@AfterGroups** El método se ejecutará al finalizar todos los test del grupo
        - **@BeforeClass** El método se ejecutará antes que cualquier método de la clase sea invocado.
        - **@AfterClass** El método se ejecutará luego que todos los métodos de la clase hayan sido invocados.
        - **@BeforeMethod** El método se ejecutará antes que cada método de test
        - **@AfterMethod** El método se ejecutará luego de cada método de test
        - **@DataProvider** Método proveedor de datos.
            - El **@Test** que lo usará debe debe usar el name del **dataProvider**.
        - **@Factory** Objetos que serán usados por clases TestNG.
        - **@Listeners** Listeners de una clase de test
        - **@Describe** como se pasan parámetros a un método **@Test**
        - **@Test** Marca una clase/método como parte del Test
            - **alwaysRun**: El método se ejecutará aun cuando fallen los anteriores
            - **dataProvider** Nombre del proveedor para este método
            - **dataProviderClass** Clase donde buscar el **dataProvider**
            - **dependsOnGroup** Lista de grupos que dependen de este método
            - **enabled** Métodos de esta clase están habilitados o no
            - **expectedExceptions** Lista de excepciones que se esperan recibir
            - **groups** Lista de grupos a los cuales pertenece el método
            - **invocationCount** Número de veces que se invocará el método
            - **invocationTimeOut** Tiempo máximo para ejecutar **invocationCount**
            - **priority** Prioridad del método (bajo número, alta prioridad)
            - **successPercentage** Porcentaje esperado de éxitos

- **singleThreaded** Todos los métodos de la clase se ejecutarán en un único hilo
- **timeOut** Tiempo que tomará como máximo el test