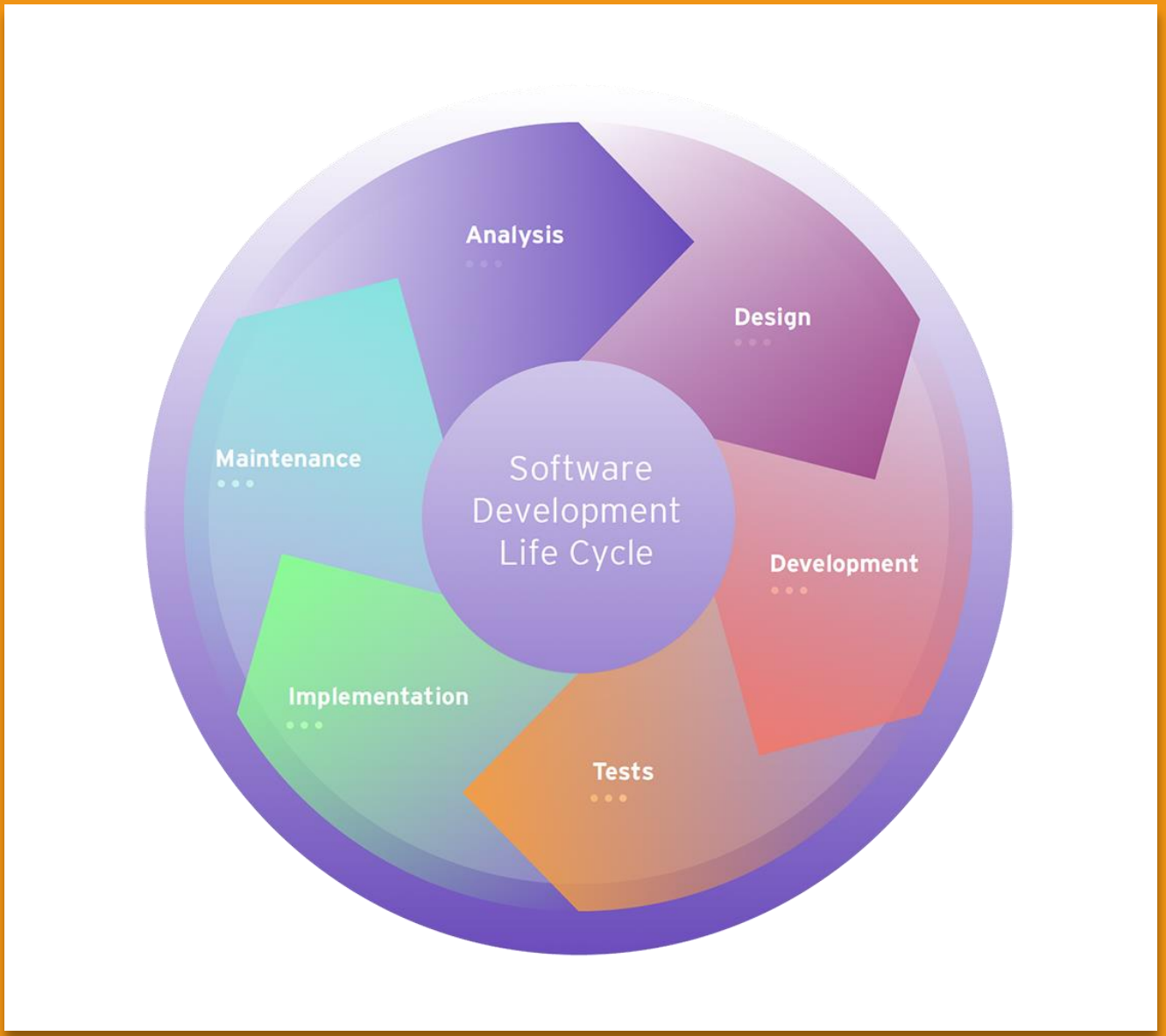
Tutoría

Ingeniería de Software

Procesos de Software



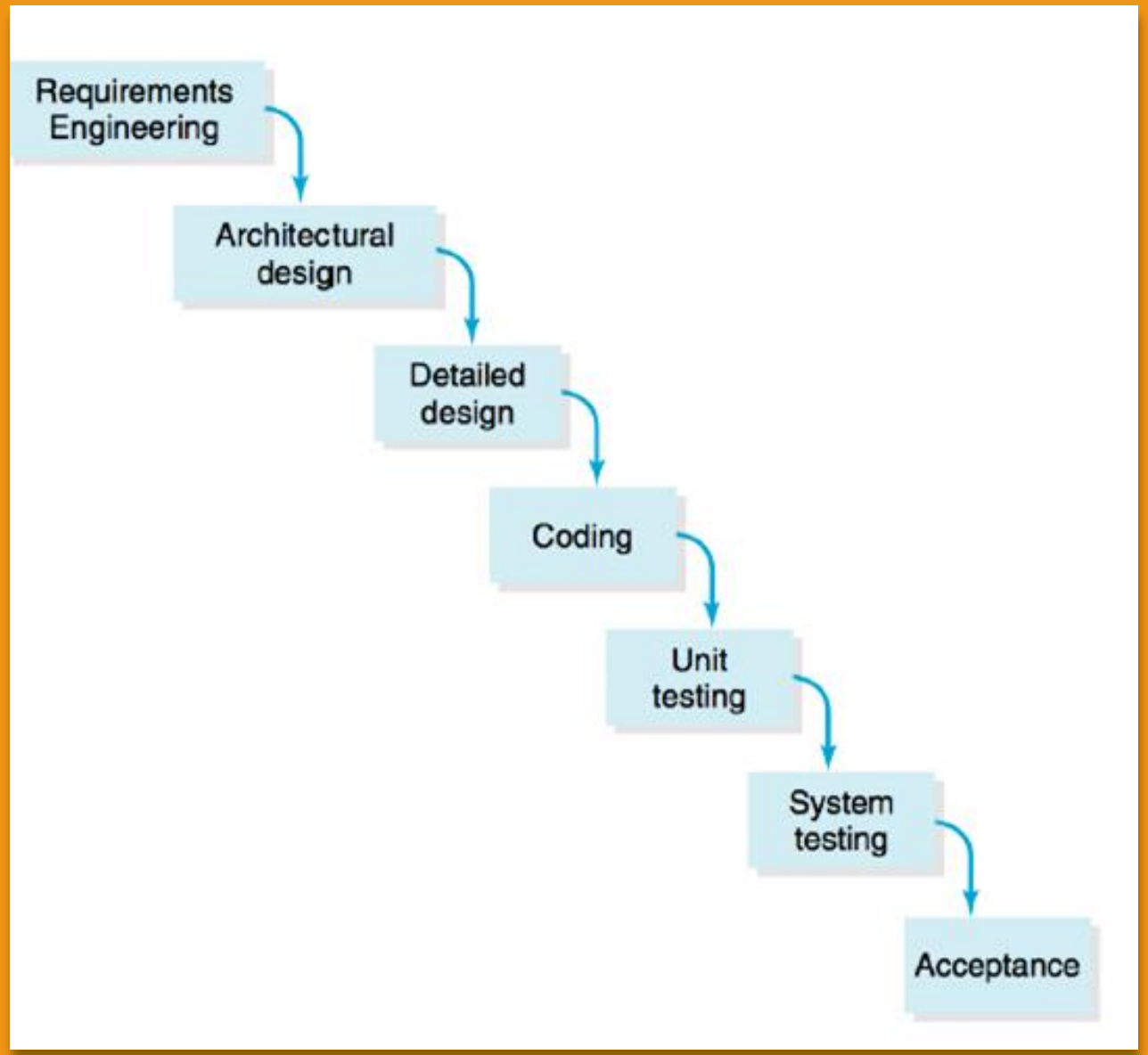
Actividades del proceso

- Involucra muchas actividades distintas:
  - Estudios de factibilidad (precio, riesgo, necesidad)
  - Ingeniería de requerimientos
  - UX/UI
  - Arquitectura
  - Diseño
  - Programación
  - Integración
  - Testing y QA
  - Poner en producción
  - Mantención
  - Documentación
  - Gestión del proyecto

¿Qué orden debemos seguir para ejecutar estas actividades? ¿Quién lo hace? ¿Se deben repetir?

El modelo en cascada

LA MÁS CONOCIDA Y SIMPLE DE TODAS.

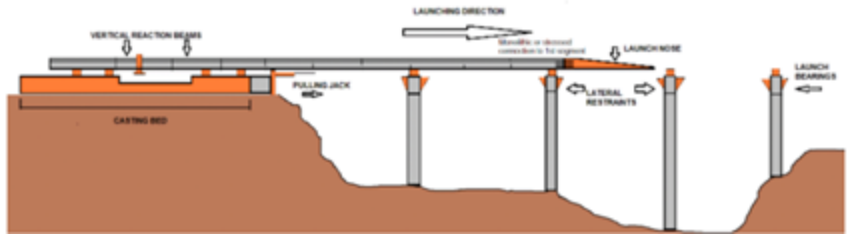


Modelo Iterativo

- Para disminuir el riesgo se puede usar este tipo de modelos.
- Se basa en diseñar una implementación inicial, exponerla al comentario del usuario y desarrollarla en diversas versiones hasta producir un sistema adecuado (Iteraciones).
- No necesariamente cada iteración produce código.

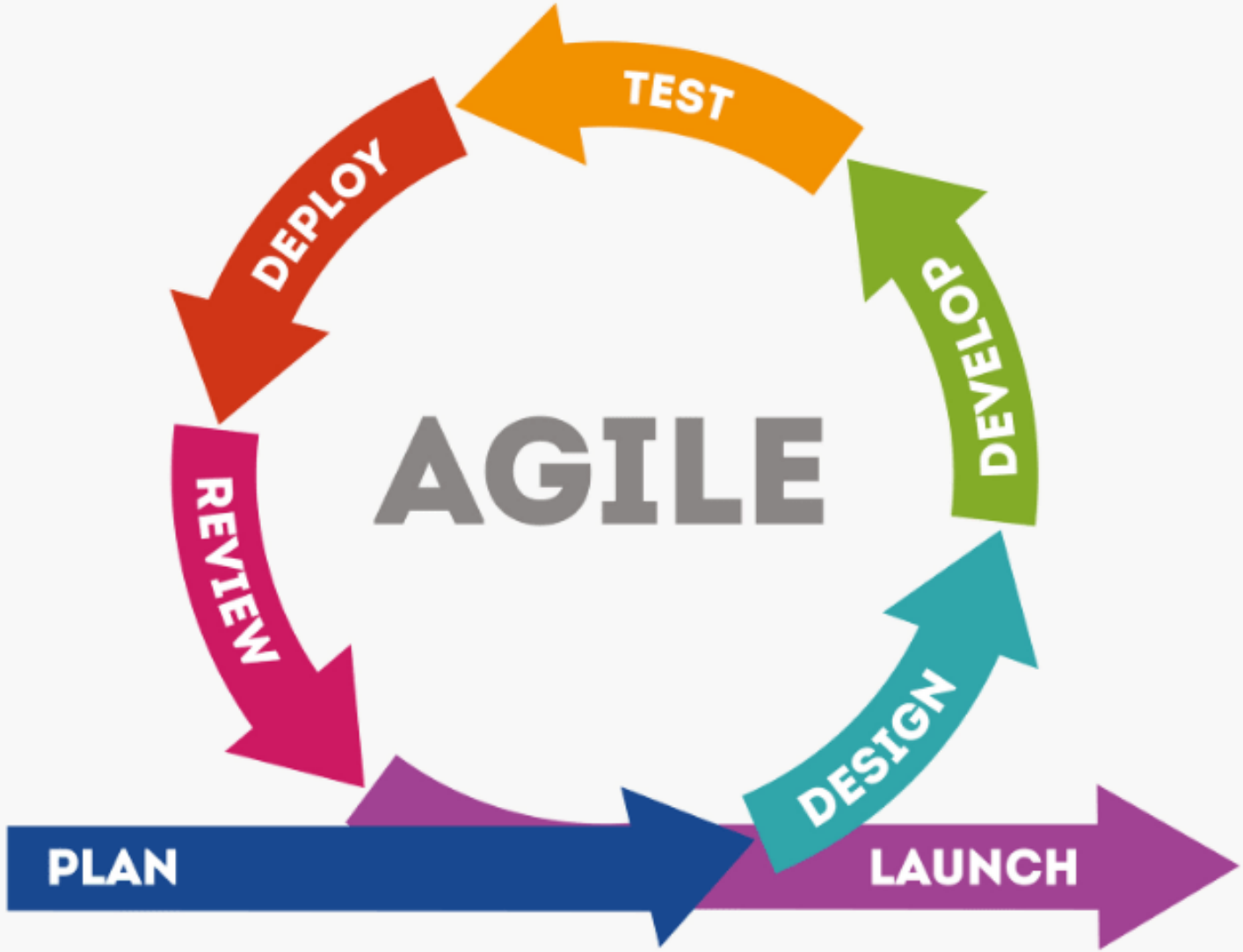
Modelo Incremental

- Software se produce en pequeños incrementos.
- Cada incremento incluye funcionalidades completas (probadas y validadas).
- Producto se completa con entrega de último incremento
- Incrementos comunmente asociados a casos de uso (o historias de usuario).



Metodologías Ágiles

- Todas las metodologías ágiles comparten características:
  - El proceso de especificación, diseño e implementación están mezclados.
  - El sistema se desarrolla incrementalmente por etapa, por lo general, se crean las nuevas liberaciones del sistema, y cada dos o tres semanas se ponen a disposición de los clientes.
  - La metodología se apoya por el uso de herramientas específicas para testing, manejo de configuraciones, integraciones de sistema, interfaz de usuario, etc.
- **¿Pueden pensar en alguna herramienta específica que conozcan?**



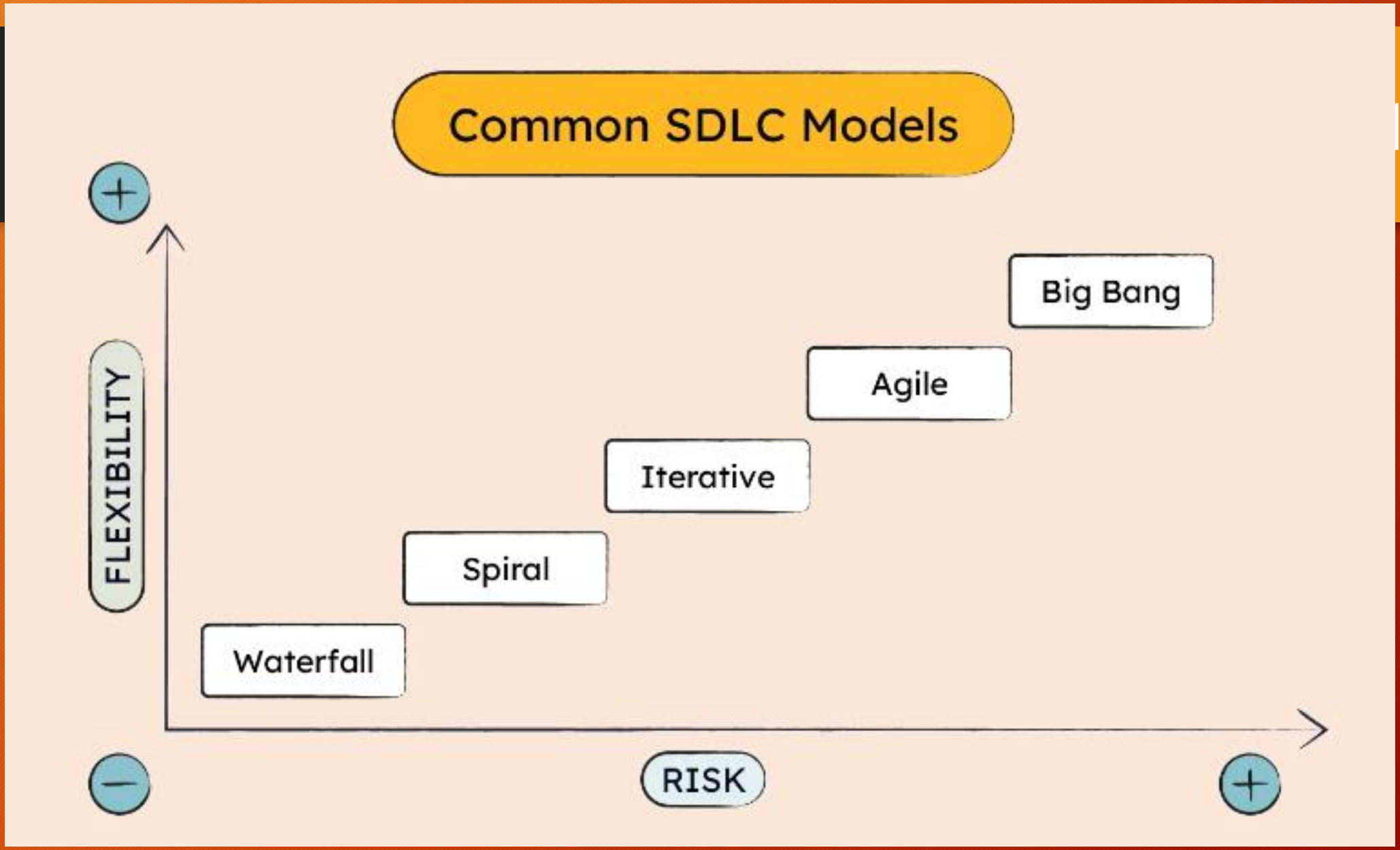
Agile Manifesto

"Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- **Individuos e interacciones** sobre procesos y herramientas
- **Software funcionando** sobre documentación extensiva
- **Colaboración con el cliente** sobre negociación contractual
- **Respuesta ante el cambio** sobre seguir un plan

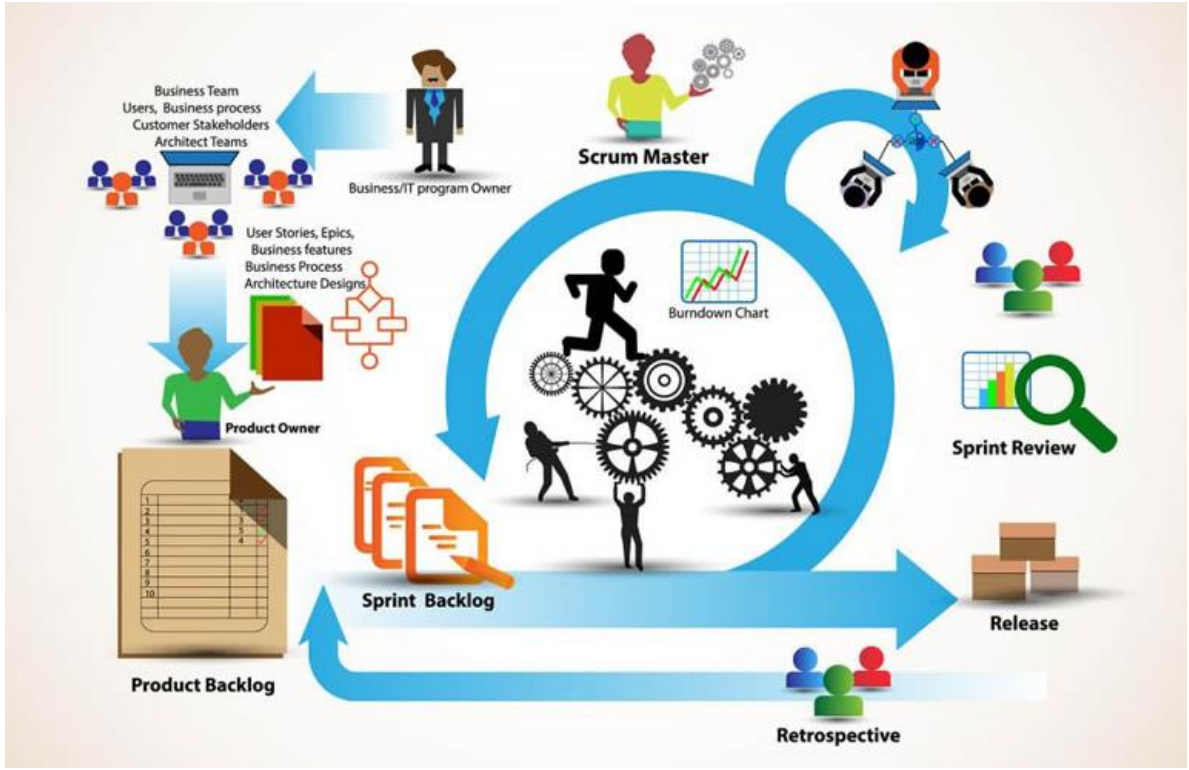
Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda."

http://agilemanifesto.org


Common SDLC Models
+
FLEXIBILITY
−
Waterfall
Spiral
Iterative
Agile
Big Bang
RISK
+

Scrum

Profesor: Matías Rojas



Factores para decidir entre enfoque basado en un plan o ágil

- Importancia de una especificación y diseño detallados antes de la implementación.
- Práctica de entrega incremental para obtener retroalimentación de los clientes.
- Tamaño del sistema que se desarrollará.
- Tipo de sistema que se desarrollará.
- Tiempo de vida que se espera del sistema.
- Tecnologías disponibles para apoyar el desarrollo del sistema.
- Organización del equipo de desarrollo.
- Problemas culturales que afecten el desarrollo del sistema.
- Habilidades del equipo de desarrollo.
- Sujeto a regulación externa.

En la práctica, muchas compañías adoptan habilidades ágiles e integran enfoques ágiles con sus procesos basados en un plan para desarrollar sistemas de software ejecutables que cubran las necesidades y funciones útiles para el usuario.

Ingeniería de Requerimientos

13

Profesor: Matías Rojas M.

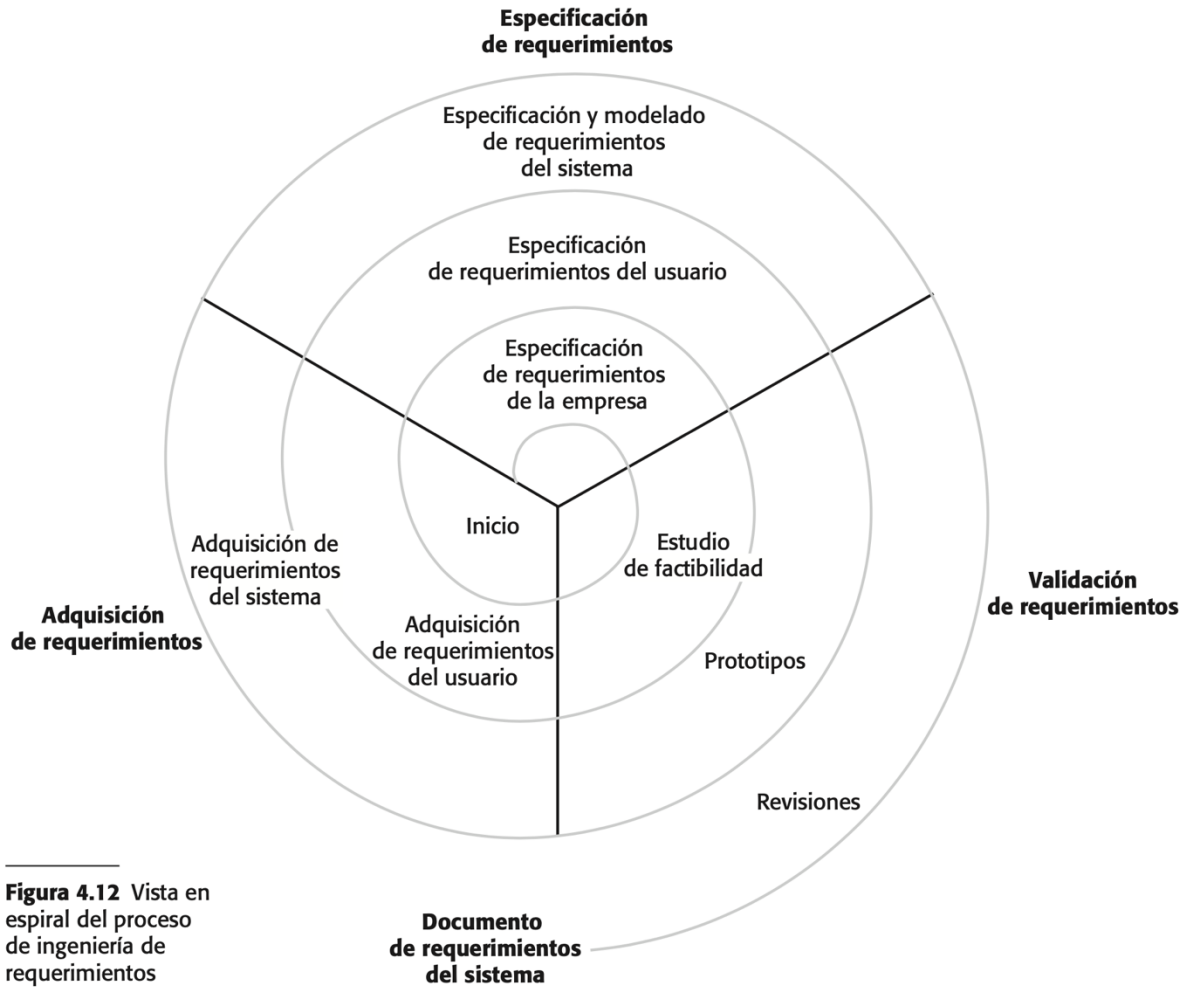


Figura 4.12 Vista en espiral del proceso de ingeniería de requerimientos

Ingeniería de Requerimientos

Establecer necesidades y limitaciones de un cliente para su sistema.

Los requisitos, en sí, son las descripciones de los **servicios del sistema** en función de las necesidades del cliente.

Además, consideran las limitaciones que se generan durante el proceso de ingeniería de requerimientos.

Clasificaciones de Requerimientos

Requerimientos del usuario:

- Servicios que el sistema debe proveer y condiciones bajo las que debe operar.
- Escrito para que los clientes entiendan y escritos en lenguaje natural.
- Pueden incluir diagramas de los servicios del sistema.

Requerimientos del sistema:

- Descripción detalladas y específica de las funciones del sistema, los servicios y las limitaciones operativas.
- Define todo lo que debe ser implementado en el sistema.
- También se conoce como especificación del software.

- **¿A quién va dirigido cada tipo de requerimiento?**

Requerimientos de Usuario y Sistema

Definición del requerimiento del usuario

1. El MHC-PMS elaborará mensualmente informes administrativos que revelen el costo de los medicamentos prescritos por cada clínica durante ese mes.

Especificación de los requerimientos del sistema

1.1 En el último día laboral de cada mes se redactará un resumen de los medicamentos prescritos, su costo y las clínicas que los prescriben.

1.2 El sistema elaborará automáticamente el informe que se imprimirá después de las 17:30 del último día laboral del mes.

1.3 Se realizará un reporte para cada clínica junto con los nombres de cada medicamento, el número de prescripciones, las dosis prescritas y el costo total de los medicamentos prescritos.

1.4 Si los medicamentos están disponibles en diferentes unidades de dosis (por ejemplo, 10 mg, 20 mg) se harán informes por separado para cada unidad de dosis.

1.5 El acceso a los informes de costos se restringirá a usuarios autorizados en la lista de control de acceso administrativo.

Requerimientos Funcionales (RF)

Enunciados acerca de los servicios del sistema que debe proporcionar, cómo el sistema debe reaccionar a entradas generales y cómo el sistema debe comportarse en situaciones particulares.

Requerimientos funcionales de los usuarios pueden ser declaraciones de alto nivel de lo que el sistema debe hacer

Requerimientos funcionales del sistema deben describir los servicios del sistema en detalle.

Requerimientos Funcionales (RF)

Para escribir un requerimiento funcional de sistema, ocupe el siguiente formato:

"El (*tipo de usuario, sistema, componente del sistema*) (*verbo en futuro*) (*descripción*)".

Por ejemplo:

El Administrador podrá crear usuarios de tipo "normal" para que puedan acceder al sistema.	El sistema solo permitirá crear usuarios de tipo normal especificando como identificador único su correo.

Imprecisión de Requerimientos Funcionales

Un RF debe ser **completo** (descripción de todos los servicios del sistema requeridos) y **coherente** (no debe haber conflictos o contradicciones en las descripciones de los servicios del sistema).

Además, un RF debe ser **atómico**, es decir, no puede ser divisible en más RF.

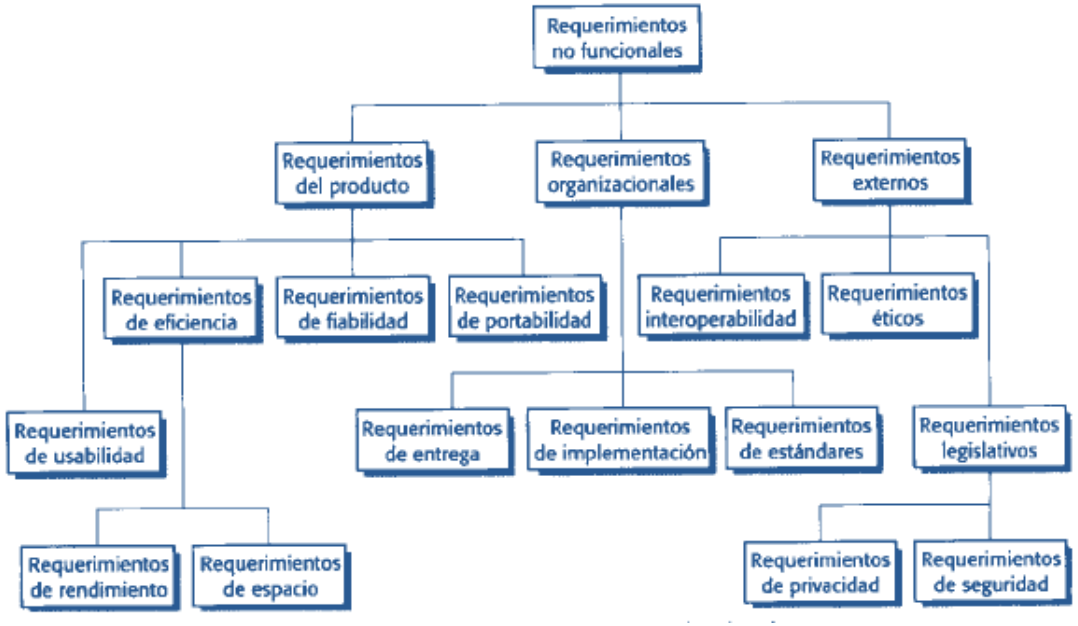
Requerimientos No Funcionales (RNF)

Limitaciones en los servicios o funciones que ofrece el sistema, como restricciones de tiempo, restricciones del proceso de desarrollo, normas, etc.

A menudo se aplica al sistema en su conjunto, en lugar de a las funciones o servicios individuales.

Considerando www.comisariavirtual.cl, específicamente la generación de permisos temporales durante pandemia ¿Qué RNF que debería tener?

Tipos de Requerimientos No Funcionales



Requerimientos No Funcionales

- Ejemplos:
  - La aplicación debe poder ejecutarse en dispositivos móviles.
  - La aplicación debe ser segura.
  - La aplicación debe ser rápida.
  - La aplicación debe ser capaz de soportar una gran cantidad de usuarios.
  - La aplicación debe ser intuitiva.

¿Crees que estos ejemplos están correctos?

Requerimientos No Funcionales

- Ejemplos:
  - La aplicación debe poder ejecutarse en dispositivos con SO Android y iOS, en sus últimas versiones publicadas.
  - La aplicación debe tener sus contraseñas encriptadas con AES 256 en su base de datos.
  - La aplicación debe ser tener un tiempo de respuesta menor a 2 segundos.
  - La aplicación debe ser capaz de soportar 1.000 usuarios accediendo simultáneamente.
  - La aplicación debe seguir estándares de material design.

iso/iec 25010

Atributo de Calidad	Subcaracterística	Métricas
Adecuación Funcional	Completitud Funcional	- Porcentaje de funciones implementadas respecto a las planificadas - Cobertura de requisitos funcionales
	Corrección Funcional	- Número de defectos funcionales detectados - Tasa de fallos por unidad de tiempo
	Pertinencia Funcional	- Eficacia de las funciones en escenarios de uso - Satisfacción del usuario
Eficiencia de Rendimiento	Comportamiento Temporal	- Tiempo de respuesta promedio - Latencia - Throughput (transacciones por segundo)
	Utilización de Recursos	- Uso promedio de CPU y memoria - Consumo de ancho de banda
	Capacidad	- Número máximo de usuarios concurrentes soportados - Volumen de datos procesados
Compatibilidad	Coexistencia	- Conflictos detectados con otros sistemas - Uso compartido de recursos
	Interoperabilidad	- Número de sistemas con los que puede interactuar - Cumplimiento de estándares de interoperabilidad
Usabilidad	Reconocimiento de la Pertinencia	- Tiempo para que el usuario reconozca las funciones disponibles - Tasas de adopción
	Aprendizaje	- Tiempo necesario para aprender a usar el sistema - Número de errores durante el aprendizaje
	Operabilidad	- Número de pasos para completar tareas - Eficiencia en la navegación
	Protección contra Errores de Usuario	- Número de errores prevenidos - Efectividad de los mensajes de error
	Estética de la Interfaz de Usuario	- Nivel de satisfacción estética en encuestas - Consistencia en el diseño de la interfaz
	Accesibilidad	- Cumplimiento de estándares como WCAG - Número de características de accesibilidad implementadas

iso/iec 25010

Atributo de Calidad	Subcaracteristica	Métricas
Fiabilidad	Madurez	- MTBF (Tiempo Medio entre Fallos) - Tasa de fallos en producción
	Disponibilidad	- Porcentaje de tiempo operativo - Tiempo total de inactividad
	Tolerancia a Fallos	- Capacidad para mantener operaciones durante fallos - Número de transacciones fallidas recuperadas
	Recuperabilidad	- MTTR (Tiempo Medio de Recuperación) - Porcentaje de datos recuperados después de un fallo
Seguridad	Confidencialidad	- Número de brechas de seguridad - Porcentaje de datos cifrados
	Integridad	- Número de alteraciones no autorizadas detectadas - Eficacia de controles de integridad
	No Repudio	- Cantidad de transacciones con registro de auditoría - Uso de firmas digitales
	Responsabilidad	- Capacidad para rastrear acciones de usuarios - Registro detallado de actividades
	Autenticidad	- Número de accesos no autorizados prevenidos - Eficacia de los mecanismos de autenticación
Mantenibilidad	Modularidad	- Grado de acoplamiento entre módulos - Nivel de cohesión interna
	Reusabilidad	- Porcentaje de código reutilizable - Número de componentes reutilizados
	Analizabilidad	- Tiempo para diagnosticar fallos - Facilidad de comprensión del código (mediciones de complejidad)
	Modificabilidad	- Tiempo para implementar cambios - Número de líneas de código afectadas por cambio
	Probabilidad	- Cobertura de pruebas - Número de casos de prueba automatizados
Portabilidad	Adaptabilidad	- Esfuerzo necesario para adaptar el sistema a diferentes entornos - Número de plataformas soportadas
	Instalabilidad	- Tiempo de instalación - Número de incidencias durante la instalación
	Sustituibilidad	- Facilidad para reemplazar componentes - Compatibilidad con sistemas anteriores

Requerimientos Funcionales

Número	Requerimiento	Descripción	Prioridad
RF1	ESPECIFICACION DE DATOS PERSONALES DE LOS ALUMNOS, PROFESORES Y ADMINISTRADORES	Permite el ingreso de los datos personales de cada uno de los usuarios del sistema.	5
RF2	ESPECIFICAR EL DETALLE DEL CURSO	El sistema deberá permitir el ingreso de código, nombre, numero de créditos, sistema de evaluación del curso	5
RF3	VIZUALIZAR LA RELACION DE ALUMNOS POR CURSO	Este caso de uso se permite al profesor visualizar el detalle de los alumnos que están llevando un determinado curso	3
RF4	GENERACION DE INFORMES Y REPORTES	Con los informes se podrá obtener resultados detallados sobre las notas del curso, notas por evaluación, además del promedio final, grado académico, clasificarlos por alumno, área, fecha, etc. Estos podrán ser impresos.	4

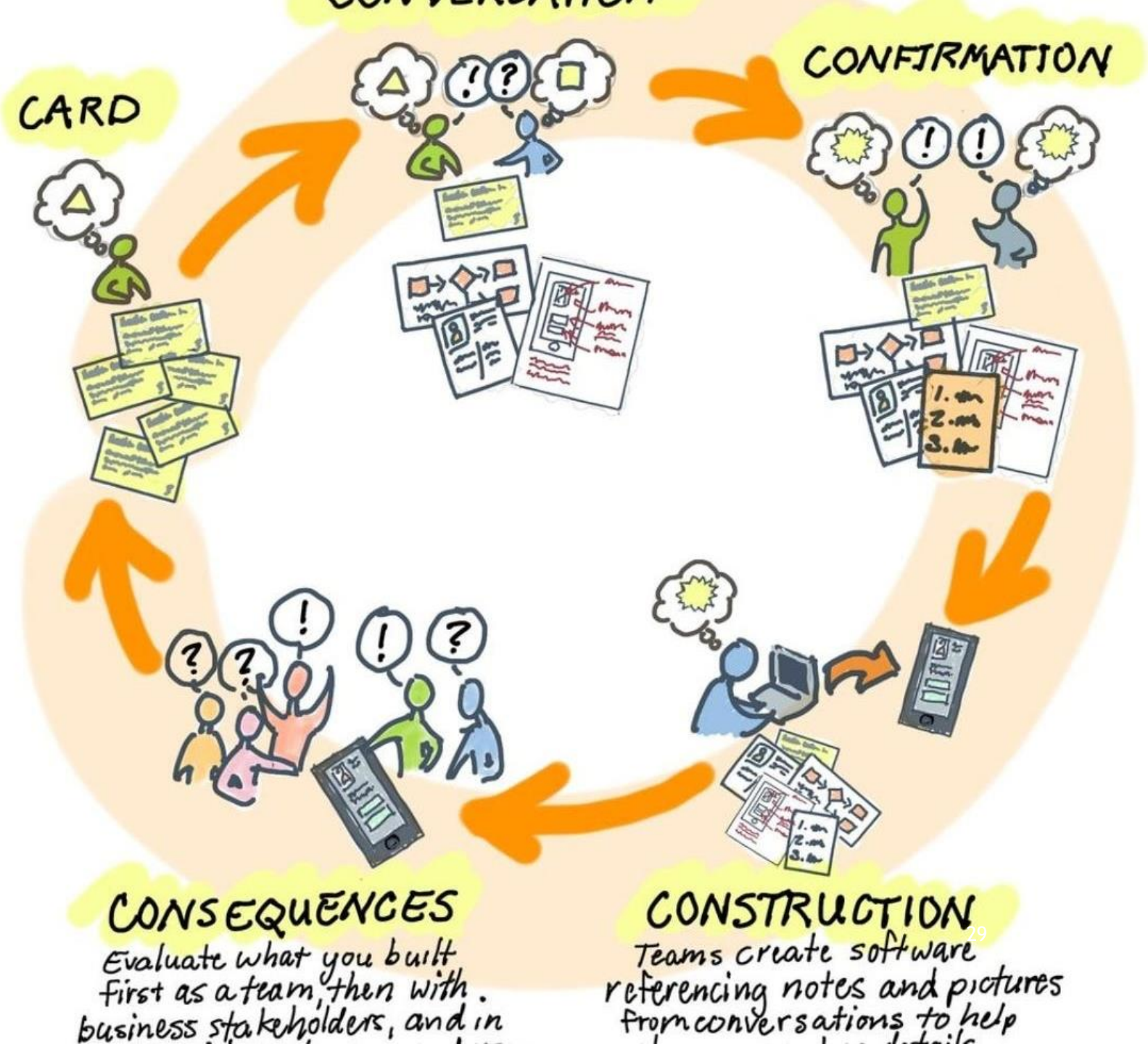
Historias de Usuario

28

Profesor: Matías Rojas M.

CCC

- Card
  - Cómo, Quiero, De manera que pueda
- Conversation
  - Aclarar dudas
  - Reglas de negocio
  - Requerimientos no funcionales
  - Otros detalles
- Confirmation
  - Criterios de Aceptación



Historias de Usuario

ID Historia de usuario	Como <tipo de usuario>	Quiero <desarrollar alguna tarea>	De manera de que pueda <lograr algún objetivo>
1	Gerente	Ver el reporte de estado de cada miembro del equipo	Asegurar que el proyecto avance según lo presupuestado
2	Analista de facturación	Ser recordado de próximas fechas límites	Completar tareas a tiempo.
3	Director	Supervisar el trabajo de la empresa	Ser consciente de lo que ocurre

HU - Criterios de Aceptación

- Definir los criterios de aceptación es crucial para garantizar que las historias de usuario sean completadas de manera satisfactoria y cumplan con las expectativas del cliente.
  - Dado que <Estado previo o contexto>
  - Cuando <Comportamiento o acción>
  - Entonces <Cambio a raíz de un comportamiento>

Historia de usuario 2: Seguimiento Detallado de Entrega de Pizza

- **Resumen**:

  Yo como un usuario enfocado en la eficiencia,

  Quiero recibir notificaciones detalladas sobre el progreso de la entrega de mi pizza,

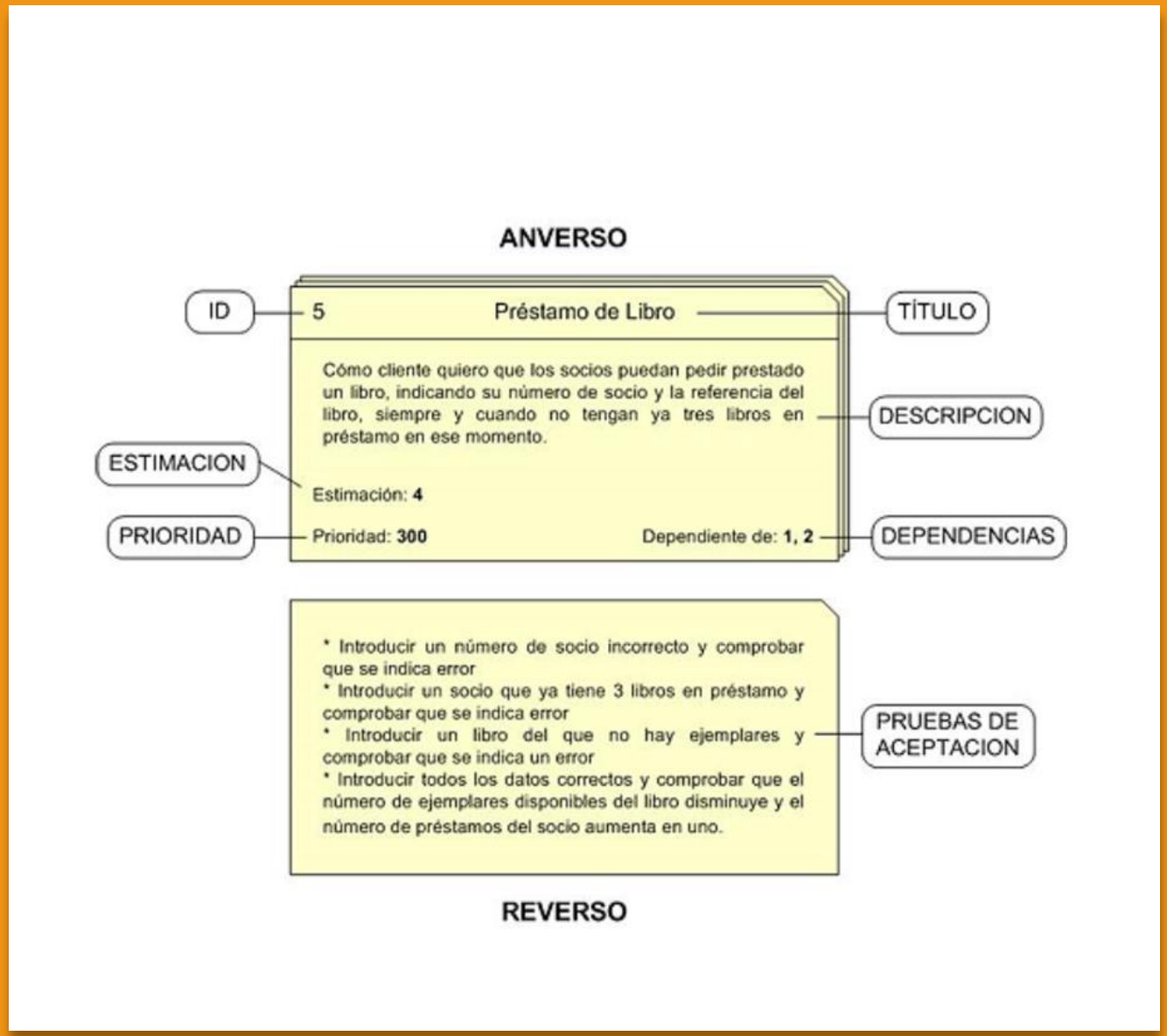
  A modo de estar informado y valorar la experiencia.

Criterios de aceptación:

- **Escenario**: Registro de Pedido y Seguimiento de Entrega
  - **Given**: He realizado un pedido de pizza
  - **and Given**: Tengo la aplicación abierta
  - **When**: Mi pedido es registrado
  - **Then**: Debo recibir notificaciones sobre el progreso de mi pedido, incluyendo la preparación, la cocción y el tiempo estimado de entrega.
- **Escenario**: Visualización de Línea de Tiempo con Gráficos y Números
  - **Given**: Estoy siguiendo el progreso de mi pedido
  - **When**: Visualizo la información del seguimiento
  - **Then**: La aplicación debería mostrar una línea de tiempo detallada con gráficos y números que representen el progreso de mi pedido y la estimación exacta de entrega

Ejemplo

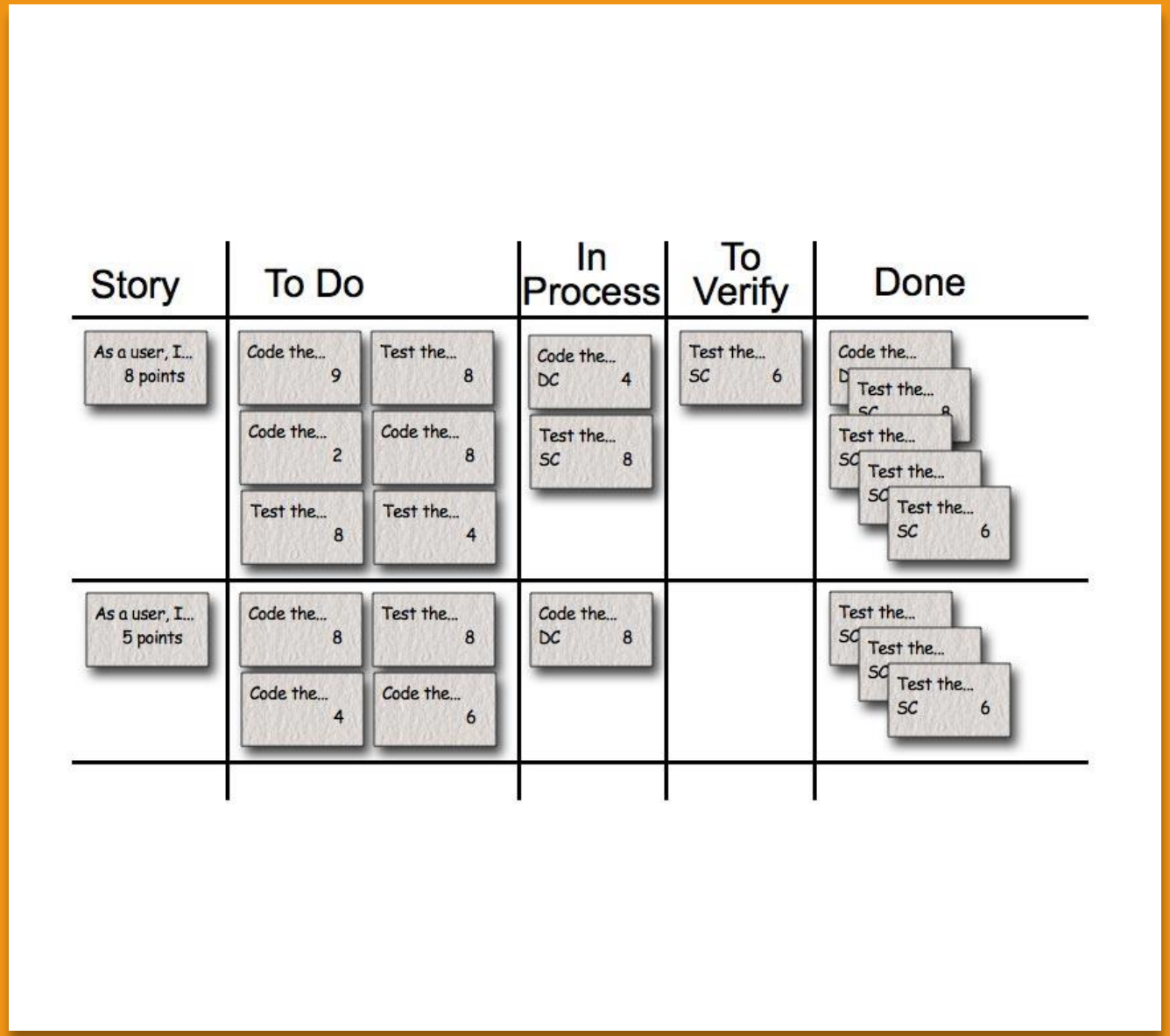
Profesor: Matías Rojas



INVEST

- Independiente
  - Al menos en un sprint
- Negociable
  - No detallar todo, en la conversación puede refinarse
- Valiosa
  - Que no dependa de otra HU para generar valor
- Estimable
  - Qué el equipo pueda decidir la dificultad de una HU en comparación a otras
- Pequeña
  - Debe caber en un sprint y evitar que tome todo el sprint desarrollarla
- Testeable
  - Que pueda decidirse cuando una HU está lista y que pueda verificarse

Tabla de Tareas



Actividad 1

Un centro deportivo universitario desea desarrollar una aplicación que permita a los estudiantes, entrenadores y al personal de administración gestionar reservas de instalaciones deportivas (como canchas de fútbol, tenis y salones de yoga), además de organizar clases y eventos deportivos. El sistema debe permitir:

1. A los **estudiantes**: reservar instalaciones, ver horarios de clases y confirmar su asistencia.
2. A los **entrenadores**: gestionar las clases, registrar la asistencia de estudiantes y definir las reglas de cada actividad.
3. Al **personal administrativo**: aprobar las solicitudes de reserva, actualizar horarios de las instalaciones y ver estadísticas de uso.

Modelamiento de Sistemas

37

Profesor: Matías Rojas M.

Modelamiento de Sistemas

La modelación de sistemas es el proceso de elaboración de modelos abstractos de un sistema

Cada modelo presenta una vista o perspectiva diferente de ese sistema.

Se basa en el Lenguaje Unificado de Modelado (*Unified Modelling Language - UML*).

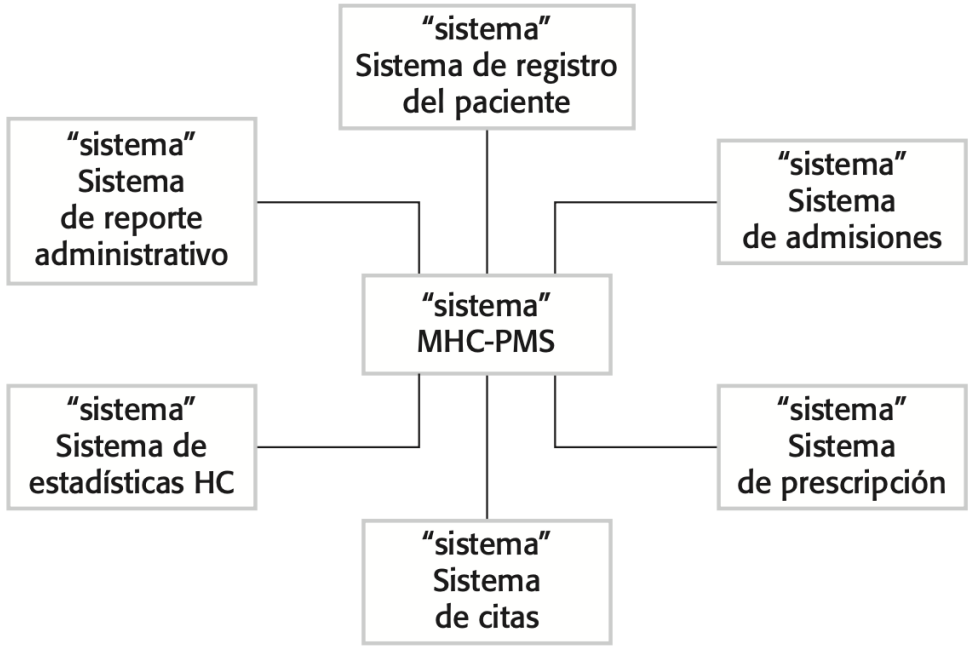
Sirven para detallar requerimientos, describir la implementación, y documentar la estructura del sistema y operación.

Se modela el sistema que se desarrollará y el que existe.

Modelos de Contexto

Modelos de Contexto

- Al empezar a especificar y detallar un sistema, se deben decidir los *límites*. Es decir, hasta dónde llega y dónde no.
- Este es un trabajo que se debe hacer con los stakeholders.
- En general, esto da pie a un modelo simple de arquitectura. Este es un Diagrama de Contexto:



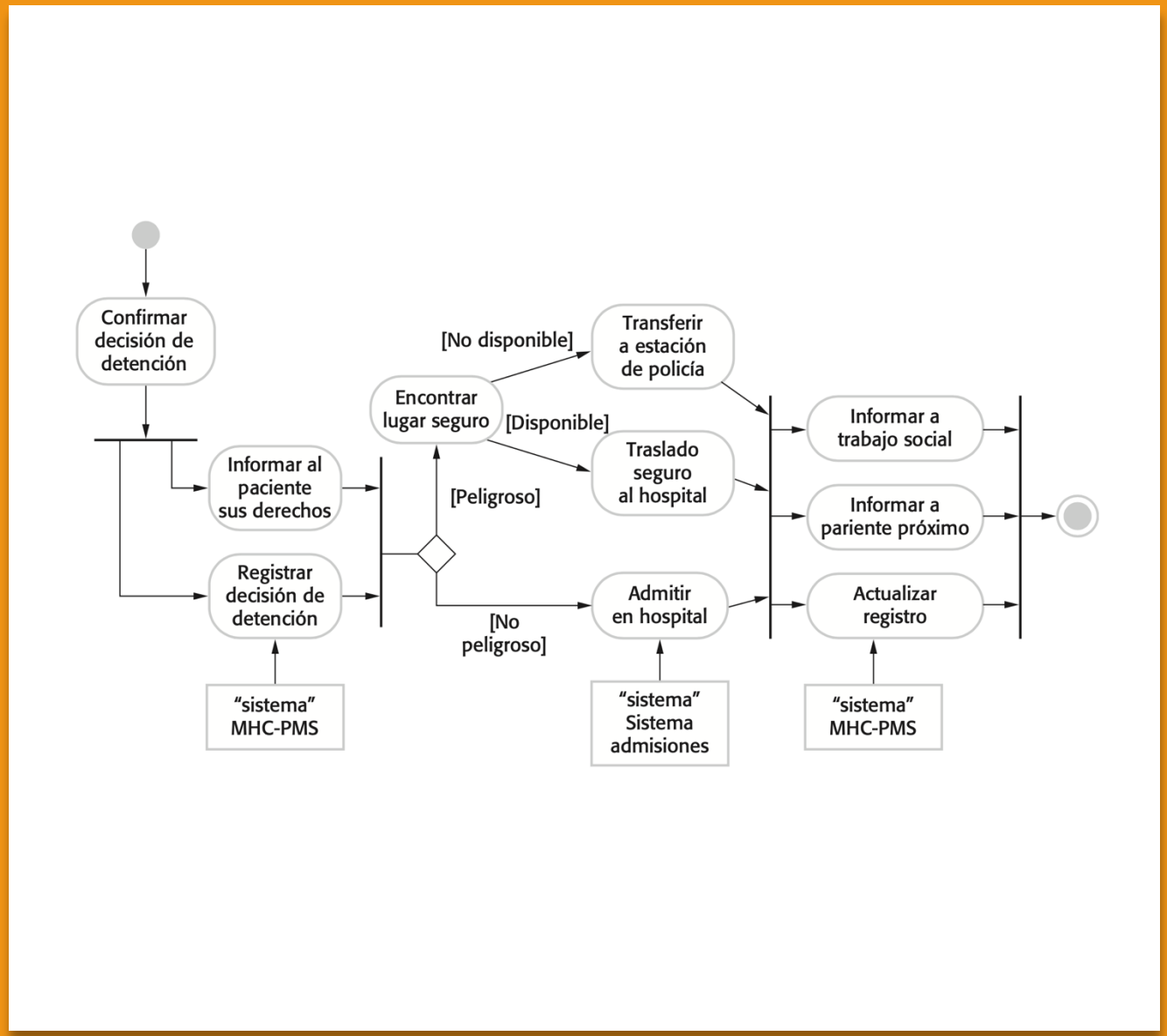
Modelos de Actividades

41

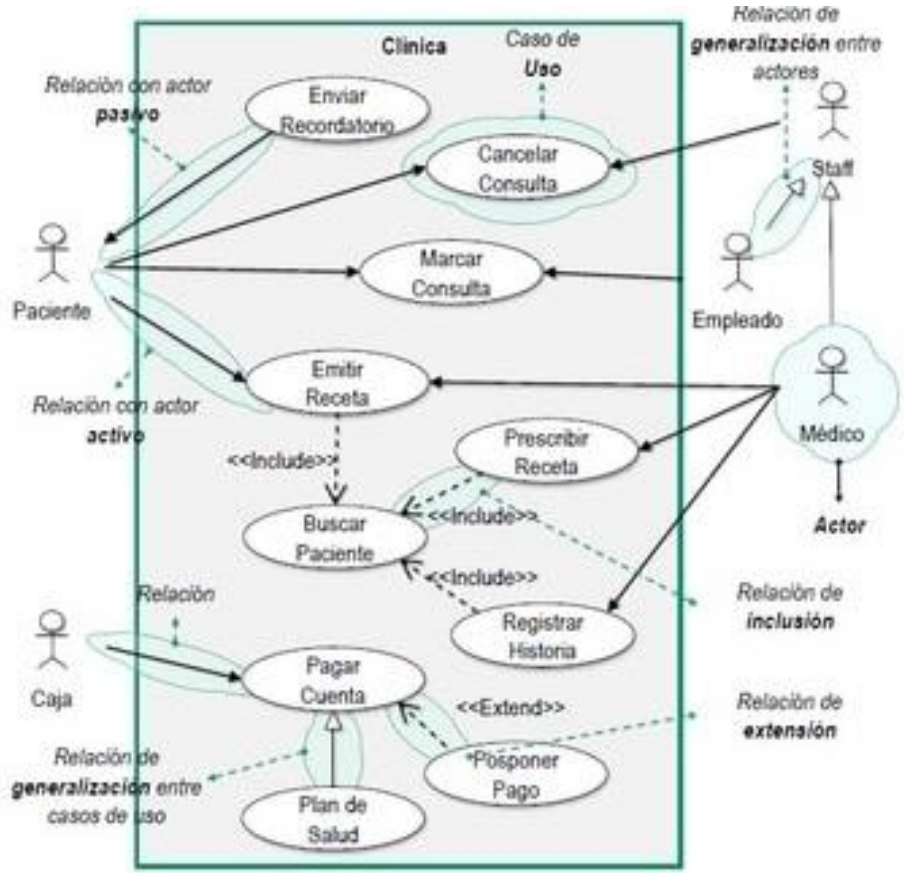
Profesor: Matías Rojas M.

Diagrama de Actividades

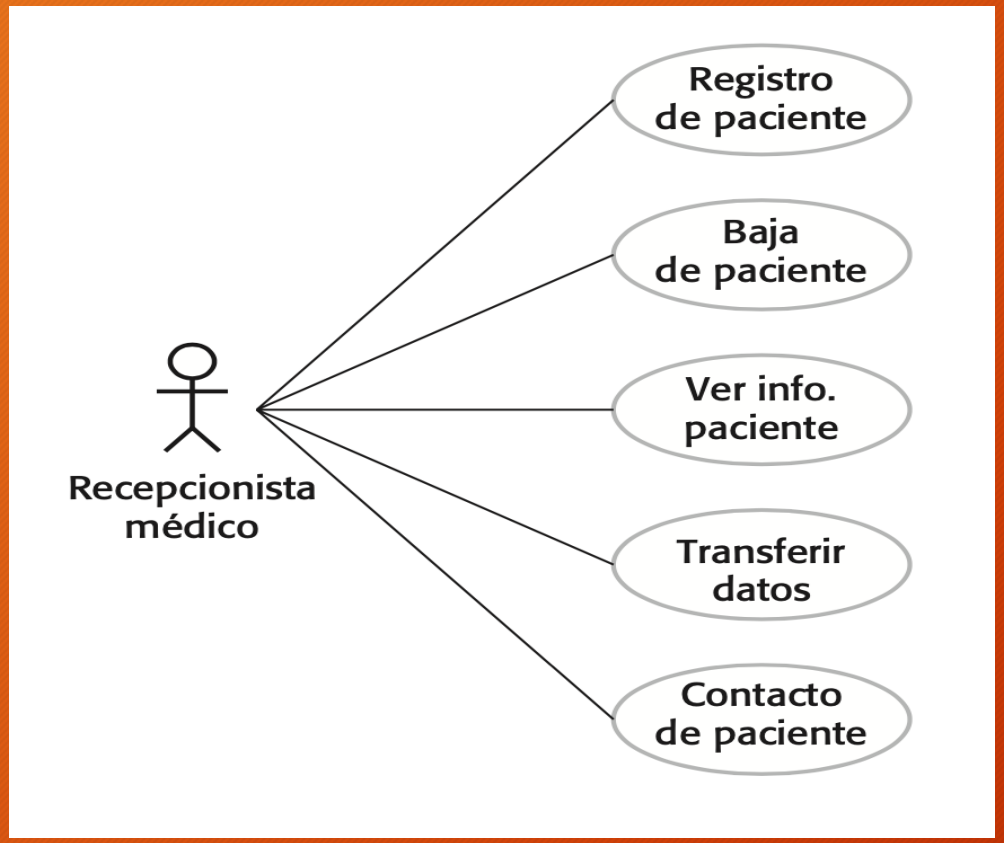
- **Ejemplo:** Un diagrama de una internación involuntaria usando *Mentcare*:

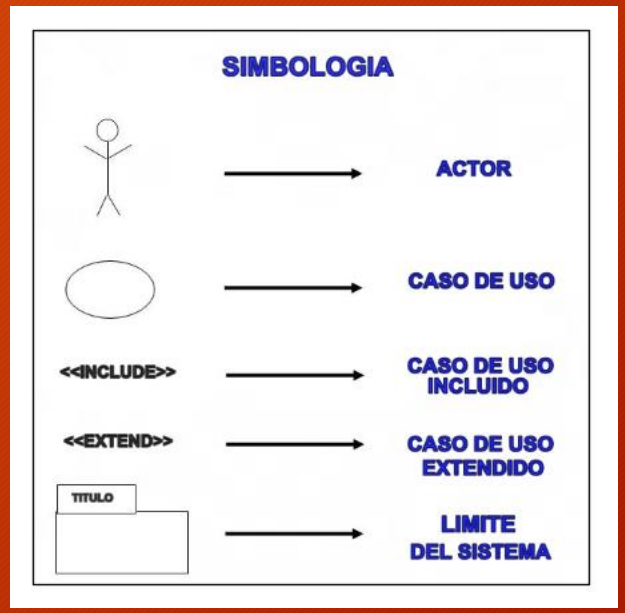


Ejemplo Diagrama Caso de Uso



Casos de Uso


Registro de paciente
Baja de paciente
Ver info. paciente
Transferir datos
Contacto de paciente
Recepcionista médico


SIMBOLOGIA
ACTOR
CASO DE USO
<<INCLUDE>>
CASO DE USO INCLUIDO
<<EXTEND>>
CASO DE USO EXTENDIDO
TITULO
LIMITE DEL SISTEMA

Actividad 2

Un centro deportivo universitario desea desarrollar una aplicación que permita a los estudiantes, entrenadores y al personal de administración gestionar reservas de instalaciones deportivas (como canchas de fútbol, tenis y salones de yoga), además de organizar clases y eventos deportivos. El sistema debe permitir:

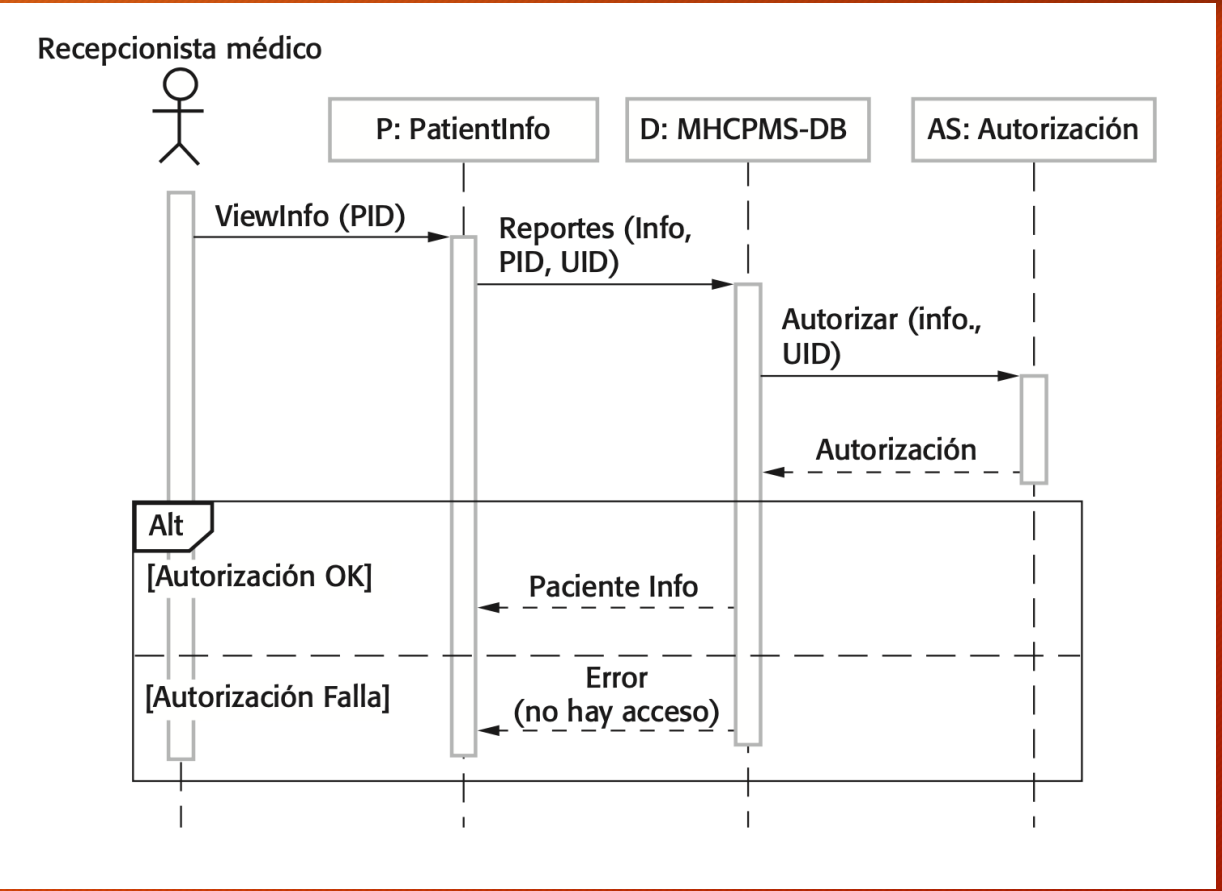
1. A los **estudiantes**: reservar instalaciones, ver horarios de clases y confirmar su asistencia.
2. A los **entrenadores**: gestionar las clases, registrar la asistencia de estudiantes y definir las reglas de cada actividad.
3. Al **personal administrativo**: aprobar las solicitudes de reserva, actualizar horarios de las instalaciones y ver estadísticas de uso.

Modele el diagrama de casos de uso

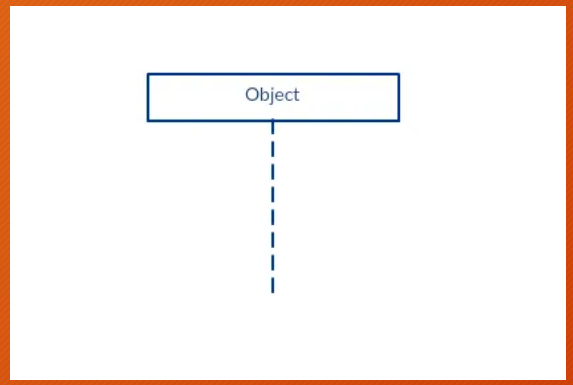
Diagramas de Secuencia

- Los diagramas de secuencia sirven para modelas interacciones entre actores y objetos en un sistema y las interacciones entre objetos.
- Muestran la secuencia de interacciones que ocurren en un caso de uso particular.
- Veamos un ejemplo:

Diagramas de Secuencia



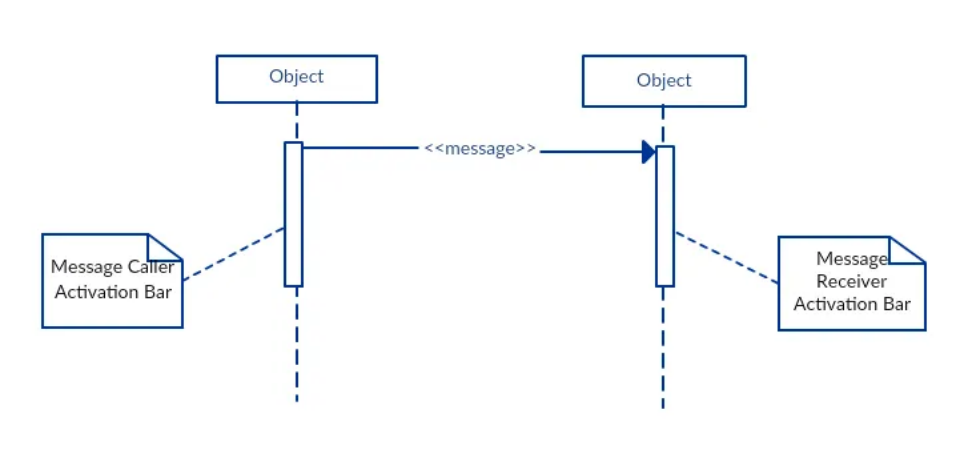
Líneas de vida



- Un diagrama de secuencia se compone de varias de estas notaciones de línea de vida que deben disponerse horizontalmente a lo largo de la parte superior del diagrama. No debe haber dos líneas de vida superpuestas. Representan los diferentes objetos o partes que interactúan entre sí en el sistema durante la secuencia.

- Una notación de línea de vida con un símbolo de elemento actor se utiliza cuando el diagrama de secuencia particular pertenece a un caso de uso.

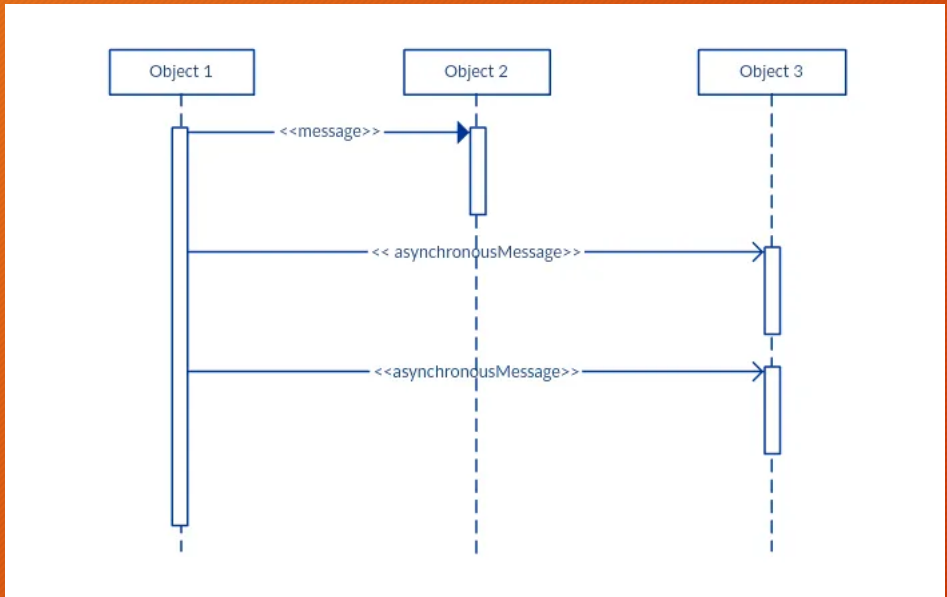
Barra de Activación



- La barra de activación es la casilla situada en la línea de vida. Se utiliza para indicar que un objeto está activo (o instanciado) durante una interacción entre dos objetos. La longitud del rectángulo indica la duración de la activación de los objetos.

- En un diagrama de secuencia, una interacción entre dos objetos se produce cuando un objeto envía un mensaje a otro. El uso de la barra de activación en las líneas de vida del Llamador del mensaje (el objeto que envía el mensaje) y del Receptor del mensaje (el objeto que recibe el mensaje) indica que ambos están activos/ se instancian durante el intercambio del mensaje.

Mensajes



Mensaje síncrono

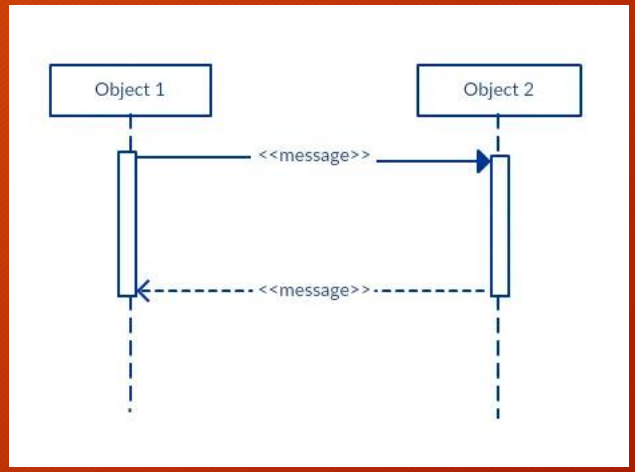
- Como se muestra en el ejemplo, un mensaje síncrono se utiliza cuando el emisor espera a que el receptor procese el mensaje y lo devuelva antes de continuar con otro mensaje. La punta de flecha utilizada para indicar este tipo de mensaje es sólida.

Mensaje asíncrono

- Un mensaje asíncrono se utiliza cuando el emisor del mensaje no espera a que el receptor procese el mensaje y vuelva antes de enviar otros mensajes a otros objetos del sistema. La punta de flecha utilizada para mostrar este tipo de mensaje es una flecha de línea.

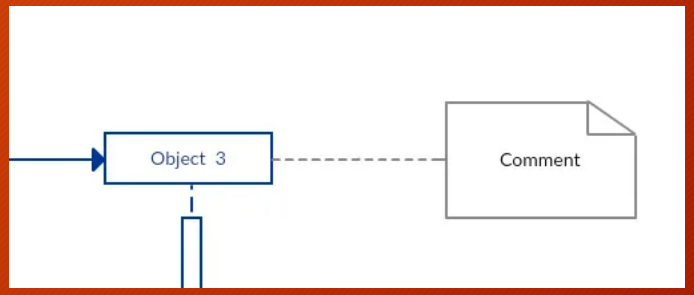
Retorno

- Un mensaje de retorno se utiliza para indicar que el receptor del mensaje ha terminado de procesarlo y devuelve el control al emisor del mensaje.

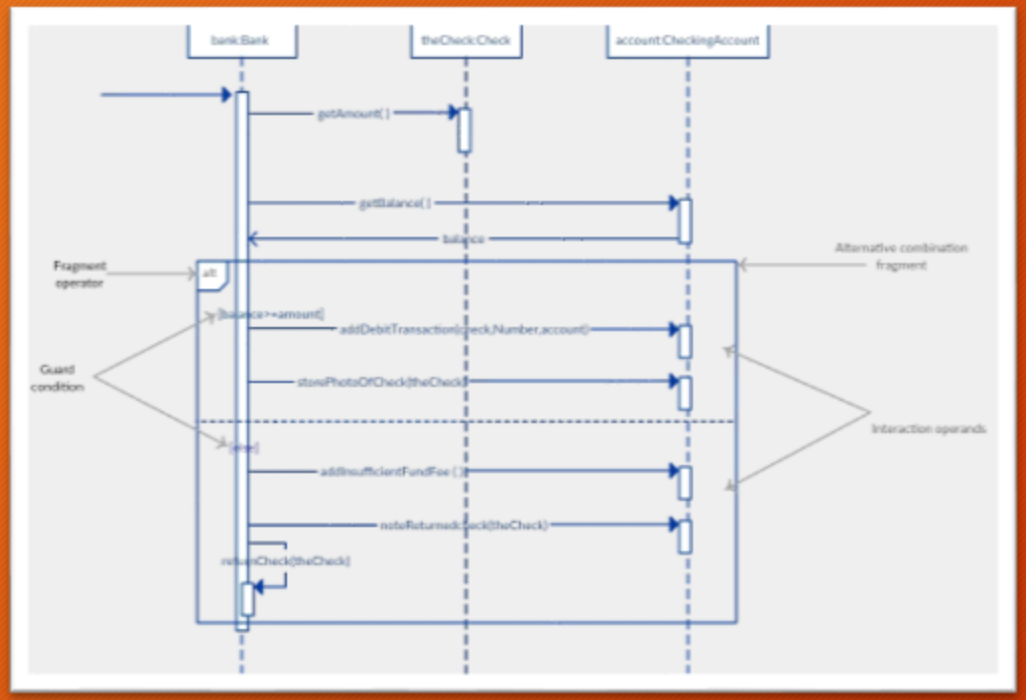


Comentarios

- En general, los diagramas UML permiten la anotación de comentarios en todos los tipos de diagramas UML. El objeto comentario es un rectángulo con una esquina doblada, como se muestra a continuación. El comentario puede vincularse al objeto relacionado con una línea discontinua.

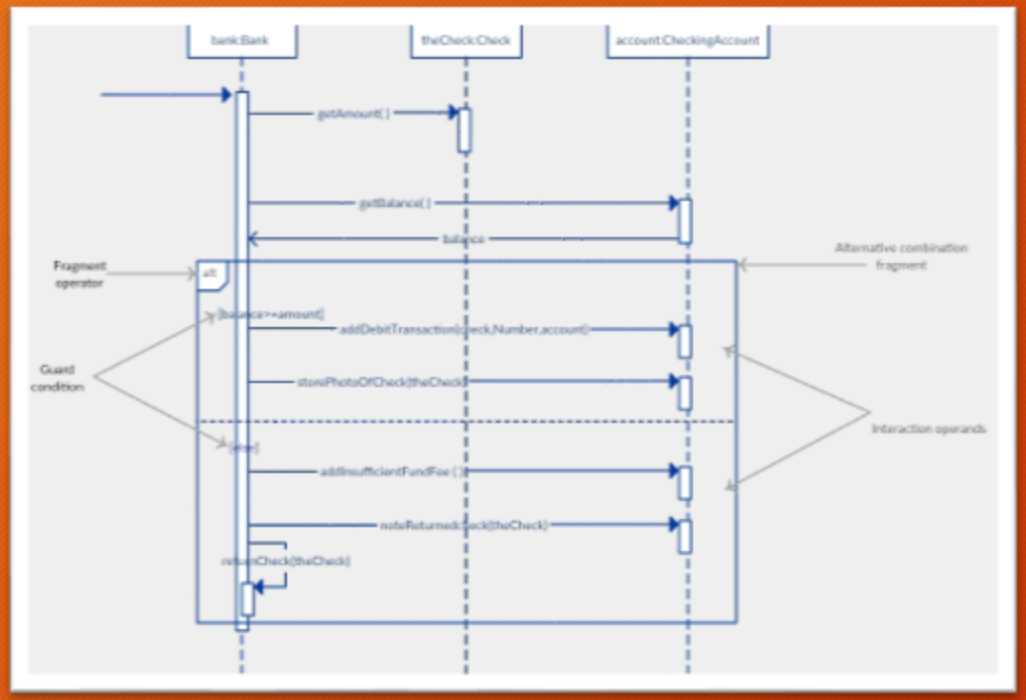


Fragmentos



- Un fragmento de secuencia se representa como un recuadro que enmarca una sección de interacciones entre objetos (como se muestra en el ejemplo) en un diagrama de secuencia.
- Se utiliza para mostrar interacciones complejas, como flujos alternativos y bucles, de una forma más estructurada. En la esquina superior izquierda del fragmento se sitúa un operador. Este, el operador de fragmento, especifica de qué tipo de fragmento se trata.

Fragmentos



El fragmento alternativa

- se utiliza cuando hay que elegir entre dos o más secuencias de mensajes. Modela la lógica "if then else".
- El fragmento alternativo se representa mediante un rectángulo grande o un marco; se especifica mencionando "alt" dentro del cuadro de nombre del marco (también conocido como operador de fragmento).

El fragmento de opciones

- se utiliza para indicar una secuencia que sólo se producirá bajo una determinada condición; en caso contrario, la secuencia no se producirá. Es un modelo de la sentencia "si entonces".
- Al igual que el fragmento alternativo, el fragmento de opción también se representa con un marco rectangular en el que "opt" se coloca dentro del cuadro de nombre.

El fragmento bucle

- se utiliza para representar una secuencia repetitiva. Coloque las palabras 'loop' en el cuadro de nombre y la condición cerca de la esquina superior izquierda del cuadro.
- Además de las pruebas booleanas, los loops puedne tener otras dos condiciones especiales. Estas son iteraciones mínimas (escritas como minint = [el número] e iteraciones máximas (escritas como maxint = [el número]).

Diagramas de Estado

Para este modelamiento se pueden usar Diagramas de Estado.

Estos muestran los estados del sistema y eventos que causaron transiciones de uno a otro estado.

No muestran flujos de datos dentro del sistema pero pueden incluir información adicional acerca del procesamiento de estos en cada estado.

Veamos un ejemplo:

Diagramas de Estado

- Este microondas sencillo tiene un interruptor para seleccionar potencia completa o media, un teclado numérico para ingresar el tiempo de cocción, un botón de iniciar/detener y una pantalla alfanumérica.
- Se supone que la secuencia de acciones al usar el horno de microondas es:
  - Seleccionar el nivel de potencia (ya sea media o completa)
  - Ingresar el tiempo de cocción con el teclado numerico.
  - Presionar “Iniciar”, y la comida se cocina durante el tiempo dado.

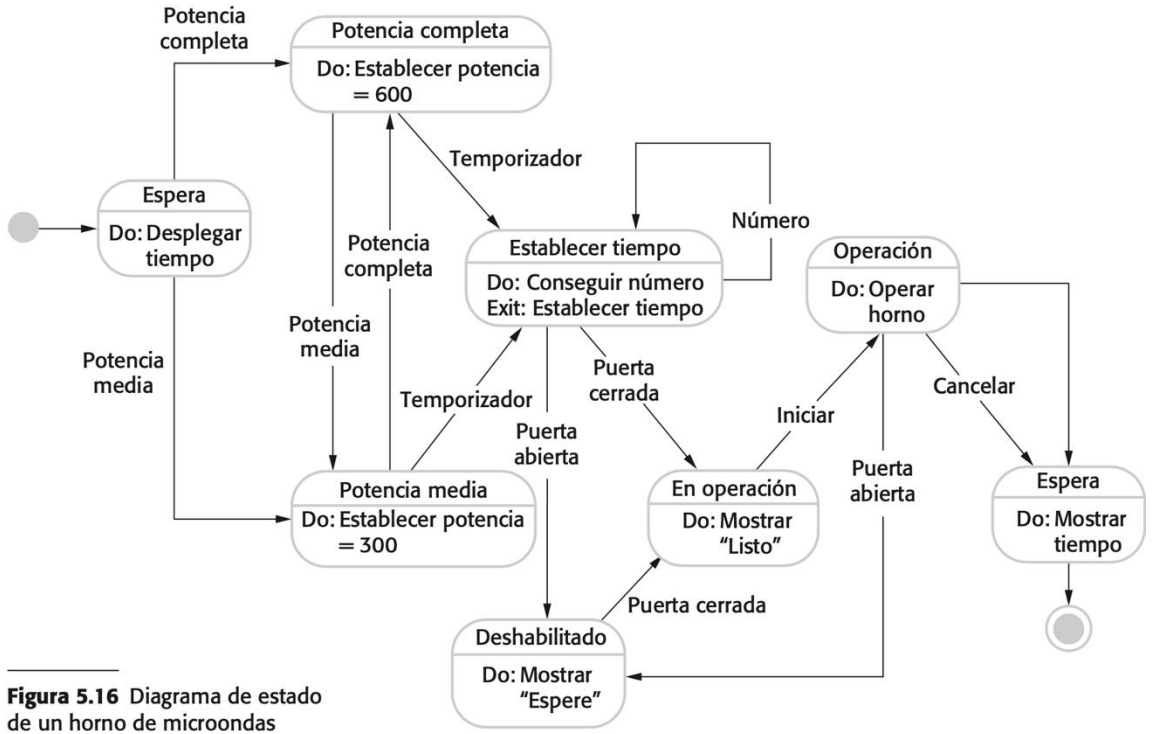


Figura 5.16 Diagrama de estado de un horno de microondas

Diagramas de Estado

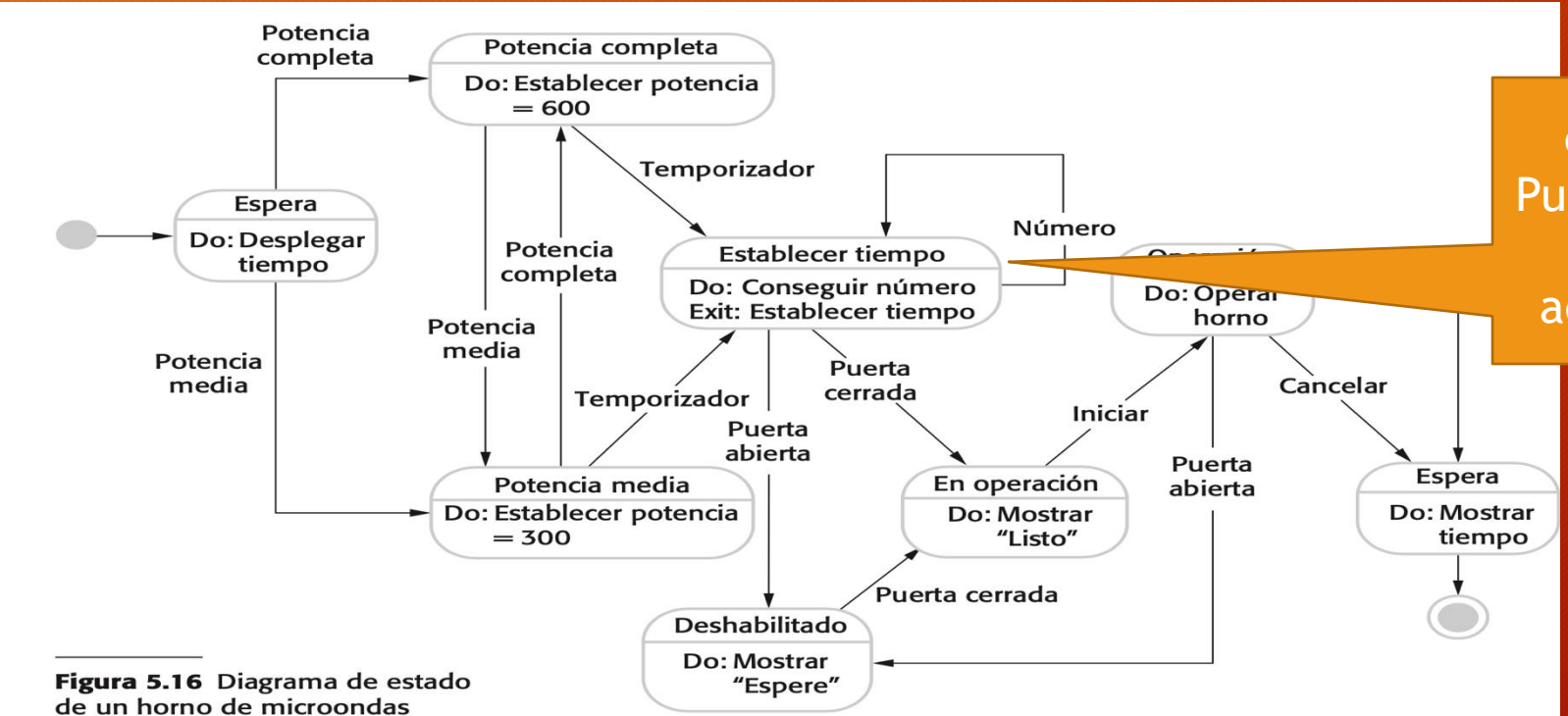


Figura 5.16 Diagrama de estado de un horno de microondas

Estos representan estados del sistema. Pueden incluir una breve descripción de las acciones que se toman en cada estado.

Diagramas de Estado

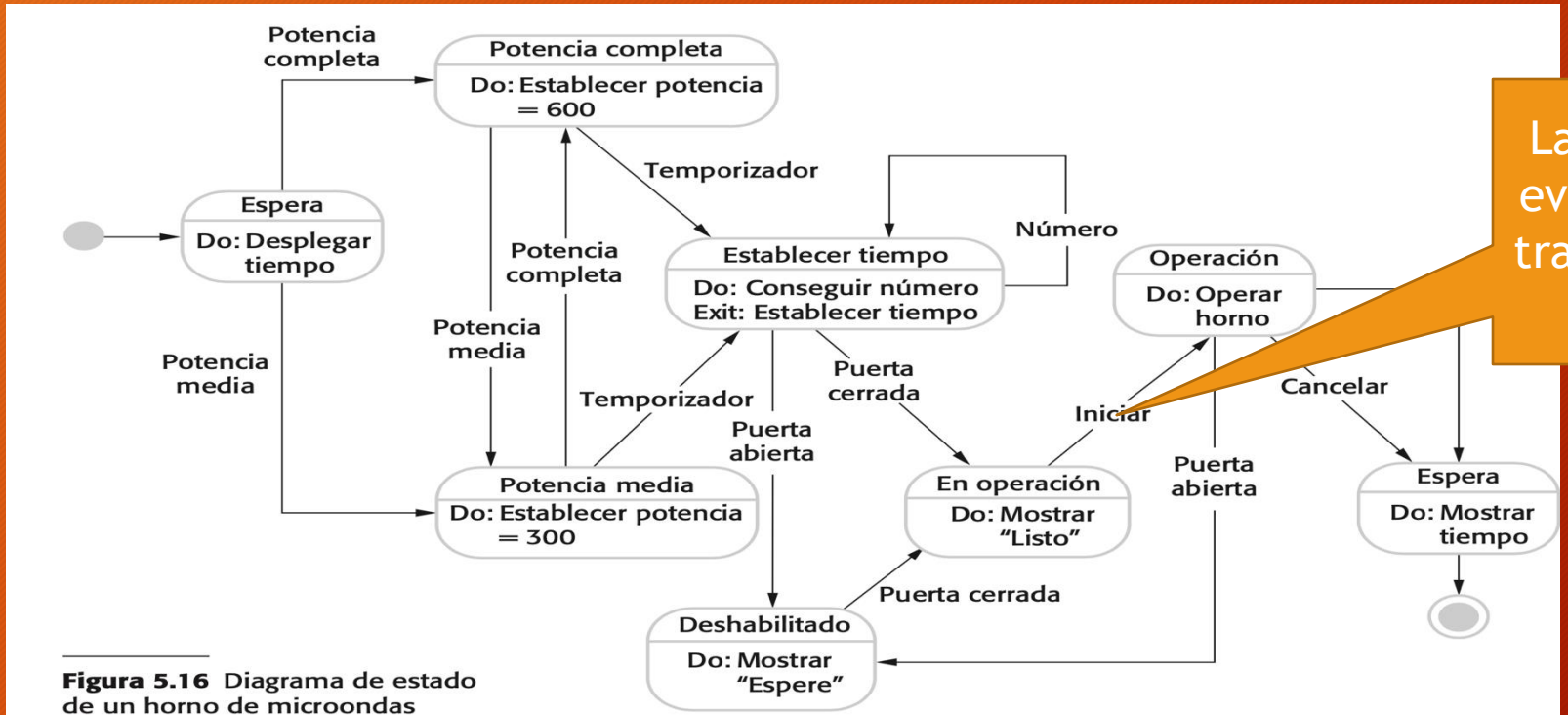


Figura 5.16 Diagrama de estado de un horno de microondas

Las flechas representan eventos que fuerzan una transición de un estado a otro.

Diagramas de Estado

Al igual que en el Diagrama de Actividades, se usan círculos para especificar el inicio y fin de un proceso

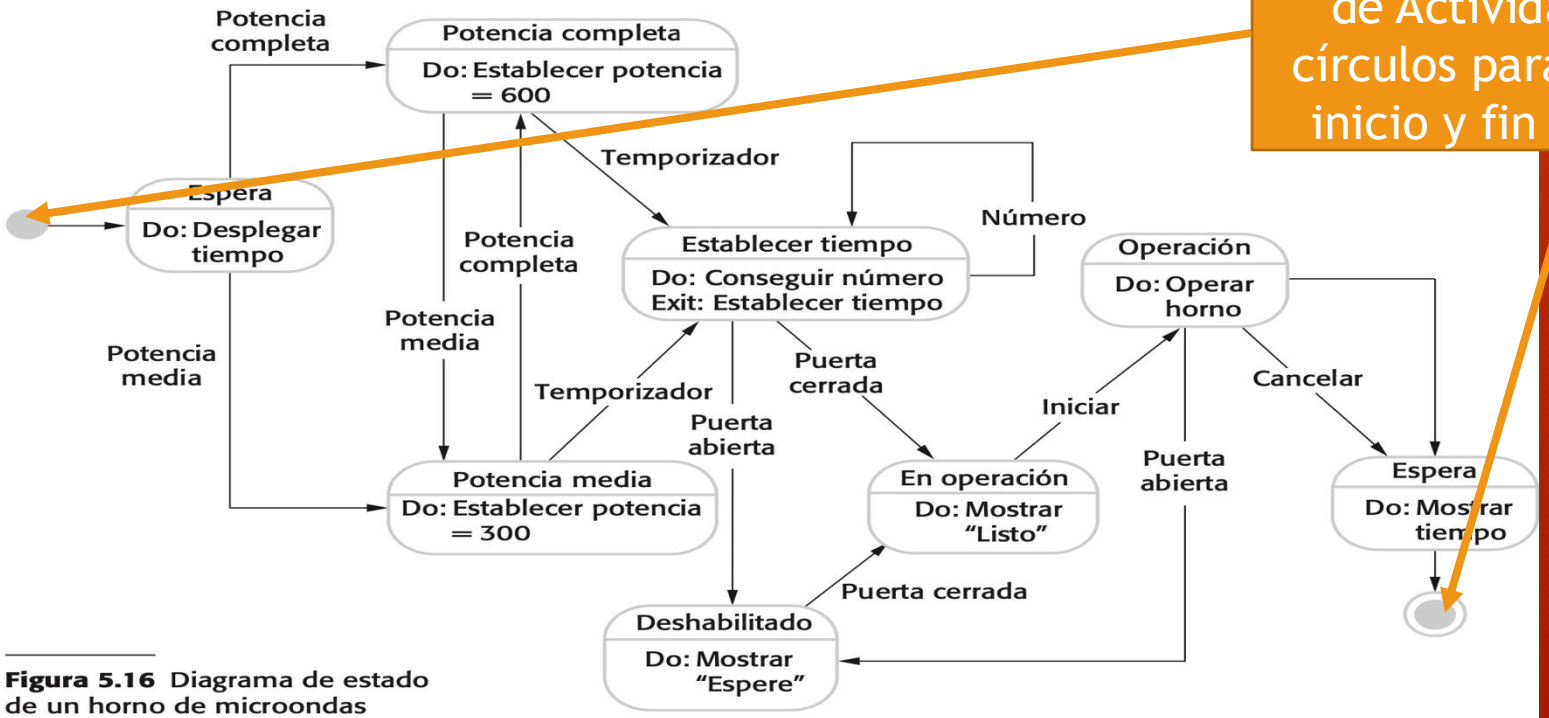


Figura 5.16 Diagrama de estado de un horno de microondas

Diagramas de Estado

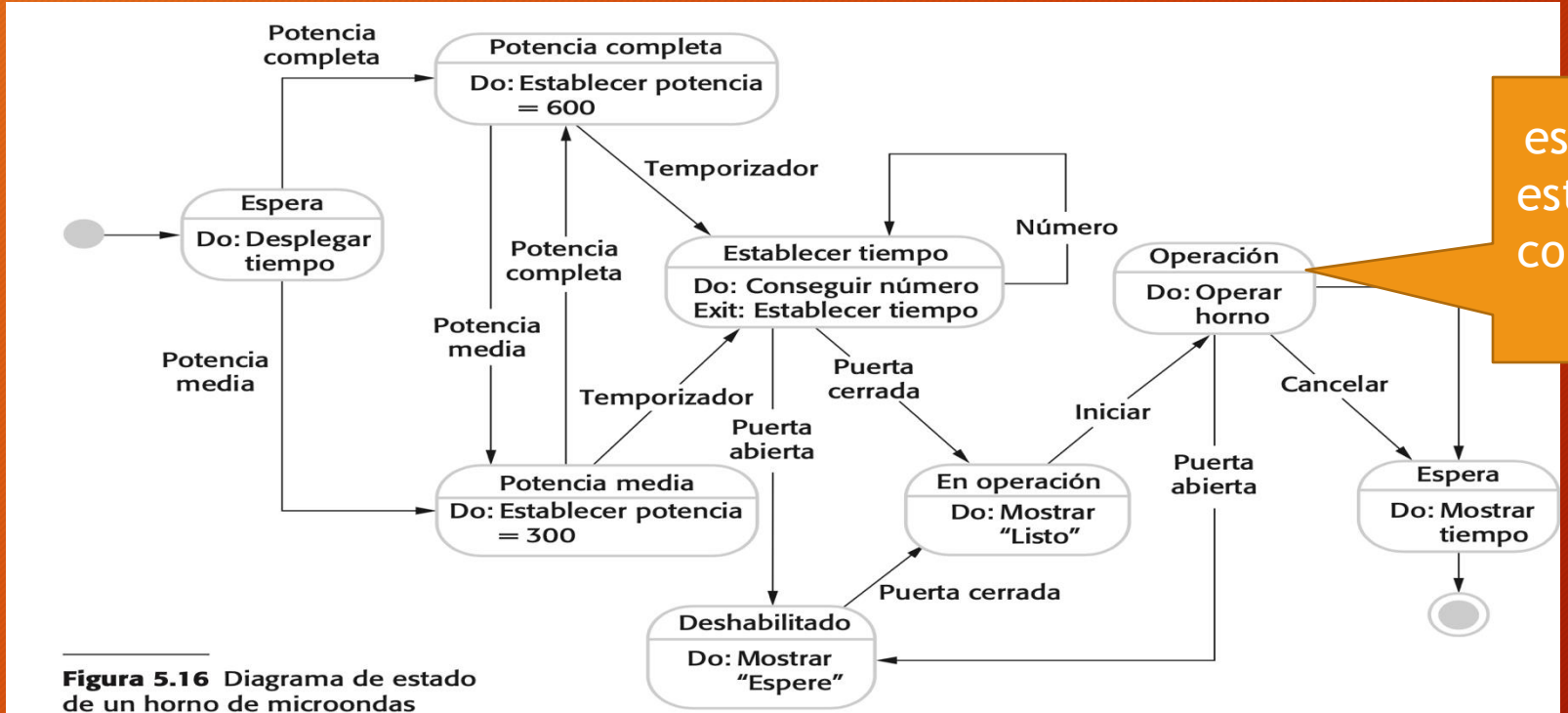


Figura 5.16 Diagrama de estado de un horno de microondas

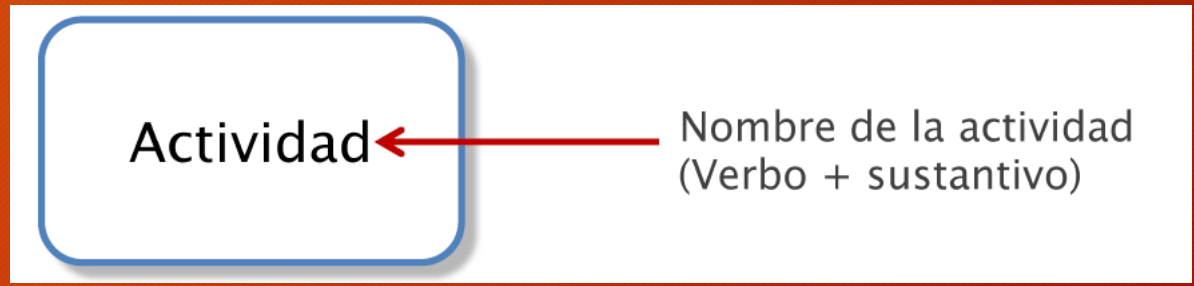
Muchas veces se esconden detalles en un estado. Estos se conocen como **superestados**, que encapsulan varios estados.

BPMN

Profesor: Matías Rojas

Actividad

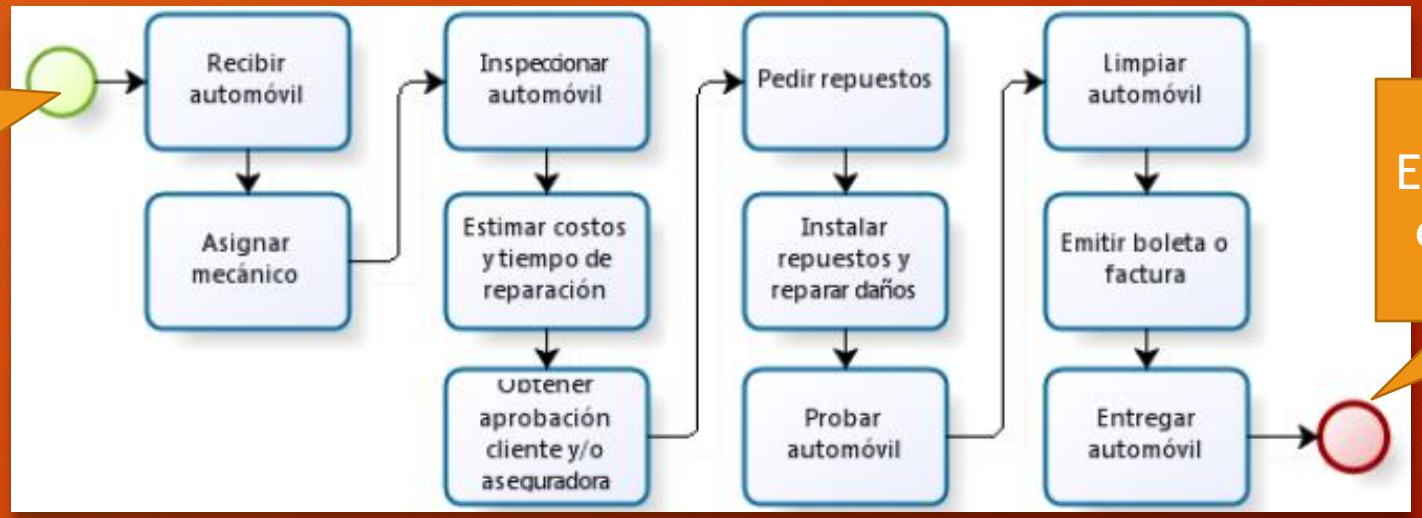
- BPMN especifica el orden y la responsabilidad de ejecución de las actividades del proceso.
- Se representan por el siguiente símbolo:



Flujo de Secuencia

- A través del **Flujo de Secuencia**, se especifica qué actividad se debe ejecutar antes que otra.
- Se representa a través de una flecha contínua.

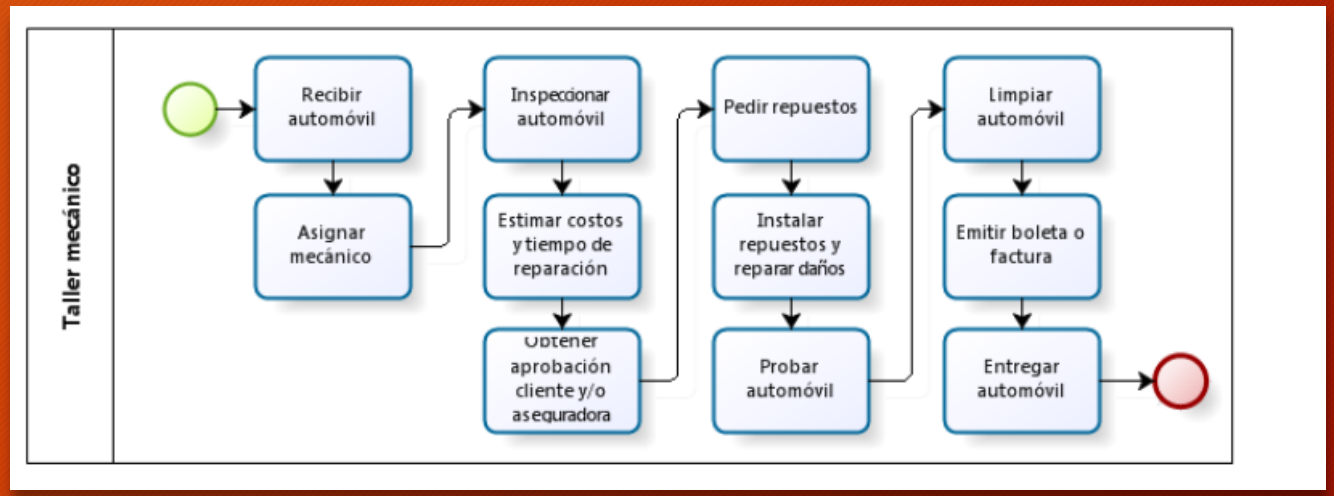
Este símbolo indica el inicio del proceso



Este símbolo indica el fin del proceso

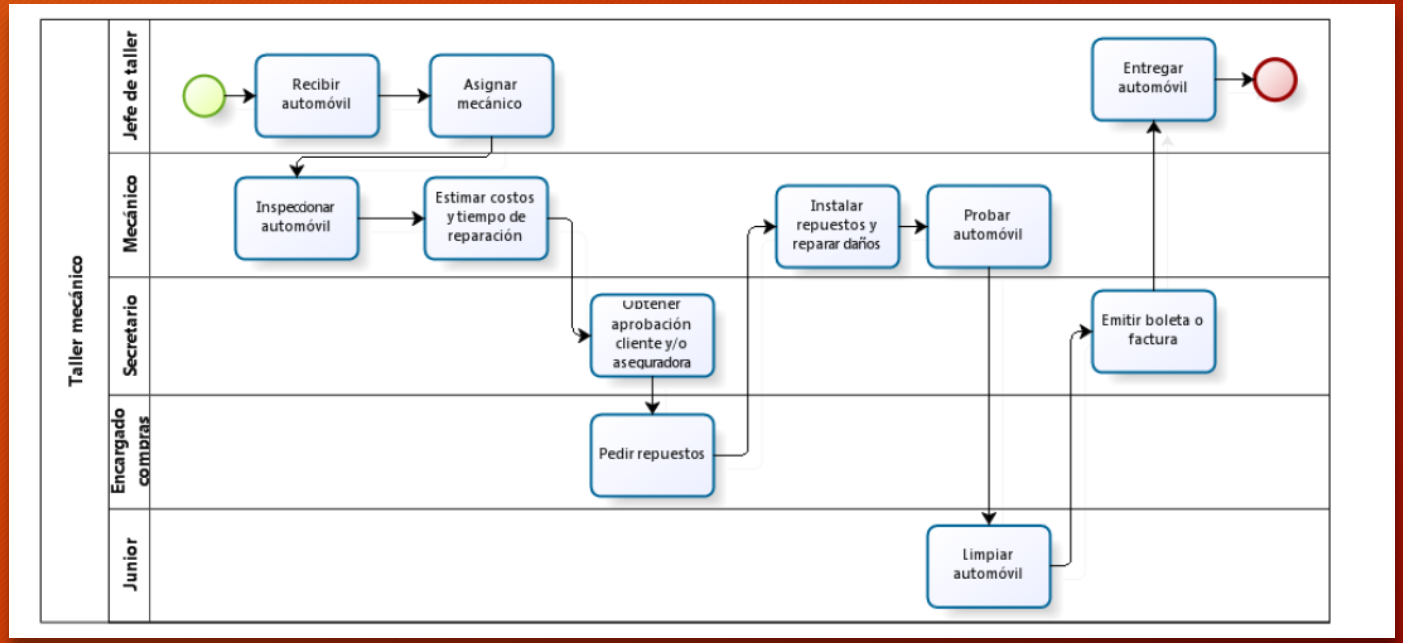
Pools y Lanes

- Los *pools* representan las distintas organizaciones involucradas en el proceso.



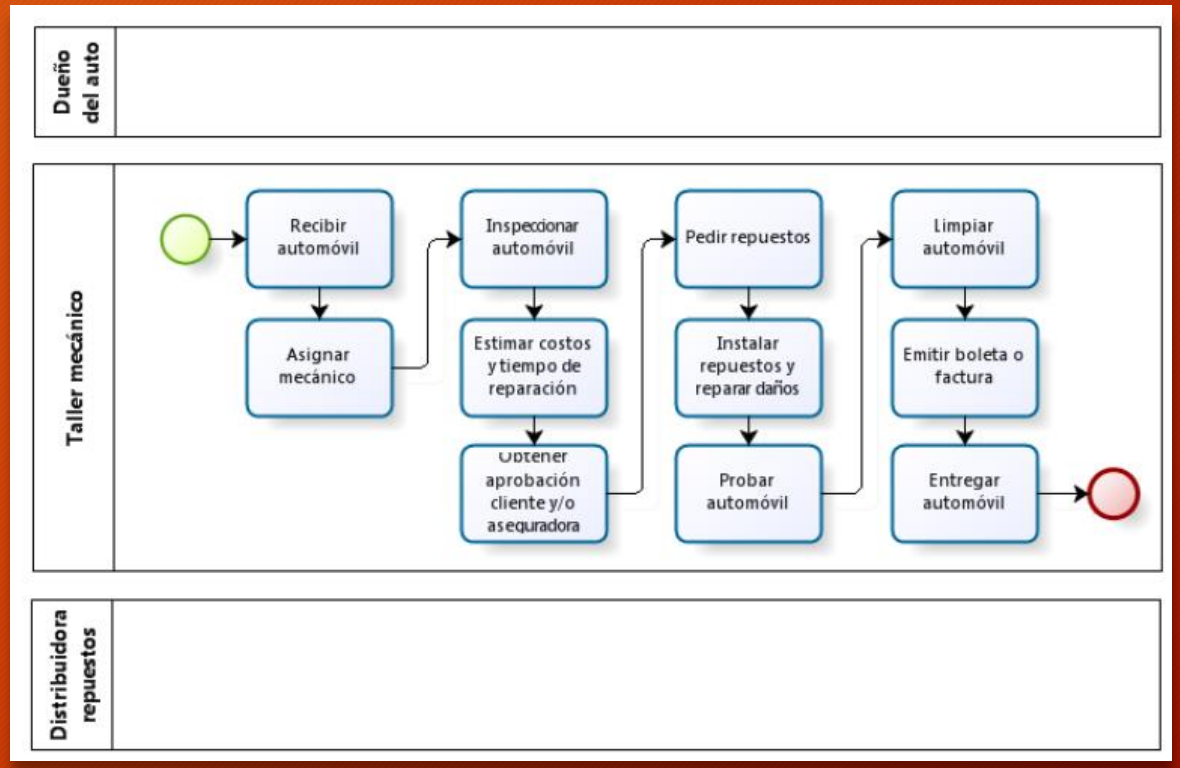
Pools y Lanes

- Para representar roles y asignar ejecución de tareas se ocupan *lanes* (carriles dentro de los pools).



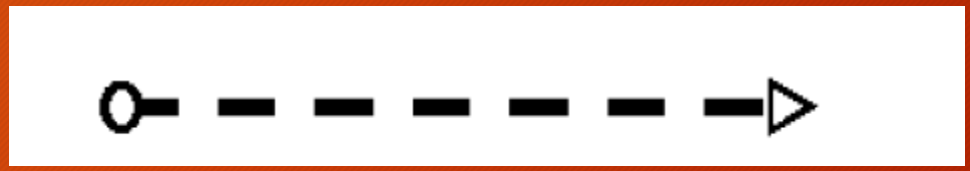
Pools y Lanes

- BPMN permite modelar el aspecto interno y externo de los procesos.

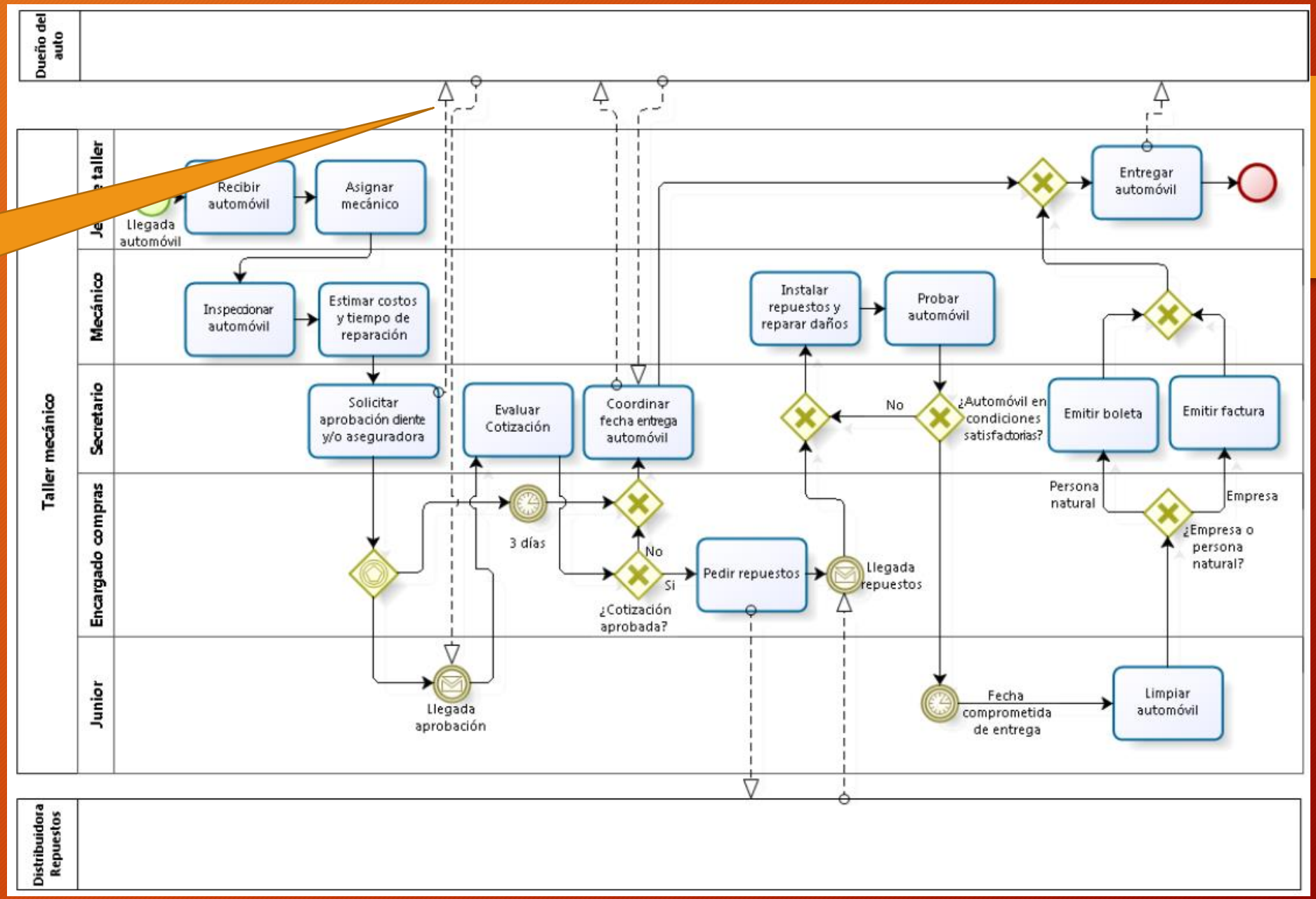


Flujo de Mensaje

- Permiten representar la comunicación entre los participantes de pools **independientes.**
- Se representa por una línea segmentada con el extremo sin relleno.
- Representa el envío de información desde un participante a otro, en una actividad o evento de un mensaje específico.



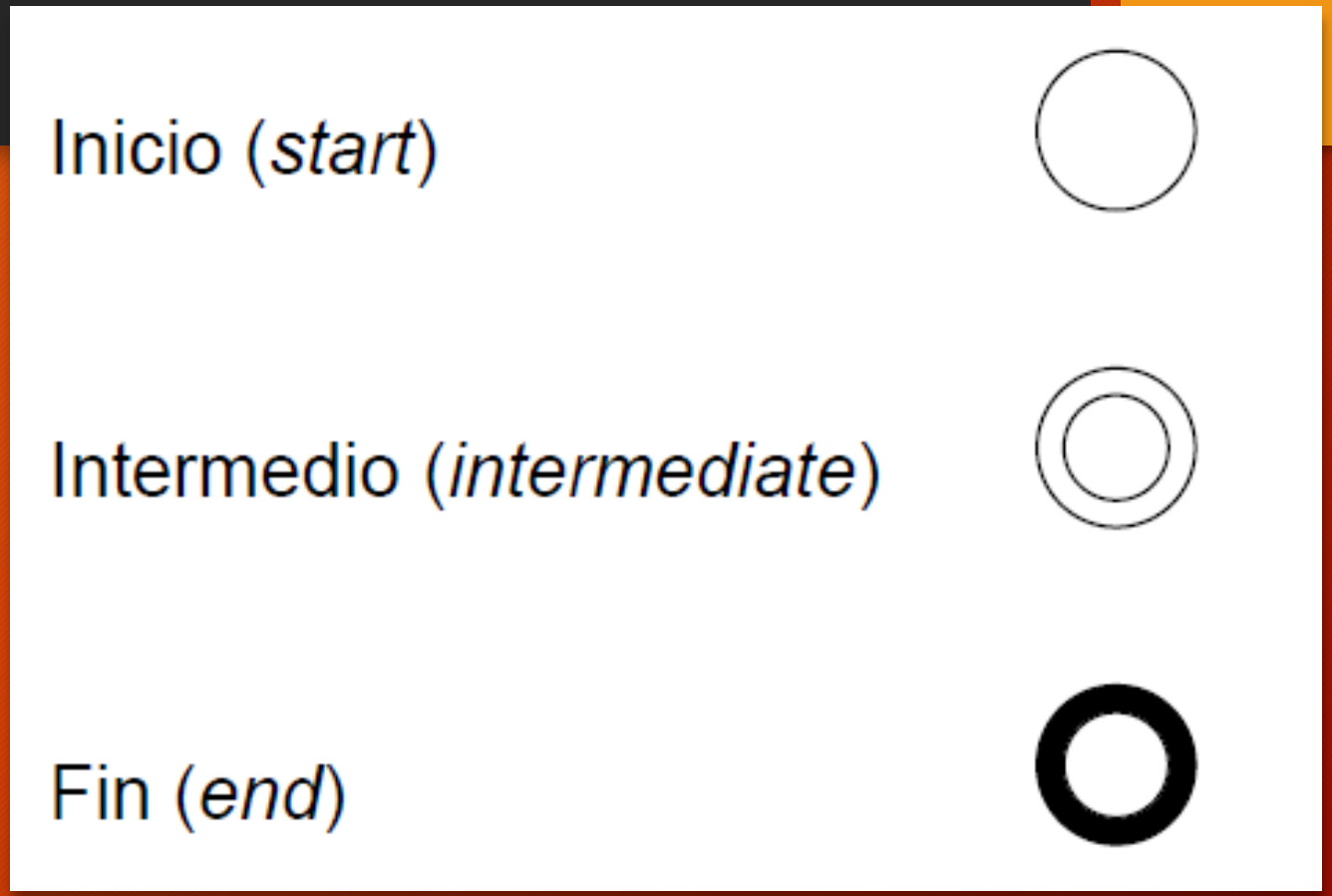
Interacción con otros pools



Profesor: Matías Rojas

Eventos: Cosas que “suceden”

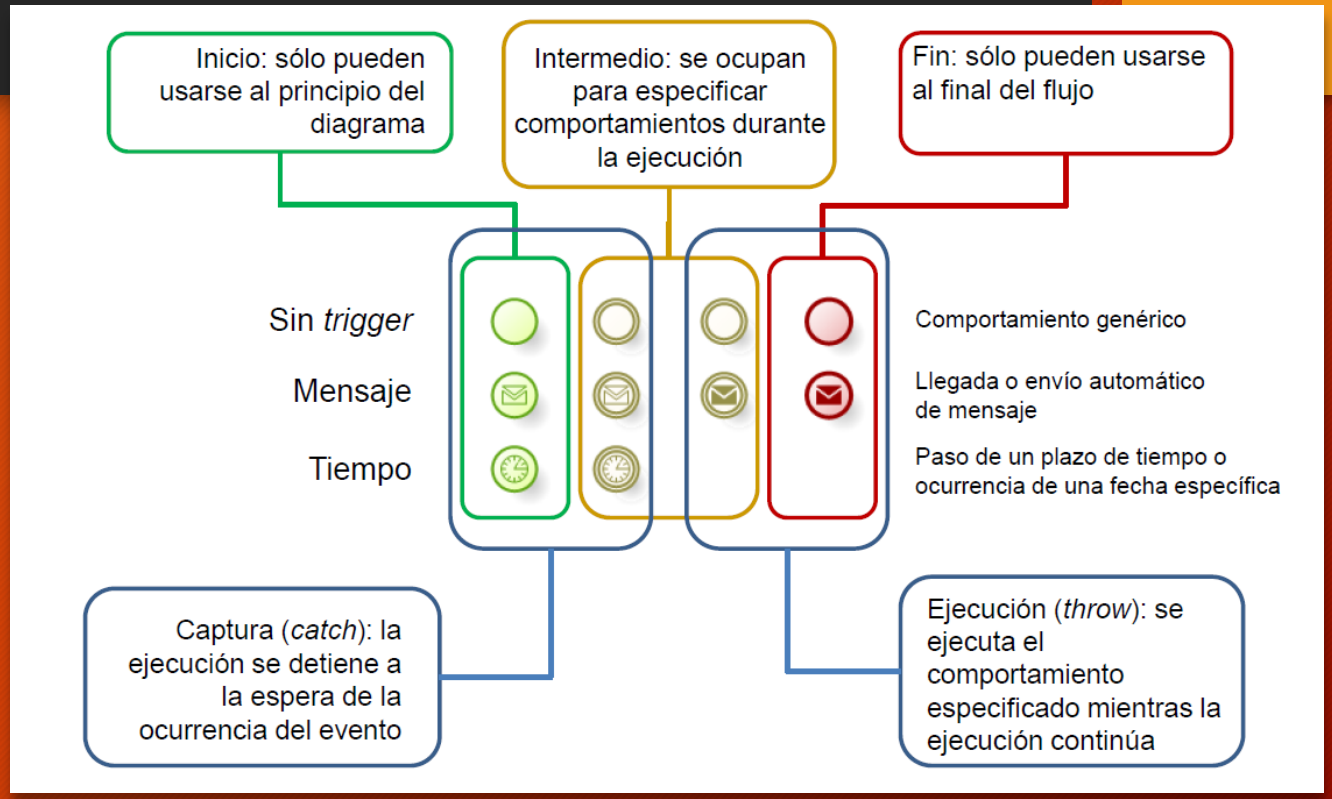
- Los procesos deben responder a cambios en el entorno: la llegada de un aviso, el paso del tiempo, etc. Estos se representan en BPMN con el concepto de evento.
- Estos afectan el flujo del proceso y tienen una causa o un efecto.
- Hay tres tipos:



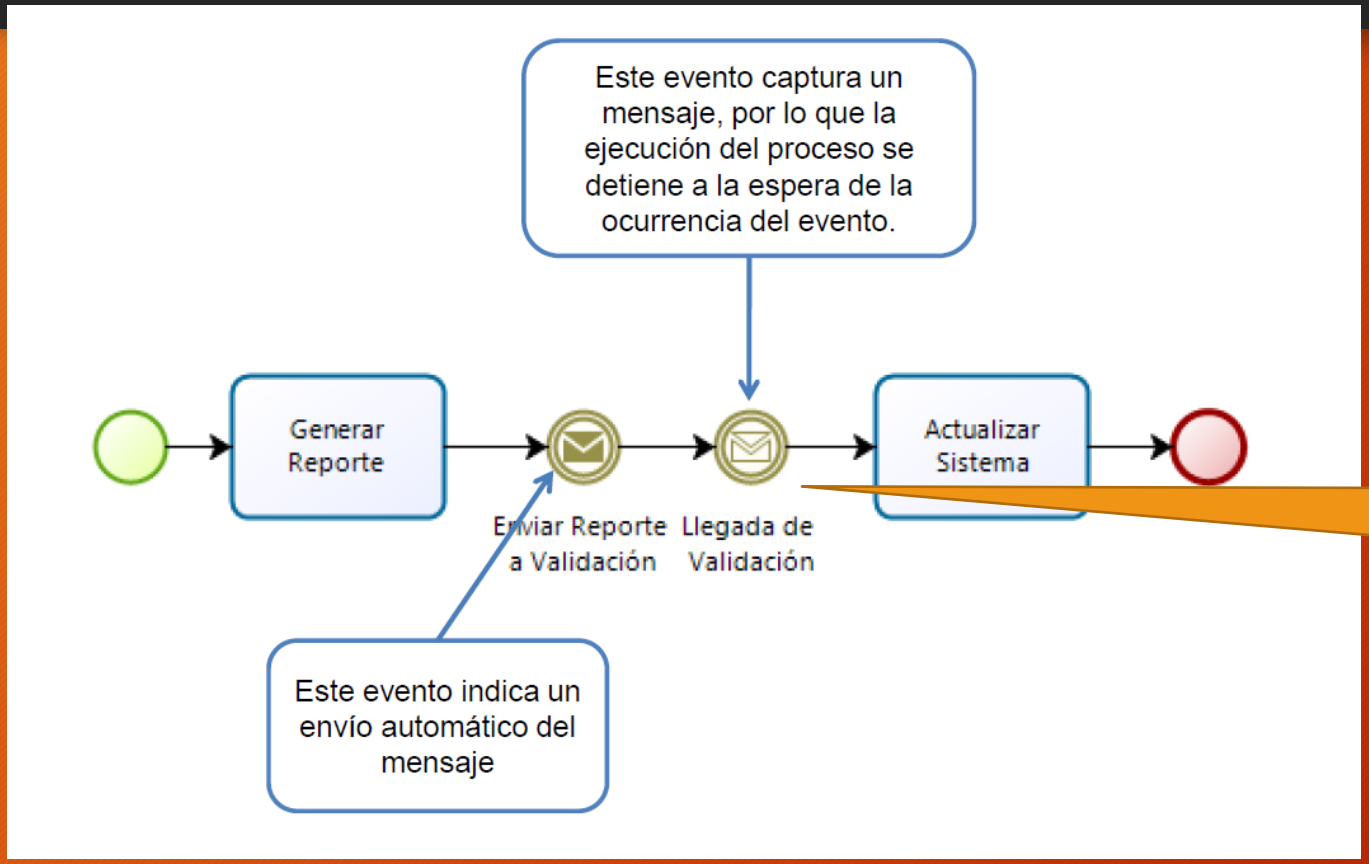
Eventos: Cosas que “suceden”

Hay diversos tipos de eventos:

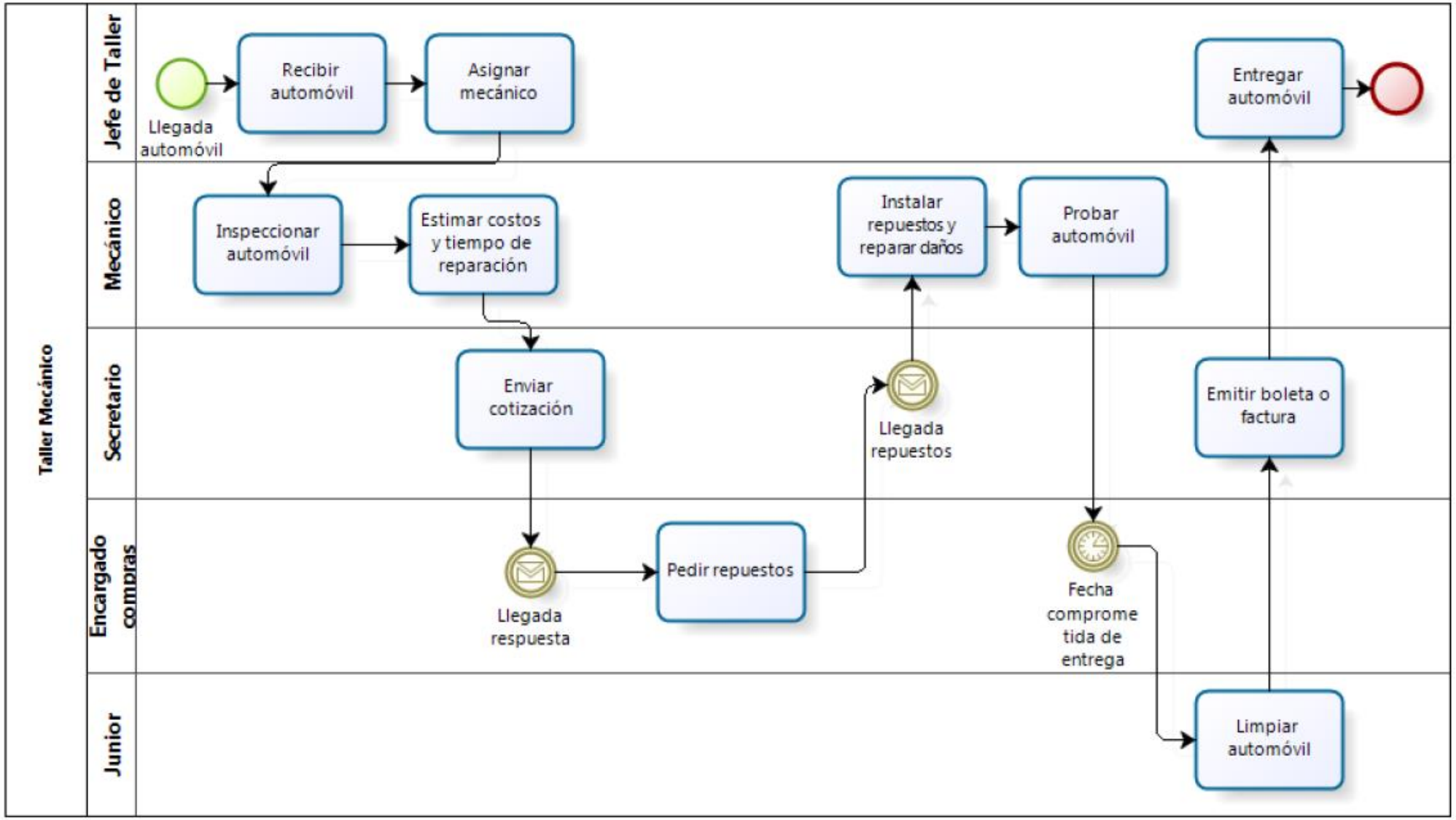
Eventos: Cosas que “suceden”



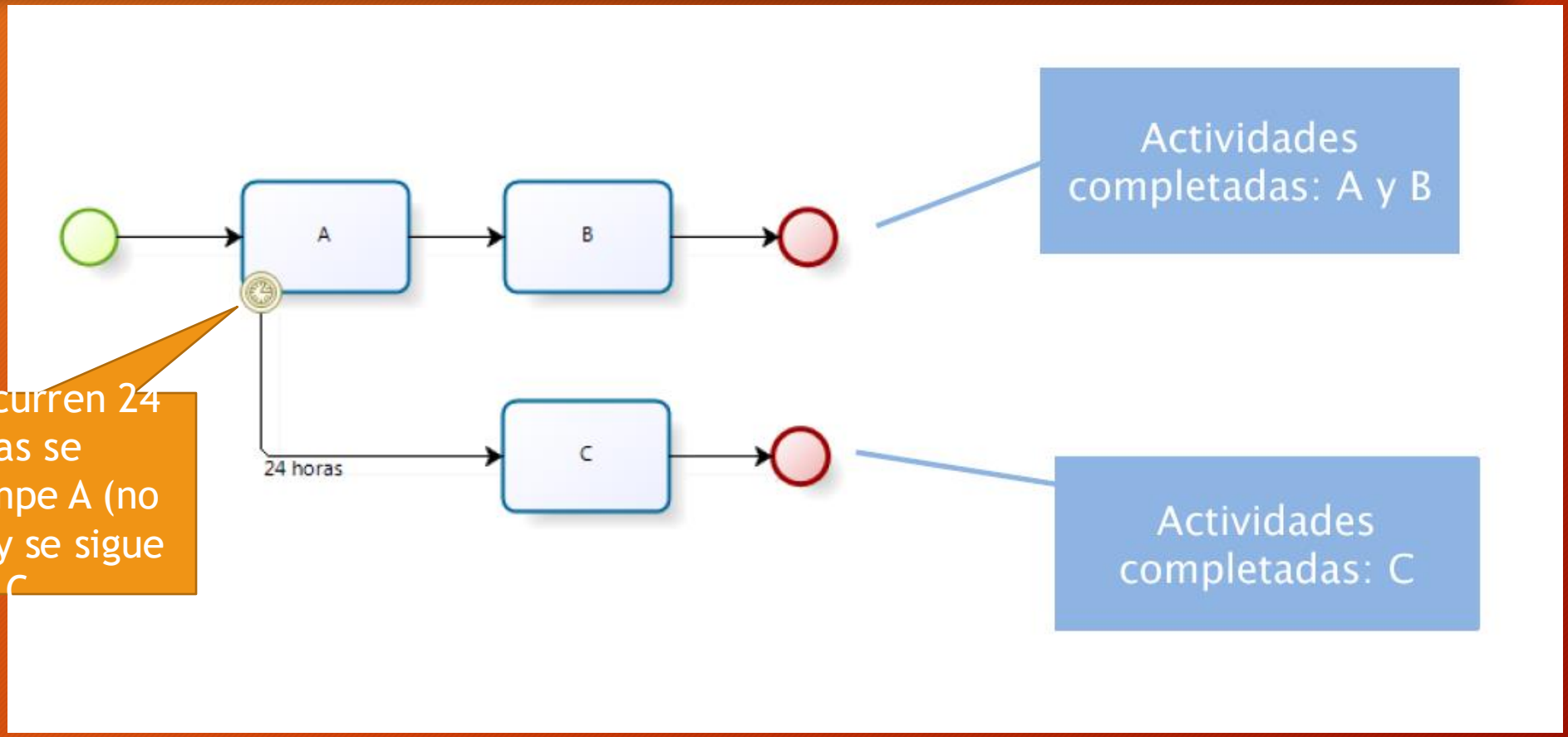
Eventos: Cosas que “suceden”



Deben tener un nombre diferente de las actividades


Taller Mecánico
Jefe de Taller
Mecánico
Secretario
Encargado compras
Junior
Llegada automóvil
Recibir automóvil
Asignar mecánico
Inspeccionar automóvil
Estimar costos y tiempo de reparación
Enviar cotización
Llegada respuesta
Pedir repuestos
Llegada repuestos
Instalar repuestos y reparar daños
Probar automóvil
Fecha comprometida de entrega
Limpiar automóvil
Emitir boleta o factura
Entregar automóvil

Eventos que interrumpen una actividad

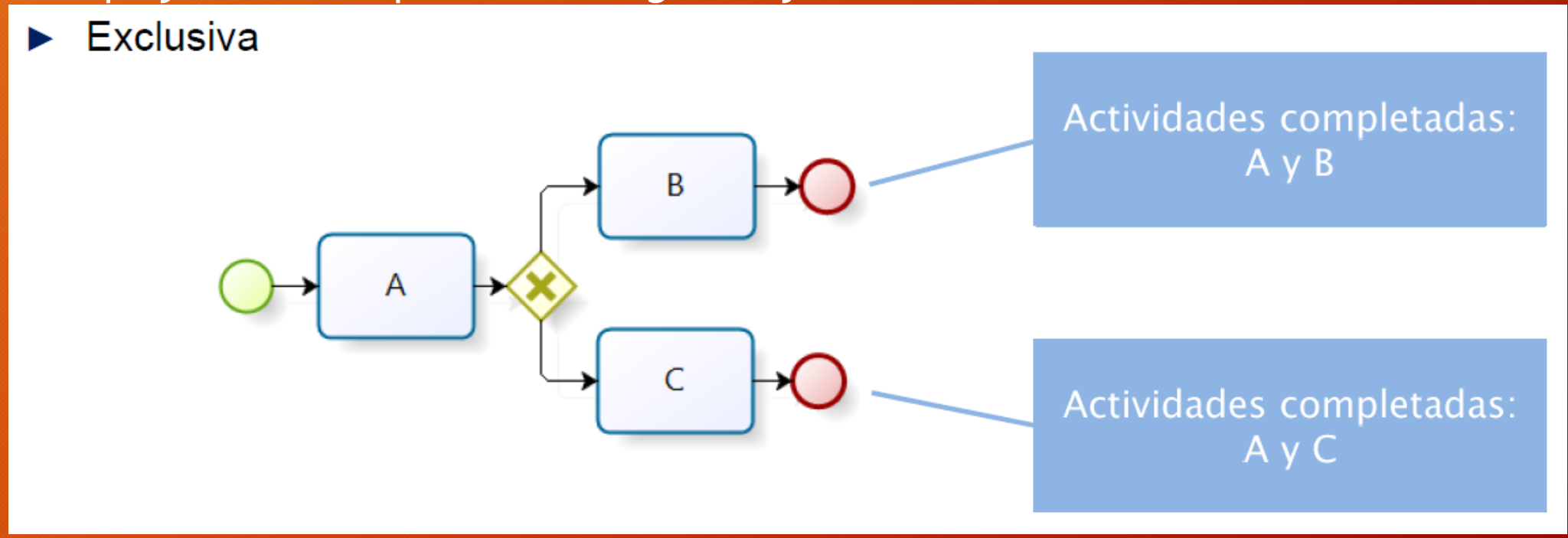


Gateways

- BPMN también incluye una forma de especificar flujos de secuencia complejos. Se hace por medio de *gateways*.

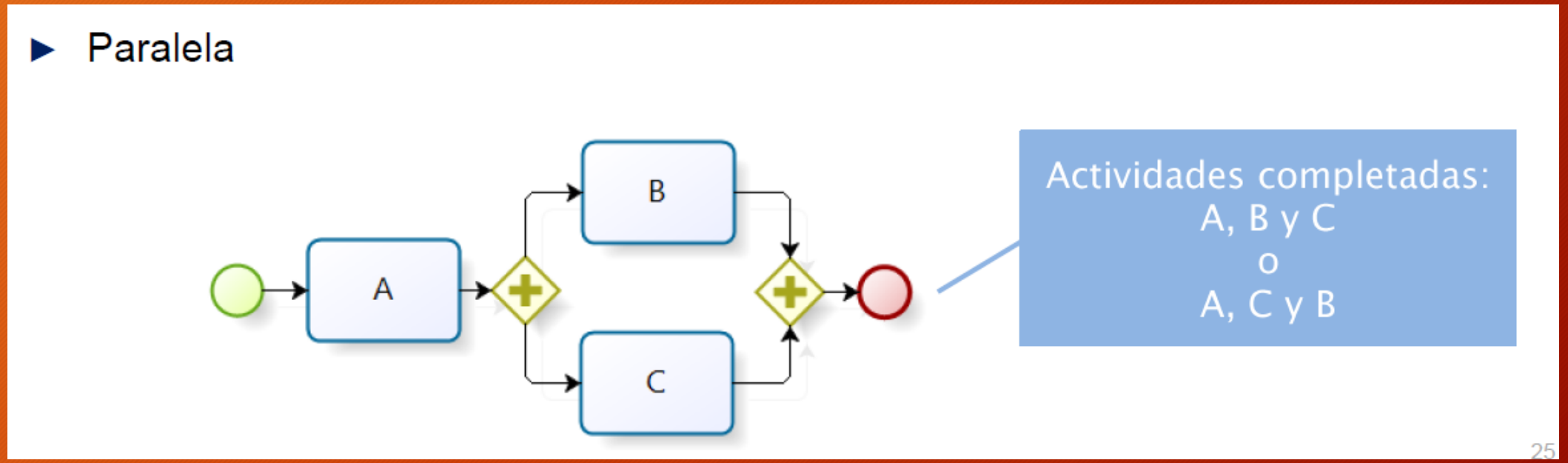
Gateways

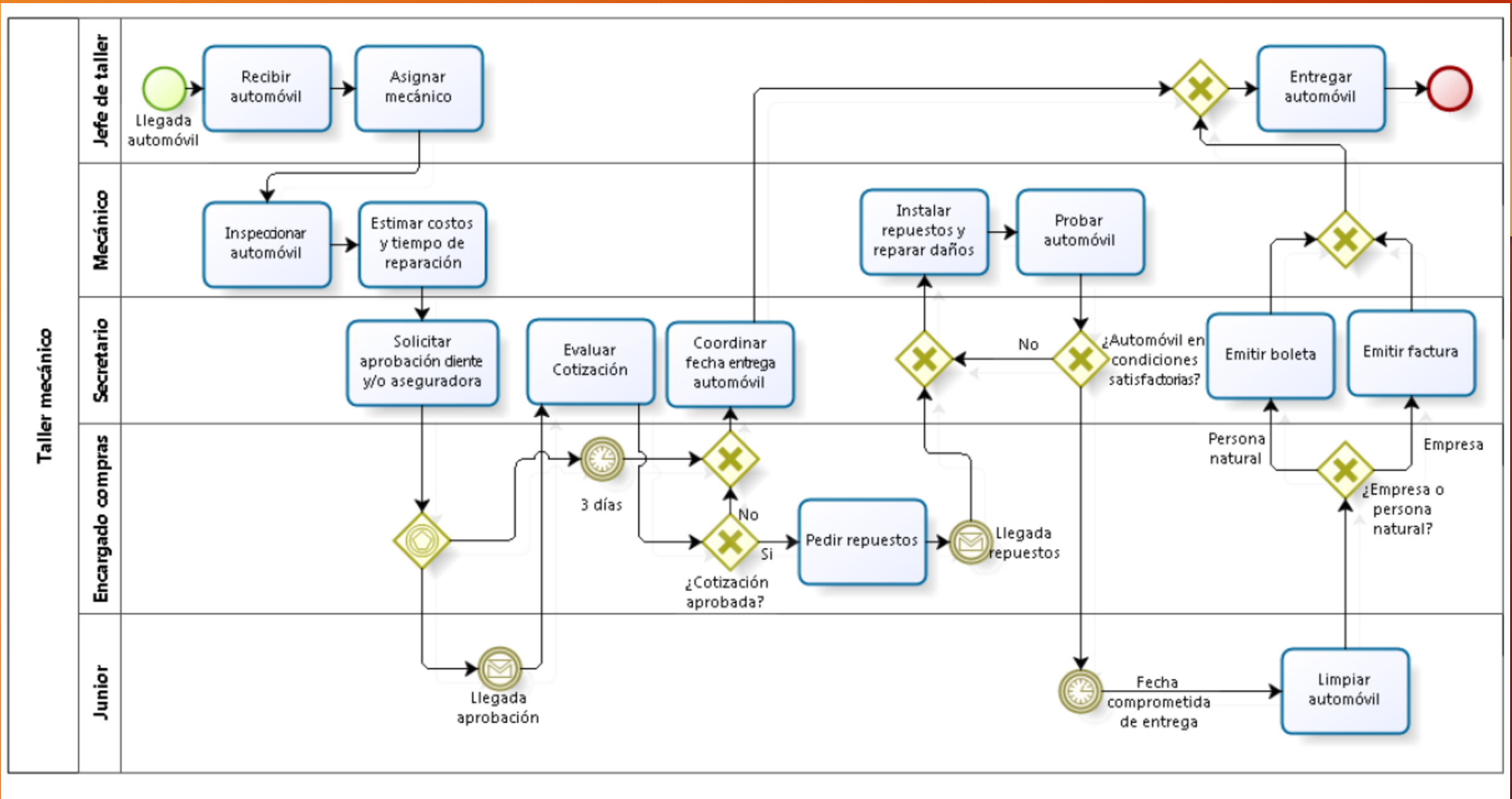
- BPMN también incluye una forma de especificar flujos de secuencia complejos. Se hace por medio de *gateways*.



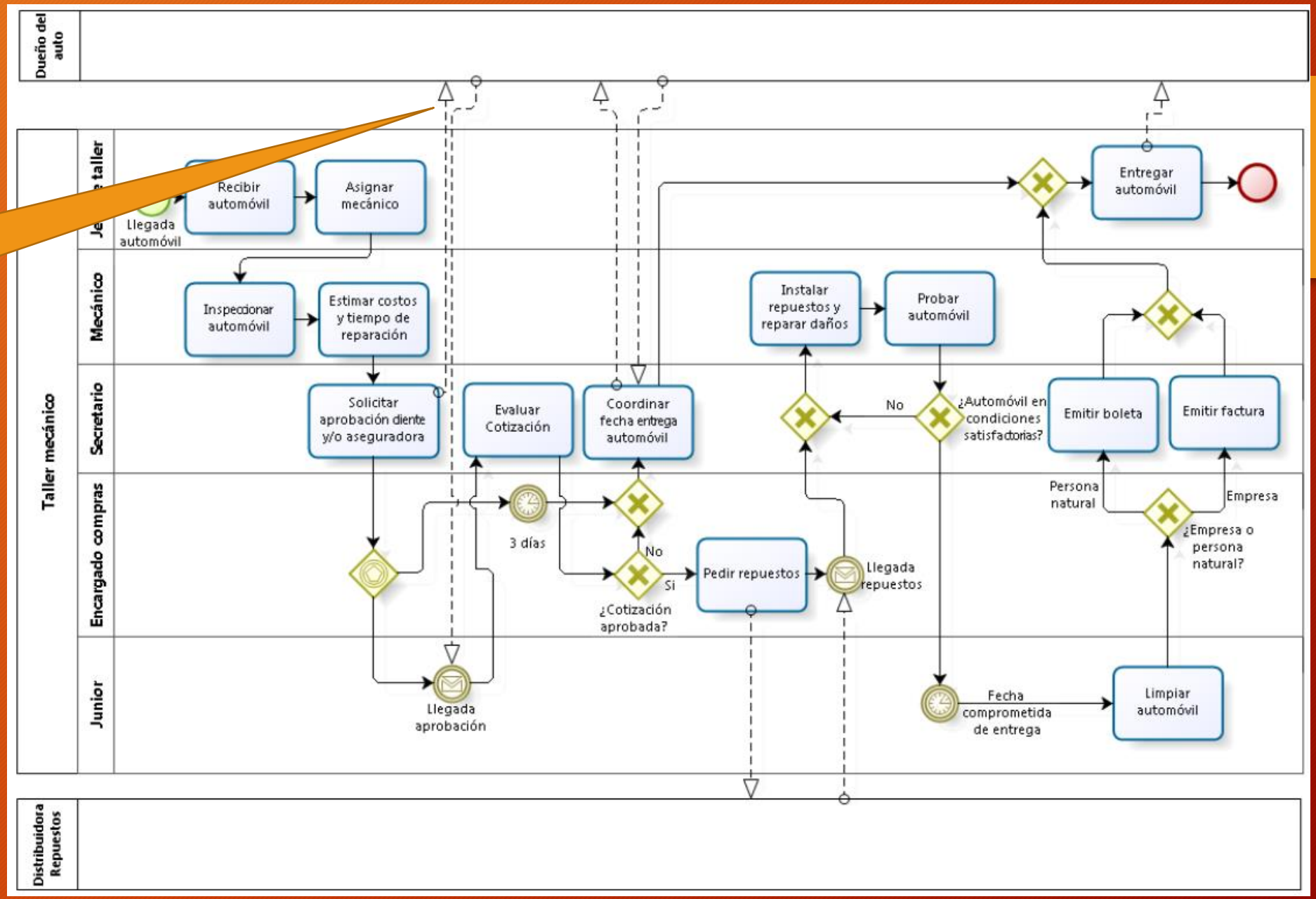
Gateways

- BPMN también incluye una forma de especificar flujos de secuencia complejos. Se hace por medio de *gateways*.




Taller mecánico
Jefe de taller
Mecánico
Secretario
Encargado compras
Junior
Llegada automóvil
Recibir automóvil
Asignar mecánico
Inspeccionar automóvil
Estimar costos y tiempo de reparación
Solicitar aprobación diente y/o aseguradora
Evaluar Cotización
Coordinar fecha entrega automóvil
3 días
No
Si
¿Cotización aprobada?
Pedir repuestos
Llegada repuestos
Llegada aprobación
Instalar repuestos y reparar daños
Probar automóvil
No
¿Automóvil en condiciones satisfactorias?
Entregar automóvil
Emitir boleta
Emitir factura
Persona natural
Empresa
¿Empresa o persona natural?
Fecha comprometida de entrega
Limpiar automóvil

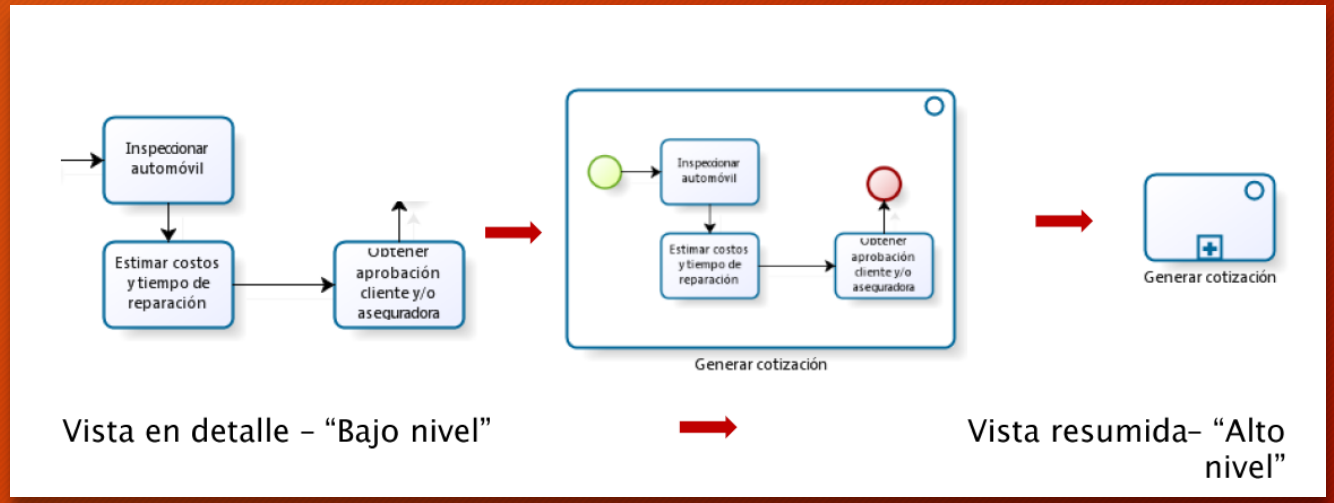
Interacción con otros pools



Profesor: Matías Rojas

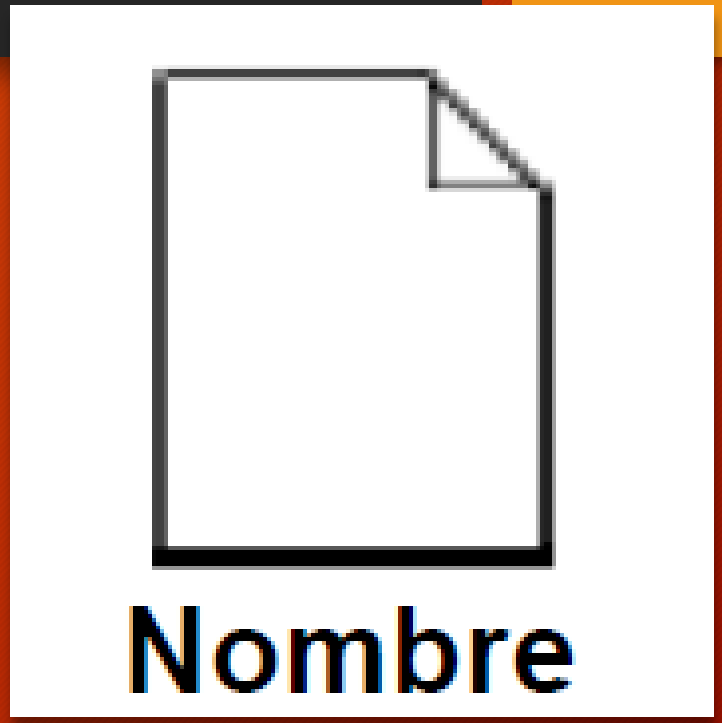
Subprocesos

- Los **subprocesos** corresponden a agrupaciones de actividades.
- Permiten definir distintos niveles de detalle en el diagrama.



Artefactos

- Elementos adicionales que enriquecen el diagrama.
- **Datos:**
  - Formularios
  - Información
  - Pueden ir incorporados en flujos de mensajes o como entrada y/o salida de actividades

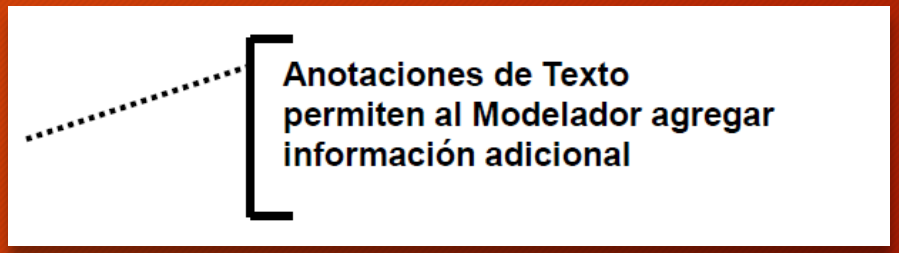


Artefactos

- Elementos adicionales que enriquecen el diagrama.
- **Grupo:**
  - Agrupación de actividades.

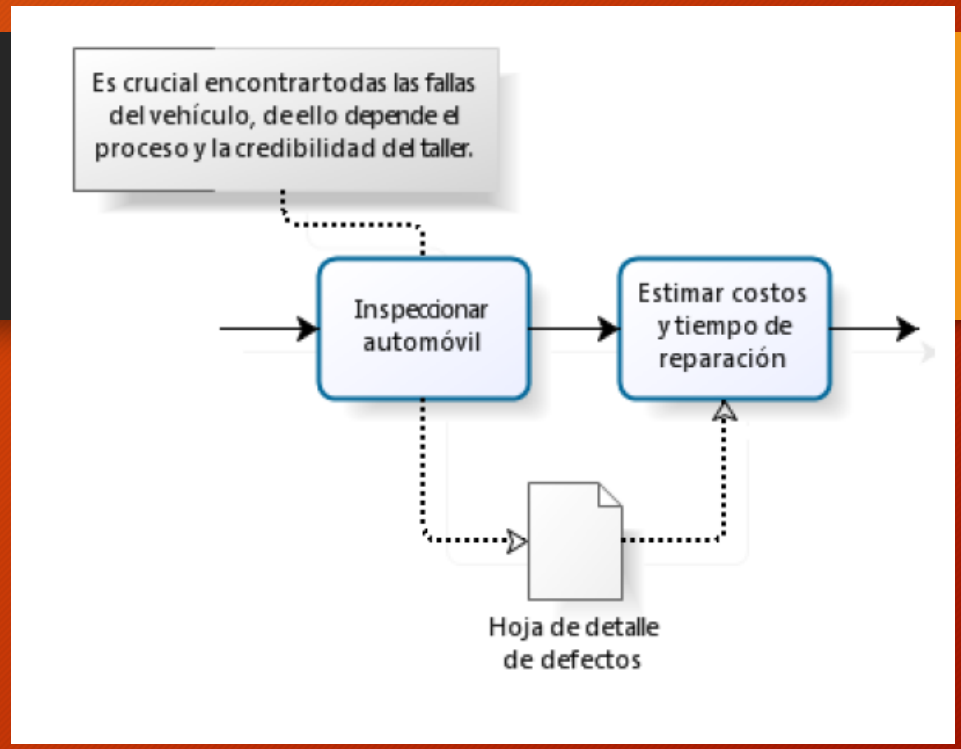
Artefactos

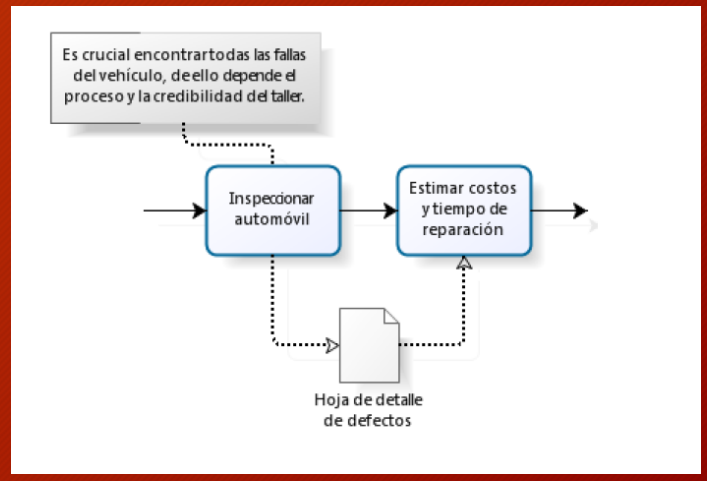
- Elementos adicionales que enriquecen el diagrama.
- **Anotación:**
  - Comentarios

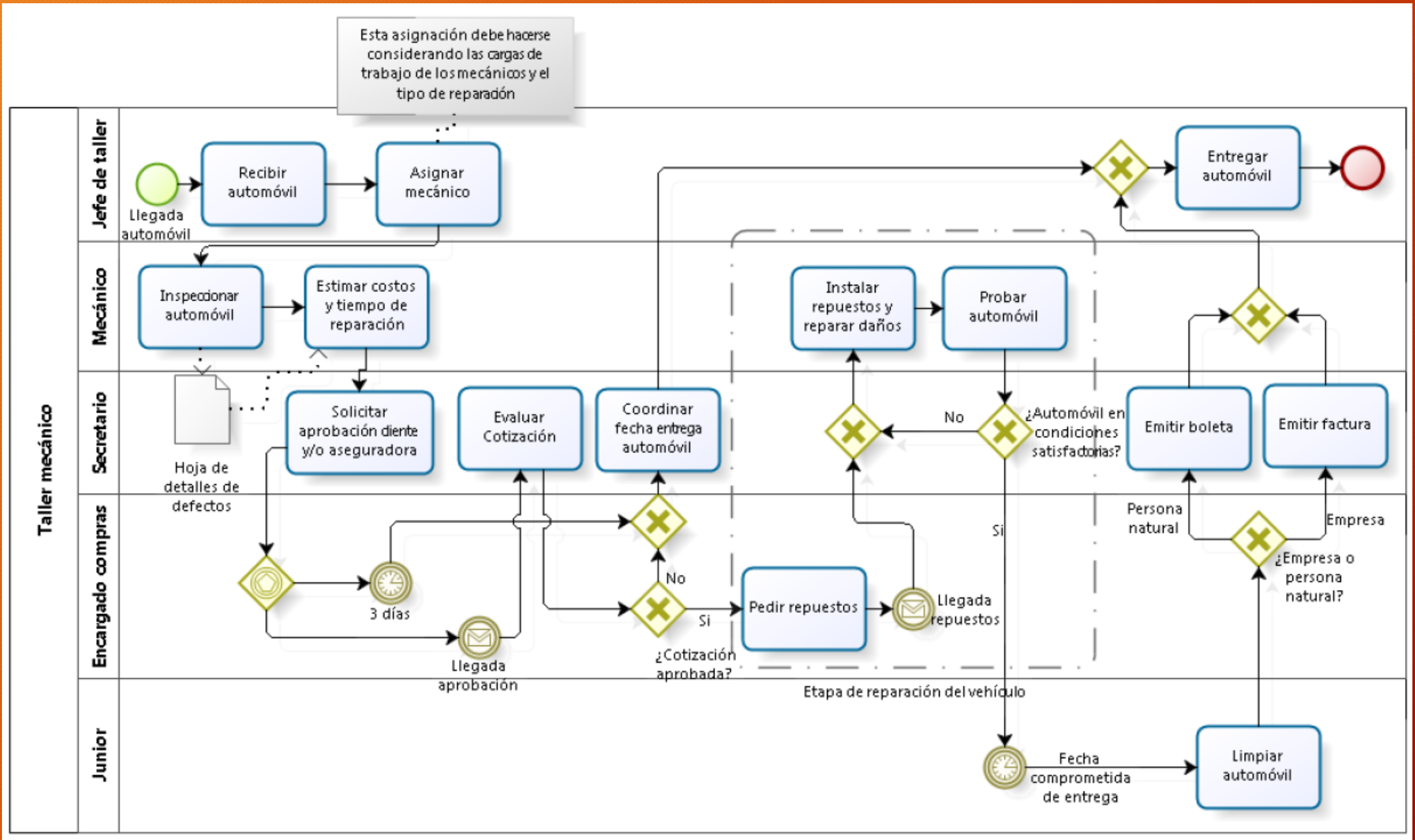


Asociaciones

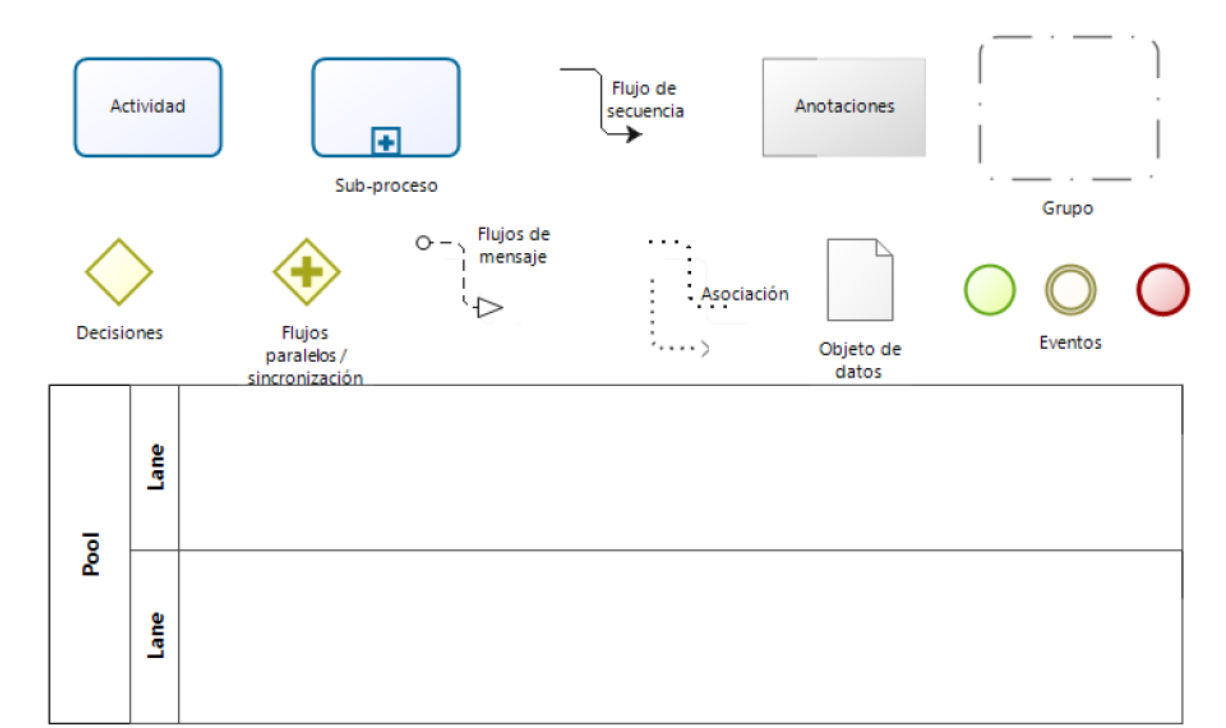
- Se usa para asociar datos, textos u otros artefactos con flujos de objetos.
- Se representa por una línea segmentada finalmente con el extremo en punta.
- También se usa para mostrar entradas y salidas de las actividades en relación al Dato.






Esta asignación debe hacerse considerando las cargas de trabajo de los mecánicos y el tipo de reparación
Taller mecánico
Jefe de taller
Mecánico
Secretario
Encargado compras
Junior
Llegada automóvil
Recibir automóvil
Asignar mecánico
Inspeccionar automóvil
Estimar costos y tiempo de reparación
Hoja de detalles de defectos
Solicitar aprobación cliente y/o aseguradora
Evaluar Cotización
Coordinar fecha entrega automóvil
3 días
Llegada aprobación
No
Si
¿Cotización aprobada?
Pedir repuestos
Llegada repuestos
Instalar repuestos y reparar daños
Probar automóvil
No
¿Automóvil en condiciones satisfactorias?
Si
Etapa de reparación del vehículo
Entregar automóvil
Emitir boleta
Emitir factura
Persona natural
Empresa
¿Empresa o persona natural?
Fecha comprometida de entrega
Limpiar automóvil

Resumen de Elementos Básicos



En las últimas semanas, los profesores de la EIT han estado sumamente extenuados debido a la carga de trabajo. Sin embargo, desean celebrar el final del semestre. Después de revisar solemnes, exámenes y memorias, muchos ya nos sentíamos agotados….es por ello que se están organizando varios carretes en la facultad. Hay tantos carretes que debemos decidir a cuáles asistir y cuándo hacerlo.

Los factores que permiten a los profesores elegir un carrete por sobre otro son:

- La comida
- Los bebestibles
- Las personas que asisten
- La música

Los profesores pueden ir a múltiples carretes, pero supondremos que el límite son 2 (si no fuera así cada profe debería hacer una aparición estelar por cada carrete y uno no alcanza a comer nada). Asimismo, el tiempo a estar en cada carrete será de un mínimo de 5 horas dentro de un lapso de 10 horas (desde las 20:00 hasta las 6:00 del siguiente día). Los carretes comienzan el día 01/08/2024 y terminan el día 07/08/2024.

Respecto del escenario general descrito, se toman los siguientes supuestos:

- Tanto los bebestibles como los platos de comida pueden consumirse en distintos carretes, además, cada carrete tiene múltiples platos y bebestibles distintos.
- Cada persona puede asistir con un máximo de 2 carretes durante este cierre de fin de semestre. Asimismo, el lapso mínimo de estadía en cada carrete es de cinco horas.
- Cada carrete pone varias canciones a lo largo de la velada, pero cada canción (por derechos) se puede escuchar en un único carrete.

Para ayudar a los profesores de la EIT a organizar su asistencia a los carretes, se desarrollará un sistema que funcionará como un espacio en donde los organizadores puedan crear eventos/carretes y los invitados (profesores, estudiantes, administrativos, entre otros) puedan gestionar su asistencia sujeto a reglas como las mencionadas (máximo 2 carretes por día). Este sistema debe permitir a los organizadores gestionar detalles del evento, mientras que los invitados pueden ver, registrarse y recibir notificaciones de los carretes.

Actividad 2

- (0,6 puntos) Proporcione un ejemplo detallado de un backlog de producto inicial que incluya al menos cinco historias de usuario, definiendo los criterios de aceptación para cada una. Además, utilizando UML, desarrolle un diagrama de casos de uso que represente las interacciones entre los profesores y el sistema, incluyendo al menos tres actores y detallando cómo cada actor interactúa con el sistema para planificar su asistencia a los carretes.

Patrones de Gof

Patrones Creacionales

Abstraen el proceso de creación de instancias de objetos. Ayudan a hacer a un sistema independiente de cómo se crean, se componen y se representan sus objetos.

Profesor: Matías Rojas

Problema

- En el desarrollo de software de simulación de vida marina, encontramos diferentes tipos de criaturas como peces, algas y corales. Cada uno de estos organismos tiene diferentes variantes, como peces de agua dulce y salada, algas verdes y rojas, corales blandos y duros.
- El reto es: ¿Cómo podemos generar estas diferentes familias de objetos (peces, algas, corales), cada una con sus diferentes variantes (agua dulce / salada, verde / roja, blando / duro) sin especificar las clases concretas?

Abstract Factory

- El patrón Abstract Factory nos permite resolver este problema al proveer una interfaz para crear familias de objetos relacionados sin necesidad de especificar sus clases concretas.
- Podríamos tener una interfaz "AbstractSeaFactory" con métodos como createFish(), createAlgae(), y createCoral(). Luego, tendríamos clases concretas como FreshWaterFactory y SaltWaterFactory que implementan esta interfaz, proporcionando las implementaciones específicas para cada tipo de organismo.

Ejemplo

```
public interface AbstractSeaFactory {
  Fish createFish();
  Algae createAlgae();
  Coral createCoral();
}

public class FreshWaterFactory implements
AbstractSeaFactory {
  // implementación de los métodos
}

public class SaltWaterFactory implements
AbstractSeaFactory {
  // implementación de los métodos
}
```

Problema

- Estás desarrollando un sistema de logística y necesitas una forma de crear diferentes tipos de transportes (camiones, barcos, aviones) dependiendo de la ruta de entrega. ¿Cómo puedes diseñar esta funcionalidad de manera que puedas agregar nuevos tipos de transporte en el futuro sin modificar el código existente?

Factory Method

- El patrón Factory Method propone definir un método de creación en una interfaz (o clase base), y luego tener subclases que implementen este método para crear objetos.
- Podrías tener una clase Logistics con un método createTransport(), y luego clases como RoadLogistics, SeaLogistics y AirLogistics que implementen este método para crear Truck, Ship y Airplane respectivamente.

Ejemplo

```
public abstract class Logistics {
  public abstract Transport createTransport();
  // más código
}

public class RoadLogistics extends Logistics {
  public Transport createTransport() {
    return new Truck();
  }
  // más código
}

public class SeaLogistics extends Logistics {
  public Transport createTransport() {
    return new Ship();
  }
  // más código
}
```

Problema

- Estás desarrollando una biblioteca de creación de documentos HTML. Cada documento HTML puede tener múltiples componentes como título, cuerpo, tablas, imágenes, listas, etc. Sin embargo, no todos los documentos necesitan todos estos componentes y su estructura puede variar mucho. ¿Cómo puedes diseñar una solución flexible para la creación de estos documentos?

Builder

- El patrón Builder nos permite separar la construcción de un objeto complejo de su representación. En este caso, podríamos tener una clase HTMLBuilder que tenga métodos como addTitle(), addBody(), addTable(), addImage(), addList(), etc.
- La clase HTMLBuilder podría tener un método build() que devuelve el documento HTML completo. De esta manera, los clientes pueden crear documentos HTML con diferentes estructuras utilizando el mismo código de construcción.

```
public class HTMLBuilder {
  // inicialización de un nuevo documento HTML
  public HTMLBuilder addTitle(String title) {
    // añadir título al documento
    return this;
  }
  public HTMLBuilder addBody(String body) {
    // añadir cuerpo al documento
    return this;
  }
  // más métodos
  public String build() {
    // retorna el documento HTML completo
  }
}
```

```
HTMLBuilder builder = new HTMLBuilder();
String document = builder.addTitle("Mi Título").addBody("Mi cuerpo").build();
```

Ejemplo

Problema

- Estás desarrollando un sistema donde necesitas un objeto que administre la configuración global de tu aplicación. Dicha configuración debe ser accesible desde cualquier punto del código y sólo debe haber una única instancia. ¿Cómo puedes implementar esto?

Singleton

101

- El patrón Singleton garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella. Este patrón es muy útil para cosas como administradores de log, conexiones de base de datos o el administrador de configuración del ejemplo.

Ejemplo

```
public class ConfigurationManager {
  private static ConfigurationManager instance;
  private String config;

  private ConfigurationManager() {
    // inicializar configuración
  }

  public static ConfigurationManager getInstance() {
    if (instance == null) {
      instance = new ConfigurationManager();
    }
    return instance;
  }

  public String getConfig() {
    return config;
  }

  // más métodos
}
```

```
ConfigurationManager configManager = ConfigurationManager.getInstance();
```

Tratan con la composición de clases u objetos. Se ocupan de cómo las clases y objetos son utilizados para componer estructuras de mayor tamaño.

Problema

- Estás desarrollando un sistema de procesamiento de pagos. Tu sistema usa un objeto CreditCard para manejar pagos, con métodos como charge(). Sin embargo, acabas de integrar un nuevo proveedor de pagos que usa un objeto PaymentGateway con un método executePayment(). ¿Cómo puedes hacer que tu sistema sea compatible con este nuevo proveedor de pagos sin cambiar mucho código?

Adapter

- El patrón Adapter te permite hacer que las clases con interfaces incompatibles trabajen juntas. En este caso, puedes crear un PaymentGatewayAdapter que implemente la interfaz de CreditCard y use una instancia de PaymentGateway.

```
public class PaymentGatewayAdapter extends CreditCard {
  private PaymentGateway paymentGateway;

  public PaymentGatewayAdapter(PaymentGateway paymentGateway) {
    this.paymentGateway = paymentGateway;
  }

  public void charge(double amount) {
    paymentGateway.executePayment(amount);
  }
}
```

Ejemplo

Problema

- Estás desarrollando un programa de dibujo. Tienes diferentes formas (círculo, cuadrado, triángulo) y cada una de estas formas puede ser dibujada en diferentes renderizadores (vectorial, rasterizado). ¿Cómo puedes evitar la explosión de clases (por ejemplo, RasterCircle, VectorSquare, etc.)?

Bridge

- El patrón Bridge te permite desacoplar una abstracción de su implementación, de modo que ambas puedan evolucionar de forma independiente. En este caso, podrías tener una interfaz Shape que usa una instancia de Renderer para dibujarse.

Ejemplo

```
public interface Renderer {
  void renderCircle(float radius);
  void renderSquare(float side);
  // otras funciones de renderizado
}

public abstract class Shape {
  protected Renderer renderer;

  public Shape(Renderer renderer) {
    this.renderer = renderer;
  }

  public abstract void draw();
}

public class Circle extends Shape {
  public Circle(Renderer renderer) {
    super(renderer);
  }

  public void draw() {
    renderer.renderCircle(radius);
  }
}
```

Problema

- Estás creando una estructura de archivos y carpetas. Ambos pueden tener un nombre y un tamaño, y las carpetas pueden contener archivos y otras carpetas. ¿Cómo puedes diseñar este sistema de manera que tratar con archivos y carpetas sea uniforme?

Composite

- El patrón Composite te permite tratar a los objetos individuales y a los grupos de objetos de la misma manera. En este caso, podrías tener una clase abstracta FileSystemElement que tenga métodos como getName() y getSize(). Luego, tendrías clases File y Folder que extienden FileSystemElement. La clase Folder podría tener una lista de FileSystemElement.

```
public abstract class FileSystemElement {
  // métodos comunes
}

public class File extends FileSystemElement {
  // métodos específicos de File
}

public class Folder extends FileSystemElement {
  private List<FileSystemElement> children;

  public void add(FileSystemElement element) {
    children.add(element);
  }

// métodos específicos de Folder
  public int getSize() {
    int totalSize = 0;
    for (FileSystemElement element : children) {
      totalSize += element.getSize();
    }
    return totalSize;
  }
}
```

Ejemplo

Problema

- Estás desarrollando un editor de texto. Necesitas una forma de cambiar el estilo del texto (negrita, cursiva, subrayado, etc.) sin cambiar la clase Text existente y permitiendo la combinación de diferentes estilos. ¿Cómo puedes implementar esto?

Decorator

- El patrón Decorator te permite añadir funcionalidades a objetos dinámicamente al envolverlos en un objeto de una clase decoradora. Podrías tener una clase abstracta StyledText que extiende a Text, y varias clases decoradoras como BoldText, ItalicText y UnderlinedText que también extienden a StyledText.

```
public abstract class StyledText extends Text {
  protected Text wrappedText;
}

public class BoldText extends StyledText {
  public BoldText(Text wrappedText) {
    this.wrappedText = wrappedText;
  }

  public String getText() {
    return "<b>" + wrappedText.getText() + "</b>";
  }
}

public class ItalicText extends StyledText {
  public ItalicText(Text wrappedText) {
    this.wrappedText = wrappedText;
  }

  public String getText() {
    return "<i>" + wrappedText.getText() + "</i>";
  }
}
```

Ejemplo

Problema

- Estás utilizando una biblioteca de compresión de archivos que tiene una API complicada con muchas clases e interacciones entre ellas. ¿Cómo puedes simplificar su uso?

Facade

- El patrón Facade proporciona una interfaz simplificada para una biblioteca o sistema complejo. En este caso, podrías crear una clase CompressionFacade que tenga métodos simples como compressFile() y decompressFile().

```
public class CompressionFacade {
  public void compressFile(String inputFile,
String outputFile) {
    // Código para comprimir un archivo
utilizando la biblioteca
  }

  public void decompressFile(String inputFile,
String outputFile) {
    // Código para descomprimir un archivo
utilizando la biblioteca
  }
}
```

Ejemplo

Problema

- Estás desarrollando un simulador de tráfico y necesitas representar miles de coches en la pantalla. Cada coche tiene un modelo y color, pero muchos coches comparten el mismo modelo y color. ¿Cómo puedes optimizar el uso de memoria?

Flywight

- El patrón Flyweight te permite compartir los datos comunes entre muchos objetos similares. En este caso, podrías tener una clase CarModel que contiene el modelo y el color del coche. Luego, cada Car tiene una referencia a un CarModel y solo almacena datos únicos como su posición.

Ejemplo

```
public class CarModel {
  private String model;
  private String color;

  public CarModel(String model, String color) {
    this.model = model;
    this.color = color;
  }

  // getters y otros métodos
}

public class Car {
  private CarModel carModel;
  private Position position;

  public Car(CarModel carModel, Position position) {
    this.carModel = carModel;
    this.position = position;
  }

  // getters y otros métodos
}
```

Caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad. Corresponden a los algoritmos y a la asignación de responsabilidades entre objetos.

Problema

- Estás desarrollando una aplicación de chat en la que varios usuarios pueden enviar y recibir mensajes. Quieres evitar que cada usuario tenga que conocer a todos los demás para enviar mensajes. ¿Cómo puedes gestionar las interacciones entre los usuarios?

Mediator

- El patrón Mediator proporciona un objeto central a través del cual los diferentes componentes de un sistema pueden comunicarse. En este caso, podrías tener un objeto ChatMediator a través del cual los usuarios envían y reciben mensajes.

Ejemplo

```
public interface Mediator {
  void sendMessage(String message, User user);
}

public class ChatMediator implements Mediator {
  private List<User> users;

  public void sendMessage(String message, User user) {
    for (User u : users) {
      if (u != user) {
        u.receive(message);
      }
    }
  }
}

public class User {
  private Mediator mediator;

  public void send(String message) {
    mediator.sendMessage(message, this);
  }

  public void receive(String message) {
    // código para manejar el mensaje recibido
  }
}
```

Problema

- Estás desarrollando un editor de texto y quieres proporcionar la posibilidad de deshacer y rehacer cambios. ¿Cómo puedes implementar esta funcionalidad sin violar el principio de encapsulamiento?

Memento

- El patrón Memento te permite capturar y externalizar el estado interno de un objeto sin violar el encapsulamiento. En este caso, podrías crear mementos del estado del texto en cada cambio, y usarlos para deshacer y rehacer cambios.

Ejemplo

```
public class TextEditor {
  private String text;

  public Memento createMemento() {
    return new Memento(text);
  }

  public void restore(Memento memento) {
    text = memento.getState();
  }

  public void setText(String text) {
    this.text = text;
  }
}

public class Memento {
  private String state;

  public Memento(String state) {
    this.state = state;
  }

  public String getState() {
    return state;
  }
}
```

```
public class Caretaker {
  private Stack<Memento> undoStack = new Stack<>();
  private Stack<Memento> redoStack = new Stack<>();

  public void saveState(TextEditor editor) {
    undoStack.push(editor.createMemento());
    redoStack.clear();
  }

  public void undo(TextEditor editor) {
    if (!undoStack.isEmpty()) {
      redoStack.push(editor.createMemento());
      editor.restore(undoStack.pop());
    }
  }

  public void redo(TextEditor editor) {
    if (!redoStack.isEmpty()) {
      undoStack.push(editor.createMemento());
      editor.restore(redoStack.pop());
    }
  }
}
```

Problema

- Estás desarrollando un sistema de notificaciones donde varios componentes deben ser notificados cuando se produce un evento específico. ¿Cómo puedes diseñar el sistema para que sea extensible y no requiera cambios en el emisor del evento cuando se añaden nuevos observadores?

Observer

- El patrón Observer te permite definir una dependencia de uno a muchos entre objetos, de manera que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.

Ejemplo

```
public interface Observer {
  void update();
}

public class ConcreteObserver implements Observer {
  public void update() {
    // realiza alguna acción en respuesta a la notificación
  }
}

public class Subject {
  private List<Observer> observers = new ArrayList<>();

  public void attach(Observer observer) {
    observers.add(observer);
  }

  public void detach(Observer observer) {
    observers.remove(observer);
  }

  public void notifyObservers() {
    for (Observer observer : observers) {
      observer.update();
    }
  }
}
```

Problema

- Estás diseñando un reproductor de música que tiene diferentes estados (reproduciendo, pausado, detenido) y el comportamiento de ciertas acciones (como presionar el botón de reproducir/pausa) depende del estado actual. ¿Cómo puedes implementar esto de forma eficiente y mantener tu código limpio?

State

- El patrón State te permite cambiar el comportamiento de un objeto cuando su estado interno cambia. Se puede implementar definiendo una interfaz de estado y subclases para cada estado específico, y luego invocando métodos en el objeto de estado actual.

Ejemplo

```
public interface State {
  void pressPlay(MusicPlayer player);
}

public class PlayingState implements State {
  public void pressPlay(MusicPlayer player) {
    player.setState(new PausedState());
  }
}

public class MusicPlayer {
  private State state;

  public void pressPlay() {
    state.pressPlay(this);
  }

  public void setState(State state) {
    this.state = state;
  }
}
```

Problema

- Estás diseñando una aplicación de navegación que puede calcular rutas utilizando varios algoritmos (por ejemplo, el camino más corto, el camino más rápido, evitar autopistas). ¿Cómo puedes organizar tu código para hacerlo extensible y mantenible?

Strategy

- El patrón Strategy te permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Puedes cambiar el algoritmo que se usa en tiempo de ejecución.

Ejemplo

```
public interface RoutingStrategy {
  Route calculateRoute(Point start, Point end);
}

public class FastestRouteStrategy implements RoutingStrategy {
  public Route calculateRoute(Point start, Point end) {
    // Implementación del algoritmo para la ruta más rápida
  }
}

public class NavigationApplication {
  private RoutingStrategy strategy;

  public void setRoutingStrategy(RoutingStrategy strategy) {
    this.strategy = strategy;
  }

  public Route calculateRoute(Point start, Point end) {
    return strategy.calculateRoute(start, end);
  }
}
```

Problema

- Estás diseñando un marco de trabajo para juegos de mesa que incluye varias etapas comunes, como la inicialización del juego, el turno de los jugadores y la finalización del juego. Sin embargo, los detalles de cada etapa pueden variar dependiendo del juego específico. ¿Cómo puedes estructurar tu código de manera eficiente?

Template Method

- El patrón Template Method define el esqueleto de un algoritmo en un método, dejando algunos pasos a las subclases. Las subclases pueden redefinir ciertos pasos del algoritmo sin cambiar su estructura.

Ejemplo

```
public abstract class Game {
  public final void play() {
    initialize();
    startPlay();
    endPlay();
  }

  abstract void initialize();
  abstract void startPlay();
  abstract void endPlay();
}

public class Chess extends Game {
  @Override
  void initialize() {
    // Implementación específica de la inicialización del ajedrez
  }

  @Override
  void startPlay() {
    // Implementación específica de cómo se juega al ajedrez
  }

  @Override
  void endPlay() {
    // Implementación específica de cómo termina un juego de ajedrez
  }
}
```

Problema

- Estás desarrollando un editor de gráficos que puede trabajar con varios tipos de formas (círculos, rectángulos, líneas). Necesitas implementar varias operaciones (como dibujar y cambiar el tamaño) que se comporten de manera diferente dependiendo de la forma específica. ¿Cómo puedes organizar tu código de manera que sea fácil agregar nuevas operaciones y nuevas formas?

Visitor

- El patrón Visitor te permite definir nuevas operaciones sin cambiar las clases de los elementos sobre los que operan. Los visitantes son objetos que almacenan el estado de la operación y proporcionan un método de visita para cada tipo de elemento.

Ejemplo

```
public interface Shape {
  void accept(Visitor visitor);
}

public class Circle implements Shape {
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

public interface Visitor {
  void visit(Circle circle);
  void visit(Rectangle rectangle);
}

public class DrawVisitor implements Visitor {
  public void visit(Circle circle) {
    // Dibuja un círculo
  }

  public void visit(Rectangle rectangle) {
    // Dibuja un rectángulo
  }
}
```

Resumen

Creacionales	
Abstract Factory	Crea familias de objetos relacionados sin especificar sus clases concretas.
Factory Method	Delega la creación de objetos a las subclases.
Builder	Separa la creación de un objeto de su representación. Permite la creación de diferentes representaciones.
Singleton	Garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a ella.
Prototype	Crea nuevos objetos clonando un objeto prototipo.

Resumen

Estructurales	
Adapter	Permite que dos interfaces incompatibles trabajen juntas.
Bridge	Desacopla una abstracción de su implementación, permitiendo que ambas varíen independientemente.
Composite	Compone objetos en estructuras de árbol para representar jerarquías todo/parte.
Decorator	Añade responsabilidades adicionales a un objeto de forma dinámica.
Facade	Proporciona una interfaz simplificada a un subsistema complejo.
Flyweight	Reutiliza objetos compartidos en lugar de crear nuevos, ahorrando memoria.
Proxy	Proporciona un sustituto o marcador de posición para otro objeto para controlar el acceso a este.

Resumen

Comportamiento	
Command	Encapsula una solicitud como un objeto, permitiendo parametrizar clientes con colas, solicitudes y operaciones.
Chain of Responsibility	Evita acoplar el remitente de una solicitud a su receptor, dando a más de un objeto la oportunidad de manejar la solicitud.
Interpreter	Define una representación de la gramática de un lenguaje y un intérprete para usar esa gramática.
Iterator	Proporciona una forma de acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente.
Mediator	Define un objeto que encapsula cómo un conjunto de objetos interactúan.
Memento	Captura y externaliza el estado interno de un objeto sin violar la encapsulación, para que el objeto pueda ser restaurado a ese estado más tarde.
Observer	Define una dependencia uno a muchos de tal manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
State	Permite que un objeto altere su comportamiento cuando su estado interno cambia.
Strategy	Define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
Template Method	Define el esqueleto de un algoritmo en una operación, aplazando algunos pasos a las subclases.
Visitor	Representa una operación a ser realizada en los elementos de una estructura de objeto, permitiendo definir una nueva operación sin cambiar las clases de los elementos en los que opera.

Actividad 3

- (0,6 puntos) Elija un patrón de diseño que considere adecuado para la implementación del sistema descrito en el caso de estudio. Justifique su elección y proporcione una implementación del patrón de diseño elegido en un diagrama de clases o en pseudocódigo, detallando cómo se aplicaría en el contexto del sistema de planificación de carretes.

Testing

Puede definirse como dos acciones:

Validation Testing: Demostrar que el sistema cumple con sus requerimientos.

Defect Testing: Encontrar inputs o secuencia de inputs donde el comportamiento del sistema es incorrecto, indeseado o se adecúa a lo especificado.

"El testing solo muestra la presencia de errores, no su ausencia" (Edsger Dijkstra, 1972)

Verificación y Validación

Validación: ¿Estamos construyendo el producto correcto?

- Consiste en verificar que el sistema satisface las expectativas del cliente.

Verificación: ¿Estamos construyendo el producto correctamente?

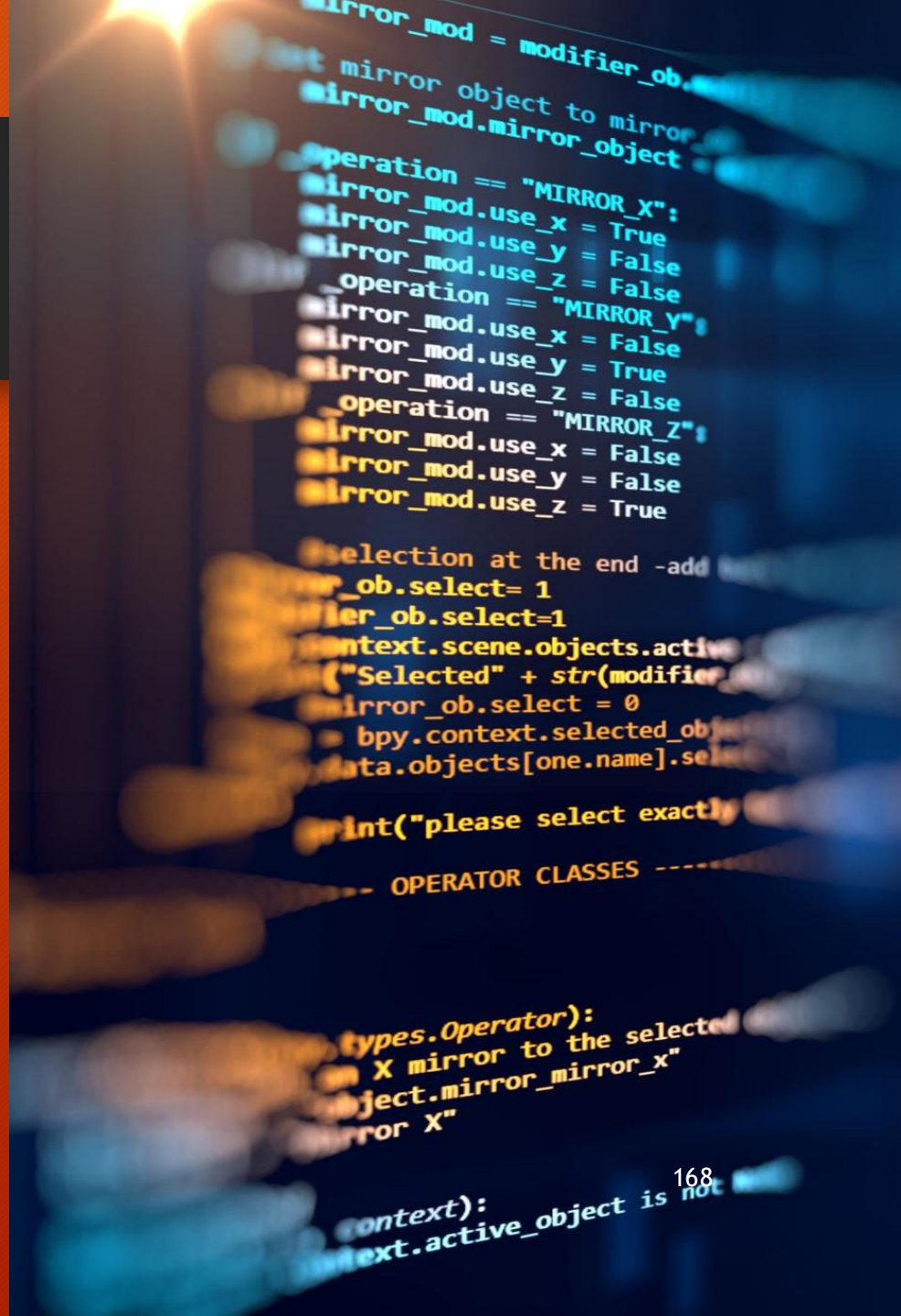
- Consiste en verificar que se cumplen los requisitos funcionales y no-funcionales.

Es básicamente evaluar si el sistema es suficientemente confiable para ser utilizado.

Este nivel de confianza depende del objetivo con el que se creó el sistema, las expectativas que se depositaron en él, y el ambiente de marketing de este.

Proceso de Testeo

- El proceso de testeo puede dividirse en tres etapas:
  - ***Development testing (Pruebas de desarrollo):*** Se ejecuta durante la etapa de desarrollo, principalmente por desarrolladores.
  - ***Release testing (Versiones de prueba):*** Donde un equipo de testeo testea una versión completa del sistema antes que sea lanzada para los usuarios.
  - ***User testing (Pruebas de usuario):*** Cuando usuarios o potenciales usuarios testean el sistema en su propio ambiente.
- En general, el proceso de testeo involucra una mezcla de testeo automático y manual.



Development Testing

- Incluye todas las actividades de testeo realizadas por el equipo desarrollador.
- Hay tres etapas en el *development testing*:
  1. **Unit testing:** Donde unidades individuales del programa o clases son testeadas. Se enfoca principalmente en el testeo de funcionalidades.
  2. **Component testing:** Donde múltiples unidades individuales se integran para crear componentes compuestos. Se enfoca principalmente en el testeo de interfaces de componentes que dan acceso a funcionalidades específicas.
  3. **System testing:** Donde alguna o todas las componentes de un sistema se integran y el sistema se prueba como un todo. Se enfoca principalmente en testear interacciones entre componentes.
- En general, este proceso de testing se desarrolla bajo la premisa de *pair programming* (un developer, y un tester, que pueden ir intercambiando). O puede haber un equipo completo dedicado a testear.

Profesor: Matías Rojas

Eligiendo *unit test cases*

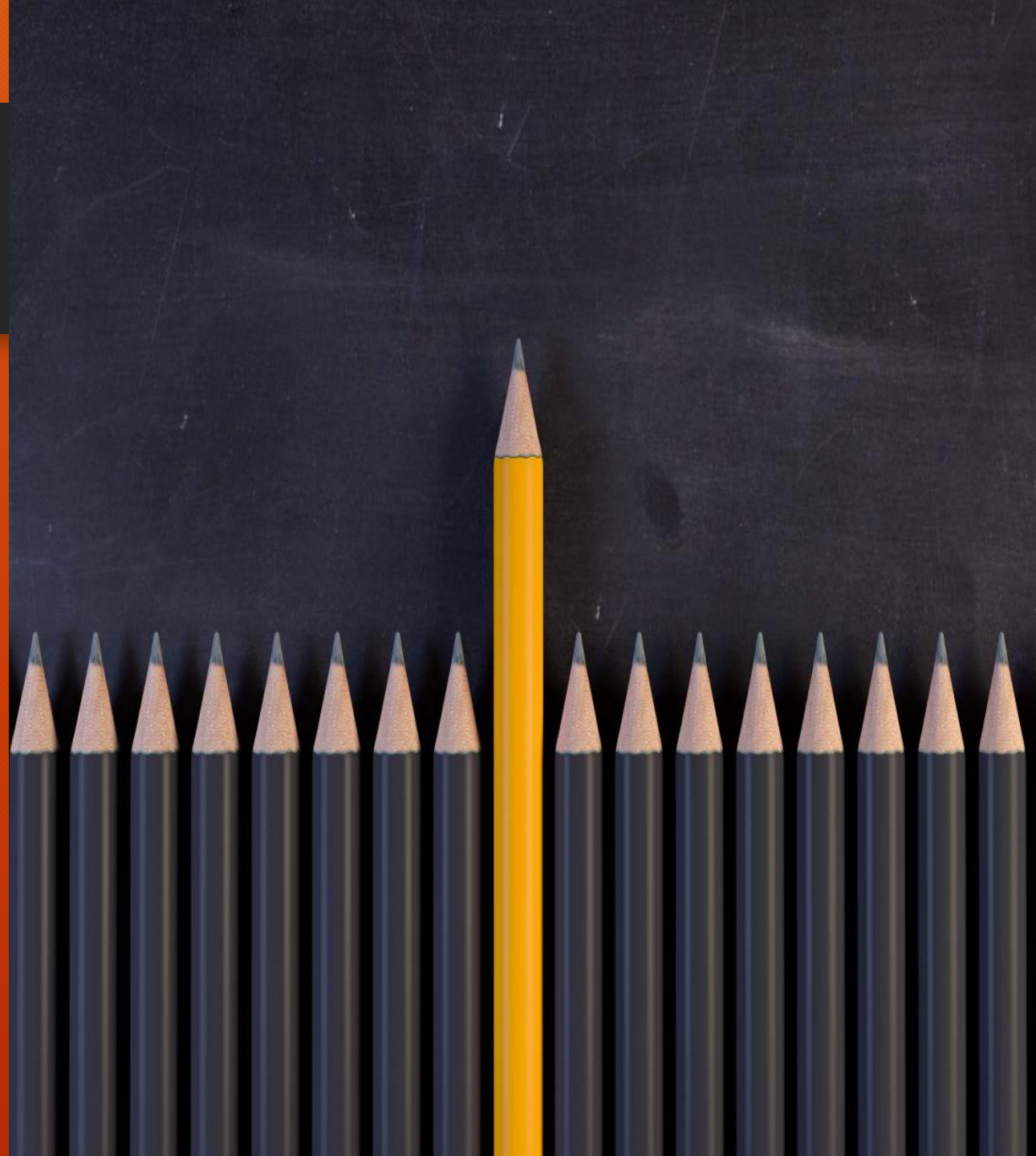
- Testing es caro y complejo, entonces hay que elegir bien qué se va a testear. En este caso, eso implica dos cosas:
  - Que muestren que el componente hace lo que debería hacer.
  - Que el test muestre defectos que podría tener el componente.
- Entonces, se deberían diseñar dos tipos de test cases:
  - Uno que refleje la operación normal de un programa y muestre lo que el componente debería hacer (un buen ejemplo es testear CRUD's)
  - Otro que refleje condiciones anormales de un programa (casos bordes, etc).

Eligiendo *unit test cases*

- Se está haciendo testing en un bar:
  - Un cliente pide 1 cerveza, se le da.
  - Un cliente pide 999999 cervezas, se le dice que son muchas.
  - Un cliente pide -1 cervezas, lo echan por borracho.
- En la vida real, entra un cliente al bar, y pide un vaso de agua… el bar se incendia.

Eligiendo *unit test cases*

- Hay dos estrategias para elegir efectivamente qué test cases realizar:
  - *Partition testing:* Donde se identifican grupo de inputs que tengan características en común y que deban ser procesados de la misma maneras.
  - *Guideline-base testing*: donde se usan lineamientos para elegir los casos de prueba. Dichos lineamientos reflejan la experiencia previa de los tipos de errores que suelen cometer los programadores al desarrollar componentes



Pruebas de Caja Negra

- La partición de equivalencia es una técnica de pruebas de caja negra que divide el conjunto de datos de entrada de un programa en clases de equivalencia, de las cuales se predicen las clases de valor de entrada que es probable que detecten errores. Por ejemplo, si un programa acepta de 4 a 8 entradas que son enteros de cinco dígitos mayores a 10,000, se pueden identificar las siguientes particiones de entrada:
  - Menos de 10,000
  - Entre 10,000 y 99,999
  - Más de 99,999"

Pruebas de Caja Blanca

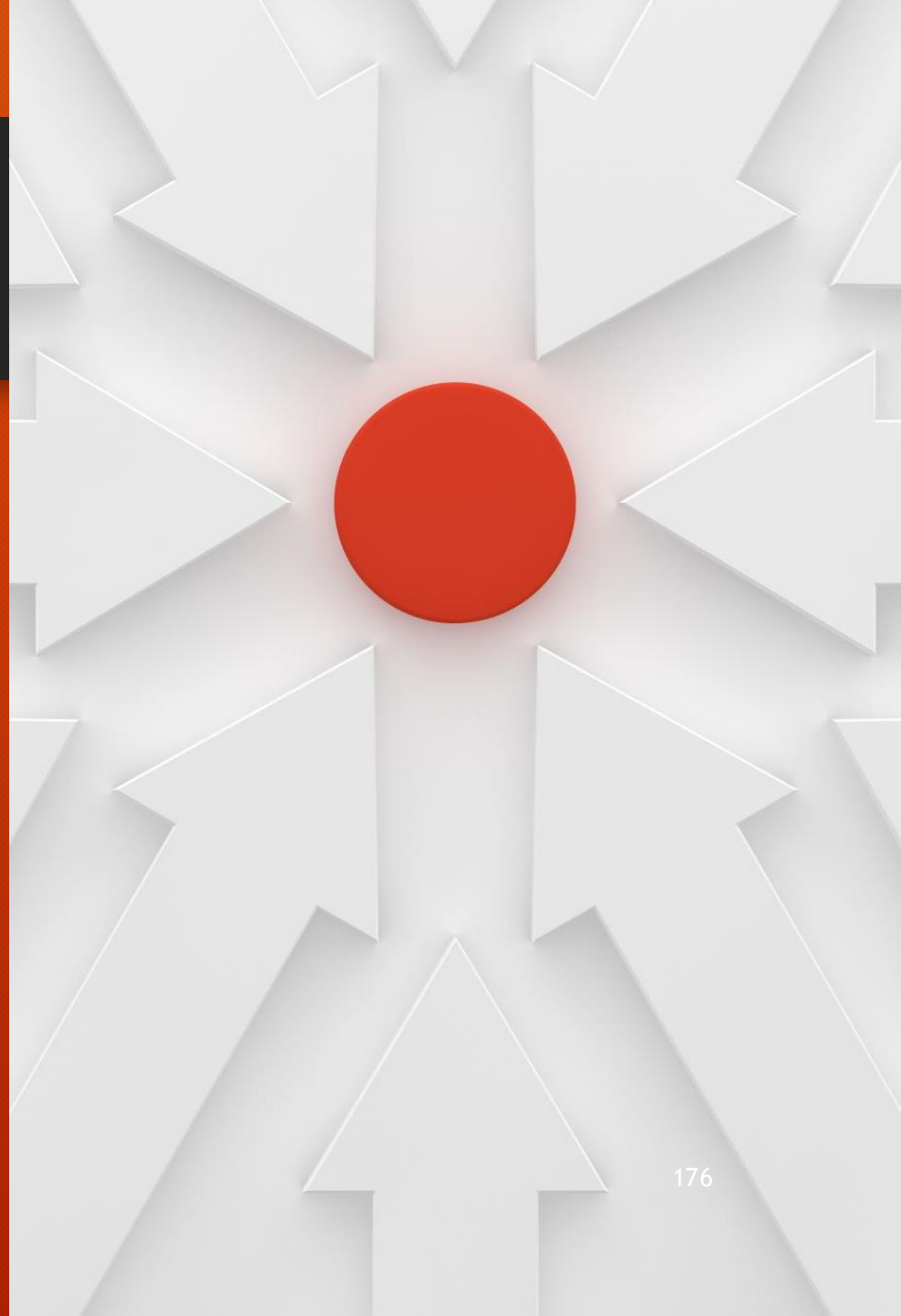
- Las pruebas de caja blanca, a diferencia de las pruebas de caja negra, requieren conocimiento del funcionamiento interno del sistema. Esta información se puede utilizar para identificar particiones de excepción: diferentes rangos donde se debe aplicar el mismo manejo de excepción.
- Por ejemplo, si tu código incluye excepciones para manejar entradas incorrectas, estos rangos pueden identificarse y probarse de forma separada.

Guidelines para *unit testing*

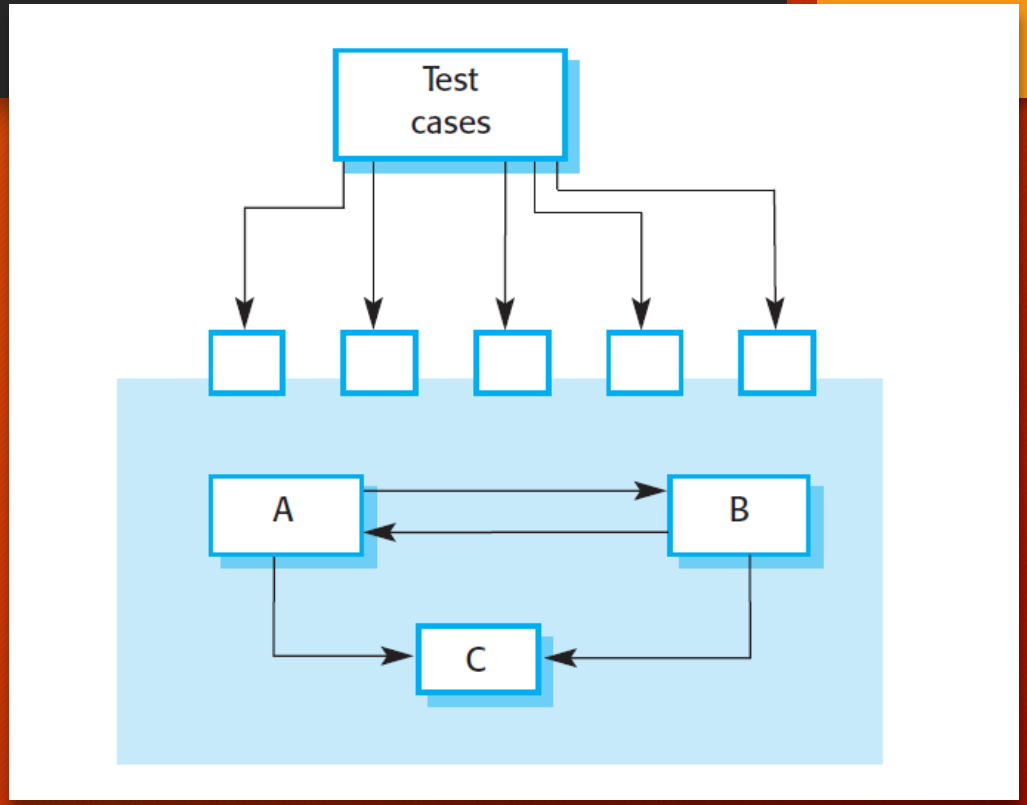
- Existen lineamientos que te ayudarán a elegir casos de prueba más efectivos para la detección de errores.
- Por ejemplo, cuando se prueban programas con secuencias, arreglos o listas, los lineamientos que pueden ayudar a revelar defectos incluyen:
  - Prueba el software con arreglos que contienen un solo valor único. Este enfoque puede revelar errores en programas que suponen que los arreglos consisten en múltiples valores.
  - Usar arreglos de diferentes tamaños en pruebas distintas para minimizar las oportunidades de que un programa defectuoso genere salidas correctas de forma accidental.
  - Diseña pruebas de forma que se acceda a los elementos primero, medio y último del arreglo. Esto revelará problemas en las fronteras de la partición.

Guidelines para *unit testing*

- En general:
  - Elegir inputs que fuercen al sistema a generar todos los mensajes de error.
  - Repetir el mismo input (o series de inputs) varias veces.
  - Forzar output’s inválidos.
  - Forzar cómputos muy grandes o pequeños.



Component testing



System testing

Se diferencia del *component testing* en:

Se testean componentes que ya hayan sido creados y testeados y sus interacciones con nuevos componentes.	Se testean componentes que hayan sido desarrollados por distintas personas.

Su objetivo es revelar bugs que solamente son visibles al momento de ver el sistema completo y las interacciones entre sus partes.

Se pueden diseñar tests basados en los casos de uso/historias de usuario.

Release Testing

El objetivo principal del *release testing* es convencer al cliente del sistema que está suficientemente bueno para su uso. Es decir, que cumpla con sus funcionalidades, rendimiento y no tiene fallas.

Generalmente se considera como **un testing de caja negra *(black-box testing)***. Es decir, el sistema se considera como una caja negra donde su comportamiento solo se determina al estudiar sus inputs y outputs.

También se le llama testing funcional, porque se encarga de funcionalidades y no implementación del software.

Se puede separar en tres perspectivas:

- Testing basado en requerimientos
- Testing basado en escenarios
- *Performance Testing*

User Testing

- Hay tres tipos de *user testing*:
  - **Alpha testing:** Donde un grupo selecto de usuarios del sistema trabajan con el equipo de desarrollo para versiones preliminares de este. En desarrollos ágiles, un producto owner debería adquirir este rol.
  - **Beta testing:** Donde una versión del sistema se entrega a un grupo más grande de usuarios para permitirles experimentar y levantar problemas que puedan ir descubriendo. Se usa entonces para descubrir problemas de interacción entre el sistema y su ambiente operacional. Se ocupa mucho cuando el sistema debe funcionar en múltiples ambientes de diversa naturaleza. Además se considera una herramienta de marketing.
  - **Acceptance testing:** Donde un grupo de usuarios (clientes) deciden si está listo o no para ser lanzado.

Casos de Prueba

- Un caso de prueba es un conjunto de condiciones y variables bajo las cuales un probador determinará si un sistema, software, o una de sus características funciona según lo especificado.
- Los casos de prueba son fundamentales para asegurar la calidad y funcionalidad de un sistema.
- Elementos de un Caso de Prueba
  - Identificador del Caso de Prueba (ID): Un identificador único para cada caso de prueba.
  - Título/Nombre del Caso de Prueba: Una breve descripción del caso de prueba.
  - Descripción: Explica en detalle qué se va a probar y por qué.
  - Precondiciones: Condiciones que deben cumplirse antes de ejecutar el caso de prueba.
  - Datos de Entrada (Inputs): Los valores o parámetros que se utilizarán en el test.
  - Pasos para la Ejecución: Instrucciones detalladas sobre cómo ejecutar el test.
  - Resultado Esperado: La salida esperada si el sistema funciona correctamente.
  - Resultado Real (opcional): Lo que realmente ocurrió durante la ejecución del test.
  - Estado: Indica si el test pasó o falló.

Ejemplo

- **ID:** TC01
- **Título:** Verificar la funcionalidad de la suma de dos números positivos
- **Descripción:** Este caso de prueba verifica si la función de suma retorna el resultado correcto al sumar dos números positivos.
- **Precondiciones:** El sistema debe estar operativo y la función suma debe estar definida.
- **Datos de Entrada (Inputs):**
  a = 1
  b = 2
- **Pasos para la Ejecución:**
  1. Llamar a la función suma con los parámetros a y b.
  2. Registrar el resultado devuelto por la función.
- **Resultado Esperado:** El resultado devuelto debe ser 3.
- **Resultado Real:** (n/a - llenado después de la ejecución)
- **Estado:** (n/a - llenado después de la ejecución)

Ejemplo

- ID: CT02
- Título: Verificar la integración del módulo de autenticación con el módulo de gestión de usuarios
- Descripción: Este caso de prueba verifica si el módulo de autenticación funciona correctamente con el módulo de gestión de usuarios.
- Precondiciones: El sistema debe tener los módulos de autenticación y gestión de usuarios implementados.
- Datos de Entrada (Inputs):
  username = "user1"
  password = "pass123"
- Pasos para la Ejecución:
  - Registrar un nuevo usuario con el username y password.
  - Intentar iniciar sesión con las mismas credenciales.
- Resultado Esperado: El usuario debe ser autenticado correctamente.
- Resultado Real: (n/a - llenado después de la ejecución)
- Estado: (n/a - llenado después de la ejecución)

Otros Ejemplos

- Test de sistema: Verificar el flujo completo de compra en una tienda en línea
- Test de versiones: Verificar si el sistema puede manejar una carga elevada de usuarios (1M) y operaciones simultáneas.
- Alfa: Verificar la funcionalidad básica del sistema en un entorno controlado de 10 usuarios reales invitados a participar.
- Beta: Verificar si el sistema funciona correctamente en un entorno real con un grupo más amplio de usuarios, mediante la liberación de una versión inicial del software.
- Aceptación: Verificar si el sistema cumple con el 80% de los requisitos especificados por el cliente.

Código del Caso de Prueba en Python

- El bloque final muestra un modo sencillo de ejecutar las pruebas. unittest.main() que proporciona un reporte básico de la ejecución de la o las pruebas.
  - Ran 3 tests in 0.000s
  - OK
- Si se le pasa una opción -v al script de prueba, se establecerá un nivel mayor de detalle del proceso de unittest.main() y se obtendrá la siguiente salida:
  - test_suma_positivos (__main__.TestSuma.test_ suma_positivos) ... ok
  - test_suma_negativos (__main__.TestSuma.test_ suma_negativos) ... ok
  - test_suma_cero (__main__.TestSuma.test_suma_cero) ... ok
  - --
  - Ran 3 tests in 0.001s
  - OK

Ejemplo

```python
import unittest

def suma(a, b):
    return a + b

class TestSuma(unittest.TestCase):
    def test_suma_positivos(self):
        self.assertEqual(suma(5, 7), 12)

    def test_suma_negativos(self):
        self.assertEqual(suma(-1, -2), -3)

    def test_suma_cero(self):
        self.assertEqual(suma(0, 0), 0)

if __name__ == '__main__':
    unittest.main()
```

Actividad 4

- (0,6 puntos) Defina tres casos de prueba (test unitarios) para la clase Evento o la clase Reserva del sistema de planificación de eventos para los profesores. Utilice el siguiente formato:
  - Título: Nombre descriptivo del caso de prueba.
  - Descripción: Breve descripción del propósito de la prueba.
  - Precondiciones: Condiciones que deben cumplirse antes de ejecutar el caso de prueba.
  - Datos de Entrada: Datos específicos que se usarán en la prueba.
  - Pasos para la Ejecución: Lista de pasos necesarios para ejecutar el caso de prueba.
  - Resultado Esperado: Resultado que debería producir el sistema si funciona correctamente.