[75.09] Análisis de la Información

María Inés Parnisari

30 de noviembre de 2013

Índice

1 Ingeniería de software

Ingeniería del software: enfoque sistemático, disciplinado y cuantificable aplicado al desarrollo, la operación y el mantenimiento del software.

Para trabajar, se necesitan metodologías que requieren:

- notación,
- herramientas automatizadas,
- proceso de desarrollo.

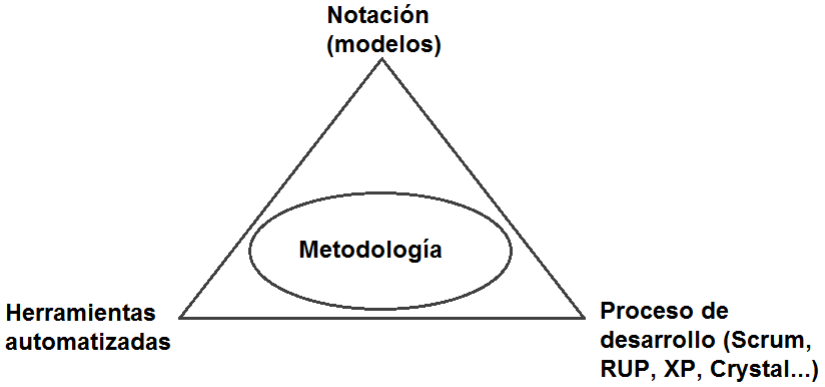


Figura 1: Pilares de las metodologías

Enfoques de la ingeniería de software:

1. Estructurado
2. Orientado a Objetos:
   a) **Objeto**: agrupa atributos y métodos en una sola entidad
   b) **Encapsulamiento**: "ocultar" un atributo de nuestra clase hacia el exterior
   c) **Abstracción**: definición de clases
   d) **Polimorfismo**: posibilidad de tener una misma función que se comporte de distinta manera dependiendo de los parámetros que le entreguemos. Declarar una función pero no implementarla, con el objetivo de que si o si las clases que la heredan tienen que implementar este método.
   e) **Herencia**: podemos "heredar" una clase más general, a otra menos general

2 Modelos

Modelo: representación simplificada de la realidad.

¿Para qué se modeliza?

1. Para comprender el problema

2. Para visualizar el problema
3. Para minimizar los riesgos de tiempo, de dinero y funcionales
4. Para comunicar ideas entre personas

	Ingeniería Civil	**Ingeniería de Software**
Captura de requisitos	¿Pileta?, ¿cantidad de habitaciones?, ¿garage?..	Encuestas, cuestionarios...
Análisis (QUÉ)	Planos esquemáticos, maquetas..	Modelos de análisis (modelos de casos de uso, de comportamiento, de negocio...)
Diseño (CÓMO)	Planos estructurales, eléctricos, hidráulicos...	Modelos de diseño (modelos de caso de uso, de componentes, de despliegue...)
Construcción		
Pruebas		Unitarias, de integración, de sistema, de usuarios
Entrega		
Mantenimiento	Correctivo y evolutivo	Correctivo y evolutivo

Cuadro 1: Paralelismo entre ingeniería civil y de software

3 Modelos en UML

UML (*Unified Modeling Language*): lenguaje para visualizar, especificar, construir y documentar sistemas bajo el paradigma de orientación a objetos. Es un lenguaje basado en modelos visuales que abarcan todo el ciclo de vida. Estas metodologías usan distintos modelos.

Modelos orientados a objetos:

1. Modelos de Requerimientos (*qué*)
2. Modelos de Diseño (*cómo*)

Modelos de Requerimientos		**Diagrama/s**	
1	Objetivos		
2	Negocio	Actividad	
3	Comportamiento	Casos de Uso	
4	Conceptual	Clases	Objetos
5	Interacción entre objetos	Secuencia	Colaboración
6	Estado de objetos	Transición de estado de objetos	
7	Componentes	Componentes	
8	Despliegue	Despliegue	

Cuadro 2: Modelos

3.1 Modelo de Objetivos

Objetivo: define en un alto nivel de abstracción, utilizando una breve descripción, y en pocas líneas, qué vamos a automatizar con el software a construir.

Alcance: define qué procesos estarán alcanzados, y cuáles no. En un principio es difuso y luego se va refinando a medida que avanza el análisis.

Hipótesis: sentencias que establecen determinadas premisas y/o restricciones sobre las cuales estará basado el sistema. Debe contemplar faltantes, ambigüedades y restricciones regulatorias.

3.2 Modelo de Negocio

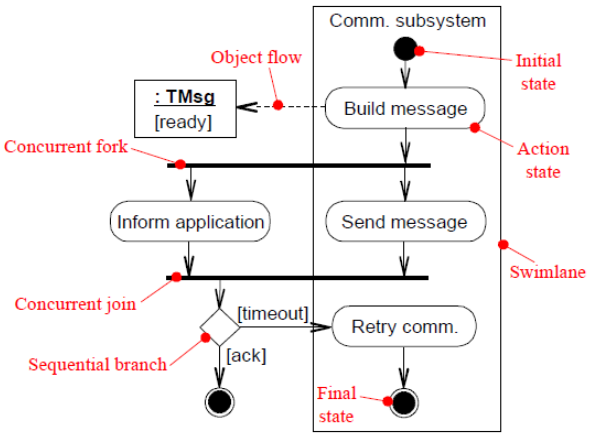
Objetivo: Entender la organización y los **procesos de negocios** para los cuales vamos a desarrollar el sistema. Los procesos de negocio son los que voy a querer "automatizar" con el sistema.

Ejemplos:

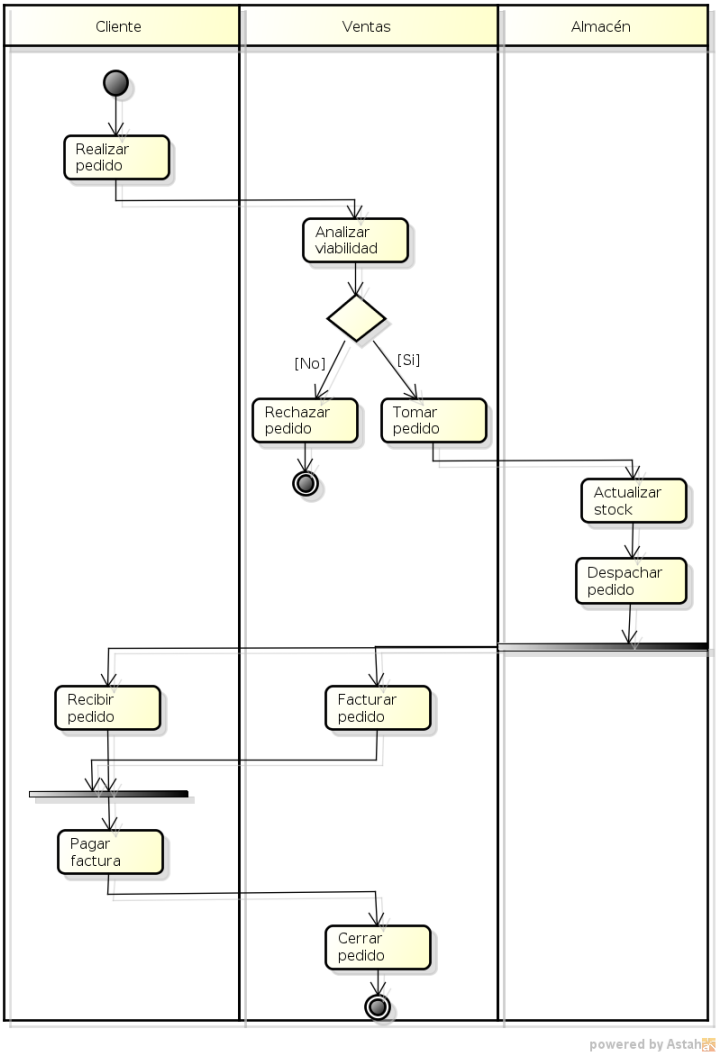
- Gestión de facturación
- Administración de pedidos
- Gestión de almacenes

3.2.1 Diagrama de actividad

Diagrama de actividad: grafo dirigido compuesto por nodos (representan actividades) y aristas (representan transición entre actividades).



(a) Componentes



(b) Ejemplo

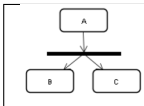
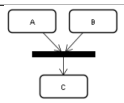
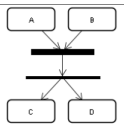
Figura 2: Diagramas de actividad

Para un diagrama de actividad hay que definir **escenarios** para los procesos de negocios.

- *Nivel Global*: visión global de los procesos. Alto nivel de abstracción.
- *Nivel Intermedio*: una parte de los procesos de negocio.
- *Método*: un proceso detallado.

Componentes:

- ***Swimming lanes***
- **Actividades**: Verbo + Objeto. Describir *qué* se hace (no el *cómo*). Las actividades tienen que hacer hincapié tal como son vistas por quienes interactúan con el sistema. Cada actividad se desglosa en 2 **tareas** como mínimo. Se debe evitar que representen algo físico (entregar o recibir paquetes) porque no agregan funcionalidad al sistema.
- **Símbolo de inicio *(uno solo)* y final**
- **Nodos de decisión**
- **Barras de sincronización, transición de actividades**: secuenciales y sincronizadas.

	Para que comiencen las actividades B y C, tiene que haber terminado la actividad A.
	Para que comience la actividad C, tienen que haber terminado las actividades A y B.
	Para que comiencen las actividades C y D, tienen que haber terminado las actividades A y B.

3.3 Modelo de Comportamiento

Objetivo: modelizar el comportamiento funcional del sistema bajo estudio.
Componentes:

1. **Casos de uso**: funciones requeridas por el sistema.
2. **Actores**: algo o alguien que interactúa con nuestro sistema pero que no es parte del mismo.
3. **Diagrama de casos de uso**: modeliza relaciones entre actores y casos de uso.

3.3.1 Actores

Cuando uno define un **actor** debe pensar en cuál es su **rol** con el sistema (cómo interactúa con el mismo).
Reglas prácticas para identificar actores:

1. ¿Quién está interesado en cierto requerimiento?

2. ¿En qué lugar de la organización se usará el sistema?
3. ¿Quién se beneficia usando el sistema?
4. ¿Quién suministrará, usará o eliminará del sistema tal información?
5. ¿Quién mantiene los datos del sistema?
6. ¿Juega una persona diferentes roles?
7. ¿Juegan varias personas el mismo rol? *Son el mismo actor.*
8. ¿Interactúa el sistema con otros sistemas?

Documentación de actores: contiene una breve descripción de cada actor, la cual debe identificar el **rol** que juega el actor al interactuar con el sistema.

Tipos de actores	
¿Dispara un caso de uso?	¿Es un actor físico?
Si: **primario**	Si: **físico**
No: **secundario**	No: **temporal**

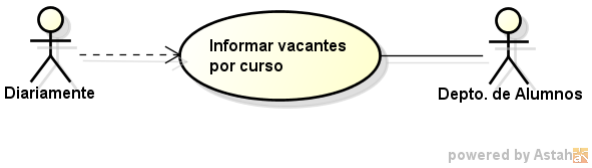


Figura 3: Actor temporal

Ultimate Primary Actor: depende del modelo en el que estamos.

- Modelo Conceptual (*qué*): actor que tiene la necesidad en última instancia.
- Modelo de Diseño (*cómo*): actor que interacciona en forma directa con el sistema.

3.3.2 Casos de uso

Caso de uso: modeliza la funcionalidad provista por el sistema, es decir los **servicios** provistos a un actor del sistema. Son una descripción de un conjunto de **secuencias de acciones**, incluyendo variaciones, que ejecuta un sistema para producir un resultado observable que sea de valor para un actor.

Reglas prácticas para identificar casos de uso:

1. ¿Cuáles son las tareas de cada actor?
2. Un actor determinado, ¿creará, modificará, eliminará o consultará información del sistema?
3. ¿Qué casos de uso crearán, modificarán, eliminarán o consultarán el sistema?
4. ¿Algún actor necesita informar al sistema de cambios?
5. ¿Necesita algún actor ser informado de cambios que ocurran?

6. ¿Qué casos de uso mantendrán actualizado el sistema?
7. ¿Todos los requerimientos funcionales están cubiertos en los casos de uso?

Un buen caso de uso posee:

- **Completitud temporal**: el caso de uso es completo de principio a fin. Ejemplo: Seleccionar curso / Registrar curso / Informar al sistema de facturación $\Longrightarrow$ Registrar curso
- **Completitud funcional**: cuando hay casos de uso que son comenzados por el mismo actor y además tratan con la misma entidad del sistema. Ejemplo: Crear / Modificar / Eliminar cursos $\Longrightarrow$ Mantener cursos.

Documentación de casos de uso:

- **Descripción**: describe el propósito del mismo en pocas sentencias, con una definición de alto nivel de la funcionalidad en cuestión provista por el sistema.
- **Flujo de eventos**: descripción de los eventos necesarios para realizar el comportamiento requerido del caso de uso. Debe hacer hincapié en *qué* debe hacer el sistema, no en el *cómo*.
  - **Precondiciones**: enumera qué casos de uso y/o subflujos deben haber sido ejecutados previo a la ejecución del caso de uso en cuestión.
  - **Flujo principal ("normal")**: debe destacar:
    1. Cuándo comienza el caso de uso
    2. Qué interacciones se dan entre el sistema y los actores
    3. La secuencia normal de eventos
    4. La descripción de flujos de excepción
    5. Cuándo termina el caso de uso
  - **Subflujos**
  - **Flujos de excepción**

Use Case: Procesar Transacciones	
Descripción: procesa transacciones pendientes contra la cuenta bancaria del cliente	
Actores participantes: Cajero	
Pre-condiciones: existen transacciones pendientes de procesamiento	
Flujos	
Flujo Principal	
1	El actor Cajero inicia (o indica que desea iniciar) el procesamiento de transacciones pendientes
2	El sistema ordena las transacciones de modo que todas las correspondientes a una cuenta en particular se encuentren juntas y, dentro de cada grupo, se procesan primero los **depósitos** para evitar saldos negativos innecesarios.
3	Por cada cuenta: **{Determinar Cuenta de Cliente}** 3.1 El sistema determina la cuenta del cliente a la cual se aplicará la transacción. 3.2 Por cada transacción: **{Aplicar Transacción}** 3.2,1 El sistema aplica la transacción a la cuenta. Las transacciones de **depósito** incrementan el saldo de la cuenta y las de **extracción** lo disminuyen. **{Registrar Transacción}** 3.2.2 El sistema registra los datos de la transacción en una bitácora para dejar evidencia de aquella. 3.2.3 El sistema marca la transacción como completada **{Resumir Transacciones}** 3.3.3 Cuando se procesaron todas las transacciones para una cuenta determinada, el sistema genera una transacción resumen.
4	Cuando se procesaron todas las transacciones, termina el caso de uso.
Flujos Alternativos	
A1	***Cuenta Inexistente*** En **{Determinar Cuenta de Cliente}**, si la cuenta no existe:
1	El sistema marca todas las transacciones de la cuenta como **suspendidas.**
2	El procesamiento continúa en el paso siguiente a **{Determinar Cuenta de Cliente}**
A2	***Manejo de sobregiro sin autorización*** En **{Aplicar Transacción}**, si la transacción genera sobregiro (saldo negativo de la cuenta) y la cuenta no tiene autorización para ello:
1	El sistema aplica la transacción y la marca como "sobregiro".
2	El sistema aplica intereses de sobregiro a la cuenta.
3	El procesamiento continua en **{Registrar Transacción}.**
A3	***Manejo de sobregiro con autorización*** En **{Aplicar Transacción}**, si la transacción genera sobregiro (saldo negativo de la cuenta) y la cuenta tiene autorización para ello:
1	Si la transacción no genera un sobregiro que supera el límite autorizado para el cliente, el sistema aplica la transacción.
2	Si la transacción genera un sobregiro que supera el límite autorizado para el cliente, el sistema aplica la transacción, la marca como "sobregiro" y aplica intereses de sobregiro a la cuenta por la cantidad que supera el límite autorizado.
3	El procesamiento continúa en **{Registrar Transacción}.**
Post-condiciones: todas las transacciones procesadas	

Figura 4: Ejemplo de caso de uso.

3.3.3 Diagrama de casos de uso

Todo caso de uso debe tener al menos un actor primario que lo inicie.
Tipos de relaciones:

1. **Asociación** entre actores y casos de uso.

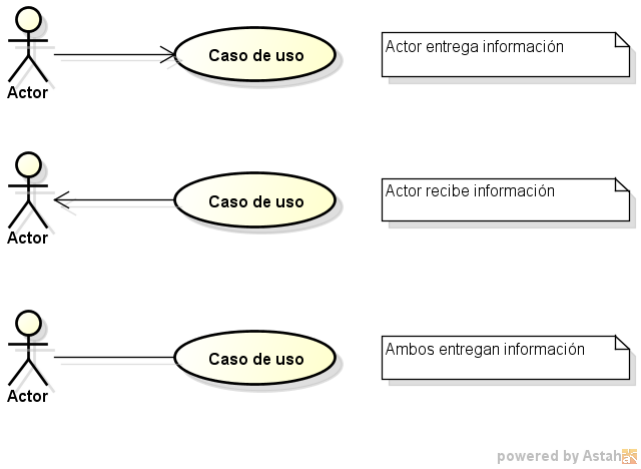
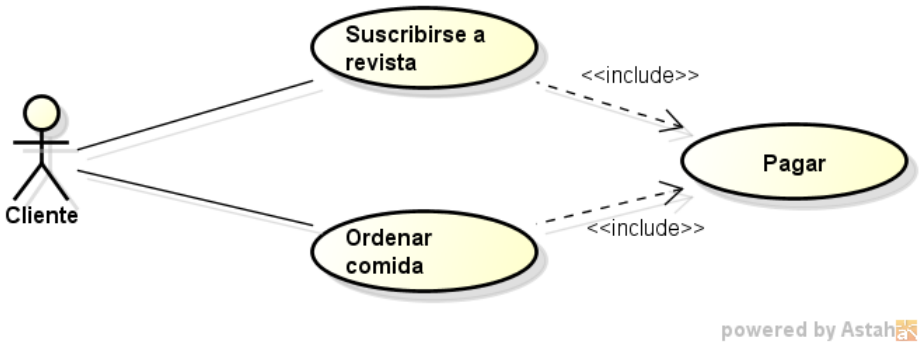


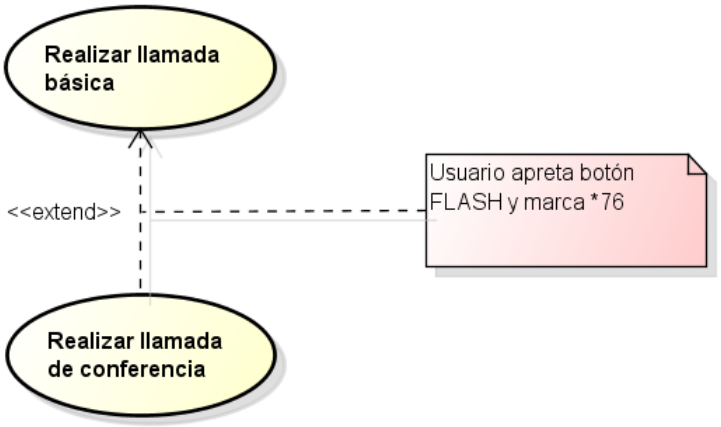
Figura 5: Tipos de asociación

2. **Dependencia** entre casos de uso.

   a) **Inclusión** (***include***): cuando dos o más casos de uso comparten parte de su funcionalidad, es conveniente factorizar la parte común y colocarla en un caso de uso separado a través de una relación de inclusión.

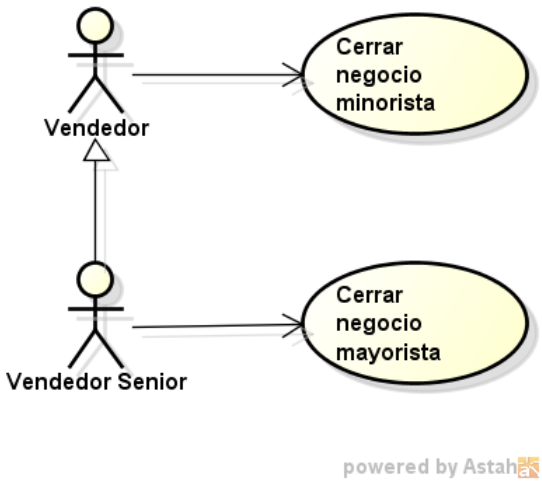


   b) **Extensión** (***extend***): un caso de uso agrega pasos condicionalmente a otro caso de uso *base*. La *etiqueta* que se coloca abajo de `<<extend>>` hace referencia a la condición de ejecución del caso extendido. El caso base no debe conocer al extendido.



3. **Jerarquía** entre actores, y entre casos de uso.

   La jerarquía inferior hereda *todos* los comportamientos de la jerarquía superior.



3.4 Modelo Conceptual

3.4.1 Diagrama de clases conceptuales

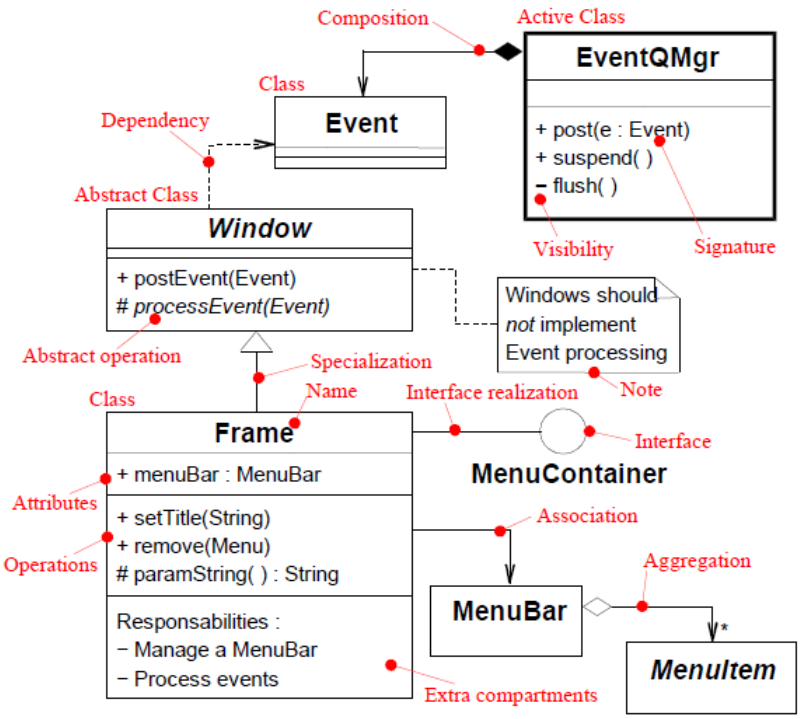
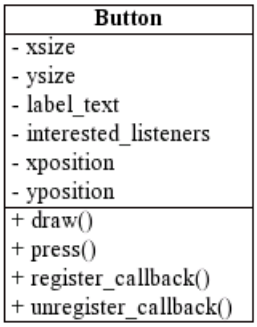


Figura 6: Diagrama de clases

Clase: representa un bloque constructivo de cualquier sistema orientado a objetos. Una clase conceptual es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.

- **Nombre de clase**
- **Atributos**: representan el estado interno del objeto.
- **Operaciones**[1]: servicios que brinda la clase.
  - \+ Público
  - \# Protegido
  - \- Privado
- **Responsabilidades**: contrato u obligación que debe prestar la clase. Como mínimo 1, como máximo 3.

[1] Una **operación** aún no está implementada. Cuando sí lo está, se llama **método**.



Reglas de identificación de clases conceptuales:

1. Regla práctica

   a) Identificar clases candidatas prestando atención a los **sustantivos** del relevamiento. Identificar las responsabilidades de cada clase.

   b) Diagrama de clases preliminar: listar las clases.

   c) Diagrama de relaciones entre clases.

2. CRC (*Collaboration Responsibility Cards*)

   a) Armar las tarjetas.

Nombre de la clase	
Superclase	
Subclases	
Responsabilidades	Colaboraciones (lista de clases con las que interactúa para cumplir sus responsabilidades)

Cuadro 3: Tarjeta CRC

   b) Armar el diagrama de tarjetas CRC.

Relaciones entre clases:

1. Relación de **Asociación**: relación estructural entre instancias de clases.

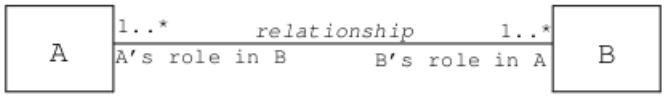


Figura 7: Asociación

   a) **Simple**

1) **Reflexiva**: relación entre instancias de una misma clase.

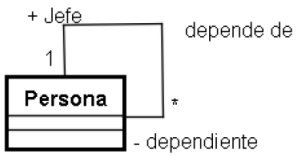


Figura 8: Relación reflexiva

2) **Clases de Asociación**: se utiliza cuando se necesita una clase para definir una relación.

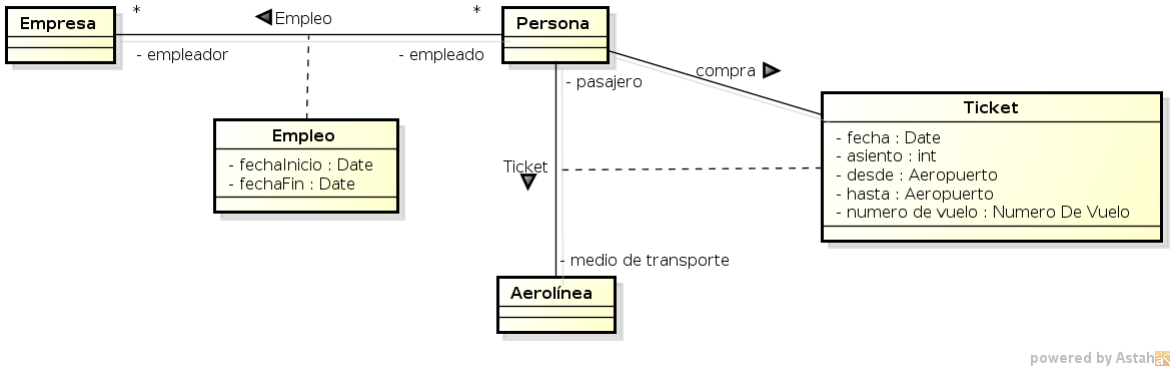


Figura 9: Clase de asociación

b) **Agregación**: si el contenedor desaparece, los contenidos puede que también, pero no siempre.

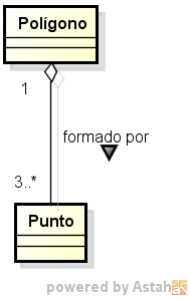


Figura 10: Relación de agregación

c) **Composición**: si el contenedor desaparece, los contenidos también.

```
1 class Departamento {
2   private string nombre;
3 };
4
5 class Empresa {
6   private Departamento depto_rrhh;
7
8   public Empresa() {
9     depto_rrhh = new Departamento();
```

```
  }
  public ~Empresa() {
    delete depto_rrhh;
  }
};
```

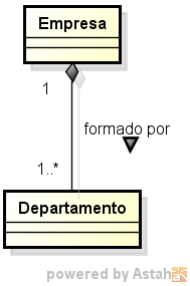


Figura 11: Relación de composición

```
class Company {
  private Employee[] peon = new Employee[n];

  public void give_me_a_raise (Employee e) {
  }
};

class Employee {
  private Company employer;
  private Employee boss;
  private Vector flunkies = new Vector();

  public void you_re_fired() {
  }
};
```

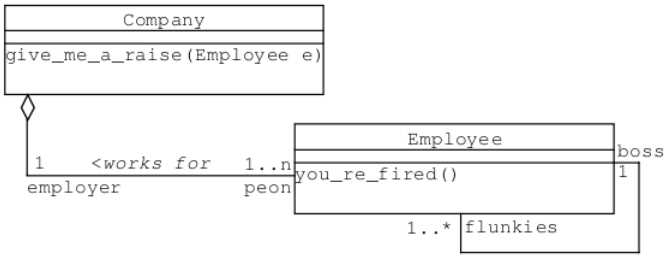


Figura 12: Ejemplo

2. Relación de **Generalización** / **Especialización**: Jerarquía de clases. Herencia. Representa relaciones de tipo “es un”.

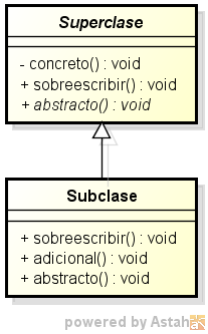
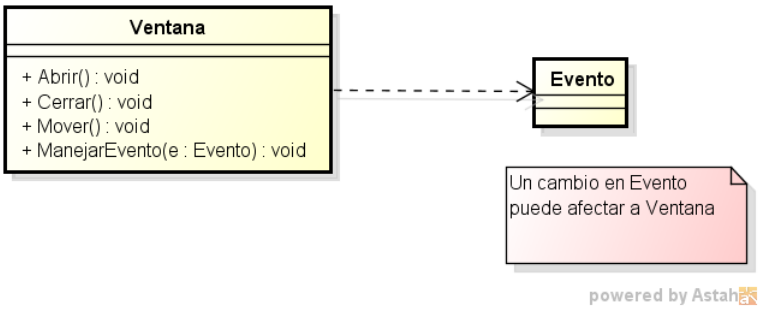


Figura 13: Herencia.

3. Relación de **Uso**: dependencia entre clases.



4. Relación de **Realización**.

   Interfaz: colección de operaciones que se utiliza para especificar un servicio de una clase. Establece un comportamiento deseado de una abstracción independiente de la implementación de dicha abstracción. La interfaz no tiene atributos y tampoco tiene instancias.

   La clase *realiza* (*implementa*) los servicios que declara la interfaz.

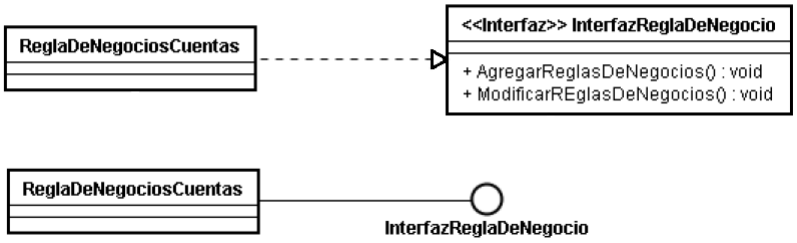


Figura 14: Relación de realización.

3.5 Modelo de Interacción entre Objetos

Se utiliza para modelar cómo se comunican entre sí los objetos para un escenario específico.

- Diagrama de secuencia: cronológico.
- Diagrama de colaboración: espacial.

3.5.1 Diagrama de Secuencia

- Las cajas superiores representan objetos, no clases.
- Las líneas verticales punteadas representan la existencia del objeto (**línea de vida**).
- Los rectángulos sobre la línea de vida indican que el objeto está activo.
- El objeto que **envía** el mensaje debe tener algún tipo de asociación con el objeto que **recibe**.
- Para crear un objeto: el objeto creado aparece al final del mensaje de creación.
- Para un mensaje condicional, la **condición** se expresa entre corchetes [] antes del mensaje.

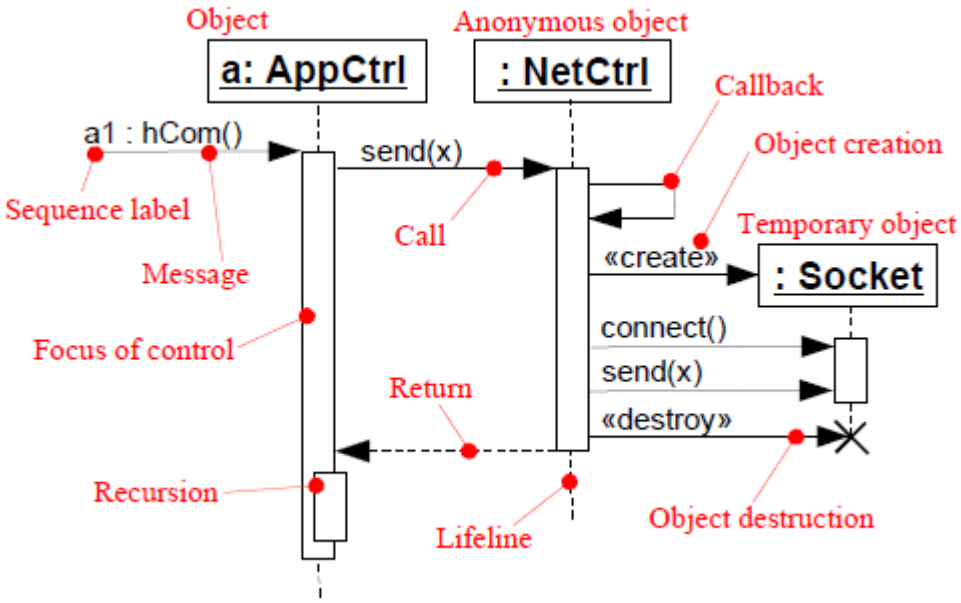
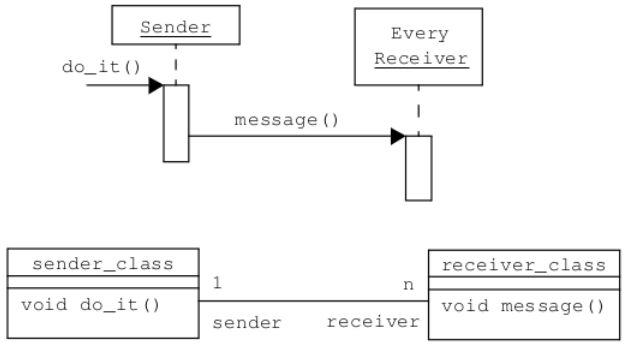


Figura 15: Diagrama de secuencia




```
class sender_class
{
   receiver_class receiver[n];
   public do_it() {
      for(int i = 0; i < n; ++i)
         receiver[i].message();
   }
}
```

Figura 16: *Loops* en un diagrama de secuencia.

3.5.2 Diagrama de Colaboración

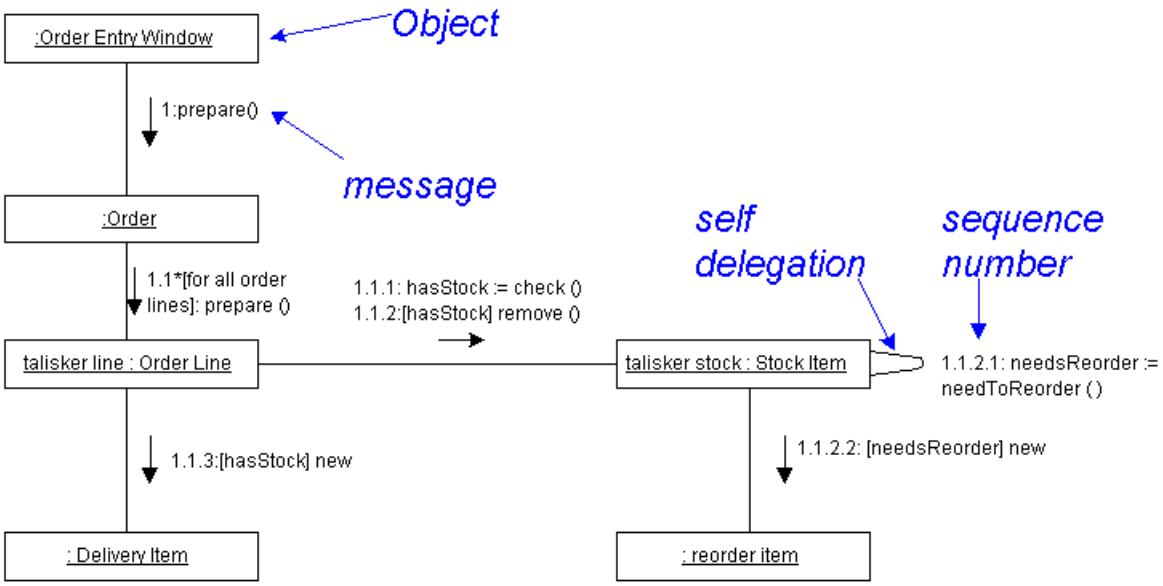


Figura 17: Diagrama de colaboración

3.6 Modelo de Estado de Objetos

Objetivo: modelizar cómo va cambiando el estado de un objeto específico con el tiempo.

3.6.1 Diagrama de transición de estado de objetos

Grafo dirigido donde los **nodos** son **estados** de un objeto en un instante determinado, y las **aristas** son las **transiciones** entre los estados.

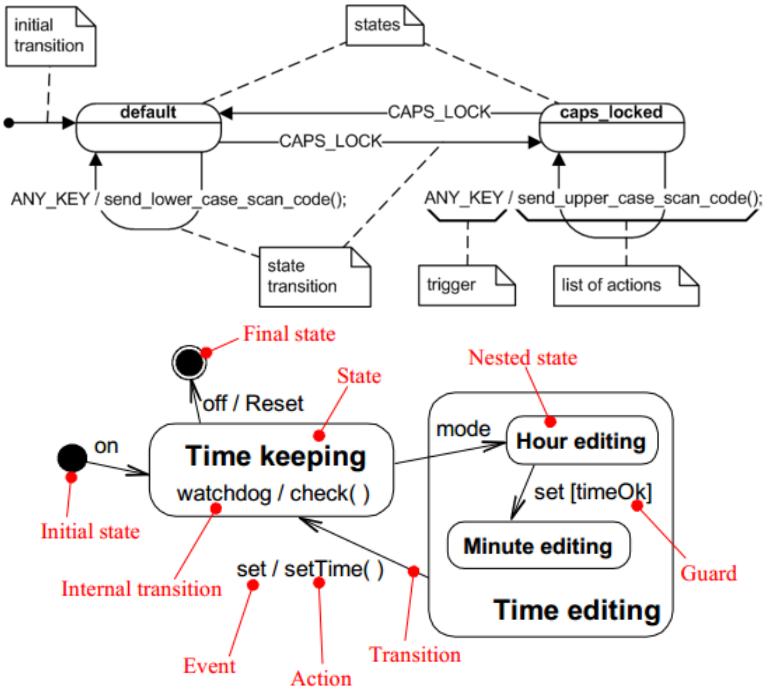


Figura 18: Diagrama de transición de estado de objetos

- **Estado**:
  - `entry`: se ejecuta siempre al entrar al estado.
  - `do`: se ejecuta mientras se está en el estado.
  - `exit`: se ejecuta siempre al salir del estado.
- **Transición**: `evento [condición] / acción`
  - Evento: qué inicia la transición
  - Condición: cuándo se inicia la transición
  - Acción: se ejecuta.
  - Transición temporizada: se representa con una línea punteada.
- **Transición inicial**: se origina del círculo relleno y especifica el estado por defecto cuando inicia el sistema. Todos los diagramas deben tener esta transición, sin etiqueta puesto que no es causada por un evento. Sí puede tener acciones asociadas.

- **Transición final**.

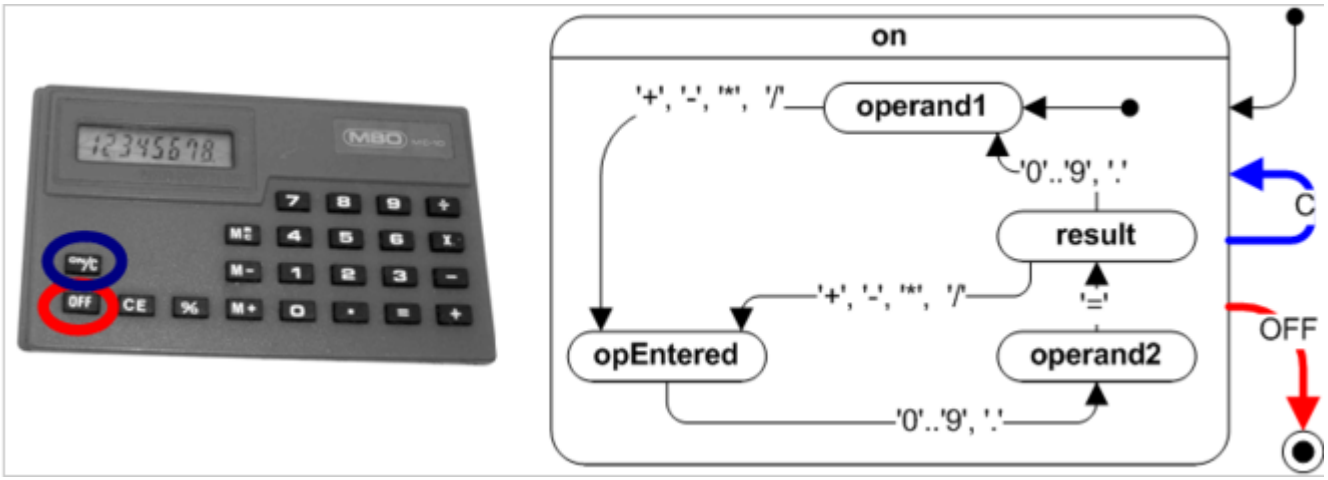


Figura 19: Generalización de estados

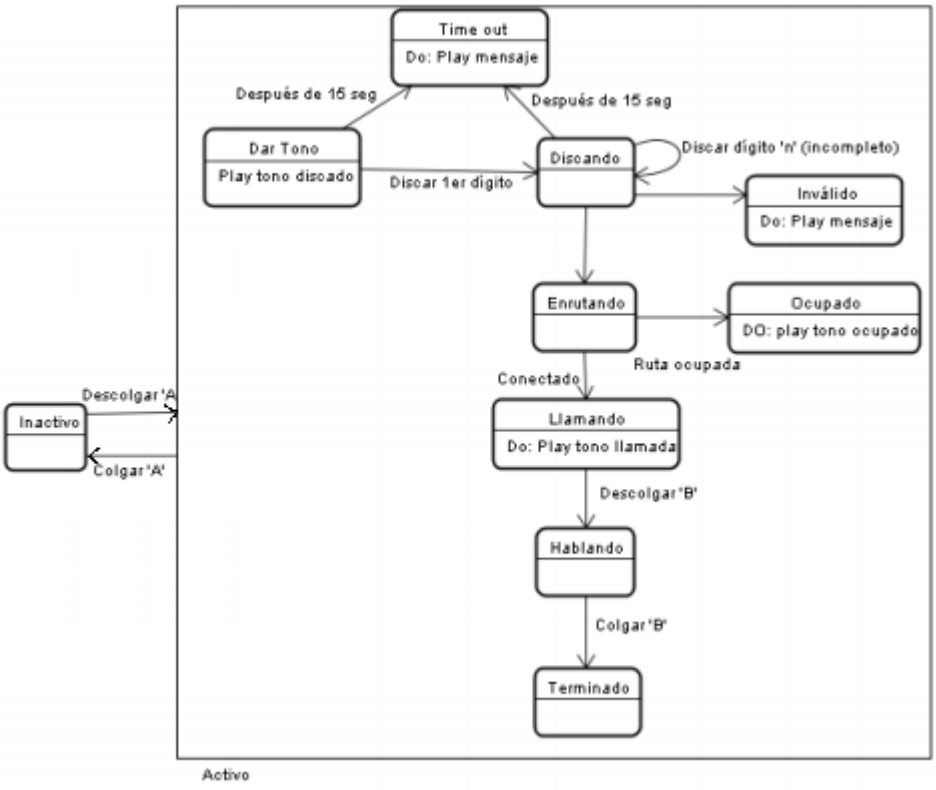
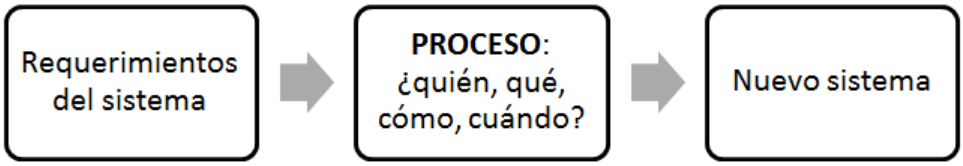


Figura 20: Diagrama de transición de estados

4 Proceso de desarrollo

Proceso de desarrollo de software: forma disciplinada de asignar tareas y responsabilidades en un proyecto de desarrollo de software, para cumplir los objetivos:

1. Software de calidad: funcional, confiable, veloz
2. Cumplir con el plazo y el costo establecido



El encadenamiento de las etapas se conoce como **ciclo de vida**.

4.1 Mejores prácticas en el desarrollo de software

Son las condiciones necesarias para el no fracaso de un proyecto.

1. **Proceso iterativo e incremental.**
   a) Ciclo de vida en cascada

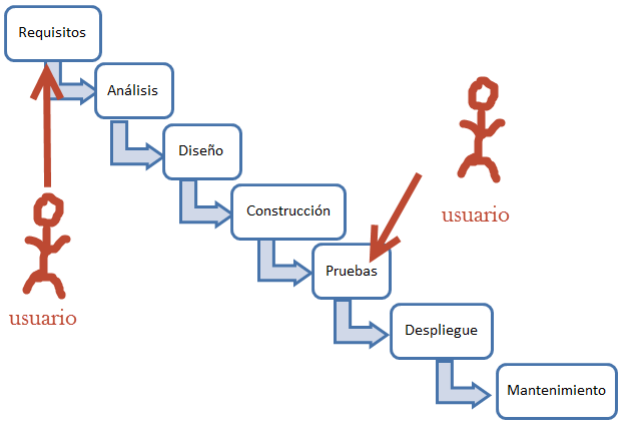


Figura 21: Ciclo en cascada.

Ventajas	Desventajas
Es simple de llevar a cabo.	*El usuario no acompaña el proceso.*
Puede servir en proyectos pequeños.	No sirve en proyectos medianos y grandes.
	No se puede paralelizar.
	No hay entregables intermedios. No se pueden detectar problemas en etapas tempranas. Si falla la captura de requisitos, hay pérdida de tiempo y dinero.

b) Ciclos de vida iterativos e incrementales

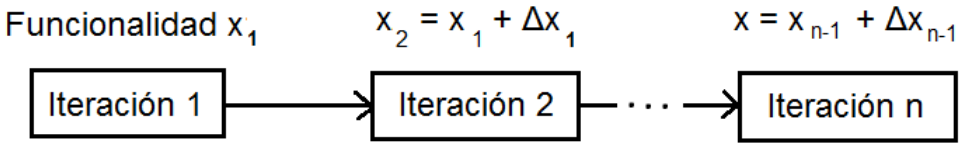


Figura 22: Ciclo iterativo

Ventajas
Permite un entendimiento incremental del problema
Permite acomodar fácilmente los cambios
Ayuda a mitigar riesgos
El equipo aprende en el camino
Mayor calidad

- Se debe comenzar por las funcionalidades de mayor riesgo. Los riesgos son más manejables cuando menor sea el ciclo de vida
- *El usuario acompaña el proceso* y proporciona retroalimentación
- Cada **iteración** termina un con *release* ejecutable. Cada iteración está compuesta por un miniciclo en cascada (3 a 6 semanas). En cada iteración se incrementa la funcionalidad del sistema.

2. **Proceso dirigido por casos de uso**.

   Los casos de uso se utilizan a lo largo de todo el ciclo de vida del proyecto, tanto para capturar requisitos (*definición de casos de uso*), servir de guía en la implementación (*realización de casos de uso*), y en las pruebas (*verificación de cumplimiento de los casos de uso*).

3. **Arquitecturas basadas en componentes**.

   El proceso se basa en diseñar tempranamente una arquitectura base ejecutable. Condiciones:

   a) Flexibilidad

   b) Fácil de modificar

   c) Se entiende intuitivamente

   d) Promover reutilización de componentes

Componentes: módulos no triviales del sistema, que cumplen una función determinada.

4. **Modelización visual del software**.

   Uso intensivo de UML. Ventajas:

   a) Permite visualizar cómo interactúan todos los elementos del sistema,

   b) Permite mantener consistencia entre el diseño y la implementación,

   c) Promueve comunicación no ambigua.

5. **Verificación de la calidad del software**.

   El aseguramiento de la calidad debe ser parte del proceso de desarrollo y **no** la responsabilidad de un grupo independiente. Se deben utilizar criterios objetivos. Se mide en cuanto a confiabilidad, funcionalidad, performance, y concordancia con los requerimientos.

6. **Gestión del cambio**.

   Los cambios son inevitables por la dinámica del negocio cambiante.

   a) Evaluación del impacto del cambio:
   ¿cómo influye en tiempo, dinero y riesgos?

   b) Evaluación de la oportunidad del cambio:
   ¿en qué momento conviene introducir el cambio? (evaluar relación costo-beneficio)

5 RUP (*Rational Unified Process*)

RUP:

- Proceso de desarrollo: forma disciplinada de asignar tareas y responsabilidades en un proyecto de desarrollo de software, con un tiempo y costo predecible.
- Proceso de producto
- Apoya la productividad del equipo, utilizando una *knowledge base* común a todos los miembros.
- Crear y mantener modelos
- Guía para usar UML efectivamente
- Apoyada por herramientas
- Proceso configurable, que se adapta a pequeños y grandes equipos
- Captura las mejores prácticas de desarrollo

Fases según RUP:

1. **Inicio**
2. **Elaboración**
3. **Construcción**

   a) Componentes del proceso

   b) Componentes de soporte de proceso:

1) Administración de proyecto
2) Gestión de cambio
3) Gestión de entornos y herramientas de desarrollo

4. **Transición**

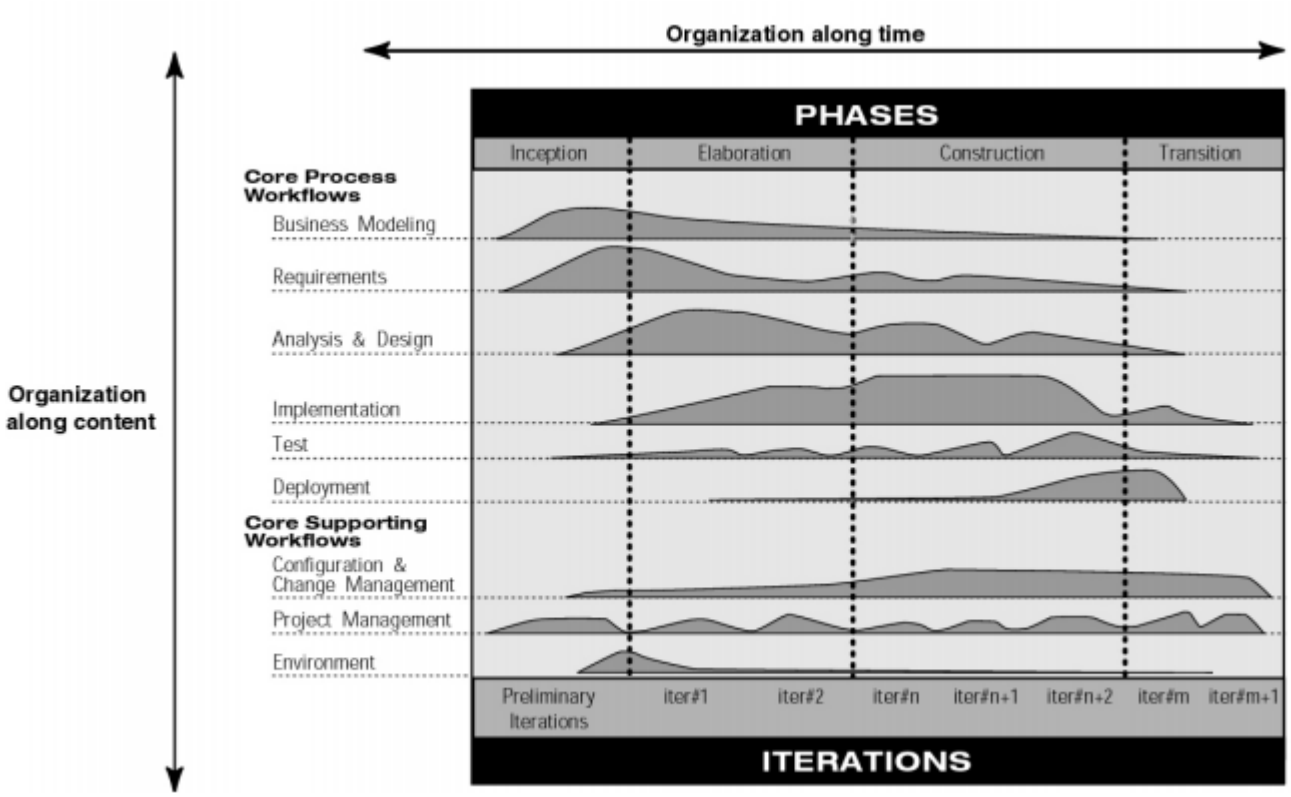


Figura 23: Matriz de esfuerzos

5.1 Matriz de fases, hitos y productos

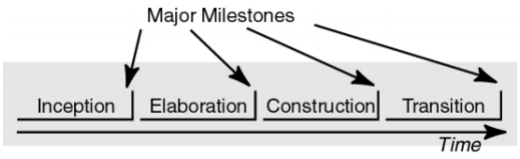


Figura 24: Hitos

Fase	Hito (*milestone*)	Conceptos involucrados	Productos obtenidos	Comentarios
Inicio	Objetivo	1. Alcance del proyecto 2. Estimación de plazos y costos 3. Comprensión inicial de requerimientos 4. Oportunidad del negocio 5. Otros productos	1 y 2. Documento de visión general 3. Modelo **inicial** de casos de uso (20 % completo) 4. *Business Case* (F.C.E. + pronóstico financiero) 5. Plan de proyecto inicial, identificación inicial de riesgos, prototipos (uno o más)	
Elaboración	Arquitectura del sistema base	1. Análisis del dominio del problema 2. Establecer arquitectura base sólida 3. Desarrollar plan de proyecto definitivo 4. Mitigar factores de riesgo 5. Otros productos	1. Modelo **final** de casos de uso 2. Descripción de arquitectura del sistema, prototipo ejecutable 3. Plan de proyecto final 4. Lista de riesgos revisada 5. Manual de usuario preliminar	A partir de este hito, la arquitectura, los requisitos y los planes de desarrollo son muy estables. Hay menos riesgos, se puede planificar el resto del proyecto con menos incertidumbre. Se construye arquitectura ejecutable que contemple casos de uso críticos y mayores riesgos identificados.
Construcción	Capacidad operacional inicial	1. Desarrollo e incorporación de componentes restantes 2. Pruebas en profundidad 3. Énfasis en eficiencia del producto 4. Otros productos	1. Producto de software con todos los componentes integrados y ejecutando en la plataforma adecuada 4. Manual de usuario final, descripción del *release* actual	Se obtiene un producto **beta** que debe decidirse si puede ponerse productivo sin mayores riesgos.
Transición	*Release* del producto	1. Traspaso de producto a usuarios 2. Pruebas beta 3. Ejecución en paralelo con sistemas antiguos 4. Migración de datos del viejo al nuevo sistema 5. Capacitación de usuarios 6. Distribución del producto y despliegue		Obtener autosuficiencia de los usuarios. Lograr consenso para liberar productos a todos los usuarios.

6 Metodologías ágiles

Metodologías Formales	Metodologías Ágiles
Atención a procesos y documentación	Atención a personas
Comunicación formal y escrita	Comunicación informal y cara a cara
Documentación de requisitos	Colaboración con el cliente
Seguimiento de planes	Agilidad en respuesta a los cambios de planes
Documentación completa	Software que funciona

6.1 Valores

1. **Individuos e interacciones** por sobre procesos y herramientas
2. **Software que funcion**a por sobre documentación extensiva
3. **Colaboración con el cliente** por sobre negociación contractual
4. **Respuesta ante el cambio** por sobre seguir un plan

6.2 Principios

1. Nuestra mayor prioridad es satisfacer al cliente mediante la **entrega temprana y continua de software** con valor.
2. **Aceptamos que los requisitos cambien**, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. **Entregamos software funcional frecuentemente**, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores **trabajamos juntos** de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a **individuos motivados**. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la **conversación cara a cara**.
7. El **software funcionando** es la **medida principal de progreso**.
8. Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La **atención continua a la excelencia técnica** y al buen diseño mejora la Agilidad.
10. La **simplicidad**, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de **equipos auto-organizados**.
12. **A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo** para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

6.3 Scrum

Scrum: "marco" (no define el proceso ni los entregables) para métodos de desarrollo ágil, iterativo e incremental. *Scrum* estructura el desarrollo del producto en ciclos que se llaman *sprints*. Un *sprint* fija objetivos y el trabajo acordado debe estar terminado al finalizar el mismo. Los *sprints* son de longitud fija (1 a 4 semanas). Durante un *sprint* no se pueden cambiar ni los integrantes ni las tareas; lo único que se puede hacer es cancelar el *sprint*.

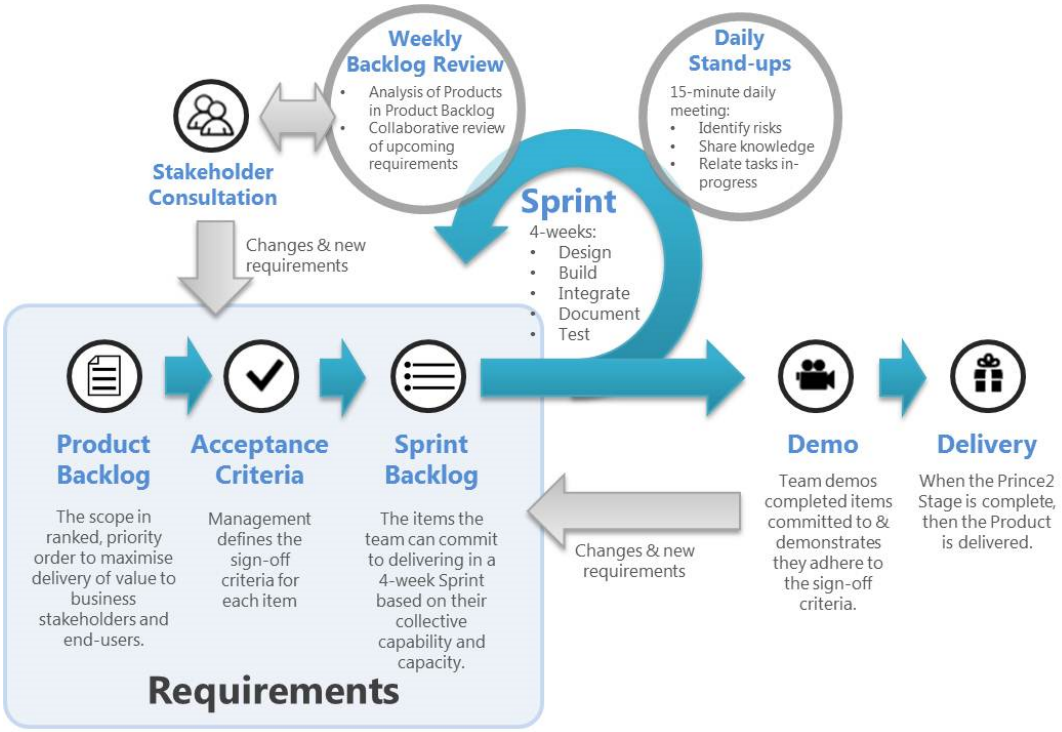


Figura 25: Scrum.

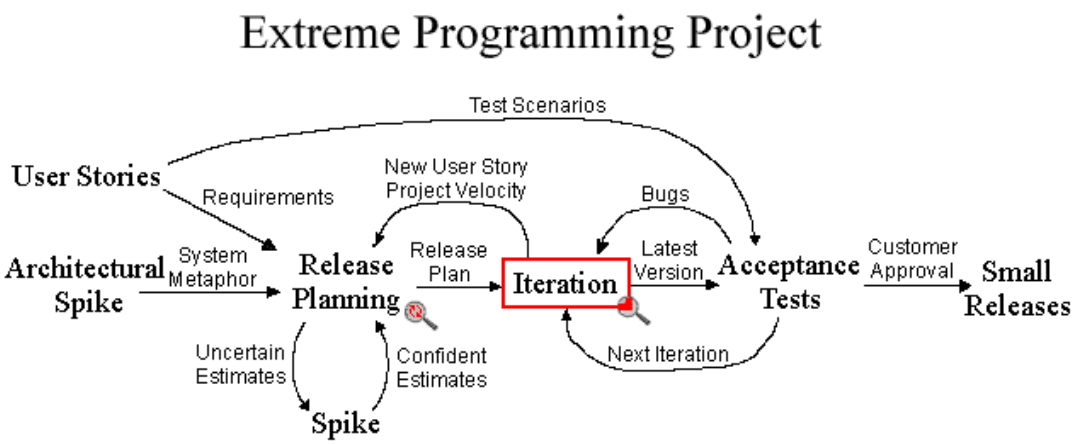
Hay 3 roles:

1. *Product Owner*: hace las veces del cliente. Elabora la lista de requisitos (*product backlog*) y les asigna prioridades. Define qué requisitos habrá que desarrollar en un *sprint* (*sprint backlog*).
2. *Scrum Master*: el líder del equipo.
3. Miembros del equipo: diseñan, codifican y prueban las funcionalidades.

Algunas características:

- Se realizan reuniones diarias de *scrum* para analizar el avance del día.
- Día a día se actualiza el *sprint burndown chart*, que muestra cuánto trabajo falta realizar dentro del *sprint*.
- Al finalizar el *sprint* se realiza una reunión para sacar conclusiones de lo aprendido.

6.4 XP (*Extreme Programming*)



Es un conjunto de prácticas ágiles de desarrollo.
Principios del XP:

1. *Test-Driven Design*: primero escribir los programas de prueba y después la aplicación.
2. *Pair Programming*: programar de a dos. Uno escribe y el otro observa el trabajo.
3. *Continuous Integration*: integraciones frecuentes y una máquina especial para las integraciones.
4. Pruebas unitarias, de integración y de aceptación.
5. Refactorizaciones al final.
6. Estándares de codificación precisos y estrictos.
7. Desarrollo incremental.
8. Comunicación frecuente con el cliente. Le mostramos el software y nos adaptamos a los cambios pedidos, y no a la inversa.
9. Implementar primero lo que tenga mayor valor para el cliente.
10. No escribir código por adelantado, ni código que no sabe si se usará. Se escribirá lo que se pida.
11. Compilaciones rápidas (10 minutos).
12. Documentación actualizada.

¿Cuándo usar XP?

- Cuando los requerimientos del cliente no son claros, cuando la funcionalidad del software pedido será cambiante.
- Cuando la fecha de entrega sea muy cercana, cuando haya mucho riesgo.
- Cuando el grupo de programadores sea pequeño (entre 2 y 12).
- Cuando haya comunicación constante entre todos (programadores y clientes).
- Cuando puedan realizarse pruebas.