**Modelos de proceso**

Un proceso se define como la colección de actividades de trabajo, acciones y tareas que se realizan cuando va a crearse algún producto terminado. Cada una de las actividades, acciones y tareas se encuentra dentro de una estructura o modelo que define su relación tanto con el proceso como entre sí.

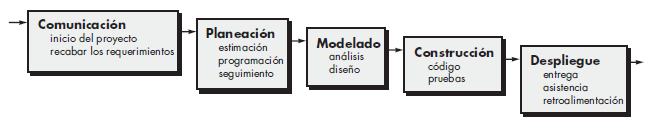
**Modelos de proceso prescriptivo o tradicional**

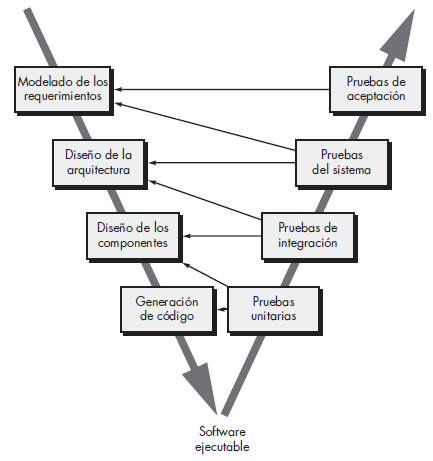
Buscan generar estructura y orden en el desarrollo del software. Prescriben un conjunto de elementos del proceso: actividades estructurales, acciones de ingeniería de software, tareas, productos del trabajo, aseguramiento de la calidad y mecanismos de control del cambio para cada proyecto. Cada modelo del proceso también prescribe un flujo del proceso (flujo de trabajo), es decir, la manera en la que los elementos del proceso se relacionan entre sí.

Modelo de Cascada (ciclo de vida clásico)

Se aplica cuando los requerimientos están bien definidos y tienen una estabilidad razonable. Esta situación se da cuando deben hacerse adaptaciones o mejoras bien definidas a un sistema ya existente.

Sugiere un enfoque sistemático y secuencial para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir con el apoyo del software terminado.



Variante de Cascada: Modelo en V

Hay una relación entre las acciones para el aseguramiento de la calidad y aquellas asociadas con la comunicación, modelado y construcción temprana.

A medida que el equipo de software avanza hacia abajo desde el lado izquierdo de la V, los requerimientos básicos del problema mejoran hacia representaciones técnicas cada vez más detalladas del problema y de su solución.

Una vez que se ha generado el código, el equipo sube por el lado derecho de la V, y en esencia ejecuta una serie de pruebas que validan cada uno de los modelos creados anteriormente.

El modelo en V proporciona una forma de visualizar el modo de asociación de las acciones de verificación y validación con el trabajo de ingeniería inicial.

*Problemas al aplicar Cascada:*

1. Es raro que los proyectos reales sigan un flujo secuencial. Aunque el modelo lineal acepta repeticiones, lo hace en forma indirecta. No se adapta a los cambios.

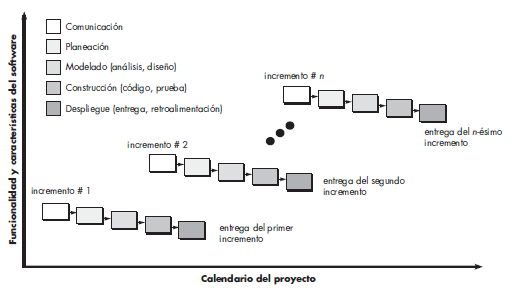
2. Es difícil para el cliente enunciar en forma explícita todos los requerimientos.

3. El cliente debe tener paciencia. No se dispondrá de una versión funcional del sistema hasta que el proyecto esté muy avanzado. Un error grande sería desastroso si se detectara al revisar el programa en funcionamiento.

4. Debido a su naturaleza lineal, el ciclo de vida llega a “estados de bloqueo” en los que ciertos miembros del equipo de proyecto deben esperar a otros a fin de terminar tareas interdependientes.

Modelo Incremental

El modelo incremental combina elementos de los flujos de proceso lineal y paralelo. Aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce “incrementos” de software que, en forma progresiva, dan más funcionalidad al cliente conforme se le entrega cada incremento.



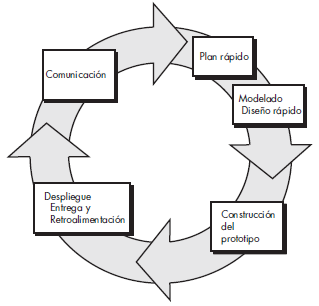
Se centra en que en cada incremento se entrega un producto que ya opera. Es frecuente que el primer incremento sea el producto fundamental. Es decir, se abordan los requerimientos básicos, pero no se proporcionan muchas características suplementarias.

Se aplica en situaciones en la que hay una necesidad imperiosa de dar rápidamente cierta funcionalidad limitada de software a los usuarios y aumentarla en las entregas posteriores de software. Es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio.

Modelo Evolutivo

Se utiliza cuando los requerimientos del negocio y del producto cambian a medida que avanza el desarrollo. Se comprende bien el conjunto de requerimientos o el producto básico, pero los detalles del producto o extensiones del sistema aún están por definirse.

Es un modelo iterativo de manera que permiten desarrollar versiones cada vez más completas del software.

* Prototipo: Es una técnica que puede implementarse en cualquiera de los modelos de proceso. Sirve de ayuda para mejorar la comprensión de lo que hay que elaborar cuando los requerimientos no están claros.

Los participantes definen los objetivos generales del software, identifican los requerimientos que conozca y detectan las áreas en las que es imprescindible una mayor definición. Se planea rápidamente una iteración para hacer el prototipo, y se lleva a cabo el modelado. Éste se centra en la representación de aquellos aspectos del software que serán visibles para los usuarios finales. El diseño rápido lleva a la construcción de un prototipo. Éste se entrega y es evaluado por los participantes, que dan retroalimentación para mejorar los requerimientos. La iteración ocurre a medida que el prototipo es afinado para satisfacer las necesidades de distintos participantes, y al mismo tiempo permite entender mejor lo que se necesita hacer.

Para construirlo pueden utilizarse fragmentos de programas existentes o herramientas que permitan generar rápidamente programas que funcionen.

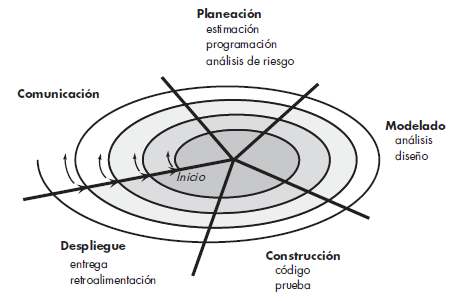
*Problemas*

1. Los prototipos no son funcionales. Se puede generar demasiada expectativa en los clientes.

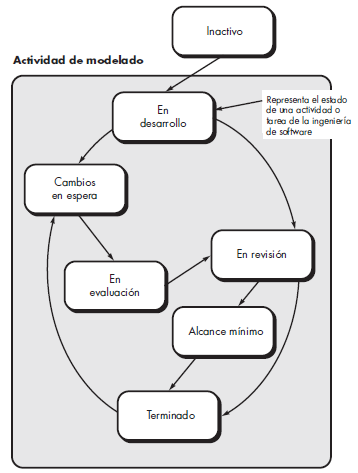
2. Por mejorar el diseño, el desarrollador se olvida de la performance y otros aspectos más importantes como el lenguaje.

* Espiral: El software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, lo que se entrega puede ser un modelo o prototipo. En las iteraciones posteriores se producen versiones cada vez más completas del sistema cuya ingeniería se está haciendo.

Un modelo en espiral es dividido por el equipo de software en un conjunto de actividades estructurales. El equipo de software realiza actividades implícitas en un circuito alrededor de la espiral en el sentido horario, partiendo del centro. En cada paso evolutivo se marcan puntos de referencia puntuales: combinación de productos del trabajo y condiciones que se encuentran a lo largo de la trayectoria de la espiral.

El primer circuito alrededor de la espiral da como resultado el desarrollo de una especificación del producto; las vueltas sucesivas se usan para desarrollar un prototipo y, luego, versiones cada vez más sofisticadas del software.

Es un enfoque realista para el desarrollo de sistemas y de software a gran escala. Como el software evoluciona a medida que el proceso avanza, el desarrollador y cliente comprenden y reaccionan mejor ante los riesgos en cada nivel de evolución. El modelo espiral usa los prototipos como mecanismo de reducción de riesgos, pero, más importante, permite aplicar el enfoque de hacer prototipos en cualquier etapa de la evolución del producto.



Modelo Concurrente

Permite que un equipo de software represente elementos iterativos y concurrentes de cualquiera de los modelos de proceso.

Es más apropiado para proyectos de ingeniería de productos en los que se involucran varios equipos de trabajo. Proporciona un panorama apropiado del estado actual del proyecto.

En lugar de confinar las actividades, acciones y tareas de la ingeniería de software a una secuencia de eventos, define una red del proceso o serie de eventos. Cada actividad, acción o tarea de la red existe simultáneamente con otras actividades, acciones o tareas. Los eventos generados en cierto punto de la red del proceso desencadenan transiciones entre los estados.

**Modelos de proceso especializado**

Tienen muchas de las características de uno o más de los modelos tradicionales. Tienden a aplicarse cuando se elige un enfoque de ingeniería de software especializado o definido muy específicamente.

Desarrollo basado en componentes

Incorpora muchas de las características del modelo espiral. Es de naturaleza evolutiva y demanda un enfoque iterativo para la creación de software. Sin embargo, construye aplicaciones a partir de fragmentos de software prefabricados. Este modelo lleva a la reutilización del software.

Las actividades de modelado y construcción comienzan con la identificación de candidatos de componentes. Éstos pueden diseñarse como módulos de software convencional o clases orientadas a objetos o paquetes de clases. El modelo incorpora las siguientes etapas (se implementan con el uso de un enfoque evolutivo):

1. Se investigan y evalúan productos disponibles basados en componentes.

2. Se consideran los aspectos de integración de los componentes.

3. Se diseña una arquitectura del software para que reciba los componentes.

4. Se integran los componentes en la arquitectura.

5. Se efectúan pruebas exhaustivas para asegurar la funcionalidad apropiada.

Modelo de métodos formales

Agrupa actividades que llevan a la especificación matemática formal del software de cómputo. Permite especificar, desarrollar y verificar un sistema muy complejo por medio del empleo de una notación matemática rigurosa.

Cuando durante el desarrollo se usan métodos formales, se obtiene un mecanismo para eliminar muchos de los problemas difíciles de vencer con otros paradigmas de la ingeniería de software. Lo ambiguo, incompleto e inconsistente se descubre y corrige con más facilidad con la aplicación de análisis matemático.

Lo utilizan los desarrolladores que deben construir software de primera calidad en seguridad y los desarrolladores que sufrirían graves pérdidas económicas si ocurrieran errores en su software.

*Problemas*:

• Consume mucho tiempo y es caro.

• Se requiere mucha capacitación.

• Es difícil utilizar los modelos como mecanismo de comunicación para clientes sin complejidad técnica.

Desarrollo de software orientado a aspectos

Consiste en definir “aspectos” que expresan preocupaciones del cliente que afectan múltiples funciones, características e información del sistema.

Los requerimientos del aspecto definen aquellas preocupaciones globales que tienen algún efecto a través de la arquitectura del software. El desarrollo de software orientado a aspectos proporciona un proceso y enfoque metodológico para definir, especificar, diseñar y construir aspectos: “mecanismos más allá de subrutinas y herencia para localizar la expresión de una preocupación global”.

**Proceso Unificado de desarrollo de SW (PU)**

Es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software. Reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema (el caso de uso). Hace énfasis en la importancia de la arquitectura del software. Sugiere un flujo del proceso iterativo e incremental, lo que da la sensación evolutiva que resulta esencial en el desarrollo moderno del software.

Es genérico y debe ser “instanciado” en cada caso:

* Estándares propios
* Plantillas de documentos
* Herramientas, compiladores, etc.
* Bases de datos, seguimiento de proyectos, seguimiento de errores, etc.
* Modificaciones al ciclo de vida: reforzar aspectos críticos (seguridad, tests, etc.)

Axiomas:

* Dirigido por requisitos y riesgo.
* Centrado en arquitectura: cuál es la forma o estructura que el software debe tener.
* Iterativo e incremental (el proceso es gradual): divide en subproyectos (módulos del sistema) que agregan funcionalidad hasta llegar al sistema final. Se requiere la totalidad de los requerimientos. Cada iteración es un subproyecto con actividades secuenciales.

Iteraciones:

Cada iteración contiene todos los elementos de un proyecto de desarrollo: planificación, análisis, diseño, construcción, integración, pruebas y versión del sistema.

En cada iteración se genera una Línea Base (LB), que es una versión interna o externa del conjunto de artefactos revisados y aprobados generados por esa iteración. Cada LB proporciona una base acordada para mayor revisión y desarrollo y se puede cambiar solamente por medio de procedimientos formales de configuración y gestión del cambio.

**Estructura**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fases** | **Comienzo** | **Elaboración** | **Construcción** | **Transición** |
| **Hitos** | Objetivos del proyecto | Arquitectura del sistema | Posibilidad operacional inicial | Versión del producto |

Las fases terminan cuando se cumplen los hitos. Cada fase puede tener varias iteraciones. Al finalizar se obtiene el subsistema.

Comienzo

* Agrupa actividades de comunicación con el cliente y de planeación.
* Se identifican los requerimientos, los cuales se describen por medio de un conjunto de casos de uso preliminares que detallan las características y funciones que desea cada clase de usuario.
* Se propone una arquitectura aproximada para el sistema, la cual no es más que un lineamiento tentativo de subsistemas principales y la función y rasgos que tienen.
* Se desarrolla un plan para la naturaleza iterativa e incremental del proyecto. La planeación identifica los recursos, evalúa los riesgos principales, define un programa de actividades y establece una base para las fases que se van a aplicar a medida que avanza el incremento del software.
* Se establece la viabilidad técnica y económica.
* Se crea un caso de negocio: recursos, alcance.
* Foco de atención: Workflows de Requisitos y Análisis.
* Entregables: Documento con requisitos, características y restricciones. Modelo inicial de caso de uso. Documento de análisis de riegos. Prototipos. Documento inicial de arquitectura.

Elaboración

* Incluye actividades de comunicación y modelado.
* Se mejoran y amplían los casos de uso preliminares desarrollados anteriormente.
* Se definen atributos de calidad, es decir, los requisitos no funcionales del sistema.
* Se aumenta la representación de la arquitectura para incluir cinco puntos de vista distintos del software: los modelos del caso de uso, de requerimientos, del diseño, de la implementación y del despliegue.
* Se crea una Línea Base ejecutable de la arquitectura que representa un sistema ejecutable de “primer corte”. La línea base de la arquitectura demuestra la viabilidad de ésta, pero no proporciona todas las características y funciones que se requieren para usar el sistema.
* Además, se revisa con cuidado el plan a fin de asegurar que el alcance, riesgos y fechas de entrega siguen siendo razonables. Es frecuente que en este momento se hagan modificaciones al plan.
* Se realiza un plan detallado de la fase de construcción.
* Se formula la oferta: recursos, tiempos, equipamientos, personas, costos.
* Foco de atención: Workflows de Requisitos, Análisis, Diseño para la arquitectura (diseño). Workflows de Implementación y Prueba (sin generar software) para formular la oferta.
* Entregables: Línea base ejecutable de la arquitectura. Modelo UML estático, dinámico y caso de uso. Documentos actualizados y plan de proyecto.

Construcción

* Se completan los modelos de requerimientos y diseño, a fin de que reflejen la versión final del incremento de software. Después se implementan en código fuente todas las características y funciones necesarias para el incremento de software. A medida que se implementan los componentes, se diseñan y efectúan pruebas unitarias para cada uno. Además, se realizan actividades de integración. Se emplean casos de uso para obtener un grupo de pruebas de aceptación que se ejecutan antes de comenzar la siguiente fase del PU.
* Se evoluciona la LB de la arquitectura.
* Foco de atención: Workflows de Implementación y Prueba (para verificar que obtuve un primer sistema que funciona correctamente).
* Entregables: producto de software, modelos UML, suite de pruebas, manuales de usuario, descripción de versión y plan de proyecto.

Transición

* Se entrega el software a los usuarios finales para las pruebas beta, quienes reportan tanto los defectos como los cambios necesarios y se modifica el SW si es necesario.
* El equipo de software genera la información de apoyo necesaria que se requiere para el lanzamiento. Al finalizar la fase de transición, el software incrementado se convierte en un producto utilizable que se lanza.
* Se preparan los entornos de usuarios (redes internas, servidores, etc.) y se adapta el software. Se ofrece una consultoría a usuario (explicación del funcionamiento).
* Foco de atención: workflows de Implementación (incluye la puesta en funcionamiento) y Prueba (incluye verificar que el software funcione en el entorno de los usuarios).
* Entregables: producto de software, plan de soporte de usuario y manuales de usuarios actualizados.

**Introducción a UML (Lenguaje Unificado de Modelado)**

Es un lenguaje de modelado visual, por lo que los diagramas son legibles por las personas. El objetivo es incorporar las mejores prácticas en las técnicas de modelado y la ingeniería de software.

Premisa básica: modelar software y sistemas como un conjunto de objetos que interactúan entre sí.

No proporciona ningún tipo de metodología de modelado, sólo una sintaxis visual.

Se puede utilizar con cualquier metodología o ciclo de vida de desarrollo y con cualquier lenguaje y plataforma de implementación. Puede soportar muchos procesos de desarrollo.

Dos aspectos atómicos del modelo:

* Estructura estática: describe qué tipos de objetos son importantes para el modelado y cómo se relacionan.
* Comportamiento dinámico: describe los ciclos de vida de los objetos y cómo interactúan entre sí.

Estructura UML:

* Bloques de construcción: elementos básicos UML, relaciones y diagramas.
* Mecanismos comunes: formas UML de conseguir objetivos específicos (especificaciones, adornos, divisiones comunes y mecanismos de extensión).
* Arquitectura: visión UML de la arquitectura del sistema.

**Especificación de requisitos**

**Workflow de requisitos**

La finalidad es descubrir y llegar a un acuerdo sobre lo que debería hacer el sistema, expresado en el idioma de los usuarios del sistema.

Usuarios del sistema (interesados): usuarios finales, clientes, programadores, diseñadores, agentes de ventas, soporte, etc.

Un requisito es una especificación de lo que se debería implementar. Se clasifican en:

* Requisitos funcionales: lo que el sistema debería hacer, comportamiento que debería ofrecer el sistema.
  + Clasificación según características, para dividir en módulos el SW: clientes, productos, pedidos, proveedores, canales de venta, pagos.
* Requisitos no funcionales: restricciones y propiedades específicas no funcionales del sistema. Ej.: velocidad de procesamiento, restricción a ciertos usuarios.
  + Clasificación según rendimiento, capacidad, disponibilidad, seguridad y seguimiento de estándares.

Ambos tipos de requisitos deben ser recolectados en el comienzo del proyecto.

Puesto que el sistema de software se basa sobre un conjunto de requisitos, la captura de requisitos es un factor crítico de éxito en proyectos de desarrollo de software.

Los requisitos deberían ser solamente una declaración de lo quedebería hacer el sistema, y no cómo lo debería hacer.

Modelo de requisitos: Requisitos Funcionales + No Funcionales

Requisito bien formado: <Id> El <sistema> debería <funcionar>

Atributos de requisitos: Capturan información adicional (metadatos) sobre el requisito. Ejemplos:

* *Prioridad*: relativa de un requisito respecto a otro. Valores posibles: requisito obligatorio para el sistema, importante que se puede omitir, opcional (se hace si hay tiempo y recursos) y aquellos que pueden incluirse en versiones posteriores.
* *Origen* del requisito.
* Otros (RUP): estado, beneficio, estimación, riesgo, etc.

Encontrar requisitos

Los requisitos provienen del contexto del sistema a modelar: usuarios directos del sistema, grupos de decisión (directores, mantenimiento, instaladores, etc.), otros sistemas con el que se interactúa, dispositivos de HW, restricciones legales y regulatorias, restricciones técnicas y objetivos del negocio.

El objetivo de definir requisitos es obtener una imagen precisa o mapa del modelo del mundo para las personas involucradas. Como no se puede capturar el mundo real completo, se utilizan 3 filtros para crear el mapa:

* *Eliminación***:** se filtra la información.
* *Distorsión***:** se modifica la información por mecanismos de creación e ilusión.
* *Generalización***:** se resume la información en reglas, creencias y principios sobre lo verdadero o falso.

Técnicas de relevamiento

* *Entrevistas*: no inventar una solución, formular preguntas abiertas, saber escuchar, no adivinar pensamientos y tener paciencia. La mejor técnica es utilizar lápiz y papel. Luego de la entrevista, se analiza la información y se extraen requisitos candidatos.
* *Cuestionarios*: no sustituyen las entrevistas, son excelentes para conseguir respuestas a preguntas cerradas y se distribuyen a una audiencia amplia.
* *Workshop de requisitos*: es una tormenta de ideas que se utiliza cuando no se tiene bien en claro los requerimientos. Todas las ideas se aceptan como buenas ideas, se registran, pero no se debaten. Los miembros del equipo deben nombrar los requisitos claves para el sistema, escribirlos en una tarjeta y pegarlos en la pared o pizarra. Luego se pasan los requisitos y se identifican atributos adicionales. Analizar los resultados y convertirlos en requisitos. Circularlos. Es un proceso iterativo y puede repetirse.

**Modelado de caso de uso**

Es una forma diferente y complementaria de crear y documentar requisitos, principalmente, funcionales. Es la entrada principal para el modelado de clases.

Proceso: Consiste en encontrar el límite del sistema candidato, encontrar los actores y encontrar los casos de uso (especificarlos e identificar flujos alternativos clave). Repetir esto hasta tener un modelo estable.

**Componentes de un modelo de CU**

**1. Límite del sistema**

****En UML se denomina sujeto. Se simboliza con un rectángulo que encierra todo lo que pertenece al sistema. Se debe definir qué es parte del sistema y qué externo. Está definido por quién o qué utiliza el sistema (ej.: actores - elementos externos). El límite indica qué utilidad específica ofrece el sistema a estos actores (CU).

**2. Actores**

Especifican un rol que cierta entidad externa o elemento adopta cuando interactúa directamente con el sistema. Puede representar un rol de usuario o un rol desempeñado por otro sistema o HW que toca el límite del sistema. Los elementos pueden tener muchos roles simultáneamente y con el tiempo, es decir, un rol puede desempeñarse por muchos elementos diferentes simultáneamente.

Los actores siempre son externos al sistema. Si bien existe una representación interna (almacenando datos) de algunos actores, no representan lo mismo.

Identificar actores:

* *Preguntas básicas* (orientadas a actores primarios): ¿Quién y qué utiliza el sistema? ¿Qué roles desempeña en la iteración? ¿Sucede algo en un momento dado? ¿Quién o qué consigue y proporciona información al sistema?
* *Preguntas complementarias* (apuntadas a características no funcionales): ¿Quién instala el sistema? ¿Quién o qué lo inicia y cierra? ¿Quién lo mantiene? ¿Qué otros sistemas interactúan con este?

Cada actor necesita: Un nombre breve que tenga sentido desde la perspectiva del negocio y una descripción de una o dos líneas que describa qué es este actor desde una perspectiva de negocio.

El “Tiempo” como actor: Aparece cuando se necesita modelar elementos que suceden en un punto específico en el tiempo pero que no parecen estar activados por ningún actor.

**3. Casos de Uso**

Es algo que el actor quiere que el sistema haga. Se inician siempre por un actor. Se escriben siempre desde el punto de vista de los actores.

Identificar CU:

* Empezar con la lista de actores y luego considerar cómo cada actor va a utilizar el sistema. Lista de CU candidatos.
* *Preguntas básicas*: ¿Qué funciones querrá un actor específico del sistema? ¿El sistema almacena y recupera información? Si es así, ¿qué actores activan este comportamiento? ¿Genera el sistema algún informe?
* *Preguntas complementarias* (orientadas a características no funcionales): ¿Interactúa el sistema con algún sistema externo? ¿Qué sucede cuando el sistema cambia de estado? ¿Se notifica a algún actor? ¿Afecta algún evento externo al sistema? ¿Qué notifica el sistema sobre estos eventos?
* Luego de encontrado los CU, es necesario especificar lo que significan y cómo deben ser resueltos.

Diagrama de CU: Representa el sujeto del modelo de CU por un cuadro etiquetado con el nombre del sujeto. Muestra actores fuera del límite, es decir, externos al sistema, y casos de uso dentro del sujeto, que constituyen el comportamiento del sistema. También se representan las relaciones entre los actores y los CU (línea continua).

**4. Glosario**

Proporciona un diccionario de términos claves de negocio y definiciones. Debería ser entendible por todo el mundo en el proyecto. Entender y capturar el lenguaje del negocio es crítico. Definir un lenguaje común (negocio – técnico) y mantener el glosario lo más sencillo y conciso posible.

Además, el glosario debe resolver sinónimos y homónimos:

* *Sinónimos*: palabras diferentes que significan lo mismo. Elegir una de estas palabras, descargar las restantes.
* *Homónimos*: la misma palabra que significa cosas diferentes a diferentes personas. Elegir uno e introducir nuevos términos para los restantes (renombrar).

Para crear un glosario se escriben términos con su definición, especificando en cada uno si hasta el momento existen sinónimos y homónimos.

**Especificación de CU**

Consiste en detallar más los CU, asignando mínimamente un nombre y una especificación.

UML no ofrece un estándar para las especificaciones de CU, pero es importante decidir uno para la especificación. Un ejemplo podría ser:

|  |
| --- |
| **Caso de uso**: el nombre debe ser único, siempre un verbo o frase verbal que lo describa. Notación libre o unir palabras con primera letra mayúscula. |
| **ID**: identificador inmutable que identifica en forma única un CU particular. |
| **Breve descripción**: un párrafo que resume el objetivo del CU. |
| **Actor principal**: activa el CU. Cada CU siempre se activa por 1 solo actor. |
| **Actores secundarios**: interactúan con CU después de haberse activado. |
| **Precondiciones**: restringen el estado del sistema antes de que el caso de uso pueda empezar, es decir, lo que debe ser verdadero antes de que el CU se pueda activar. |
| **Flujo principal**: detalle de los pasos del CU. Capturan el “mundo perfecto”: todo sucede según lo esperado y lo deseado. Siempre empieza por el actor principal utilizando una función para activar el CU: El caso de uso empieza cuando un <actor> <función>  Consta de una secuencia de pasos breves declarativos, numerados y ordenados en el tiempo, de la forma: <numero> El <algo> <alguna acción>  Se pueden utilizar ramificaciones (si, sino), repeticiones (para) y condiciones booleanas (mientras). |
| **Postcondiciones**: restringen el estado del sistema después de que el CU se ha ejecutado. Especifican qué será verdadero después de que el CU se haya ejecutado. |
| **Flujos alternativos**: son rutas de acceso alternativas a través del CU que capturan errores, ramificaciones e interrupciones en el flujo principal. Frecuentemente no regresan al flujo principal. Hacer su especificación aparte. |

Consideraciones especiales para la elaboración de los CU

* El CU debe ser siempre una declaración precisa de una parte de la funcionalidad del sistema.
* Para evitar ambigüedades, eliminaciones o generalizaciones: específicamente quién, qué, cuándo, dónde.
* Encontrar flujos alternativos
  + - Alternativas posibles al flujo principal.
    - Errores que puedan surgir en el flujo principal.
    - Interrupciones que puedan ocurrir en un punto particular en el flujo principal.
    - Interrupciones que puedan ocurrir en cualquier punto en el flujo principal.
* Limitar el número de flujos alternativos al mínimo posible.
* Documentar sólo los flujos alternativos relevantes.
* Mantener las especificaciones de CU al mínimo nivel necesario.

Seguimiento de Requisitos

Relacionar el modelo de requisitos y el modelo de CU para averiguar si hay algo en alguno de los modelos que no se trató. Un CU tratará muchos requisitos funcionales individuales y un requisito funcional puede estar presente en varios CU diferentes. UML proporciona herramientas para el seguimiento de requisitos, pero se puede utilizar la siguiente matriz de trazabilidad:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Requisitos Funcionales** | **CU 1** | **CU 2** | **CU 3** | **…** |
| RF1 |  | X |  |  |
| RF2 | x | X |  |  |
| RF3 |  |  | X |  |
| … |  |  |  |  |
| RFn |  |  |  | X |

**Modelado de CU (Avanzado)**

**Extensiones de CU**

* + Include (<<include>>):
    - Permite incluir el comportamiento de un caso de uso en el flujo de otro caso de uso.
    - El CU que incluye se denomina CU base**.**
    - El CU incluido se denomina CU de inclusión.
    - El CU de inclusión proporciona su comportamiento a su CU base.
    - El CU base se ejecuta hasta que se alcanza el punto de inclusión, luego la ejecución se pasa al caso de uso de inclusión. Cuando este termina, continúa en el CU base.
    - El CU base no está completo sin todos sus CU de inclusión.
    - Los CU de inclusión suelen no poder instanciarse por sí mismos, sino que deben ser invocados por otros CU.
  + Extend (<<extend>>): Una relación entre CU que permite
* Permite que un CU extienda su comportamiento con otro CU.
* El CU base proporciona puntos de extensión donde se incorporan los CU de extensión.
* El CU base no sabe nada de los CU de extensión.
* El CU base está completo sin sus extensiones.
* Los puntos de extensión no están realmente insertados en el flujo de eventos de CU base, están añadidos a una capa en la parte superior del flujo principal.
* Proporciona una forma de tratar casos excepcionales.
* Un caso de extensión podría proveer más de un comportamiento. Es por ello que, en la notación, se utiliza “Segmento X” en cada sección, indicando el orden de ejecución.

**Generalización**

* + Generalización de Actor:
    - Es una relación de generalización entre un actor más general y un actor más específico.
    - Se presentan cuando dos o más actores se comunican con el mismo conjunto de casos de uso de la misma forma.
    - Los actores descendentes heredan los roles y relaciones a casos de uso albergados por el actor padre.
    - Se puede sustituir un actor descendente en cualquier lugar que se espere el ascendiente.
  + Generalización de Casos de Uso:
    - Se utiliza cuando tiene uno o más CU que son realmente especificaciones o un caso más general.
    - Los CU hijo representan formas más específicas del padre.
    - Los hijos pueden heredar características, añadir nuevas características y anular o cambiar las características heredadas.

Sugerencias de escritura de CU

* Utilizar características avanzadas solamente cuando simplifiquen el modelo.
  + Los mejores modelos de CU son sencillos.
* Mantener los CU breves y sencillos.
  + Incluir el detalle suficiente para capturar los requisitos.
  + Regla: asegurarse de que el flujo principal de un CU cabe en una hoja de papel.
* Centrarse en el qué y no el cómo.
* Evitar descomposición funcional

**Workflow de análisis Orientado a Objetos (Proceso)**

En la fase de elaboración se crean modelos que capturan el comportamiento deseado del sistema. La finalidad es generar un modelo de análisis que se centre en lo que el sistema necesita hacer (no cómo lo hará).

Se generan:

* Clases de análisis: modelan conceptos clave en el dominio de negocio.
* Realizaciones de CU: ilustran cómo las instancias de clase de análisis pueden interactuar para realizar el comportamiento del sistema especificado por un CU.

El modelo de análisis puede contener muchos paquetes de análisis y cada paquete puede a su vez contener otros paquetes anidados. Estos paquetes contienen clases de análisis y realización de CU.

Los paquetes de análisis se pueden dividir en otros paquetes de análisis.

Consideraciones del modelo de análisis:

* Limitarse solamente a aquellas clases que son parte del dominio del problema. No en particularidades de diseño, o del software que espero construir.
* El modelo de análisis está siempre en el lenguaje del negocio.
* Cree modelos que “cuenten una historia”. Que nos muestre algo que el usuario pueda validar (decir si está bien o no).
* Concéntrese en capturar la idea general: lo que hay que resolver.
* Distinga claramente entre el dominio del problema (los requisitos del negocio) y del dominio de solución (consideraciones detalladas de diseño).
* Siempre trate de minimizar el acoplamiento (el nivel de comunicación entre dos entidades).
* Explore herencia si parece haber una jerarquía natural de abstracciones. Se incorporan solo cuando es necesario, no siempre es bueno. Se hacen cuando los elementos en común entre dos entidades son muchos.
* El modelo debe ser de utilidad para todos los grupos de decisión. Hago solo lo que alguien necesita.
* Mantener el modelo sencillo.

**Clases**

El primer paso para crear un SW OO es aclarar el dominio del problema: es el dominio en el que surge primero la necesidad de un sistema de software.

Una clase representa una abstracción que modela un elemento específico del dominio del problema. El aspecto más importante de una clase es que debería mapear en forma clara y no ambigua con algún concepto de negocio del mundo real. Raramente los conceptos son claros y no ambiguos.

Es crucial que cada clase tenga un conjunto cohesivo de responsabilidades que coincidan con el propósito de dicha clase y con el elemento del mundo real que representa.

Tener en cuenta que ninguna clase permanece sola, todas colaboran entre sí.

Estructura de una clase:

|  |
| --- |
| Nombre Clase |
| Atributos |
| Operaciones |

Poner sólo los atributos importantes y las operaciones claves, sino ninguna.

Debe haber sí o sí una descripción representativa.

**Técnicas para encontrar clases de análisis**

* Análisis nombre/verbo
* Análisis CRC.
* Estereotipos RUP (proceso unificado racional): se sale del dominio del problema.

Análisis nombre/verbo

Se analiza el texto descriptivo de requisitos funcionales, especificaciones de CU, o descripciones narrativas para encontrar clases, atributos y responsabilidades.

* Nombres y frases nominales: Clases o atributos de clases. Ej: Profesor, Alumno, Número de Legajo.
* Verbos y frases verbales: Responsabilidades u operaciones de una clase. Ej: “registra la asistencia de los alumnos”, “solicita un certificado de alumno regular”, “pagar”.

Procedimiento:

1. Recopilar la información relevante: desde el modelo de requisitos, modelos de CU, glosario del proyecto u otros documentos.
2. Analizar la documentación para destacar nombres, frases nominales, verbos y frases verbales.
3. Asignación tentativa de clases, atributos y responsabilidades.

Análisis CRC (Clase, Responsabilidades y Colaboradores)

Es una buena forma de hacer que el usuario se implique en la búsqueda de clases. Este análisis siempre se debería utilizar con el análisis nombre/verbo, a menos que el sistema sea muy sencillo.

Es una técnica manual. Se utilizan post-it (notas):

|  |  |
| --- | --- |
| Nombre de la clase | |
| Responsabilidades | Colaboradores |

Los colaboradores son otras clases que pueden colaborar con esta clase para realizar una parte de la funcionalidad del sistema. Proporciona una forma de grabar relaciones entre clases.

Procedimiento: Recopilar información y analizarla para decidir cuáles post-it van a convertirse en clase y cuáles en atributos. Es un proceso repetitivo porque hay que hacerlo en varias veces hasta llegar a las clases, atributos y operaciones.

Tener en cuenta:

* Si una nota parece ser parte de otra nota, puede ser un atributo.
* Si una nota no parece ser importante o tiene un comportamiento poco interesante puede ser un atributo de otra clase.
* Si se tiene dudas sobre una nota, se puede armar una clase.

Estereotipos RUP

Se consideran tres tipos distintos de clases de análisis durante la actividad de análisis. Es una forma de centrar el análisis en aspectos específicos del sistema. Es una técnica opcional que puede utilizarse para complementar las dos técnicas anteriores.

**Relaciones entre clases**

Las relaciones son conexiones semánticas (significativas) entre elementos de modelado. Son la forma de UML de conectar elementos.

Conexiones entre objetos:

En un sistema OO los objetos tienen que estar conectados para colaborar entre sí. Las conexiones se llaman *vínculos*.

**Vínculo**

Es una conexión semántica entre dos objetos que permite enviar mensajes de un objeto a otro.

Para que exista un vínculo, al menos uno de los objetos tiene que tener una referencia de objeto al otro.

Si existe un vínculo entre dos objetos, debe existir una conexión semántica entre sus clases (las clases deben conocer los otros objetos). Las conexiones entre clases se denominan *asociaciones*.

**Diagramas de Objetos**

Muestra objetos y sus relaciones en un punto en el tiempo. Los objetos vinculados pueden tomar diferentes rolesrelativos entre ellos. Tener en cuenta que los vínculos son conexiones dinámicas entre objetos, por lo que no están fijos en el tiempo.

**Asociación**

Son relaciones entre clases. Para que exista un vínculo entre objetos, debe existir una asociación entre sus clases debido a que un vínculo es una instancia de una asociación.

Semántica: una asociación entre clases indica que puede haber vínculos entre objetos de esas clases.

En tiempo de ejecución, las clases no existen; existen sus instancias.

Sintaxis de la asociación

Las asociaciones pueden tener:

* + Nombre de asociación
* Debería ser una frase verbal que indique una acción que el objeto fuente está realizando sobre el objeto destino.
* Se podría prefijar o postfijar con una punta de flecha negra para indicar la dirección en que debería leerse el nombre.
* Se escribe la primera palabra en minúsculas y las siguientes empiezan en mayúsculas.
  + Nombre de rol
* Referencia el rol que los objetos desempeñan cuando están vinculados por instancias de la asociación. Deberían ser nombres o frases nominales.
* Se pueden ponen en uno o ambos extremos de la relación.

*Nota: Poner nombre de asociación o de rol, no los dos.*

* + Multiplicidad
* Es un tipo de restricción.
* Restringe el número de objetos de una clase que se pueden implicar en una relación determinada en cualquier momento en el tiempo.
* Esta restricción es muy importante porque puede codificar reglas clave de negocio en el modelo.
* En UML no existe multiplicidad predeterminada, por lo que si no se indica esta restricción, significa que está pendiente.
* Sintaxis:

|  |  |
| --- | --- |
| **Adorno** | **Semántica** |
| 0..1 | Cero o 1 |
| 1 | Exactamente 1 |
| 0..\* | Cero o mas |
| \* | Cero o mas |
| 1..\* | 1 o mas |
| 1..6 | 1 a 6 |
| 1..3, 7..10, 15, 19..\* | 1 a 3 o 7 a 10 o 15 o 19 o más |

* + Navegabilidad
* Muestra que es posible pasar desde un objeto de la clase fuente a uno o más objetos de la clase destino, dependiendo de la multiplicidad.
* Útil para minimizar el acoplamiento entre clases.
* Sintaxis:

|  |  |  |  |
| --- | --- | --- | --- |
| **Sintaxis** | **Modelo 1: Navegabilidad estricta** | **Modelo 2: No navegabilidad** | **Modelo 3: Práctica estándar** |
| Clase A ↔ Clase B | A a B es navegable  B a A es navegable |  |  |
| Clase A x→ Clase B | A a B es navegable  B a A no es navegable |  |  |
| Clase A → Clase B | A a B es navegable  B a A no está definido |  | A a B es navegable  B a A no es navegable |
| Clase A – Clase B | A a B no está definido  B a A no está definido | A a B no está definido  B a A no está definido | A a B es navegable  B a A es navegable |
| Clase A x–x Clase B | A a B no es navegable  B a A no es navegable |  |  |

El modelo 1 hace que la navegabilidad sea completamente visible, pero tiende a saturar los diagramas. El modelo 2 se debería evitar porque oculta demasiada información.

El modelo 3 es la opción que se utiliza generalmente en la práctica porque no satura los diagramas, aunque tiene desventajas:

* No permite reconocer si la navegabilidad está presenta o todavía no está definida.
* Cambia el significado de la flecha navegable/indefinido a navegable/no navegable.
* No permite mostrar asociaciones que no son navegables en ninguna dirección.

**Asociaciones Reflexivas**

Asociación de una clase consigo misma. Los objetos de la clase tienen vínculos a otros objetos de la misma clase.

**Jerarquías de Objetos**

Una jerarquía tiene un objeto raíz y cualquier otro nodo en la jerarquía tiene exactamente un objeto directamente sobre él.

**Redes de Objetos**

No hay objeto raíz, aunque puede haberlo. Cada objeto puede tener muchos objetos directamente conectado a él. Cardinalidad: 0..\* a 0..\*

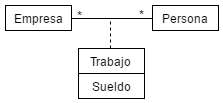
**Asociaciones y Atributos**

Una asociación entre una clase origen y una clase destino significa que los objetos de una clase origen pueden albergar una referencia de objeto a objetos de la clase destino.

El nombre del rol proporciona el nombre del atributo y la clase en el extremo de la asociación proporciona el nombre de la clase del atributo.



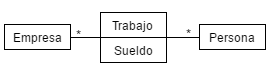
Consideraciones:

* Utilizar asociaciones sólo cuando la clase destino es una parte importante del modelo.
* Si la multiplicidad destino es exactamente 1, el objeto destino puede ser una parte de la fuente y por lo tanto no merece la pena mostrarse como una asociación. Se puede modelar como un atributo.
* Si la relación es bidireccional (1 a 1), se modela como atributo.

**Clases de asociación**

Surge cuando hay una relación de muchos a muchos entre dos clases con atributos que no se pueden acomodar en ninguna de dichas clases. La relación se simboliza con una línea de puntos desde la relación entre clases hacia la clase asociación.

Tener en cuenta:

* La clase asociación está compuesta por la clase, la asociación y la línea de puntos. Pueden tener atributos, operaciones y otras relaciones.
* Las instancias son vínculos que tienen atributos y operaciones.
* La identidad única de los vínculos está determinada por la identidad única de los objetos en cada extremo. Solamente se puede utilizar cuando existe un solo vínculo único entre dos objetos en cualquier momento en el tiempo.