INGENIERÍA DE SOFTWARE
Un enfoque crítico

"Probadlo todo y quedaos con lo bueno" (Pablo de Tarso)

Curso 2020-21

Guiones de clase

Tema I. INGENIERÍA DE REQUISITOS

1. Introducción a la ingeniería de requisitos

Qué es la Ingeniería de Requisitos

- Es la *rama de la ingeniería del software* que se ocupa de una de las primeras etapas en el proceso de desarrollo del software: la *comprensión de las necesidades* del cliente y su definición en forma de conjunto estructurado de requisitos que debe satisfacer un sistema informático.
- Peculiaridades de la ingeniería del software, frente a otras ramas de la ingeniería:
  - El "producto" software es inmaterial: transformar información en lugar de materia.
  - El beneficio ya no se obtiene por la repetición de un proceso material bien diseñado.
  - El producto software cambia a un ritmo vertiginoso: exige adecuada gestión del cambio.
- Se pueden distinguir *dos fases* en el proceso:
  - *Captura*: interacción cuidadosa con todos aquellos interesados en la aplicación o sistema informático; *adquisición de información*, "requisitos en bruto".
  - *Análisis*: estudio metódico de la información adquirida para lograr una verdadera *comprensión de* lo que debe hacer el sistema para satisfacer las necesidades del cliente; expresar los requisitos de forma concreta y detallada; formalización, refinamiento, estructuración, proceso eficazmente gestionado, "requisitos depurados".

Un caso práctico

- Agenda de compromisos: "Necesito algo para organizar mejor mis actividades, una agenda para llevar al día mi horario, mis compromisos, etc."
- ¿Cuánto se tardaría en desarrollar esta aplicación? "¡Esto me lo curro yo en una semanita!"
- En el momento de la entrega de la agenda el cliente no queda satisfecho. Quiere más, quiere otra cosa. Colores, sonidos, funciones, copiar de un día a otro… La divertida semanita se convierte en una pesada carga. ¿Hasta qué punto me he comprometido?
- Antes de entregarla, quiero probarla… ¿Qué pruebo? ¿Qué porcentaje de funcionalidad he alcanzado, cómo lo mido?
- La ingeniería del software no trata de "programar bien", sino de:
  - Cumplir plazos, presupuestos y expectativas.
  - Gestionar riesgos y recursos.
  - Transformar la producción artesanal en industrial.
- ¿Termina el ciclo de vida del software a la entrega del producto? Grave error: mantenimiento.

Definición de requisito

- Varias definiciones según IEEE:
  - Una condición o capacidad que *un usuario necesita* para resolver un problema o llegar a un objetivo.
  - Una condición o capacidad que *debe poseer un sistema* para satisfacer un contrato, un estándar, una especificación u otros documentos impuestos formalmente.
  - Una *representación documentada* de una condición o capacidad.
- Dos tipos principales de requisitos:
  - Requisitos de *capacidad (funcionales)*: funciones y operaciones requeridas por los clientes para resolver un problema o alcanzar un objetivo; describe una operación, o secuencia de operaciones, que el software debe ser capaz de realizar.
  - Requisitos de *restricción (no funcionales)*: restricciones impuestas por los clientes sobre la manera en que el problema es resuelto o el objetivo es alcanzado; restringe la manera en que el software es construido o funciona, sin alterar o describir las capacidades del software.

El documento de especificación de requisitos

- El *resultado del proceso* es el "documento de requisitos", que es uno de los productos (o *artefactos*) del proceso de desarrollo de software. Otros artefactos son: modelos de diseño, código fuente, pruebas, manuales...
- Es el único lugar donde se pone por escrito la naturaleza exacta de la aplicación.
- El nivel de detalle debe ser completo, pero no redundante.
- Los requisitos son la base para el diseño y la implementación.
- A cada requisito se le sigue la pista hasta la implementación.

Necesidad de la Ingeniería de Requisitos

- Para construir algo, antes hay que entender qué es ese "algo".
- Si la aplicación "funciona", pero no satisface las necesidades del cliente... ¿de qué sirve?
- En general, los requisitos expresan *qué* debe hacer una aplicación, sin decir *cómo* debe hacerlo: expresan el punto de vista del cliente, que sabe lo que quiere (en el mejor de los casos), pero no tiene por qué saber cómo conseguirlo:
  - A menudo el cliente ni siquiera sabe lo que quiere.
  - Tarea del analista es ayudar a que el cliente se exprese.
- Ejemplo:
  - El sistema permitirá al usuario consultar el saldo de su cuenta (sí).
  - Los saldos de clientes se almacenarán en una tabla llamada Saldo en una base de datos Access (no).
- Excepciones: puede ocurrir que usar Access sea un requisito:
  - Existen distintos niveles de abstracción en los requisitos, debido a que suelen provenir de distintas fuentes (distintos interlocutores con distinto nivel de conocimientos tecnológicos).
  - Ejemplo: el informático de la empresa nos habla de los sistemas con los que deberá interactuar el nuestro, mientras que el jefe de contabilidad nos proporciona una mejor visión sobre el conjunto de procesos del área.

Por qué es necesario *escribir* los requisitos

- Es obvio incluso para el novato, pero con demasiada frecuencia se ignora o se deja pasar.
- Tarea descuidada durante mucho tiempo: trivial, "literaria", poco tecnológica...
- ¿No quedan bien expresados en el código fuente? No, no funciona.
- Sin requisitos escritos, el equipo de desarrollo:
  - No sabe cuál es su objetivo.
  - No puede inspeccionar su trabajo.
  - No puede probarlo (¡y las pruebas sí se consideran esenciales!).
  - No puede analizar su productividad.
  - No puede reflexionar sobre sus prácticas.
  - No puede predecir el tamaño y esfuerzo del siguiente trabajo.
  - No puede satisfacer a sus clientes.
  - En resumen, no hay ingeniería profesional sin requisitos escritos.
- Una forma de NO escribirlos es no mantener las versiones de los requisitos, o no publicarlas.
- Cada uno de los requisitos debe ser...
  - Expresado con propiedad.
  - Fácilmente accesible para todo el personal involucrado.
  - Numerado de forma unívoca.
  - Acompañado por pruebas que lo verifiquen.
  - Tenido en cuenta en el diseño y el código.
  - Probado aisladamente y con otros requisitos.
  - Verificado por las pruebas al finalizar la construcción de la aplicación.
- Sin requisitos escritos sería imposible hacer todo esto.

The Chaos Report

- Estudios de Standish Group International
  - Exitoso: termina bien (tiempo, dinero, requisitos).
  - Completo pero deficiente: termina y funciona (+ tiempo, + dinero, – requisitos).
  - Cancelado: no termina.
- Conclusiones: la adecuada gestión de los requisitos es *el factor más importante de éxito* en un proyecto, por encima de las pruebas, el diseño, la programación...

Dificultades de la Ingeniería de Requisitos

- Obtener los requisitos correctos es un proceso difícil:
  - Adivinar los deseos y necesidades que habitualmente el cliente no es capaz de describir más que en forma confusa, incompleta y desordenada.
  - Las necesidades *se descubren,* los requisitos *se inventan*; los requisitos no están ahí esperando que alguien los descubra, sino que son creados, construidos o inventados en un proceso interactivo entre el cliente y el ingeniero.
  - La mayor parte de los *defectos en el software* entregado tienen su origen en el análisis de requisitos, y son en general los más difíciles de reparar.
  - El éxito en el producto requiere *colaboración y comunicación* fluida entre clientes y desarrolladores: cuanto más completo y menos ambiguo sea el conjunto de requisitos, más probabilidades de éxito.
- No es una tarea simple: organizar *todos* los requisitos bien detallados es una tarea difícil que requiere saber organizar personas y documentación. Anécdota de la NASA en 1999.
- El precio pagado: un defecto en los requisitos es 20-50 veces más caro de reparar si se desliza en el resto del proceso de desarrollo (aparte del daño que sufre el prestigio del desarrollador).
- La inversión es tanto más valiosa cuanto mayor sea el proyecto.
- Si el beneficio es tan grande, ¿por qué tan a menudo se omite el análisis de requisitos?
  - Los clientes normalmente no tienen claro lo que quieren al principio, sólo una idea vaga, y esto convierte el análisis de requisitos en una tarea más difícil.
  - Solución: sucesivas iteraciones Requisitos-Diseño-Implementación.
- Es una necesidad, no un lujo ("voy despacio porque tengo prisa", "no puedo permitirme el lujo de gastar tan poco").
- Muchas organizaciones no logran escribir los requisitos: esto no significa que no los usen, sino que *sólo existen en las mentes* de los ingenieros. Otro peligro: la *rotación de personal*. No es extraño que muchos proyectos nunca terminen.
- Un problema más sutil: *escribir sólo la primera versión* de los requisitos, pero sin mantenerlos al día en sucesivas iteraciones. Para poder actualizar el documento de requisitos es necesario que esté bien organizado. Gestión de requisitos cambiantes. Herramientas.
- *Escribir es pensar*. Escribir código prematuramente es como echar cemento sin saber cómo va a ser el puente. Documentar no es sólo registrar decisiones, es *pensar por escrito*.
- El rechazo o desgana para escribirlos no es porque sea una tarea trivial e inútil, sino porque es demasiado difícil. La profesionalidad lo exige.

La Ingeniería de Requisitos en el contexto del proyecto

- Los requisitos tienen *valor contractual*, establecen los límites de la aplicación.
- Pueden estar sujetos a normas de *confidencialidad* y propiedad intelectual.
- Son la base para los *estudios de viabilidad*: ¿merece la pena realizar el proyecto?
- La mejor comprensión de los requisitos facilita refinar las *estimaciones* de esfuerzo necesario.
- La obtención de requisitos afecta a la *planificación* del proyecto. El proceso puede ser iterativo, pero con ciertos límites: el cliente quiere saber el coste antes de empezar, el desarrollador no quiere comprometerse al menos hasta congelar los requisitos.

2. Obtención y descripción de requisitos

Las fuentes de los requisitos[ggf1]

- Restricciones[ggf2] naturales (físicas, químicas, biológicas, sociológicas...)
- Documentos (leyes, documentos organizativos de la empresa...)
- Personas: deseos y necesidades
- Expertos del dominio
- Observar cómo se usa el sistema actual, cómo funciona el negocio.
- Otras aplicaciones en uso en el dominio: reuso de requisitos[ggf3].

Plan de trabajo para obtener los requisitos [ggf4]

- Identificar al usuario (usuarios posibles, seleccionar entre ellos)
- Entrevistar al usuario (antes, planificar la entrevista)
- Escribir los requisitos del usuario (sucesivas versiones)
- Revisar los requisitos con el usuario
- Repetir los pasos hasta que el usuario firme el "conforme" del documento

Identificación de *stakeholders* (interesados: usuarios, clientes, propietarios, etc.) [ggf5]

- ¿Quiénes tienen interés en el producto? (Ejemplo: sitio web de comercio electrónico)
  - Los usuarios (pueden ser millones... y saber mucho más que nadie del tema)
  - Los propietarios
  - Los administradores[ggf6]
  - Incluso los desarrolladores (negociar requisitos para protegerse)
- ¿Quién es el cliente?
  - La persona que te contrata.
  - El departamento de Marketing, que diseña un producto para un cliente ideal.
  - Aplicaciones a medida, aplicaciones genéricas (*product lines*).
- Los distintos *interesados* pueden tener intereses contrapuestos y generar requisitos contradictorios (¿puede un profesor ver el expediente académico de un alumno?)
  - Ejemplo: El cliente de Aula Global es la Universidad, pero los usuarios principales son los profesores y alumnos, que no fueron consultados...
- Negociar el equilibrio requisitos-presupuesto-planificación: tarea del director del proyecto, que requiere habilidades de gestión, personales, negociadoras y políticas.
- El cliente confía en el ingeniero de requisitos para aclarar sus necesidades (como en un arquitecto para definir la casa que quiere construir).
- Trabajo conjunto para determinar los deseos del cliente: idea general, estudio del problema, descubrir matices, tomar decisiones clave.

El punto de vista del cliente[ggf7]

- El gran desafío es *averiguar y expresar claramente* lo que los clientes necesitan.
- A veces bastan las *palabras*, pero otras veces las *figuras* o tablas son de gran ayuda.
- El cliente suele tener una idea vaga, inconsciente e incompleta de lo que espera de su aplicación (*punto de vista del cliente*).
- Diferentes personas pueden concebir de modo muy distinto lo que implica un sistema.
- Ejemplo: una "aplicación meteorológica" puede significar:
  - Una utilidad para presentar gráficamente información recibida en bruto del servicio meteorológico.
  - Un sistema en tiempo real para predecir el tiempo.
  - Una aplicación para alertar a los usuarios de anomalías climáticas.

Cómo llevar adelante una entrevista [ggf8]

- La entrevista es un recurso caro:
  - Consume un tiempo significativo de más de una persona.
  - → Requiere una planificación cuidadosa.
- Es importante escuchar con atención, pero no basta con escuchar pasivamente:
  - Hay que saber preguntar, el cliente necesita ayuda.
  - Ganarse la confianza del cliente.
  - El resultado es producto del trabajo conjunto.
- Escribir los requisitos y validarlos por escrito; continuar las reuniones hasta lograr el acuerdo.
- *Antes de la entrevista:*
  - Enumerar y priorizar los entrevistados.
  - Planificar la hora de comienzo y finalización de cada entrevista.
  - Enumerar los puntos que es necesario tratar en la entrevista.
- *Durante la entrevista:*
  - Asistir al menos dos personas del equipo de analistas: es preferible que haya dos entrevistadores (uno pregunta, otro anota); uno solo puede verse desbordado.
  - Usar equipo de grabación (pedir permiso).
  - Concentrarse y escuchar.
  - No ser pasivo: preguntar y animar.
  - Insistir hasta entender los deseos y las necesidades.
  - Utilizar diagramas (si son útiles y según la formación de los entrevistados).
  - Tomar notas detalladas.
  - Concretar la siguiente entrevista.
- *Después de la entrevista:*
  - Redactar el borrador de requisitos.
  - Revisar entre los dos entrevistadores.
  - Enviar a los clientes para comentar y aprobar.

Técnicas para la *obtención* y *descripción* de requisitos del usuario[ggf9]

- Textuales: relativamente accesibles a un cliente sin formación específica
  - Texto en prosa común y corriente, tablas, etc.
  - Texto estructurado, lenguaje técnico
  - Casos de uso
    - Describir los *requisitos* como *interacciones* entre la aplicación y agentes externos.
    - Expresan el *punto de vista del usuario* de cómo debe funcionar la aplicación.
    - Más adecuados para obtener requisitos funcionales que no-funcionales.
    - Alternativa más sencilla[ggf10]: tabla de roles de usuario y servicios/funciones requeridos por cada uno.
- Gráficas: requieren un cierto grado de formación técnica en el cliente; tienen el peligro de convertir el análisis de requisitos en diseño de la aplicación
  - Diagramas de flujo de datos
    - Para requisitos que se describan de modo *natural* como un flujo de datos (flechas) entre elementos de procesamiento (nodos).
    - La utilidad de los DFD's depende del tipo de aplicación.
  - Diagramas de actividad
    - El énfasis no se pone en el flujo de datos sino en el *flujo de control* entre acciones.
  - Diagramas de estados
    - Dividir la aplicación en estados, de modo que el sistema siempre se encuentre exactamente en uno de ellos.
    - Útiles para *aplicaciones dirigidas por eventos.*
- Interfaces de usuario y prototipos: no confundir con diseño, valorar la inversión

Interfaces de usuario[ggf11]

- Diseño de la interfaz de usuario: ¿requisitos o diseño?
- Visualizar la GUI ayuda a los clientes a concebir y describir la aplicación.
- Al ver los bocetos, el cliente se da cuenta de que *necesita más o quiere algo diferente.*
- Tarea de un profesional especializado (especialmente el diseño detallado).
- Once pasos para desarrollar la GUI (ver Braude, pp. 151-157).

Prototipos[ggf12]

- Prototipado rápido:
  - Implementación parcial, con un componente GUI importante.
  - Muy útil para extraer requisitos del cliente y para identificar, y tal vez eliminar, partes arriesgadas de un proyecto.
  - Una comprensión más profunda ahorra trabajos futuros de corrección.
- El beneficio del prototipo depende de su coste y de su valor para el proyecto.
- ¿Debe transformarse el prototipo en la aplicación? Decisión planificada, no accidental, porque, en caso contrario, puede darse un empeoramiento de calidad.
  - Contruidos rápidamente, mal documentados (no será necesario mantenerlos)…
  - Implementados en lenguajes que rápidamente obtienen resultados, pero tal vez inadecuados para la aplicación final: seguridad, fiabilidad...
  - Desventaja: el cliente puede impresionarse y pensar que estamos cerca del final.
  - ¿Transformarías un prototipo de casa con balas de paja en la casa final?
- Beneficios: extraer requisitos, ensayar soluciones[ggf14].

Cómo escribir buenos requisitos
Lectura recomendada

- I. Alexander, R. Stevens. *Writing Better Requirements*. Addison-Wesley, 2002.

Cómo debería ser una especificación de requisitos

- Las 7 características que debe tener una buena especificación de requisitos (IEEE).
- Los 7 pecados del especificador (Bertrand Meyer).
- Especificación inteligente de requisitos

Claves para lograr una buena especificación de requisitos

- Correctamente estructurada
- Completa
- Consistente
- Estilo literario adecuado: una especificación de requisitos no es una novela
- Indicadores de calidad: medidas automáticas que ayudan a mejorar la calidad

Errores típicos

- Modo condicional
- Detalles de diseño
- Opcionalidad
- Atomicidad
- Acrónimos y vaguedad
- Puntuación y legibilidad
- Pronombres
- Pseudocódigo
- Número de términos
- Subjetividad
- Negaciones
- Falta de precisión

3. Propiedades, atributos y organización de los requisitos

PROPIEDADES deseables de los requisitos del software

- Propiedades que deben tener *todos los requisitos* para estar bien especificados.
- Al *inspeccionar* los requisitos deben cuestionarse estas propiedades.
  - Utilizar *cuestionarios de validación* para inspeccionar requisitos.
- Dos tipos:
  - Propiedades *globales*: completitud, consistencia...
  - Propiedades *individuales*: tamaño, claridad, comprobabilidad, condiciones de error…
- No confundir con los *atributos* de los requisitos: toman un valor distinto en cada caso.
  - Necesidad, prioridad, estabilidad, riesgo…

Completitud [ggf16]

- Significa que *no hay omisiones* que comprometan la integridad de los requisitos.
  - No faltan requisitos (propiedad global).
  - No faltan detalles en la especificación de cada requisito (propiedad individual).
- Es una propiedad *difícil de determinar* (tan sólo podemos alcanzar una aproximación).
  - Contrastar con el cliente.
  - Comparar con proyectos semejantes.
- Buscar la *visión de conjunto*, detectar huecos o partes infra-especificadas.
  - Para cada requisito, comprobar que están presentes los demás requisitos relacionados.
- La buena *organización* facilita la detección de faltas.
  - Ejemplo: organización por tipos de requisitos.
- Técnicas estadísticas para estimar el número de requisitos aún no descubiertos.
  - Ritmo temporal de creación/modificación de requisitos.
  - Ajustar la "nube de puntos" (tiempo-requisitos conocidos) a una curva monótona creciente acotada (¿hipérbola?).
  - Dificultad: estas técnicas se ven afectadas por el propio ritmo de trabajo de los analistas.

Consistencia (o coherencia)[ggf17]

- Significa que *no hay contradicciones* entre requisitos.
- Contradicción ≠ Ambigüedad, pero las ambigüedades difultan detectar contradicciones.
- Es más difícil de comprobar si el número de requisitos crece.
- Una buena organización facilita la detección de contradicciones.
- Ejemplo: matriz de referencias cruzadas, que además facilita la detección de requisitos afectados por la modificación de uno dado:
  - Conflicto: contradicción, no se pueden satisfacer simultáneamente.
  - Acoplamiento[ggf19]: hablan de lo mismo (si cambia uno, puede afectar al otro).
  - Redundancia: dicen lo mismo (sobra uno de los dos).
  - Independencia.
- En la versión final no puede haber Conflictos ni Redundancias, sí Acoplamientos.

Tamaño adecuado[ggf20]

- Para manejar con mayor facilidad un requisito, deberá tener un *tamaño adecuado*:
  - Ni tan grande que sea inmanejable.
  - Ni tan pequeño que no valga la pena seguirle la pista por separado.
- Es posible aplicar los principios de *modularidad* y *anidamiento* a los requisitos.
  - Paquetes y subpaquetes [ggf21]de requisitos (áreas temáticas).
  - Requisitos y subrequisitos.
- Nivel de *granularidad*: la misma cantidad de información puede repartirse en un número grande/pequeño de requisitos (grano fino/grueso).

- Nivel de *detalle*: los requisitos contienen más detalles, globalmente hay más información.

Claridad

- Significa que *no hay ambigüedad* en la especificación de cada requisito.
- Utilizar un *vocabulario controlado*, y tabla de términos equivalentes (*sinónimos*).
- Para cualquier labor de escritura:
  - Tenga siempre a mano *diccionarios* (normal, sinónimos, estilo, idiomas, corrector ortográfico y sintáctico).
  - Escribir, corregir, escribir, corregir… y hacerlo *entre varios* (uno escribe, otro corrige).
  - Respetar *normas* ortográficas, sintácticas, gramaticales, estilísticas… no es un capricho: lo que no está bien escrito no se entiende.
  - Estructurar bien, proceder con orden, proporcionar las referencias necesarias.
  - Sintetizar, resaltar ideas importantes, resaltar más lo menos obvio.

Comprobabilidad[ggf22]

- Incluye dos tipos distintos de defectos que se desea *descubrir y eliminar*:
  - *Validación*: defectos de interpretación (*do the right thing*) – taburete en lugar de mesa.
  - *Verificación*: defectos de implementación (*do the thing right*) – mesa coja.
- Muchos defectos se pueden descubrir y eliminar mediante *pruebas*, pero salvo que se usen métodos formales, las pruebas no garantizan que *todos* los defectos desaparecen.
- Atención: las pruebas de verificación *no sirven* como pruebas de validación – la mesa.
- La validación requiere interacción con el cliente: pruebas con prototipos, etc.
- Para validar y verificar un requisito es necesario que éste sea comprobable (*testable*). Un requisito no comprobable o ambiguo tiene escaso valor y debe ser rechazado.
- Ejemplos:
  - El sistema mostrará la diferencia entre el valor observado y la media mundial (*no comprobable*).
  - El sistema mostrará la diferencia entre el valor observado y el valor publicado por Naciones Unidas (¿cuándo? todavía es *ambiguo*).
  - El sistema mostrará la diferencia entre el valor observado y el último valor publicado por Naciones Unidas en su página web.
- Conviene asegurar que no es ambiguo, ni siquiera si se interpretase mal a propósito.
- Esbozar una prueba específica que establecería la verificación del requisito.
- La definición de pruebas ayuda a clarificar el sentido de cada requisito.

Condiciones de error [ggf23]

- Para estar completa, la especificación del requisito debe tener en cuenta las condiciones de error, es decir, qué ocurre con el requisito en una situación con errores.
- Las condiciones de error son especialmente importantes para realizar las pruebas, ya que al probar un requisito se fuerzan condiciones de error, y es necesario prever qué debe ocurrir en estos casos.
- Caso típico: datos de entrada incorrectos.
  - *Ejemplo*: clasificar un triángulo a partir de las coordenadas de sus tres vértices.
- No confiar en que no se producirán condiciones de error: esto aumenta la dependencia entre módulos (un módulo confía en la detección de errores de otro módulo).
  - Pero recordar que no se debe abusar del manejo de errores internos.
- Redundancia para promover la seguridad.
- Determinar lo que hay que hacer en situaciones no previstas (*prever lo imprevisto*).
- Prevenir posibles errores de programación en lugares críticos.

ATRIBUTOS definibles en cada requisito[ggf24]

- Los atributos son los "campos" de la "plantilla[ggf25]" usada para los requisitos.
- El número y tipo de atributos dependen del proyecto concreto y de la organización.
- Ejemplo (si se usa una herramienta de gestión de requisitos):
  - Atributos automáticos: identificador, creador, fecha de creación…
  - Atributos obligatorios en todo proyecto: tipo, estado, descripción breve.
  - Atributos opcionales, dependiendo del proyecto u organización: descripción detallada, fuente, necesidad, prioridad, estabilidad, complejidad, riesgo, coste, pruebas, errores…
- Uso y abuso del identificador de requisito (único y permanente para cada requisito):
  - Mejor si no significa nada, ni siquiera el tipo de requisito. Evitar "códigos parlantes".
  - No preocuparse por los "huecos" o el "desorden" en la secuencia de requisitos.
  - El orden de los requisitos viene dado por su organización lógica, no por su ID.

Necesidad y prioridad[ggf26]

- Para negociar entre funcionalidad, planificación, presupuesto y nivel de calidad, es conveniente que los requisitos funcionales tengan asignado un nivel de *necesidad* (tres o cuatro categorías: *esencial, deseable, opcional...*), así como un nivel de *prioridad* temporal (numerados 1, 2, 3… el número más bajo indica mayor prioridad).
- La necesidad de un requisito hace referencia al *interés* de los usuarios/clientes en que la aplicación lo realice, y hasta qué punto estarían dispuestos a pasarse si él.
- La prioridad de un requisito hace referencia al *orden temporal*: indica en qué fase de construcción del sistema se incluirá la funcionalidad que realice el requisito.
- Si el 20% de la aplicación proporciona el 80% de su valor... no más del 20% de los requisitos deberían ser "esenciales".

Estabilidad[ggf27] **y riesgo**

- Un requisito no estable o variable es aquel que está sujeto a *posibles modificaciones*, ya sea por el propio cliente o bien por otros factores que pueden afectar a nuestra aplicación.
- Tener en cuenta esta posibilidad es fundamental para *reducir riesgos*: prever la posibilidad de que un requisito cambie desde las fases iniciales del proyecto ayudará a tomar las medidas oportunas para evitar costes innecesarios en caso de que finalmente cambie.
- Pero cuidado: muchos requisitos del cliente pueden cambiar, sin embargo se trata de marcar como inestables sólo aquellos en los que hemos acordado con el cliente esta posibilidad, porque ya desde el principio es previsible la inestabilidad (porque el cliente no acaba de decidirse, porque depende de una Ley que se sabe va a cambiar próximamente, etc.).
- Cada requisito puede ir acompañado de una estimación del riesgo que supone realizarlo (relacionado con la dificultad de implementarlo). Ej.: *alto, moderado, bajo.*
- Seguir la pista con especial atención a los de mayor riesgo.

Para escribir un requisito: resumen

- Enunciado breve del mismo, numerado.
- Explicación detallada.
- Atributos y propiedades tal como se han discutido hasta aquí, por ejemplo:
  - Necesidad, Prioridad y Riesgo.
  - Condiciones de error.
  - Pruebas de verificación requeridas: *en este curso son obligatorias*.
- Matrices de trazabilidad hacia requisitos de usuario y hacia diseño/implementación.
- Matriz de referencias cruzadas entre requisitos: conflicto, acoplamiento, redundancia.
- Usar un formato que permita distintos *niveles de visibilidad:*
  - Sólo enunciado
  - Sólo enunciado y descripción
  - Enunciado, descripción y propiedades...

ORGANIZACIÓN – Métodos para organizar los requisitos del software[ggf28]

- Una buena organización de los requisitos del software es esencial, porque si no están organizados no se pueden manejar (localizar, actualizar, aplicar, comprobar...).
- Técnicas ya mencionadas:
  - Matriz de trazabilidad hacia requisitos de usuario (o hacia diseño, en ADD/DDD).
  - Matriz de referencias cruzadas (consistencia): conflicto, acoplamiento, redundancia.
  - Principios de *modularidad* y *anidamiento* en los requisitos (áreas temáticas).
- Organizarlos tomando como pauta el *modelo del sistema* (ver la nomenclatura de modelos).
- Utilizar tablas para representar explícitamente las relaciones entre los requisitos y el modelo, y entre distintas partes del modelo, por ejemplo:
  - Casos de uso – requisitos derivados del caso de uso
  - Requisitos – clases derivadas del requisito
  - Casos de uso – clases mencionadas en el caso de uso

Organización de los requisitos según el modelo de casos de uso [ggf29]

- Estructurar los requisitos en torno a los *servicios requeridos por los usuarios*, descritos como secuencias de operaciones. Cada caso de uso agrupa:
  - Un requisito principal descrito en el *objetivo* del caso de uso
  - Uno o varios requisitos subordinados, que estarán relacionados con:
    - Las *clases* de objetos que se mencionan (elementos de información).
    - Las *operaciones* individuales de que consta el caso de uso.
- *Ejemplo*: caso de uso "alquilar una película de videoclub":
  - Las películas alquilables tienen título, director, actores, duración...
  - Las películas alquilables son ordenables o filtrables según distintos criterios...

Organización de los requisitos según el modelo conceptual (clases)

- Estructurar los requisitos en *unidades atómicas de información y comportamiento*.
- Las clases resultantes de la organización de los requisitos pueden tener mucho o nada que ver con las clases del diseño y la implementación. Si la relación es estrecha...
  - *Ventajas*: trazabilidad, una de las banderas de la OO; las clases próximas a conceptos del dominio es más probable que sean reutilizadas.
  - *Desventajas*[ggf30]: acoplamiento análisis-diseño, selección de clases de diseño antes de tiempo, simular innecesariamente la estructura del entorno en el diseño.
- Procedimiento:
  - Identificar y enumerar las *clases conceptuales*: las que mencionan los requisitos.
  - Para cada clase, describir la *funcionalidad requerida* de la aplicación que pertenezca principalmente a esa clase, en forma de atributos y operaciones:
    - Atributos: unidades de información.
    - Operaciones: unidades de comportamiento, reacciones frente a eventos (atención: es mucho más fácil repartir los atributos que repartir las operaciones).

Ciclo de vida de los requisitos[ggf31] **(estados por los que pueden pasar)**

- *Propuesto*: el ingeniero de requisitos lo ha propuesto (podría borrarse si no se llega a aceptar).
- *Validado*: el cliente lo ha aceptado y validado.
- *Cancelado*: el cliente ha cancelado el requisito, ya no se debe implementar (un requisito que ha llegado a ser Validado no debe ser borrado sin más).
- *Implementado*: el equipo de desarrollo lo ha implementado.
- *Verificado*: el equipo de pruebas ha verificado que se satisface correctamente.

- Variantes (dentro de una organización, proyecto, etc.):
  - Ningún requisito Propuesto puede ser eliminado, sino en todo caso Cancelado.
  - Modificación de requisitos: transiciones desde cualquier estado hacia Propuesto.

Uso de herramientas para el análisis de requisitos[ggf32]

- Las herramientas pueden ayudar a...
  - Capturar y gestionar los requisitos: ordenación, priorización, asignación, seguimiento…
  - Valorar el estado del análisis de requisitos.
- Son especialmente útiles los hipervínculos:
  - Desde requisitos hacia otros requisitos (trazabilidad, consistencia).
  - Desde requisitos hacia otros documentos relevantes del proyecto (modelos, diseño).
  - Desde el código hacia los requisitos que implementa.

Características que debería tener una buena herramienta de gestión de requisitos[ggf33]

- Se explican sólo las menos obvias.
- *Atributos habilitados en función del tipo de requisito, valores desplegables.*
  - Una herramienta genérica proporciona más atributos de los que son necesarios en cada proyecto. Para guiar a los analistas, el jefe de proyecto debe poder determinar los atributos que hay que rellenar en un proyecto concreto.
  - Además, los atributos de un requisito pueden ser distintos por tipo de requisito; por ejemplo, evaluar el Riesgo que conlleva implementar un requisito puede ser importante para requisitos de rendimiento, pero no para los requisitos de información.
  - Los valores desplegables deben ser configurables por el jefe de proyecto; por ejemplo, en un proyecto los niveles de Necesidad pueden ser tres, en otro pueden ser cinco.
- *Control de versiones: control obligatorio, control opcional, sin control.*
  - Cada vez que se modifica un requisito se guarda su versión, o se pregunta si se desea guardar (por ejemplo, son modificaciones irrelevantes), o no se guarda (por ejemplo, todavía estamos en fase de elaboración y no tiene sentido guardar versiones anteriores).
- *Notificación de modificaciones (opcional).*
  - Cada vez que se modifica un requisito se notifica al creador (o al implementador). Intenta resolver este problema: "¿Quién ha cambiado lo que yo había escrito?"
- *Seguimiento del ciclo de vida de los requisitos.*
  - Porcentaje de requisitos en cada uno de los estados.
- *Relaciones entre requisitos: navegabilidad y asimetría. Requisitos "sospechosos".*
  - Las relaciones deben ser navegables en ambos sentidos: debo poder ir desde el padre al hijo y viceversa. En cambio, algunas relaciones son asimétricas (traza y subordinación jerárquica), mientras que otras son simétricas (acoplamiento, conflicto, redundancia).
  - El concepto de requisito sospechoso, o sucio, indica que si se modifica un requisito, se marcarán como sospechosos todos aquellos requisitos que estén relacionados con el requisito modificado, con el fin de identificarlos más fácilmente y someterlos a revisión. La "suciedad" se transmite en la dirección marcada por la asimetría de la relación, o en ambos sentidos si la relación en cuestión es simétrica.
- *Utilidades.* Las cuatro mencionadas requieren técnicas de procesamiento lingüístico.
  - El glosario se puede crear identificando los términos significativos, por ejemplo en función de su frecuencia de aparición (ni muy frecuentes, ni muy escasos).
  - La coherencia requisitos-modelo exige que los términos del glosario, que han sido extraídos de los requisitos, estén reflejados en el modelo.
  - Detección automática de solapamientos mediante la coincidencia de términos: si dos requisitos contienen los mismos términos, probablemente hay algún tipo de solapamiento (acoplamiento, conflicto, redundancia).
  - Métricas de calidad obtenidas midiendo propiedades lingüísticas del texto.

CALIDAD – Qué sentido tiene realizar medidas de calidad en los requisitos del software[ggf34]

- Los requisitos deben cumplir determinados criterios de calidad (*cuantificables*).
- Si no se exige que los requisitos cumplan los criterios de calidad, entonces más adelante no se podrá comprobar que la aplicación satisface estos mismos criterios (es decir, no se podrá *buscar la calidad en fases posteriores* del proyecto).
- La calidad de los requisitos debe ser *medida por personas distintas* de los autores.
- Toda medición (o métrica) tiene un *coste añadido* en dinero y tiempo para obtenerla, almacenarla, analizarla e informar acerca de ella.
  - Se trata de optimizar la relación coste/beneficio, lo cual depende de muchos factores: cultura de la organización, estado del proyecto, naturaleza del proyecto, etc.
  - No se trata de obtener mediciones sin saber para qué.
  - Idea general: clasificar las mediciones en tres categorías (*bien, regular, mal*).
- Las medidas son más útiles cuando sus valores esperados se especifican por adelantado.
  - Ejemplo: basados en experiencias anteriores, los requisitos se consideran "completos" cuando la tasa de creación y modificación es menor del 1% semanal.

Métricas de calidad: cómo de bien están escritos los requisitos [ggf35]

- ¿Qué debemos medir?
  - Propiedades deseables (cualitativas) – taxonomía de propiedades.
  - Indicadores medibles (cuantitativos) – ¿cómo se relacionan con las propiedades?
- ¿Cómo podemos medir?
  - Medidas manuales, basadas en la inspección directa con ayuda de cuestionarios.
    - Completitud global: ¿están todos los requisitos?
    - Organización: ¿están bien clasificados los requisitos, están relacionados con casos de uso o clases incorrectos?
    - Completitud individual: ¿está completo cada requisito, con todos sus atributos?
    - Consistencia: ¿hay problemas de consistencia entre requisitos?
  - Medidas automáticas, basadas en propiedades lingüísticas del texto de los requisitos.
    - Indicadores morfológicos: caracterísiticas puramente formales del texto.
    - Indicadores léxicos: comparación con listas predefinidas de términos.
    - Indicadores analíticos: requieren análisis lingüístico del texto.
    - Indicadores relacionales: requieren información relacional, no sólo textual.

Métricas de calidad: cómo de efectivos son los procesos[ggf36]

- Medidas de la efectividad de la inspección de requisitos:
  - Número de requisitos que faltan o son defectuosos, encontrados por hora de inspección.
- Medidas de la efectividad del proceso de análisis de requisitos:
  - Coste por requisito específico:
    - Coste medio total (tiempo total / número de requisitos).
    - Coste marginal (coste de obtener uno más).
    - Ritmo al que los requisitos son: modificados / eliminados / añadidos.

4. Tipos de requisitos

Dos niveles en los requisitos[ggf37], ventajas y desventajas de distinguirlos

- ¿Del cliente o del sistema, del usuario o del software?
  - Sería más correcto decir: requisitos *del* software *para* el cliente / *para* el desarrollador.
  - Diferencia: punto de vista del cliente / del desarrollador.
- Primer nivel: *requisitos del usuario* (o del cliente):
  - Deseos y necesidades del cliente, expresados en lenguaje comprensible por él.
- Segundo nivel: *requisitos del software* (o del sistema, del desarrollador, detallados...):
  - Forma estructurada y específica, carácter mucho más técnico, modelos.
  - Audiencia primaria, el desarrollador.
  - El cliente puede estar interesado en ellos e incluso entenderlos y hacer observaciones.
- La finalidad última es la misma. La distinción entre los dos niveles no es muy clara:
  - Forma: no estructurada / estructurada.
  - Audiencia: cliente / desarrollador.
  - Contenido: mayor o menor nivel de detalle, precisión y formalidad.
  - Texto / Texto + Diagramas (modelos).
  - Requisitos en bruto → Requisitos depurados.
- Distintas nomenclaturas y clasificaciones, misma idea de fondo:
  - Clásica/IEEE: una única fase, un único documento.
  - USDP/ESA: dos fases (captura y análisis), dos documentos.

Clasificación de requisitos del software[ggf40]

- *Requisitos negativos*[ggf41]: lo que la aplicación no debe o no necesita hacer (son infinitos...).
  - Pueden aclarar posibles malentendidos, el alcance del sistema.
  - Seleccionar aquellos que sirven para aclarar los verdaderos requisitos.
  - Ejemplo: la aplicación no *asesorará* al usuario sobre...
  - ¿Dónde? Alcance del software. Anotación a un requisito concreto.
- *Requisitos funcionales* (derivados de los requisitos de capacidad – *decir el qué*).
  - Especifican la funcionalidad o *servicios* que la aplicación debe proporcionar.
    - Ejemplo: un tiempo de respuesta no es un requisito funcional.
  - Acompañados de un *modelo del sistema*.
    - *Modelo conceptual*: requisitos de información.
    - *Modelo de casos de uso*: requisitos de operación.
    - Modelos de comportamiento que sean necesarios.
- *Requisitos no funcionales* [ggf42](derivados de los requisitos de restricción – *restringir el cómo*).
  - Imponen *restricciones*: en el producto desarrollado, en el proceso de desarrollo.
  - Características distintivas:
    - *Cómo* debe realizar algo el software, NO *qué* debe realizar.
    - Requieren *operacionalización*, pero eso no significa que sean funcionales.
    - Cortan *transversalmente* a los requisitos funcionales.
    - Son *negociables* hasta cierto punto, dentro de límites aceptables.
    - La *medida de satisfacción* (o verificación) no es todo/nada, sino gradual.
  - A veces denominados requisitos de calidad, o requisitos de diseño.
    - "La funcionalidad se presupone, la calidad satisface (o no)".
  - Habitualmente peor analizados y documentados.
    - Descritos de modo vago, informal, ambiguo (dificultan la verificación).

        - Repetidos en muchos lugares (transversalidad).
        - Peor contemplados en los modelos.
    - Ejemplos: portabilidad, mantenibilidad, modificabilidad, reusabilidad, amigabilidad, uso de estándares, verificación... (y los de la página siguiente).

Consumo de recursos (*Resource expenditure*)

- Especifican la cantidad de recursos que la aplicación requiere.
- Ejemplos:
  - Uso de memoria (volátil / permanente; o primaria / secundaria).
  - Capacidad de tráfico, líneas de atención simultánea, etc.

Rendimiento (*Performance*)[ggf43]

- Especifican restricciones temporales que la aplicación debe cumplir.
- Son críticas en los sistemas de tiempo real.
  - Ejemplos: velocidad, tiempo de respuesta, etc.

Fiabilidad y disponibilidad (*Reliability and availability*)[ggf44]

- En estos dos factores se reconoce que las aplicaciones difícilmente son perfectas, pero sí se exige en ellas un nivel de calidad mínimo y cuantificable.
- Fiabilidad: cuantifica los *errores* permisibles y su gravedad.
  - Ejemplo: "el sistema no sufrirá más de dos fallos de nivel uno al mes".
- Disponibilidad: cuantifica el tiempo en que la aplicación estará *operativa* para los usuarios.
  - Ejemplo: "la aplicación no estará operativa como máximo durante 10 horas en cualquier periodo de 30 días, y como máximo durante 2 horas seguidas".

Manejo de errores (*Error handling*)[ggf45]

- Cómo debe responder la aplicación a *errores de su entorno*, o a *errores internos.*
  - Ejemplo: cómo reaccionar ante un mensaje en formato no reconocido.
- No se debe abusar del manejo de errores internos, que no deben existir (no debemos cubrir nuestros errores con un código interminable de manejo de errores).
  - Ejemplo: determinar lo que hay que hacer si una función es llamada con parámetros equivocados; esto sólo se debe hacer si es preferible manejar el error a la terminación de la aplicación.

Requisitos de interfaz (*Interface requirements*)[ggf46]

- Describen el formato con el que la aplicación se comunica con su entorno.
  - Ejemplo (*comunicación con usuarios*): El coste del envío se mostrará en la esquina inferior derecha.
  - Ejemplo (*comunicación con otras aplicaciones*): Los pedidos se transmitirán en formato XML utilizando la plantilla DTD detallada en el anexo IV.

Restricciones (*Constraints*)[ggf47]

- Describen límites o condiciones sobre cómo diseñar o implementar la aplicación.
- Estos requisitos no deben suplantar el proceso de diseño, simplemente especifican condiciones impuestas en el proyecto por el cliente, el entorno, u otras circunstancias.
- Ejemplos:
  - Exactitud, precisión: las tarifas aplicadas se medirán en diezmilésimas de euro[ggf48].
  - Lenguaje, herramienta, plataforma: se empleará el lenguaje Fortran, el gestor de base de datos SQLServer, la aplicación deberá funcionar sobre Windows 95.
  - Arquitectura: se emplearán tecnologías de intranet y cliente-servidor.
  - Estándares: los informes generados se ajustarán al estándar XX-123.

Seguridad (*Security / Safety*)

- *Security*: seguridad del sistema frente a amenazas externas (confidencialidad, integridad, disponibilidad, etc.).
- *Safety*: seguridad de las personas frente a fallos del sistema.

Trazabilidad *hacia atrás*: RS ← RU (u otras fuentes)

- La relación es típicamente *uno-a-pocos*. Todo RU debe ser desarrollado por al menos un RS, pero puede haber un RS cuya fuente no sea un RU (PSS-05-03, pp. 3 y 47):
- No confundir con el caso de nuevos RU que se descubren durante el desarrollo de los RS, y que deberán ser validados por el usuario: nueva versión URD.
- Verdadero RS sin RU: todo RS debe tener una buena razón para existir, con mayor razón si no existe su correspondiente RU: el cliente no querrá pagar por cosas que no ha solicitado.
- Es más fácil que ocurra con NFRs: requisitos impuestos por el analista al diseñador.
- Son muy útiles las matrices de trazabilidad de requisitos:
  - Dos matrices: hacia atrás en el URD/SRD; hacia delante en el ADD/DDD.
  - No confundir con la matriz de referencias cruzadas entre requisitos (Unidad 8), que se usa para gestionar la consistencia entre ellos (conflictos, acoplamientos y redundancias).
  - Evitar matrices de doble entrada (demasiado dispersas). Usar matrices de tres columnas.

Trazabilidad *hacia delante*: requisitos → diseño / implementación [ggf50]

- Una forma de conseguirla es relacionar cada requisito funcional con una función específica en el lenguaje destino.
  - Esta técnica es poco generalizable y limitada; produce excesivo acoplamiento entre la estructura de los requisitos y la estructura de la implementación.
  - En OO *no es trivial* preguntarse qué clase es responsable de qué función:
    - Función con un solo parámetro: F(x) → x.F( )
    - Función con dos o más parámetros: G(x, y) → ¿x.G(y) / y.G(x)?
  - La transición Análisis-Diseño no es sencilla ni tiene por qué serlo.
- La trazabilidad es necesaria para poder *comprobar* que la aplicación satisface los requisitos, para afrontar que los requisitos cambian, y para gestionar su *evolución*:
  - Si la gestión de la trazabilidad es difícil y lleva demasiado trabajo, los desarrolladores tenderán a evitar la actualización del documento de requisitos e irán directamente a hacer cambios al código: en consecuencia el documento de requisitos se hace inútil para las pruebas y la verificación.
  - Si además el documento no es fiable por culpa de algunos miembros del equipo menos responsables, entonces ni siquiera los más responsables se tomarán la molestia de mantenerlo al día.
  - Por tanto, el sistema para hacer corresponder requisitos con diseño y código debe ser claro y concreto, y en lo posible automatizado con herramientas adecuadas.
- Un requisito puede corresponderse con varias partes del código (clases, funciones), que a su vez pueden implementar varios requisitos cada una:
  - Por tanto, antes de cambiar el código para adaptarse a un cambio en un requisito, hay que comprobar que otros requisitos no quedan comprometidos.
  - La correspondencia Requisito-Función es muchos-a-muchos; si no puede ser *uno-uno*, al menos puede intentarse que sea *pocos-pocos*.
  - El *diseño* juega un papel fundamental en establecer esta correspondencia.

Trazabilidad de requisitos no funcionales [ggf51]

- La correspondencia con elementos del diseño y de la implementación es mucho más difícil, ya que depende en mayor medida de la *colaboración entre varios elementos* (emergencia).
- La *verificación* de estos requisitos es más complicada, normalmente requiere un mayor nivel de integración del sistema (no es fácil verificar NFRs en componentes aislados).
- Ejemplo: *rendimiento*.
  - Identificar los elementos que consumen más tiempo o memoria.
  - La mayor parte de los recursos son consumidos por un número reducido de elementos, de modo que esta tarea es provechosa para mejorar el rendimiento.
  - Bucles, gráficos y comunicaciones suelen ser los que más tiempo consumen.

Tema II. MODELADO CONCEPTUAL

5. Introducción al modelado conceptual

La necesidad de modelar

- Analogía arquitectónica
- Comunicación y representación del conocimiento
- Qué es un modelo. Propiedades deseables. Sistema, modelo, diagrama
- Herramientas de modelado

Qué significa "modelo conceptual" (modelo especificativo lógico del sistema)

- Qué NO representa el modelo conceptual:
  - Mundo real o "universo del discurso" → modelo de dominio.
  - Implementación o "vista tecnológica" → modelo de diseño.
- Qué representa ("modelo del sistema"):
  - Vista conceptual del sistema: conceptos que debe manejar/implementar el sistema.
  - Abarca aspectos estáticos (estructura) y dinámicos (comportamiento).
- Cómo lo representa ("modelo lógico"):
  - Omite la implementación: sin "tecnología", sin artefactos de diseño.
  - Recoge el vocabulario del dominio (a partir del "modelo de dominio[ggf53]").

6. Modelado conceptual (1)

Objetos y clases

- Dos niveles de abstracción. La relación de clasificación / instanciación. En análisis y diseño.
- No al "realismo ingenuo": los objetos no vienen con una etiqueta que nos indica a qué clase pertenecen. Clasificar es una tarea creativa.
- Notación básica de objetos y clases: supresión opcional de compartimentos.
- Tipos de clases según los objetos representados[ggf54]: físicos, lógicos, históricos…
- Caso especial: objetos que representan una colección, familia o tipo de cosas. Un mismo concepto del mundo real puede ser modelado como objeto o clase según el contexto:
  - "Pastor Alemán":
    - clase cuyas instancias son Friedrich, Wolfgang y Heinz (nombres de perros).
    - objeto que es instancia de la clase "Raza Perruna".
  - "Lata de Atún":
    - clase cuyas instancias son lata1, lata2... (todas ellas latas de atún "tangibles").
    - objeto que es instancia de la clase "Productos que vendo en mi supermercado".
  - "El Lenguaje Unificado de Modelado":
    - clase cuyas instancias son los diversos ejemplares del libro en la Biblioteca.
    - objeto que es instancia de la clase "Títulos de libros registrados en la biblioteca".
- Cómo decidir sin un concepto debe modelarse como clase o como tipo de dato[ggf56].
  - Un concepto será modelado como clase o como tipo de dato dependiendo del sistema en el que sea usado, por tanto *la diferencia es relativa* al sistema en cuestión.

Atributos

- Definición: propiedad compartida por los objetos de una clase.
- Atributos derivados.
- Notación: definida en UML, pero más importante ajustarse al lenguaje de implementación.
  - Elementos opcionales, propiedades predefinidas, ejemplos.

Operaciones

- Definición: función o transformación que puede aplicarse a los objetos de una clase.
- Diferencia entre "operación" y "método".

- Notación: definida en UML, pero más importante ajustarse al lenguaje de implementación
  - Elementos opcionales, propiedades predefinidas, ejemplos.

7. Modelado conceptual (2)

Asociaciones[ggf58]

- Definición: especificación de un conjunto de conexiones entre instancias (enlaces) que representan la estructura y posibilidades de comunicación del sistema.
- Propiedades básicas:
  - Nombre de asociación[ggf59] y nombre de rol.
  - Multiplicidad.
  - Navegabilidad[ggf60].
- Toda asociación es asimétrica desde el punto de vista lingüístico.
  - Persona-caza-Animal, Persona-contrata-Empresa: ¿quién caza o contrata a quién?
  - Aunque sea la misma palabra, el nombre de la asociación significa cosas distintas según cómo se lea (no es lo mismo que una persona contrate a una empresa para que le realice un servicio, o que una empresa contrate a una persona como empleado).
  - Los nombres de rol y la dirección del nombre de asociación rompen esta ambigüedad, ayudan a darse cuenta de que cada extremo desempeña un papel distinto.
  - Ni siquiera una asociación reflexiva puede ser simétrica (en UML).
  - Toda asociación es asimétrica, pero puede ser uni- o bi- direccional (navegabilidad).
- Relación con otros elementos:
  - Asociación vs. Atributo: equivalencia parcial con asociación unidireccional.
  - Asociación vs. Operación: invocación de operaciones a través de la asociación.

Generalizaciones

- Definición: relación entre un elemento general y un elemento específico, donde el elemento específico puede añadir información y debe ser consistente con el elemento general.
- Generalización y clasificación.
- Generalización y especialización.
- Jerarquías de clases.
- Dimensiones de especialización.
- Generalización múltiple vs. Clasificación múltiple.
- Subclase vs. Atributo: *en análisis* es posible modelar mediante generalización múltiple, clasificación múltiple o clasificación dinámica determinados aspectos del problema que resultan "naturales" en el dominio, pero que no tienen traducción directa en los lenguajes de programación habituales, por lo que *en diseño* a veces hay que usar atributos como solución.

8. Modelado conceptual (3)

Diagramas

- Diagramas de clases y diagramas de objetos.
  - No es correcto que "un diagrama de objetos es una instancia de un diagrama de clases".
- Restricciones y notas.
  - Pueden afectar a propiedades, clases, o relaciones.
- Legibilidad de los diagramas.[ggf61]
- Restricciones en asociaciones.

Asociaciones especiales

- Asociaciones actor-sistema y clase-clase:
  - El actor modela un elemento del *entorno*, la clase modela un elemento del *sistema*.
- Asociaciones reflexivas[ggf62]:
  - Asimetría esencial de toda asociación en UML, que dificulta especialmente el modelado de *relaciones de equivalencia* (asociaciones reflexivas, simétricas y transitivas).
- Ciclos de asociaciones y asociaciones derivadas:
  - Los ciclos de asociaciones son prácticamente inevitables, y hay que tener claro su significado en cada caso.
- Agregación, composición y anidamiento:
  - Contrariamente a lo que pueda parecer, la relación todo-parte en UML no implica encapsulamiento ni acceso restringido (necesidad de acceder a las partes a través del todo, imposibilidad de acceder directamente a las partes).
  - Tampoco debe confundirse con el anidamiento [ggf63]entre clases.
- Diseño e implementación de asociaciones binarias
- Clase-asociación:
  - El problema de los enlaces repetidos.
  - Transformación en clase intermedia.
- Asociación n-aria:[ggf64]
  - En opinión de muchos autores, no son verdaderamente orientadas a objetos.
  - Cuidado con la tendencia a ver asociaciones n-arias donde no las hay.
  - Multiplicidad engañosa.
  - Transformación en clase intermedia.

Tema III. MODELADO ARQUITECTÓNICO

9. Introducción a la Arquitectura del software

¿Qué es la arquitectura de software?

- La arquitectura se refiere principalmente a la *descomposición del software* en módulos relacionados entre sí (arquitectura lógica, arquitectura del software), y secundariamente a su distribución entre las partes físicas (arquitectura física, arquitectura del hardware).
- Es probable que la descomposición del sistema en subsistemas (arquitectura) influya en la forma de estructurar los requisitos, para lograr una más clara correspondencia entre los requisitos y el diseño.
- La especificación clara de arquitecturas software, importante para todas las aplicaciones, es indispensable en el *trabajo en equipo*: modularización y ensamblaje.
- Recapitulación (IEEE Std 1471-2000[ggf66]): la arquitectura del software es…
  - la *organización fundamental* de un sistema
  - encarnada en sus *componentes*,
  - las *relaciones* entre ellos y con el entorno,
  - y los *principios* que orientan su diseño y evolución.

El modelo de 4+1 vistas (Philippe Kruchten)[ggf67]

- Se ha hecho clásica la representación de la arquitectura de un sistema mediante 4+1 vistas:
  - Vista lógica (o conceptual)
  - Vista de proceso (o de ejecución)
  - Vista de desarrollo (o de implementación)
  - Vista física (o de despliegue)
  - Vista de casos de uso: unifica (*supuestamente*[ggf68]) las otras cuatro vistas y sus relaciones.
- En cada vista podemos distinguir diferentes características:
  - *Aspecto* del sistema tratado de modo especial en la vista (complementario a los demás).
  - A quién va dirigida de modo particular esta vista (*stakeholders*).
  - Tipos de *requisitos* relacionados más estrechamente con la vista.
  - *Notación*: elementos y conectores esenciales en el lenguaje diagramático de cada vista.
- No todas las vistas son igualmente necesarias e importantes en cada proyecto.
- Relaciones entre las cuatro vistas:[ggf69]
  - VL→VP: identificar las clases que tendrán un hilo de control propio (*clases activas*).
  - VL→VD: descomponer en subsistemas de clases estrechamente relacionadas (*minimizar dependencias*); aparecen nuevas clases de diseño que no son conceptuales.
  - VP y VD→VF: tener en cuenta los *requisitos no funcionales* de la vista física.
- En proyectos grandes no debe esperarse una correspondencia 1-1 entre las distintas vistas.

¿Cómo lograr una descomposición modular eficaz?

- *Máxima cohesión, mínimo acoplamiento.*
  - *Cohesión* dentro de un módulo: grado de comunicación entre sus elementos.
  - *Acoplamiento* entre módulos diferentes: grado de comunicación entre ellos.
- Alta intra-cohesión, bajo inter-acoplamiento = *minimizar dependencias.*
  - Estas dos características hacen la arquitectura mucho más fácil de modificar, ya que los cambios sólo tienen efectos locales.
- Por lo tanto... diseñar para el cambio, diseñar para la extensión (= *mantenibilidad*).
- La clave para lograr una descomposición modular eficaz es *minimizar las dependencias*.
- Criterios para la selección de una arquitectura (o estilo arquitectónico)
  - Para un proyecto software dado puede haber varias arquitecturas apropiadas.
  - Se elige una teniendo en cuenta los *objetivos*, que deben estar priorizados, ya que unas arquitecturas satisfacen mejor unos objetivos que otros.

10. Componentes, dependencias [ggf71]

Noción de componente y dependencia

- Un componente (en la vista de desarrollo) representa una *unidad de código fuente* que encapsula el estado y comportamiento de una parte de la implementación.
- La relación de dependencia (representada como una *flecha discontinua*) significa que:
  - el elemento dependiente *requiere la presencia* del elemento independiente, y
  - los *cambios* en el elemento independiente *pueden afectar* al elemento dependiente.
- La dependencia puede establecerse entre cualesquiera dos elementos (componentes, clases…).
- La dependencia queda mejor definida si se establece con respecto a una interfaz.
  - De esta manera se establece una *dependencia parcial* o restringida: la dependencia ya no es hacia el componente/clase, sino sólo hacia la interfaz.
  - Una misma interfaz puede ser realizada por varios componentes/clases.
  - Un componente UML equivale aproximadamente a un paquete Java.

Diagrama de componentes (o de estructura)

- Muestra principalmente componentes e interfaces, y relaciones entre ellos.
- Distintas formas de expresar el cableado (*wiring*) de componentes:
  - Dependencia y realización.
  - Uso y realización (conector de ensamblaje *ball and socket*).
- Cableado en árbol: cuando varios componentes requieren/proporcionan una misma interfaz.
  - Las dos representaciones (independiente/en árbol) son totalmente equivalentes.
  - Aunque en el diagrama se puede expresar que dos o más componentes *proporcionan* la misma interfaz, habrá que escoger uno de ellos en el diseño final (no si *requieren*).

11. Interfaces [ggf72]

Noción de componente y dependencia

- Un componente (en la vista de desarrollo) representa una *unidad de código fuente* que encapsula el estado y comportamiento de una parte de la implementación.
- La relación de dependencia (representada como una *flecha discontinua*) significa que:
  - el elemento dependiente *requiere la presencia* del elemento independiente, y
  - los *cambios* en el elemento independiente *pueden afectar* al elemento dependiente.
- La dependencia puede establecerse entre cualesquiera dos elementos (componentes, clases…).
- La dependencia queda mejor definida si se establece con respecto a una interfaz.
  - De esta manera se establece una *dependencia parcial* o restringida: la dependencia ya no es hacia el componente/clase, sino sólo hacia la interfaz.
  - Una misma interfaz puede ser realizada por varios componentes/clases.
  - Un componente UML equivale aproximadamente a un paquete Java.

Noción de interfaz [ggf73]

- *Dos formas alternativas* de representación:
  - Abreviada: icono circular.
  - Completa: rectángulo «interface» con compartimentos para atributos y operaciones.
- Interfaz (definición tradicional): conjunto de operaciones que ofrecen un servicio coherente.
  - Nota: la operación *imprimir( )* de la transparencia significa *ser impreso*, por eso tiene sentido que la clase *Documento* realice la interfaz *Imprimible*. Si *imprimir( )* significase

imprimir algo, entonces la interfaz debería llamarse *Imprimidor*, y podría ser realizada por la clase *Impresora*.

- Características de una interfaz:
  - No contiene la implementación de las operaciones (métodos).
  - Puede contener también atributos[ggf74] (*properties*). Esto es nuevo en UML2.
  - Si un componente/clase realiza una interfaz, se compromete a implementar *todas* las operaciones definidos en la interfaz. Puede tener más operaciones, pero no menos.
  - Dos interfaces ofrecidas por un mismo componente no son necesariamente disjuntas.
- Semejanza con clases abstractas.
- Generalización de clases vs. Realización de interfaces.

Interfaces proporcionadas y requeridas[ggf75]

- Un componente/clase puede relacionarse de dos formas distintas con una interfaz:
  - El componente proporciona (realiza, implementa) la interfaz: *interface realization*.
  - El componente requiere (usa, depende de) la interfaz: *interface usage*.
- Es posible mostrar las interfaces requeridas sin mostrar quién las proporciona.
- Cómo se deben nombrar las interfaces:
  - *Suelen* nombrarse empezando por "i" (regla de estilo *no obligatoria*).
  - El nombre de la interfaz debe escogerse para que signifique adecuadamente el "nombre común" (en singular) de todas sus instancias indirectas. Ejemplos:
    - iCaminante: implementada por Hormiga, Robot…(operaciones: avanzar, girar…).
    - iImprimible: implementada por Texto, Imagen … (operaciones: imprimir…).

- Cableado en árbol: cuando varios componentes requieren/proporcionan una misma interfaz.
  - Las dos representaciones (independiente/en árbol) son totalmente equivalentes.
  - Aunque en el diagrama se puede expresar que dos o más componentes *proporcionan* la misma interfaz, habrá que escoger uno de ellos en el diseño final (no si *requieren*).

Uso de interfaces en OO

- Definición de interfaz en OO[ggf76]:
  - De modo general (no sólo en OO/UML, también en programación estructurada) una interfaz es un *conjunto de operaciones*[ggf77] que ofrecen un servicio coherente.
  - Pero en OO/UML toda operación se invoca sobre una instancia (salvo el caso especial de las operaciones estáticas[ggf78]):
    - Para usar las operaciones de la interfaz es necesario crear en el componente una clase que realiza (o implementa) la interfaz, y es necesario instanciar esta clase.
    - Las operaciones se invocan sobre las instancias "indirectas" de la interfaz (instancias de la clase o clases compatibles con la interfaz).
  - Así pues, *una interfaz define un tipo* (un tipo abstracto de datos) que proporciona un conjunto coherente de operaciones sobre las instancias compatibles con ese tipo.
  - Análoga a una clase abstracta con todas sus operaciones abstractas: no puede tener instancias *directas*.
- Cómo usar/instanciar las interfaces de un componente:
  - Problema: para crear una instancia indirecta de una interfaz necesito una operación (las interfaces no tienen constructores), pero sólo puedo utilizar la operación si previamente ya tengo una instancia sobre la que invocarla.
  - Existen dos soluciones generales, más una intermedia:
    - Proporcionar una o más clases públicas compatibles, usar sus constructores.
    - Proporcionar una fábrica y gestor de instancias (patrón *AbstractFactory*).
    - Proporcionar la interfaz, pero no las clases concretas.
  - Ejemplo: componente GestiónEmpleados:

        - interfaz iEmpleado, clases públicas EmpleadoFijo, EmpleadoTemporal, etc.: más flexibilidad, menos encapsulación (las clases se puede usar en cualquier lugar).
        - interfaz iGestorEmpleados, clase pública GestorEmpleados, clases de paquete (no visibles desde fuera del mismo) EmpleadoFijo, etc.: más encapsulación, menos flexibilidad (las operaciones de GestorEmpleados no pueden recibir ni devolver valores de tipo Empleado).
        - interfaz iGestorEmpleados, interfaz iEmpleado, clase pública GestorEmpleados clases privadas EmpleadoFijo, etc.: situación intermedia (no se puede usar el constructor directamente, pero sí el tipo abstracto iEmpleado).
    - Recapitulación: ¿cómo se deben nombrar las interfaces de clases y componentes?
        - Regla: el nombre de la interfaz debe escogerse para que signifique adecuadamente el "nombre común" (en singular) de todas sus instancias indirectas.
        - Realización clase-interfaz: *es-un-tipo-de*. La instancia de la clase es una instancia (indirecta) de la interfaz. El nombre de la interfaz denota un superconjunto respecto al nombre de la clase. EmpleadoFijo es-un-tipo-de Empleado, EmpeladoTemporal es-un-tipo-de Empleado. El nombre de la interfaz es un nombre común más general que los nombres de las clases que la realizan.
        - Realización componente-interfaz: *ofrece*. El componente no tiene instancias, las clases incluidas en el componente sí. El nombre de la interfaz denota un superconjunto de alguna de las clases contenidas en el componente. No decimos *GestiónEmpleados es-un-tipo-de iEmpleado, iGestorEmpleados, decimos que el componente ofrece esas interfaces. El nombre de la interfaz no necesariamente es un nombre común más general que el nombre del componente que la realiza.

12. Diseño por contratos

Diseño por contratos (*design by contract: DbC*)

- La *productividad* del software depende de su *calidad*, que a su vez depende de su *fiabilidad*.
  - Fiabilidad = corrección + robustez.
  - La búsqueda sistemática de la fiabilidad empieza por *definir con precisión* lo que se espera de cada elemento del software.
  - No es una condición *suficiente* para la fiabilidad, pero sí es una condición *necesaria*.
- Beneficios de DbC:
  - Mejor comprensión de la construcción de software (en particular, orientado a objetos).
  - Construcción sistemática de sistemas libres de errores.
  - Marco eficaz para depuración, pruebas, y aseguramiento de la calidad.
  - Documentación de componentes: el contrato explica la finalidad del componente.
  - Mejor comprensión del mecanismo de herencia.
  - Técnica para tratar los casos anormales (excepciones).

Noción de contrato

- Diseñar contra una interfaz es como emplear un contrato[ggf80]: el proveedor se compromete a proporcionar un servicio, de modo que el cliente puede ser diseñado sin saber cómo se implementa este servicio.
- Analogía con contratos en asuntos humanos: cliente y proveedor, obligación y beneficio.
- Si ambos satisfacen su parte del contrato:
  - El cliente puede confiar en que obtendrá lo que el proveedor promete.
  - El proveedor puede confiar en que no se le pedirá más de lo que promete.
- El uso de interfaces (o clases abstractas) facilita la incorporación de otros subtipos concretos al diseño sin tener que cambiar el código cliente (*menor acoplamiento*).

Especificación formal del contrato[ggf81]

- Todo contrato tiene dos partes *igualmente esenciales*: sintaxis y semántica.
- La *sintaxis* del contrato (su interfaz) especifica el *aspecto externo* del contrato, la información necesaria para proporcionar el servicio, dada por la firma o *signatura* de la operación, que define los valores entregados y devueltos, con sus tipos.
- La *semántica* del contrato (su significado) especifica el *contenido* propiamente dicho del contrato, mediante pre- y post- condiciones (expresiones lógicas entre atributos y parámetros).
  - Precondición:
    - El cliente debe asegurarse de que se cumple (es su obligación en el contrato).
    - El proveedor la comprueba, y no hace nada si no se cumple (su beneficio).
  - Postcondición:
    - El proveedor debe asegurarse de que se cumple (es su obligación en el contrato).
    - Expresa lo que el cliente espera del servicio (su beneficio).
- El nombre del servicio/operación debe estar relacionado con la semántica, pero no la expresa completamente, ni menos aún la garantiza. La signatura del contrato también es insuficiente para expresar y garantizar la semántica. *Necesitamos* pre- y post- condiciones (PPCs).
- *Ejemplo*: cuenta corriente.
  - Los nombres no *expresan* completamente la semántica del contrato: ¿qué pasa con *sacarDinero* si la cuenta está vacía? ¿y con *meterDinero* si la cuenta está cerrada?
  - Tampoco *garantizan* la semántica: ningún lenguaje de programación puede garantizar que *meterDinero* y *sacarDinero* significan lo que el cliente espera que signifiquen.
- Es un formalismo perfectamente aplicable en análisis (recordar casos de uso)[ggf82].

Uso de pre- y post- condiciones

- La expresión de las PPCs es *puramente declarativa*, no dice lo que tiene que hacer una operación, sino sólo el efecto que puede percibir su cliente.
- Las PPCs se *especifican* en el diseño, y se *comprueban* en tiempo de ejecución.
  - En determinadas circunstancias se puede omitir[ggf83] la comprobación de la postcondición: por ejemplo, servicios proporcionados por componentes muy fiables.
  - En cambio, en general conviene al proveedor comprobar siempre la precondición: es más fácil que el código cliente se equivoque, y comprobarlo puede evitar desastres.
- ¿Quién las comprueba? Tanto el cliente como el proveedor pueden comprobarlas (ver lo dicho más arriba sobre la semántica del contrato).
- ¿Qué ocurre si no se cumplen las PPCs?
  - Precondición: indica un *error en el cliente*, que no debería haber invocado el servicio. Diversas estrategias: devolver un código de error (forma abreviada de comprobar la precondición y ejecutar la operación si se cumple), levantar una excepción, etc.
  - Postcondición: indica un *error en el proveedor*, que no ha cumplido su parte del contrato. Error grave que exige levantar una excepción u otro mecanismo equivalente.
- Además de ser útiles para documentar el diseño, las PPCs tienen utilidad especialmente en la *depuración* del software.

Invariantes de clase

- Además de las pre- y post- condiciones de cada operación, pueden especificarse también *invariantes de clase*, que son aserciones que deber ser verdaderas "en todo momento".
- Las operaciones pueden violarlos temporalmente y de modo controlado durante su ejecución.
- Obviamente, la expresión lógica del invariante sólo puede contener atributos (no parámetros).
- La violación de un invariante es un error grave que también exige levantar una excepción.
  - Igual que con las postcondiciones, a veces se puede omitir la comprobación.

Contratos y herencia: subcontratación

- La superclase subcontrata a las subclases para las instancias directas de la subclase.
- Pero existe el peligro de redefinir mal un servicio, es decir, modificar su semántica.
- La especificación de contratos ayuda a entender y usar correctamente la herencia.
- Las instancias de la subclase son también instancias de la superclase, por tanto todo invariante aplicable a la superclase se aplica también a la subclase (*principio de sustitución*). El invariante puede ser más restrictivo en la subclase, pero no menos restrictivo.
- *Regla o principio de subcontratación*: la operación redefinida en una subclase debe…
  - mantener o debilitar la precondición (no puede exigir más), y
  - mantener o reforzar la postcondición (no puede ofrecer menos).
- Por lo tanto, para que pueda existir una relación jerárquica entre dos clases, deben cumplirse las siguientes implicaciones lógicas ("si A entonces B"):
  - subclase.invariante → superclase.invariante
  - Y para cada operación redefinida:
    - subclase.operación.Pre ← superclase.operación.Pre
    - subclase.operación.Post → superclase.operación.Post
- Si esto no se cumple, entonces no debe existir una relación jerárquica entre las clases.
- *Ejemplo*: cuenta corriente y cuenta de crédito.
  - La operación sacarDinero está redefinida correctamente.
  - Pero el invariante de clase redefinido es menos restrictivo: el saldo podría ser negativo.
  - Conclusión: *la generalización es incorrecta*.
  - Solución[ggf85]:
    - añadir una subclase CuentaDebito, hermana de CuentaCredito.
    - traladar el invariante (saldo ≥ 0) a esta subclase.
- *Ejemplo*[ggf86]: distancia "oblicua" y distancia "esquinada". La jerarquía es también incorrecta.