

# Análisis de la información - Teoría

## Tabla de contenido

<b>1 - Introducción a la ingeniería de software.....</b>	<b>9</b>
<i>Orígenes .....</i>	9
<i>Ingeniería de software .....</i>	9
Ingeniería tradicional Vs. Ingeniería de software .....	9
Ataque a los accidentes .....	11
<i>¿Qué define una profesión?.....</i>	12
Ingeniería de software.....	12
<i>Ingeniería de software .....</i>	12
Áreas de conocimiento .....	12
Requisitos.....	12
Diseño.....	13
Construcción .....	13
Testing .....	13
Gestión .....	13
Gestión de la configuración.....	13
Métodos y modelos .....	13
Procesos .....	14
Calidad y procesos .....	14
Procesos, ciclos de vida .....	14
<i>Desarrollo de software.....</i>	15
Un proceso iterativo e incremental.....	15
Ciclo de vida en cascada .....	15
Un proceso iterativo e incremental.....	16
Casi todo es diseño .....	16
<i>Proceso de desarrollo de software .....</i>	16
Disciplinas y ciclo de vida.....	16
Modelos de referencia: (IBM) Rational Unified Process .....	17
<i>Desarrollo iterativo e incremental.....</i>	17
Agile Manifesto (2001) .....	17
<i>Proceso de desarrollo de software .....</i>	18
Modelos de referencia.....	18
Scrum.....	18
Extreme Programming (XP) .....	19
Feature Driven Development (FDD).....	19
Disciplined Agile Delivery (DAD).....	20
<i>Conclusiones.....</i>	20

<b>1.1 - Calidad y procesos en la industria del software .....</b>	<b>22</b>
<i>Calidad.....</i>	22
Definición: ISO 8402.....	22
<i>Calidad y procesos.....</i>	22
Gestión cuantitativa.....	22
<i>Definiciones: Verificación Vs. Validación.....</i>	23
<i>Un poco de historia .....</i>	23
<i>Calidad y procesos.....</i>	24
PDCA.....	24
Enfoques vigentes hoy .....	25
Mejora continua.....	25
Calidad del software .....	25
<i>Calidad y procesos en la industria del software.....</i>	26
Perspectiva histórica .....	26
<b>1.2 - Modelos en la ingeniería de software.....</b>	<b>28</b>
<i>¿Qué es un modelo?.....</i>	28
<i>Modelos y métodos.....</i>	28
Una breve (e incompleta) perspectiva histórica .....	28
<i>Modelos.....</i>	29
La influencia del lenguaje de programación .....	29
Perspectiva histórica .....	29
IDEFO.....	29
Análisis estructurado (DFD, DER, DD) .....	30
Diseño estructurado (DE) .....	30
Unified Modeling Language (UML) .....	30
UML .....	30
Diagrama de clases .....	30
Diagrama de casos de uso .....	31
Diagrama de secuencia.....	31
Diagrama de estado.....	31
Diagramas de actividad.....	31
Business Process Modeling and Notation (BPMN) .....	32
<i>C4.....</i>	33
Arquitectura de software .....	33
Nivel 1: Diagrama de contexto .....	34
Nivel 2: Diagrama de contenedores .....	34
Nivel 3: Diagrama de componentes .....	35
Nivel 4: Diagrama de código.....	35
<i>Resumen.....</i>	36

Escenarios de utilización.....	36
<b>1.1 - INCOSE.....</b>	<b>38</b>
<b>1.2 - Videos 1, 1.1 y 1.2 y Bourque .....</b>	<b>40</b>
<b>1.3 - No silver bullet .....</b>	<b>42</b>
<b>1.4 - Boehm y Royce.....</b>	<b>43</b>
<b>1.5 - Fowler.....</b>	<b>45</b>
<b>2 - Introducción a la ingeniería de requisitos .....</b>	<b>46</b>
<i>Introducción .....</i>	46
<i>Requerimientos y requisitos.....</i>	46
Definiciones.....	46
<i>Requisitos .....</i>	46
Distintos tipos de requisitos .....	46
Distintas perspectivas .....	47
<i>Ingeniería de requisitos.....</i>	48
¿Qué tipos de actividades incluye? .....	48
¿Qué implica?.....	49
Los requisitos y el ciclo de vida del proyecto.....	49
Complicaciones .....	49
Expectativas e involucramiento .....	50
Ventajas.....	50
¿Quién hace ingeniería de requisitos? .....	50
Un rol, muchos roles, diferentes nombres .....	50
Habilidades necesarias .....	50
¿Cuán formal debe ser?.....	51
Diferentes contextos.....	51
<i>Actividades y técnicas .....</i>	52
<i>Ingeniería de requisitos en el ciclo de desarrollo .....</i>	53
Un ejemplo: Scrum.....	54
<i>Disciplinas relacionadas.....</i>	55
Business Analysis.....	55
Design Thinking .....	55
<b>2.1 - Design Thinking .....</b>	<b>57</b>
<i>Design Thinking.....</i>	57
Definición .....	57
Etapas .....	57
<i>Design Thinking for Requirements Engineering (DT4RE) .....</i>	58
Métodos y herramientas (algunos) .....	58
Design Thinking & Scrum: ejemplo de aplicación.....	58
<b>2.1 - Videos 2 y 2.1 .....</b>	<b>59</b>

<b>2.2 - Wiegers capítulos 1 y 2 .....</b>	<b>60</b>
<b>2.3 - Wiegers capítulo 14 .....</b>	<b>63</b>
<b>3 - Técnicas de descubrimiento de requisitos .....</b>	<b>64</b>
<i>Ingeniería de requisitos.....</i>	<i>64</i>
Actividades .....	64
<i>Descubrir requisitos.....</i>	<i>64</i>
Complicaciones .....	64
<i>Partes interesadas.....</i>	<i>65</i>
¿Quiénes son?.....	65
<i>¿Cómo descubrir requisitos?.....</i>	<i>65</i>
Técnicas .....	65
Entrevistas.....	65
Observación .....	66
Talleres (de requisitos) .....	66
Encuestas .....	66
Grupos de discusión (Focus Groups).....	66
Análisis de reglas de negocio.....	67
Brainstorming.....	67
Análisis de documentos .....	67
Análisis de causa raíz .....	67
Prototipado .....	67
Open Space (Technology).....	67
Documentación de los resultados .....	68
Minuta de reunión .....	68
Otras técnicas.....	68
Recomendaciones .....	68
<i>Conclusiones.....</i>	<i>69</i>
Resumen.....	69
<b>3.1 - IIBA y PMI .....</b>	<b>71</b>
<b>3.3 - IIBA y PMI .....</b>	<b>72</b>
<b>3.4 - Bartyzel.....</b>	<b>74</b>
<b>4 - Técnicas de análisis, modelado y especificaciones de requisitos.....</b>	<b>75</b>
<i>Introducción .....</i>	<i>75</i>
<i>Requisitos .....</i>	<i>75</i>
Distintos interesados, distintos puntos de vista .....	75
Alternativas de representación .....	76
Visión (visión y alcance) .....	76
SRS (Especificación de Requisitos de Software) .....	76
Desarrollo ágil .....	76

Atributos de una especificación bien escrita .....	77
Análisis, modelado, especificación.....	77
<i>Cómo analizar, modelar y especificar requisitos .....</i>	78
Técnicas .....	78
Organización .....	78
Priorización.....	79
Escenarios .....	79
Una familia de técnicas.....	79
Usuarios y objetivos.....	79
Casos de uso.....	79
Historias de usuario .....	79
Guiones (Storyboards).....	80
Modelado .....	80
Lenguajes y notaciones.....	80
Modelos y diagramas.....	80
Diagrama de contexto .....	80
Modelo de casos de uso .....	80
Modelo de dominio .....	81
Árbol de funcionalidades (prestaciones) [feature tree] .....	81
Diagrama de Estados .....	81
Diagrama de actividades (UML) .....	82
Flujo de actividades en un proceso de negocio (BPMN).....	82
Prototipado .....	82
EARS (Easy Approach to Requirements Syntax) .....	82
Criterios de aceptación.....	83
Reglas de negocio .....	83
User Story Mapping .....	83
Impact mapping .....	83
Recomendaciones .....	84
Generales .....	84
Modelado .....	84
Especificación.....	84
Conclusiones.....	85
Para recordar.....	85
<b>4.3 - Modelado de dominio .....</b>	<b>86</b>
<i>Introducción .....</i>	86
Modelos.....	86
Modelado del dominio .....	86
Modelado del dominio: objetos, asociaciones y atribuciones.....	87

<i>Modelo del dominio</i> .....	87
¿Cómo se construye?.....	87
¿Cómo encontrar clases conceptuales?.....	87
Análisis lingüístico .....	88
Lista de categorías .....	88
Atributos.....	89
Asociaciones.....	90
Asociaciones binarias.....	90
Asociaciones unarias.....	91
Asociaciones ternarias .....	91
Clases asociativas.....	91
Composición y agregación.....	91
Asociaciones calificadas.....	92
Generalización y especialización .....	92
Roles en una asociación.....	92
¿Cómo encontrar asociaciones? .....	93
<i>Patrones</i> .....	93
Personas y organizaciones.....	93
Productos y especificaciones.....	94
Facturas .....	94
Facturas: otros escenarios.....	97
Facturas: (algunos) otros escenarios.....	97
Inventario y contabilidad .....	98
Mediciones .....	99
<i>Modelado de datos</i> .....	100
Métodos y modelos: una breve (e incompleta) perspectiva histórica .....	100
Diagrama de Entidad-Relación (ERD o DER) .....	100
Funciones y datos.....	101
<i>Modelado de datos y modelado de objetos</i> .....	101
Tres comunidades relacionadas, pero con perspectivas diferentes .....	101
Conciliación de perspectivas .....	102
<i>Conclusiones</i> .....	102
Resumen.....	102
<b>4.1 - Video 4.....</b>	<b>103</b>
<b>4.4 - Larman.....</b>	<b>104</b>
<b>4.2 - Historias de usuario .....</b>	<b>105</b>
<i>Historias de usuario</i> .....	105
Definición .....	105
Formato .....	105

Ejemplo.....	105
<i>Historias de usuario y requisitos</i> .....	105
<i>Historias de usuario</i> .....	105
3 Cs: Card, Conversation, Confirmation .....	105
Criterios de aceptación.....	105
Formato.....	105
Ejemplo.....	106
Epopeyas .....	106
Nombres.....	106
<i>Épicas e historias de usuario</i> .....	107
Formatos .....	107
<i>Calidad de las historias de usuario</i> .....	107
Regla INVEST .....	107
QUS Framework.....	107
Ejemplo 1.....	108
Ejemplo 2.....	108
Ejemplo 3.....	109
<i>Historias de usuario</i> .....	109
¿Cómo encontrarlas?.....	109
User Story Mapping .....	109
Impact Mapping.....	110
En el proceso de desarrollo .....	111
Product & Sprint Backlogs .....	111
Kanban.....	112
<b>4.3 - Lucassen.....</b>	<b>113</b>
<b>4.1 - Casos de uso.....</b>	<b>114</b>
<i>Casos de uso</i> .....	114
Definición (informal) .....	114
Definición (formal) .....	114
<i>Actor</i> .....	114
Definición .....	114
<i>Casos de uso</i> .....	114
Tecnológicamente neutros .....	114
El modelo visual: diagramas de casos de uso en UML .....	114
<i>Escenario</i> .....	115
Definición .....	115
<i>Casos de uso</i> .....	115
Un contrato .....	115
Formato breve.....	115

Formato secuencia de acciones.....	115
Formato de dos columnas o diálogo .....	116
Más completo: precondiciones, postcondiciones, escenarios alternativos .....	116
<i>¿Cómo encontrarlos?</i> .....	117
<i>Casos de uso</i> .....	117
Relaciones de inclusión.....	117
Relaciones de extensión .....	118
Relaciones de generalización.....	118
Adicionales .....	119
Casos de uso CRUD / ABMC.....	119
<i>Casos de uso en el proceso de desarrollo</i> .....	119
Proceso unificado.....	119
Proceso unificado.....	120
Modelo de análisis en el proceso unificado.....	120
<i>Modelo de casos de uso</i> .....	121
à la Larman .....	121
<i>Contratos y operaciones</i> .....	121
Diagrama de secuencia del sistema .....	121
Operaciones del sistema.....	122
<i>Casos de uso</i> .....	122
2.0 .....	122
<i>Conclusión</i> .....	123
<b>4.2 - IIBA y PMI .....</b>	<b>124</b>
<b>4.11 - Reglas de negocio .....</b>	<b>125</b>

# 1 - Introducción a la ingeniería de software

## Orígenes

El término ingeniería de software se le atribuye a Margaret Hamilton, quien en 1960 lo acuñó cuando buscaba describir las actividades que realizaba en el contexto del programa Apolo de la NASA. Esta persona tenía a cargo el desarrollo del sistema de navegación que permitió la llegada del hombre a la Luna. Margaret Hamilton fue reclutada del MIT, había que desarrollar una disciplina que no existía hasta ese entonces y aparece entonces ese término, software engineering. Hamilton se dio rápidamente cuenta que hacía falta hacer un montón de cosas, además de programar: hacía falta clarificar cuáles eran los requisitos que tenían que satisfacer el software; había que diseñarlo; había que tomar decisiones de diseño; había que resolver temas relacionados con la concurrencia, con la disponibilidad; había que integrar el software con el resto de los componentes; manejar las versiones de código; gestionar las actividades del equipo, et.

Este término se popularizó en una famosa conferencia de la OTAN que se realizó en 1968 y que se repitió al año siguiente. El tema de estas conferencias fue ingeniería de software. Por ese entonces, se hablaba de la crisis del software. Gracias al desarrollo de la tecnología, el hardware era cada vez más potente lo que hacía posible desarrollar software cada vez más complejo. Lo que a su veces generaba demanda para desarrollar aplicaciones cada vez más grande, cada vez más complicadas. En la década de 1960 no había experiencia suficiente como para poder enfrentar este desafío exitosamente, por eso surge la necesidad de estudiar el tema en profundidad. En ese entonces la comunidad se preguntaba algunas cosas que hoy nos parecen triviales. Los grandes interrogantes pasaban por: cómo desarrollar código que fuera mantenible; cómo satisfacer requisitos complejos y cambiantes; cómo desarrollar software en grandes equipos, y otras cuestiones por el estilo. Para tener una idea y a manera de ejemplo, en ese entonces ni siquiera había sistemas que permitían administrar el versionado de código

## Ingeniería de software

### Ingeniería tradicional Vs. Ingeniería de software

Ahora bien, formalmente: ¿Qué es la ingeniería de software? En términos generales:

*La **ingeniería** es la creación de soluciones costo efectivas a problemas prácticos mediante la aplicación de conocimiento codificado para construir cosas al servicio de la humanidad.*

*La **ingeniería de software** es entonces un tipo particular de ingeniería que consiste en la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software. En definitiva, la aplicación de los principios de la ingeniería al desarrollo de software, al mantenimiento y a la operación.*

Más informalmente podríamos decir que la ingeniería de software es una *ingeniería de cosas abstractas*. ¿Por qué? Porque fundamentalmente, *los elementos con los que nosotros tratamos no existen en el mundo real*.

[V / F] Fundamentalmente la ingeniería del software es una ingeniería de cosas abstractas ya que los elementos con los que se trata son parte del mundo real.<sup>1</sup>

<sup>1</sup> Fundamentalmente la ingeniería del software es una ingeniería de cosas abstractas ya que los elementos con los que se trata NO son parte del mundo real.



La ingeniería clásica se caracteriza por estar gobernada por las leyes de la física, por estar limitada por las propiedades de los materiales y, obviamente, por siglos de experiencia acumulada. Construimos casas, construimos puentes, construimos caminos desde hace milenios, fabricamos productos desde hace centenares de años. El producto de la ingeniería tradicional es tangible. Los ingenieros diseñan y son los operarios y las máquinas los que construyen. En muchas veces se replica un mismo diseño. Pensemos que se pueden fabricar millones de unidades de una misma pieza. Y el cuerpo de conocimiento está basado en las ciencias básicas matemáticas, física y química.

En cambio, el desarrollo de software no está restringido por las leyes físicas ni por las propiedades de los materiales. El producto es intangible. Casi toda la actividad es diseño. Si partimos de la base de que el código es un modelo de cómo se tiene que comportar el software, claramente cuando estamos programando en realidad estamos diseñando, estamos definiendo un comportamiento. El código fuente no es el producto, el producto de software es, en definitiva, comportamiento, funcionalidad. No hay replicación de un mismo diseño, cada producto que desarrollamos es único, diferente, cada pieza de código distinta. Y el cuerpo de conocimiento es propio. Por sus características, el desarrollo de software se parece más a la producción de una obra teatral, la escritura de un libro, la producción de una película, o inclusive, un emprendimiento de tipo económico.

[V / F] La ingeniería de software se asemeja a la ingeniería tradicional.<sup>2</sup>

Hace ya más de 30 años, Frederick Brooks analizó en un artículo famoso llamado “No silver bullet”<sup>3</sup> la naturaleza del software. En ese artículo, Brooks describe las **cuatro características esenciales del software**:

- **Invisibilidad:** El software es invisible porque sólo se aprecian sus resultados, no se puede visualizar, es invisualizable. En la ingeniería clásica, una realidad geométrica se puede modelar mediante una abstracción, hacemos un plano de una pieza, hacemos el plano de un puente. Esa descripción nos describe gráficamente, una realidad geométrica. En el caso del software, esto no es posible porque es invisualizable, no podemos representar geométricamente los softwares. Forzosamente tenemos que emplear diversas perspectivas y diversos niveles de abstracción para representar los distintos aspectos que los compone.
- **Complejidad:** El software es complejo y es uno de las entidades más complejas que alguna vez haya producido el hombre. El software está compuesto de muchas partes y de muchos estados posibles.
- **Mutabilidad:** Es mutable, puede cambiar. El cambio parece natural dado lo variable del medio. Como decimos, es agregar un if. Es un cambio simple, muchos creen, y sin embargo, estamos ante un pequeño ajuste que puede llegar a producir impactos sumamente importantes.
- **Conformidad:** Siempre el software va a tener que cumplir con restricciones que le impone el medio ambiente, requisitos que le imponen sus usuarios.

Brooks dice que estas características son esenciales porque no se pueden resolver mágicamente, no se pueden resolver como con una bala de plata como plantea en el artículo. Son inherentes a la naturaleza propia del software.

<sup>2</sup> La ingeniería de software NO se asemeja a la ingeniería tradicional.

<sup>3</sup> “No silver bullet – Essence and Accident in Software Engineering”, Frederick P. Brooks Jr.

Parte de la complejidad radica en la escala. Veremos que hace 30 años atrás, 40 años atrás, la primera versión de UNIX tenía 70.000 líneas de código. Hoy cualquiera de los navegadores que usamos en nuestras computadoras tiene 8, 9, 10 millones de líneas de código, lo que hace 10 años atrás tenía un sistema bancario. El software va aumentando de tamaño justamente aprovechando esa capacidad creciente del hardware. Y desarrollar software de un tamaño importante tiene profundas implicancias. Ya Melvin Conway hace varios años acuñó algo que hoy conocemos como ley de Conway, que dice que: "Las organizaciones humanas que desarrollan software, tienden a replicar en sus diseños la estructura de comunicación de la organización".

Todos hemos tenido la oportunidad de desarrollar algún programa para resolver un problema puntual. Convertir eso en un producto implica generalizar la solución, probarla, documentarla, estar dispuesto a dar soporte, mantener ese producto en el tiempo, eso implica un esfuerzo notable, muchas veces de un equipo grande de personas. Si nuestro pequeño programa se transforma en un sistema, es decir, que debe interactuar con otros programas, aparecen otros problemas. Vamos a necesitar definir cómo se va a hacer la integración entre los distintos componentes, cuáles van a ser las interfaces, etc. Si queremos transformar nuestro programa en un producto, hay que claramente pensar en una solución más completa, hay que generalizarla, hay que pensar en la documentación, hay que pensar bien en la prueba. Y si lo transformamos en un paquete, es decir, en un producto que va a estar integrado por más de un producto, la complejidad todavía crece más.

Lo que no hay que perder nunca de vista es que el software forma parte siempre de un sistema mayor, que está integrado por otros elementos. En un celular, por ejemplo, nos vamos a encontrar con hardware, con software. El usuario final, no tiene demasiado claro cuál es la frontera entre uno y otro, es un producto, es un sistema, es una cosa integrada.

### Ataque a los accidentes

Volviendo al artículo de Brooks, en él se nos advierte de que muchos de los desarrollos que han prometido resolver mágicamente los problemas del desarrollo de software han atacado más bien los accidentes y no los problemas esenciales. Hemos visto con el transcurrir de las décadas, lenguaje de alto nivel, ambientes de desarrollo más evolucionado, hemos visto también la aparición de la programación orientada a objetos, la programación visual, la inteligencia artificial y, en mayor o menor medida, en algún momento todos ellos, se han vendido como recetas mágicas, una bala de plata para resolver las limitaciones y los problemas del desarrollo de software. Lo que nos advierte Brooks es que las herramientas apuntan a resolver los accidentes, y no a resolver la esencia. El software es inherentemente complejo. Podemos tener algunas herramientas, pero va a ser menos difícil escaparse de esas cuatro características esenciales que mencionábamos hace unos momentos.

Está claro que en la economía de hoy es crítico poder manejar adecuadamente el desarrollo de software. Todos sabemos el profundo impacto que tiene el software en la economía mundial. Ya en 2011, Mark Andreessen, creador de Netscape (el primer navegador comercial que hubo) y actual accionista de importantes empresas de tecnología, aseguraba en un artículo que apareció en el Wall Street Journal, que el software se estaba comiendo al mundo. En este artículo, Andreessen describía cómo empresas basadas en software estaban desplazando del mercado a empresas más tradicionales. Daba el ejemplo de Netflix que derrotó a Blockbuster (una cadena de alquiler de videos), Amazon que desplazó a las librerías tradicionales y que actualmente está desafiando el concepto mismo de supermercados tal como lo conocemos. También describía como el software está reemplazando determinados elementos de hardware en autos y en dispositivos de comunicación. Actualmente estamos viendo como algunos dispositivos genéricos con software parametrizable están reemplazando a elementos de hardware específicos.

El dinero que se gasta en software es impresionante. Solamente en 2019 se gastaron 439.000 millones de dólares, 8.3% más que el año anterior.

Lo que complica el panorama es que no todos los proyectos de desarrollo de software terminan bien. Se calcula que alrededor del 30% de los proyectos no cumple con alguno de sus objetivos de costo, duración y resultado. Por eso es sumamente crítico aprender a gestionar, a manejar adecuadamente los esfuerzos de desarrollo de software. Hace

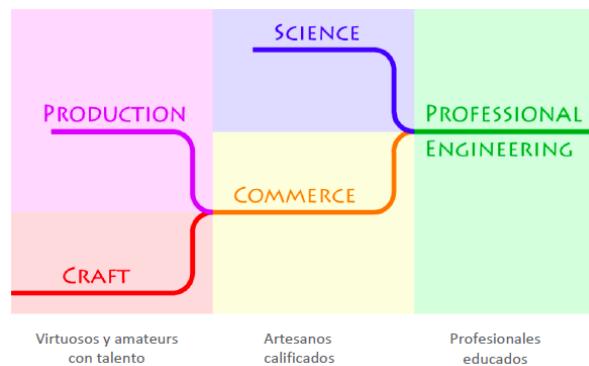


falta una profesión, y ahí es donde aparece la ingeniería de software. Cabría preguntarse entonces si realmente la ingeniería de software es una profesión.

## ¿Qué define una profesión?

### Ingeniería de software

Históricamente, las ingenierías han surgido desde la práctica ad hoc, en 2 etapas: En la primera, las técnicas de producción y de gestión permitieron transformar en producción rutinaria la actividad de los artesanos y de los virtuosos. Posteriormente, en una segunda etapa, surge como resultado de los problemas que iban apareciendo en la producción rutinaria, fundamentos científicos para ayudar a mejorar esa producción. Y es ahí en donde tiene lugar y aparece la práctica profesional de la ingeniería.

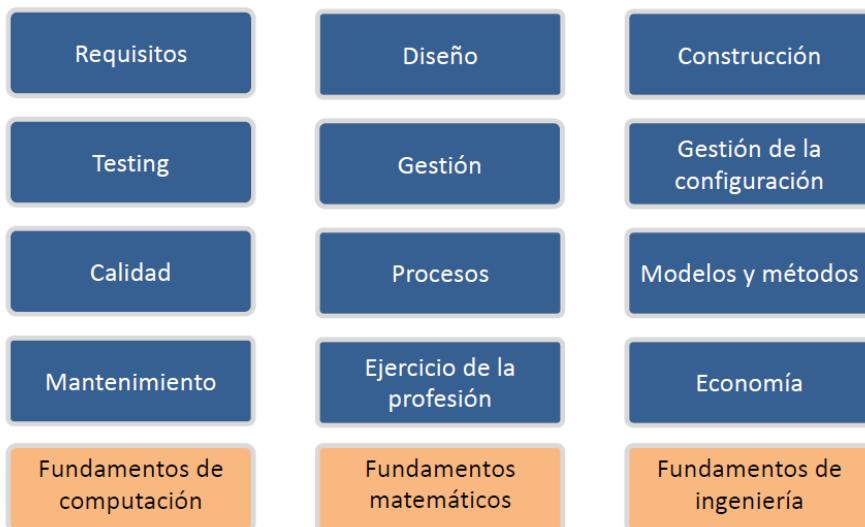


Un ejemplo interesante es el de la construcción de puentes, en el ámbito de la ingeniería civil. Al comienzo, la construcción de puentes era meramente artesanal. No se conocía prácticamente nada de las propiedades de los materiales, y mucho menos de mecánica de fluidos. Con el tiempo se fue desarrollando un cuerpo de conocimiento que le dio soporte a la práctica y se fue optimizando la construcción, se fue optimizando el uso de los materiales, se fue aprendiendo cuáles eran los mejores diseños que se podrían aplicar a la construcción de puentes.

## Ingeniería de software

### Áreas de conocimiento

Probablemente, el principal problema de la ingeniería de software sea su juventud, es una disciplina muy nueva, estamos todavía aprendiendo. No podemos comparar con ingenierías más tradicionales, claramente. Eso no impide que la ingeniería de software tenga un cuerpo de conocimiento. Un cuerpo de conocimiento que está en parte constituido por algunos de los fundamentos de la ingeniería que son relevantes (fundamentos matemáticos, fundamentos de ciencias de la computación) y áreas de conocimiento que le son propias.



En el gráfico podemos ver, en el área de conocimiento de construcción, que tiene que ver con la promoción y la prueba. Pero como ya hemos mencionado antes, en cualquier desarrollo de software no trivial, aparece la necesidad de otro tipo de actividades que no tienen que ver específicamente con la programación. Así que vamos a ir recorriendo rápidamente alguna de estas áreas de conocimiento que vemos aquí.

[ V / F ] El desarrollo de software es una disciplina relativamente reciente.

### Requisitos

La primera es la de **Requisitos**. El área de conocimiento de requisitos cubre todo lo que tiene que ver con el descubrimiento, el análisis, la especificación, la validación, y la administración de los requisitos del software. ¿Qué es

un requisito? Es una descripción, una definición de cómo el sistema se debe comportar o de una propiedad o atributo que debe poseer. En definitiva, esta área de conocimiento tiene que ver con definir cuál es el producto que tenemos que desarrollar, tenemos que entender el problema y definir qué es lo que necesitamos desarrollar. Por ejemplo, un requisito puede ser durante una semana, el teléfono debe indicar con un tono suave la entrada de otras llamadas. Eso es una descripción de cómo el sistema se debe comportar.

### Diseño

Otra área del conocimiento es la de **Diseño**. Es la definición de la arquitectura, los componentes y las interfaces de un producto de software. Esto sí está más cercano con la implementación, los requisitos no necesariamente. Por supuesto que el diseño tiene como input los requisitos. Se analizan esos requisitos con el objetivo de producir una descripción de la estructura interna del software.

### Construcción

Otra de las áreas de conocimiento es la **Construcción** que ya mencionábamos antes y que tiene que ver con la creación del software mediante la combinación de actividades de codificación, verificación, pruebas unitarias y pruebas de integración. Es importante aclarar este punto, que los límites entre áreas de conocimiento no son tan claras ni son tan lineales. Cuando uno está codificando, de alguna manera también está diseñando. Hace unos momentos dijimos que el código es un nivel de abstracción más, es un modelo más, cuando lo estamos modificando, también estamos diseñando, estamos de alguna manera modelando, que es lo que plantean algunos autores como por ejemplo Martin Fowler. Estamos tomando decisiones de diseño en el momento de codificado, pero también es cierto que algunas decisiones de más alto nivel que tienen más que ver con el diseño a nivel de arquitectura del software que van a condicionar todo lo que hagamos luego. Eso también es diseño.

[ V / F ] Si el diseño es un modelo de cómo se debe comportar el software, entonces al programar se está diseñando.

### Testing

El testing es la verificación dinámica del software. Cuando ejecutamos casos de prueba, lo que estamos haciendo es verificar en forma dinámica que el software funcione como se supone que debe funcionar.

### Gestión

Todo esfuerzo no trivial de desarrollo de software implica que tenga que haber un plan, que se tenga que organizar, que se tenga que utilizar ese plan para controlar las actividades, etc. Bueno, eso tiene que ver con la **gestión** del desarrollo de software. En definitiva esta área de conocimiento tiene que ver con la aplicación de actividades de gestión al desarrollo de software con el propósito de asegurar la entrega de productos y servicios de manera eficiente y efectiva.

### Gestión de la configuración

Otra área del conocimiento de este cuerpo es **gestión de la configuración**. ¿Qué es la gestión de la configuración? La identificación de la configuración de un producto de software en momentos específicos con el fin de mantener su integridad y trazabilidad a lo largo del ciclo de vida. ¿Qué es la configuración? Es la foto que le sacamos al conjunto de elementos (no necesariamente software) que forman parte del producto o sistema combinado de acuerdo a procedimientos específicos. Ustedes, probablemente, han hecho gestión de versiones de software. Bueno eso es una parte de gestión de la configuración. Cuáles son las versiones correctas de los distintos componentes que me permiten a mí integrar esos componentes y entregar el producto de software.

### Métodos y modelos

Todo software tiene algún tipo de **método** para su desarrollo. Los métodos nos proporcionan un enfoque sistemático para la especificación, diseño, construcción, prueba y verificación. Entonces, esta área de conocimiento tiene que ver con los métodos que se pueden utilizar para desarrollar software, y por supuesto la producción de modelos que nos ayudan a entender, definir y comunicar. Un método es una manera sistemática de hacer algo. En la ingeniería del software tenemos los heurísticos que están basados en la experiencia, como por ejemplo, el análisis y el diseño estructurado, el análisis y el diseño orientado a objetos, el modelado de datos, etc.; y los formales que tienen alguna justificación desde el punto de vista más formal y utilizan en general una notación matemática. El 80,

90% de los métodos de desarrollo de software que utilizamos son empíricos, son heurísticos, están basados en la experiencia de gente que ha trabajado mucho en esto.

Un **modelo**, por otra parte, es una simplificación de la realidad. Construimos modelos para entender mejor el sistema que estamos desarrollando o el sistema que queremos desarrollar y por supuesto que en este contexto hay distintos aspectos que nos van a interesar desarrollar. Algunos van a estar más orientados al diseño del producto, otros modelos que vamos a desarrollar van a estar más relacionados con los requisitos. Los modelos permiten visualizar un sistema existente o que queremos desarrollar, permiten especificar la estructura o el comportamiento de un sistema, nos proveen por supuesto una guía para la construcción y también nos permiten documentar las decisiones que tomamos.

[V / F] Dentro de la ingeniería de software los modelos son relativamente simples de desarrollar.<sup>4</sup>

## Procesos

Un proceso, es un conjunto de actividades que transforma una entrada en una salida y que consume recursos. En ingeniería de software esta área de conocimiento estudia cuáles son los procesos más adecuados para producir software.

## Calidad y procesos

Es importante entender que el desarrollo de software, la calidad del software, depende en buena medida de la tecnología que empleemos, las herramientas que empleemos, la capacidad de los recursos humanos y los procesos.

En ingeniería de software tenemos una gran variedad de procesos. Todos apuntan a poder minimizar los costos que implica salir con un producto al mercado, con un producto a nuestros usuarios que no tienen el nivel de calidad adecuada. No hemos definido formalmente calidad. **Calidad** es la totalidad de características que permiten que un producto o servicio satisfaga las necesidades explícitas o implícitas. Como podemos ver en el gráfico, es económicamente conveniente invertir recursos en la capacitación de la gente, en la definición de estándares, en la definición de procesos estándar, en lugar de tener que invertir, gastar recursos en trabajos, en devoluciones, en quejas.



## Procesos, ciclos de vida



<sup>4</sup> Por el contrario, el modelado tiende a ser lo más complejo de desarrollar.

El tema de procesos, en el ámbito de la ingeniería de software, se ha discutido ampliamente. Nos vamos a encontrar con que hay 2 grandes etapas en el ciclo de vida de los productos de software: el desarrollo y la producción. El software normalmente es el resultado de la ejecución de un proyecto, es lo que ocurre durante el desarrollo. En ese proyecto se instancia un proceso o un grupo de procesos. A veces esos procesos están definidos formalmente, otras veces son más informales. Esas actividades que forman parte del proceso mapean contra alguna de las áreas de conocimiento que hemos visto antes. Algunas de esas actividades van a estar orientadas a la programación, otras al diseño, y por supuesto que va a haber actividades que van a mapear con más de un área de conocimiento.

En la etapa de **producción**, hay por supuesto una serie de actividades y procesos de las que no vamos a hablar mucho por ahora, pero que sí es importante que las diferenciamos de las que ocurren durante el desarrollo. El desarrollo normalmente, decíamos recién, lo que se hace es ejecutar un proyecto. Un proyecto es un esfuerzo temporal que crea nuevos productos, servicios o resultados y que genera, en definitiva, la capacidad de producir valor. A su vez este proyecto tiene su propio ciclo de vida con fases, entregables, actividades, y es de lo que en general nos vamos a ocupar. La operación, la producción, es un esfuerzo permanente que produce los resultados de manera repetitiva, produce valor. Entonces, pensemos que la etapa de desarrollo nos permite obtener un nuevo producto, en la etapa de producción utilizamos ese producto, lo explotamos.

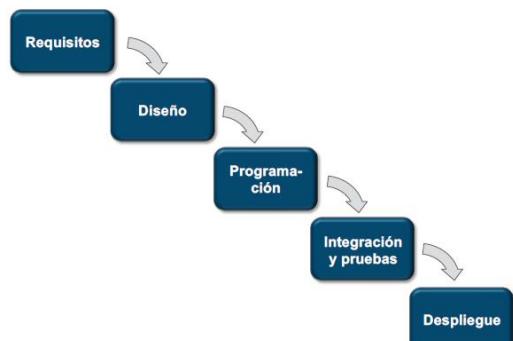
## Desarrollo de software

### Un proceso iterativo e incremental

El desarrollo de software es diferente, como hemos dicho ya, a el desarrollo de productos tradicionales. En la ingeniería tradicional, hay primero un análisis del problema, en el que se descompone al problema en etapas, y después una síntesis de la solución, en donde se definen los elementos que los integran entre sí. En la ingeniería tradicional, por las limitaciones que ya hemos hablado y por lo costoso que es equivocarse, hay una etapa muy grande de definición y planificación. ¿Por qué? Porque hay mucha incertidumbre con respecto a los requisitos, el alcance, los riesgos, etc.; y esto está bien que sea así porque imaginense que si nos largamos a hacer un puente sin tener un plan detallado, si tener un diseño detallado, sería una catástrofe y estaríamos gastando millones de dólares.

### Ciclo de vida en cascada

Con el desarrollo de software se intentó hacer algo parecido al comienzo. Se intentó aplicar un proceso más bien secuencial en donde primero se entiende qué es lo que se quiere hacer, después se diseña, después se planifica, después se construye. Esto se le atribuye una primera referencia a Herbert Bennington, que lo mencionan en un paper en el año 56 acerca de cómo desarrollar software, cómo construir programas en ese entonces. Erróneamente se atribuye la autoría del ciclo de vida en cascada a Winston Royce por un paper que publicó en 1970. En realidad, Royce lo daba como un ejemplo de mal proceso. En general lo que pasó con este ciclo de vida secuencial fue que se aplicó sin considerar las propuestas de Royce para mejorarlo. Royce proponía la posibilidad de volver para atrás, de diseñar 2 veces, etc.



Lo que termina pasando en el desarrollo de software es que el proceso es iterativo e incremental. Entonces analizamos un poco el problema, hacemos una síntesis de la solución y esa definición de la solución nos ayuda a entender un poco más el problema, y así vamos refinando. Es relativamente fácil de hacer esto porque el medio es maleable, no estamos construyendo puentes, estamos desarrollando software.

## Un proceso iterativo e incremental



Entonces nos encontramos en una situación que se ilustra muy bien con estas imágenes que estamos viendo aquí. Vamos paulatinamente entendiendo el problema y definiendo la solución.

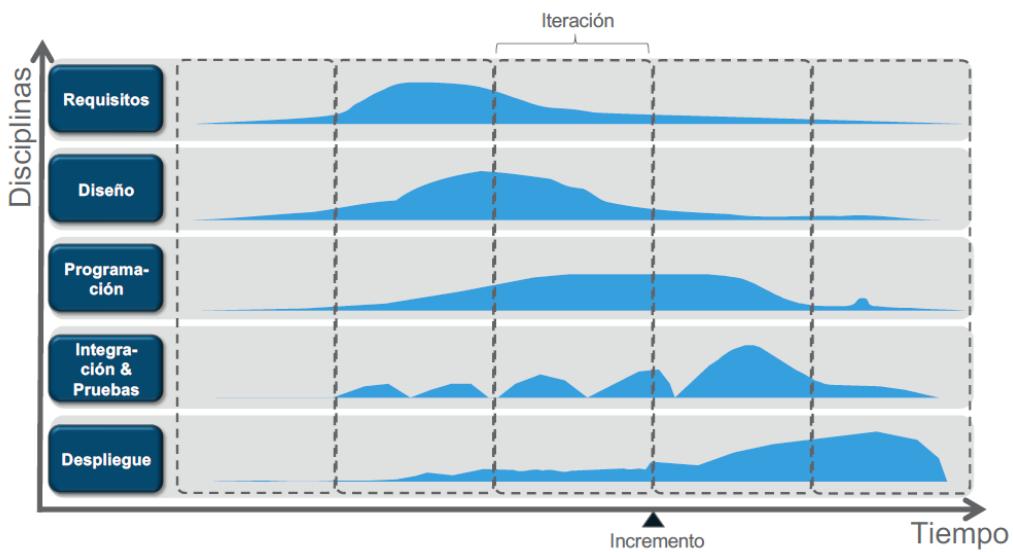
### Casi todo es diseño

Por eso los ciclos de vida en general del desarrollo de software, tienden a tener una etapa un poco más corta de planificación y de definición, y luego se comienza con lo que sería el desarrollo propiamente dicho. La diferencia fundamental es que en el desarrollo de productos tradicionales hay una etapa de planificación y definición muy grande porque el costo, la ejecución después de la fabricación de la producción es en donde se concentra la mayoría de los costos, es en donde hay que utilizar las maquinarias, los operarios. En cambio en el desarrollo de software casi todo es diseño. Es como hacer la comparación entre lo que hace, por ejemplo Toyota. Toyota utiliza un sistema de producción llamado “lean”, muy eficiente por cierto, pero tiene a su vez un proceso distinto para el desarrollo de productos. Es como si yo les dijera, para fabricar cada unidad de autos utilizamos un proceso repetitivo, pero para diseñar un nuevo modelo de auto utilizamos un proceso diferente, menos predecible, menos planificable, un poco caótico porque naturalmente el diseño es así.

## Proceso de desarrollo de software

### Disciplinas y ciclo de vida

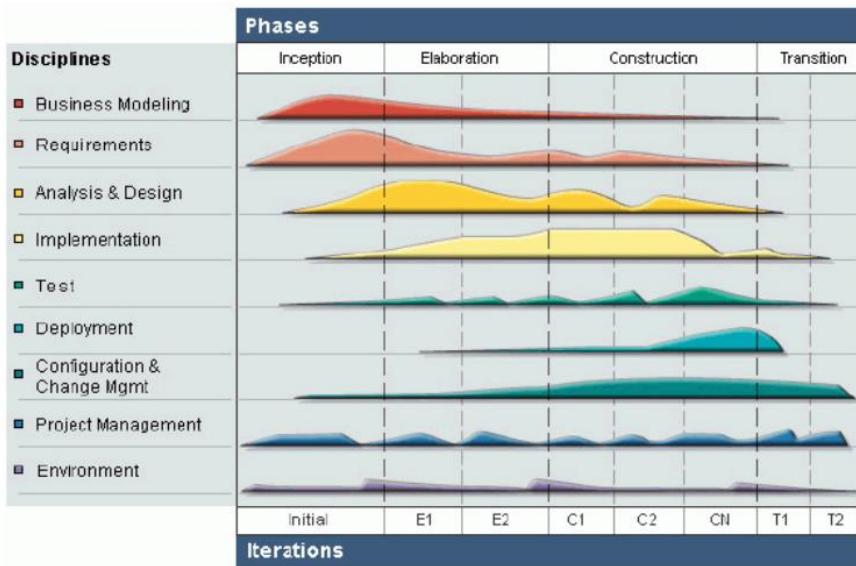
En lugar de encontrarnos con actividades secuenciales, con fases secuenciales, lo que nos vamos a encontrar en el desarrollo de software es que las actividades del cuerpo de conocimiento se ejecutan con distinto grado de intensidad a lo largo de todo el ciclo de vida.



Si tomamos en cuenta que el eje de las x nos describe el tiempo, y el eje de las y nos describe las áreas de conocimientos o disciplinas, lo que busca ilustrar el gráfico, es la distinta intensidad con que se van ejecutando las actividades relacionadas con cada una de esas disciplinas. No todos los requisitos se definen al comienzo del proyecto. No todo el diseño se define en las primeras etapas sino que a medida que vamos avanzando entendemos mejor los requisitos, estamos en condiciones de diseñar la solución, el diseño de la solución nos permite entender mejor los requisitos.

## Modelos de referencia: (IBM) Rational Unified Process

Uno de los procesos que tomó en cuenta estas ventajas, es el Rational Unified Process que surge a finales de los años 90, a principios del 2000, y lo que busca justamente es lo que ilustra:



El desarrollo de software se separa en fases que son secuenciales a lo largo del ciclo de vida. Cada fase a su vez se separa en iteraciones, que son pequeños esfuerzos de 2, 3, 4 semanas con una cantidad de trabajo acotado, que lo que busca es ir refinando la definición del problema y refinando la definición de la solución. Se sabe ya desde fines de los años 60 del siglo pasado que el desarrollo de software naturalmente es iterativo e incremental, que no conviene utilizar un enfoque secuencial como el que se planteaba a fines de la década del 50.

## Desarrollo iterativo e incremental

### Agile Manifesto (2001)

*We are uncovering better ways of developing software by doing it and helping others do it.*  
*Through this work we have come to value:*

- Individuals and interactions over processes and tools*
- Working software over comprehensive documentation*
- Customer collaboration over contract negotiation*
- Responding to change over following a plan*

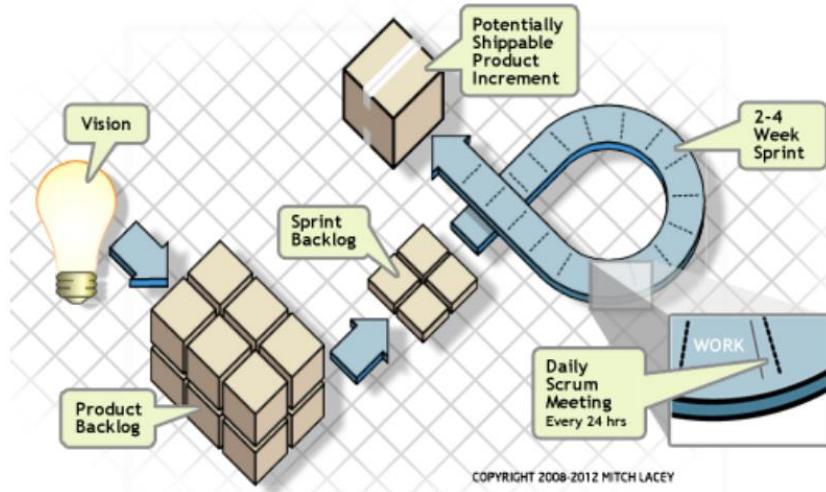
*That is, while there is value in the items on the right, we value the items on the left more.*

Esto se hace todavía más presente cuando, ya comenzado este siglo, aparece el manifiesto ágil que da como resultado una declaración que es este famoso manifiesto en donde se declara que: "Es más importante o se valoran más los individuos y las interacciones, que los procesos y las herramientas. El software trabajando que sobre la documentación. La colaboración con los clientes sobre la negociación del contrato. La capacidad de responder a los cambios sobre seguir un plan". Y esto es en respuesta, un poco, a los enfoques que algunos llaman "plan driven" o sea muy orientados a tener una definición y un plan detallado antes de comenzar. Lo que sí es cierto, es que no todo el mundo toma en cuenta lo que dice el último párrafo, es decir, hay valor en los ítems de la derecha (procesos, herramientas, documentación, etc.) pero ellos valoran más los ítems que están a la izquierda. Lo cual no quiere decir que no haya procesos ni estándares documentados ni que haya planes, lo que quiere decir es que tienen que ser mucho más dinámicos, tienen que ser bastante diferentes de como uno lo ve por ahí en las ingenierías tradicionales.

## Proceso de desarrollo de software

### Modelos de referencia

#### Scrum



Uno de los modelos de proceso, uno de los modelos de referencia muy popular hoy y que fue en parte contemporáneo a la aparición del manifiesto ágil, es **Scrum**. En Scrum se plantea, como ilustra claramente el gráfico, una serie de ciclos. El desarrollo de software se separa en ciclos, en **releases**. A su vez dentro de cada release que nos permite generar una versión del producto, lo que tenemos son **sprints**, son iteraciones. O sea que, si por ejemplo, decidimos que vamos a tener una versión del producto cada 3 meses, vamos a separar ese lapso de 3 meses en iteraciones, en sprints de por ejemplo 2 o 4 semanas. Si decidimos que vamos a tener sprints de 4 semanas, claramente para producir un reléase, en este ejemplo, lo que vamos a necesitar son 3 meses. Vamos a tener 3 sprints de 4 semanas cada uno. ¿Cómo funciona este proceso? Bueno, hay un dueño de producto que es el que establece la visión del producto: nos define cuál es el mercado; nos define cuáles son los objetivos; nos define las grandes features. A partir de esa visión, se elabora un "**product backlog**". ¿Qué es un product backlog? Una definición de las funcionalidades que tiene que tener el producto. Al comienzo de cada release, se define, estimativamente, cuáles van a ser las features que van a salir en cada release. Al comienzo de cada sprint, el equipo de desarrollo en conjunto con el product owner, toma del product backlog los ítems que corresponden a la iteración, al release, y deciden cuáles van a salir en el sprint. Esa información se utiliza para preparar lo que se llama el "**sprint backlog**". Ese sprint backlog es un detalle de las actividades que el equipo tiene que llevar adelante para poder transformar esas features que están en el product backlog asignadas y que se definió que iban a ser desarrolladas en el sprint a los distintos individuos que forman parte del equipo de trabajo.

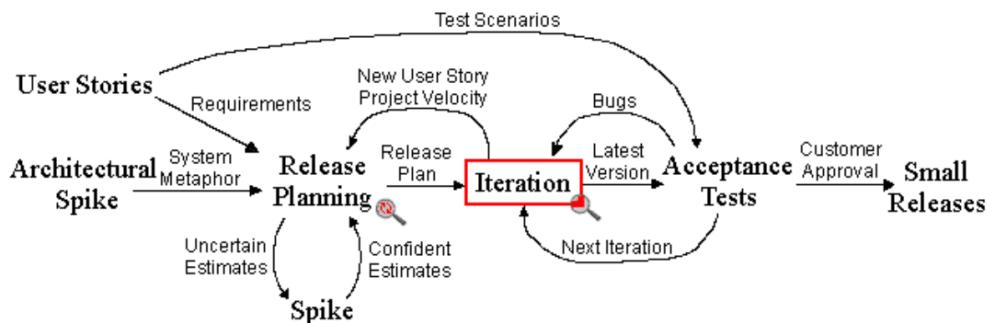
También hay una serie de actividades, rituales o ceremonias diarias. El día comienza con una reunión llamada "Daily Stand up Meeting". Es una reunión corta en donde la gente está de pie y en donde se plantea: qué fue lo que se hizo el día anterior, qué es lo que se va a hacer durante el día y se plantean algunos issues que podrían haber surgido. De ninguna manera se intentan resolver los problemas en esa reunión sino que se depura la lista de temas pendientes para que cada uno pueda asignarle la resolución de esos issues (problemas) a quién a quién corresponda. Entonces al final de cada día, algunas aplicaciones de scrum lo que proponen es que haya una integración con el resto del código producido por los distintos programadores del equipo. O sea, cada uno trabaja en su propio espacio de trabajo y al final del día se propone integrarlo. Lo cierto es que más allá de esta práctica de integración continua e integración diaria, lo que puede llegar a pasar o lo que normalmente pasa es que al final del sprint hay que integrar y hay que probar, con lo cual al final del sprint el producto estaría potencialmente entregable. Lo que queremos decir

<sup>5</sup> El manifiesto ágil promueve la planificación.

con esto es que, a pesar de que tenga funcionalidad limitada, el producto tiene una funcionalidad mínima que va a ir creciendo a lo largo de cada sprint y a lo largo de cada release.

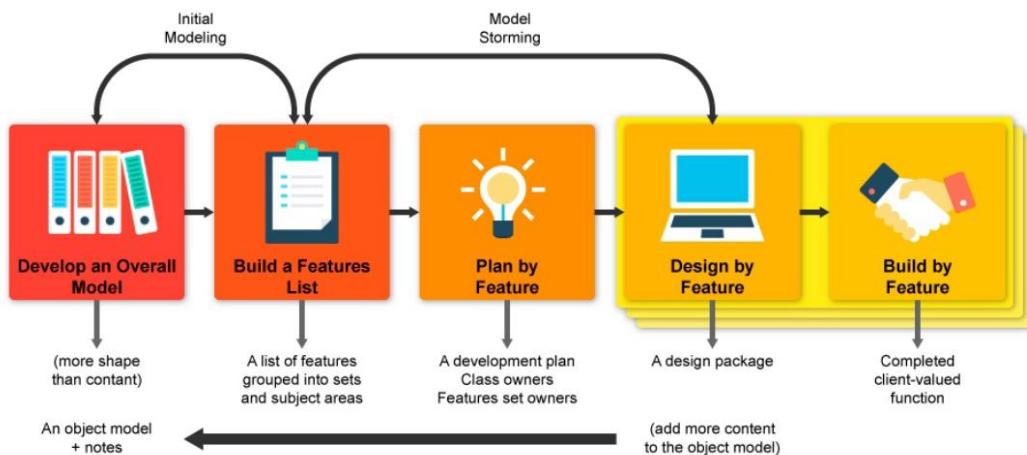
Lo otro que también propone Scrum es que, al final de cada sprint, se revise el sprint backlog, se hace una retrospectiva del sprint para ver qué fue lo que funcionó, qué fue lo que no funcionó, qué cosas hay que mejorar en el proceso de desarrollo, se decide cuáles son los elementos del backlog que se van a tomar para el próximo sprint, y se analiza si quedó algún tema pendiente.

## Extreme Programming (XP)

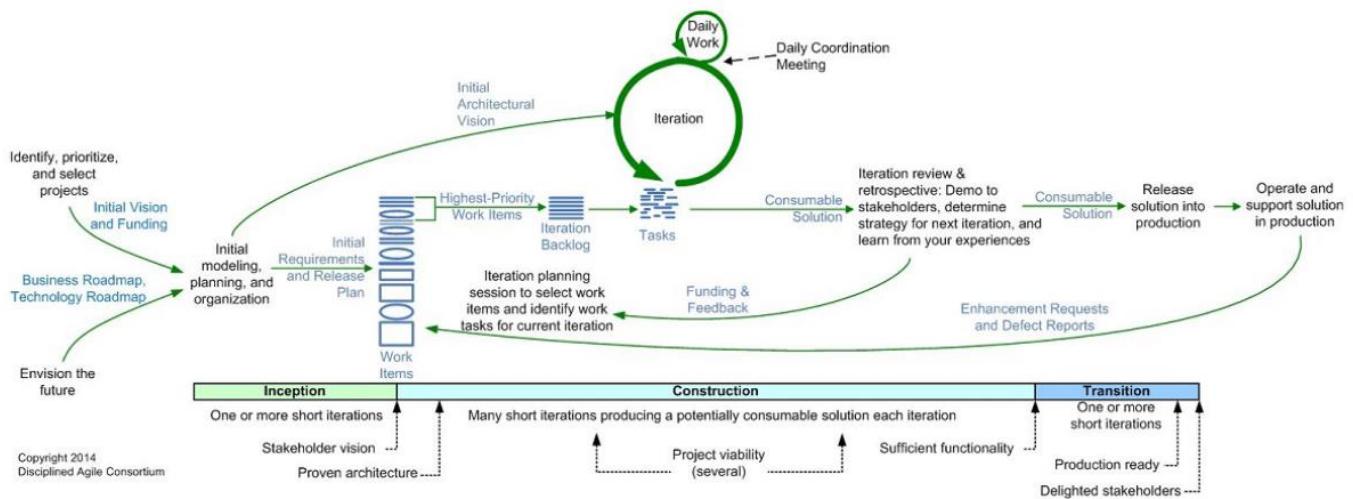


Hay otros procesos similares como XP (**Extreme Programming**) que también propone un desarrollo iterativo e incremental en donde es justamente, como el nombre se indica, se toman al extremo algunas prácticas que tienen que ver con algunas prácticas de desarrollo muy puntuales. Con Extreme Programming surge la técnica de User Stories (de la cual hablaremos más adelante)

## Feature Driven Development (FDD)



## Disciplined Agile Delivery (DAD)



No se agota acá el tema. Hay otros métodos como **Feature Driven Design** o como **Disciplined Agile Delivery** que es una aplicación de scrum en un contexto un poco más disciplinado. Es decir, lo que nos vamos a encontrar es con que hay varios tipos de procesos disponibles, varios modelos de referencia disponibles que podemos aplicar en distintos contextos. Aplicar y cómo aplicarlo van a depender por supuesto de nuestra experiencia, del contexto y de una serie de variables. Vamos a estudiar con más profundidad y vamos a ver qué papel juegan en este contexto los procesos, los métodos y los modos.

## Conclusiones

Para cerrar entonces, a manera de resumen de lo que hemos visto en este vídeo:

- El software es claramente una entidad compleja, es una de las entidades más complejas que haya fabricado el hombre. Es una entidad relativamente poco conocida que es el resultado de un proceso instanciado durante la ejecución de un proyecto
- El desarrollo de software hemos visto que no se parece demasiado a la ingeniería clásica. La ingeniería tradicional comparte algunos elementos pero tiene algunos aspectos que son particulares y que tienen que ver más con actividades creativas como por ejemplo escribir un libro, producir una película. Pensemos que cuando estamos produciendo una película a veces está el guión, y a veces no está el productor, no está el director, a veces está el director, están los actores y no se definió todavía el guión, a veces está el presupuesto, está la fecha de entrega, pero no está el guión; todo en definitiva se termina combinando y un guion original, con una serie de ideas y de diálogos, muchas veces es ajustado en el mismo set y la película termina de cobrar forma en la isla de edición. Eso no pasa con la ingeniería tradicional en general y si pasa con la ingeniería de software. ¿Por qué? Porque casi todas las variables de un proyecto desarrollo se pueden ajustar: si tenemos una fecha de entrega fija y tenemos un costo fijo, bueno por ahí podemos ajustar qué es lo que entregamos, qué funcionalidades va a tener el producto y cuáles no: Si la funcionalidad es fija bueno tendremos que negociar tiempo y recursos. Todo en definitiva es el resultado de una ecuación que combina determinadas variables.
- Naturalmente el software se desarrolla de manera iterativa e incremental. Claramente, el enfoque secuencial no es el más adecuado, aunque a veces puede haber circunstancias que nos fuercen adoptar un enfoque de ese tipo.
- La ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software.

El desarrollo de software se compone de:

- Personas, procesos y productos
- Procesos, personas, métodos y modelos
- Procesos, personas y tecnologías**
- Individuos, técnicas y procesos

[V / F] Un proceso es un conjunto de pasos que producen un resultado.<sup>6</sup>

Identifique todos los modelos o metodologías de desarrollo que son iterativos e incrementales:

- Cascada<sup>7</sup>
- Scrum**
- RUP (Rational Unified Process)**

<sup>6</sup> Un proceso es un conjunto de actividades que transforma una entrada en una salida y que consume recursos.

<sup>7</sup> Cascada es de tipo secuencial.

# 1.1- Calidad y procesos en la industria del software

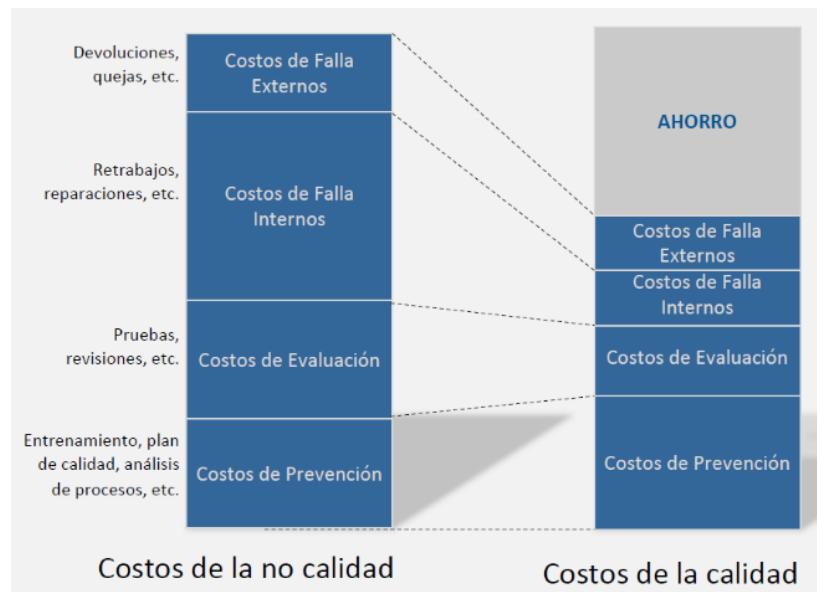
## Calidad

### Definición: ISO 8402

Antes que nada deberíamos definir qué es calidad. Hay múltiples definiciones posibles. Prácticamente todos los autores que han escrito acerca de calidad han dado su propia definición, nosotros nos quedamos con esta que es la estándar de la ISO y es que calidad es:

*Totalidad de prestaciones y características de un producto o servicio que generan su capacidad de satisfacer necesidades explícitas o implícitas*

Claramente la calidad no es gratis. Cuando no hay calidad, incurrimos en los llamados **costos de falla externos**, que son aquellos que tienen que ver con todas las acciones necesarias para atender los reclamos de los clientes. Pensemos en lo que nos pasa en nuestra vida personal cuando tenemos que llamar a algún fabricante por algún electrodoméstico que acabamos de comprar y que no funciona de acuerdo a lo previsto o que tiene algún tipo de falla. Ese fabricante tiene que tener dispuesta una mesa de ayuda, tiene que tener una línea de teléfono dedicada, tiene que tener personal capacitado para atender todos esos reclamos. También hay una serie de costos llamados **costos de falla internos**, que son todos los relacionados con las actividades necesarias para reparar esos productos defectuosos que nos han llegado. La situación habría sido muy diferente si hubiéramos invertido más en prevención y en actividades de evaluación.



Como muestra el gráfico, invertir en estos 2 rubros claramente produce ahorros, ya que un producto de mayor calidad, una vez entregado para su uso, necesitará menos retrabajo y producirá menos reclamos. Invertir en entrenar a la gente, en definir procesos estándar, en tener buenas herramientas, claramente es económicamente conveniente.

## Calidad y procesos

### Gestión cuantitativa

La preocupación por la calidad y los procesos no es algo nuevo, lleva siglos. La industria tiene una larga historia acerca de su interés por tener procesos estables, bajo control estadístico. Hoy hablamos de cuatro grandes elementos en el mundo de la calidad:

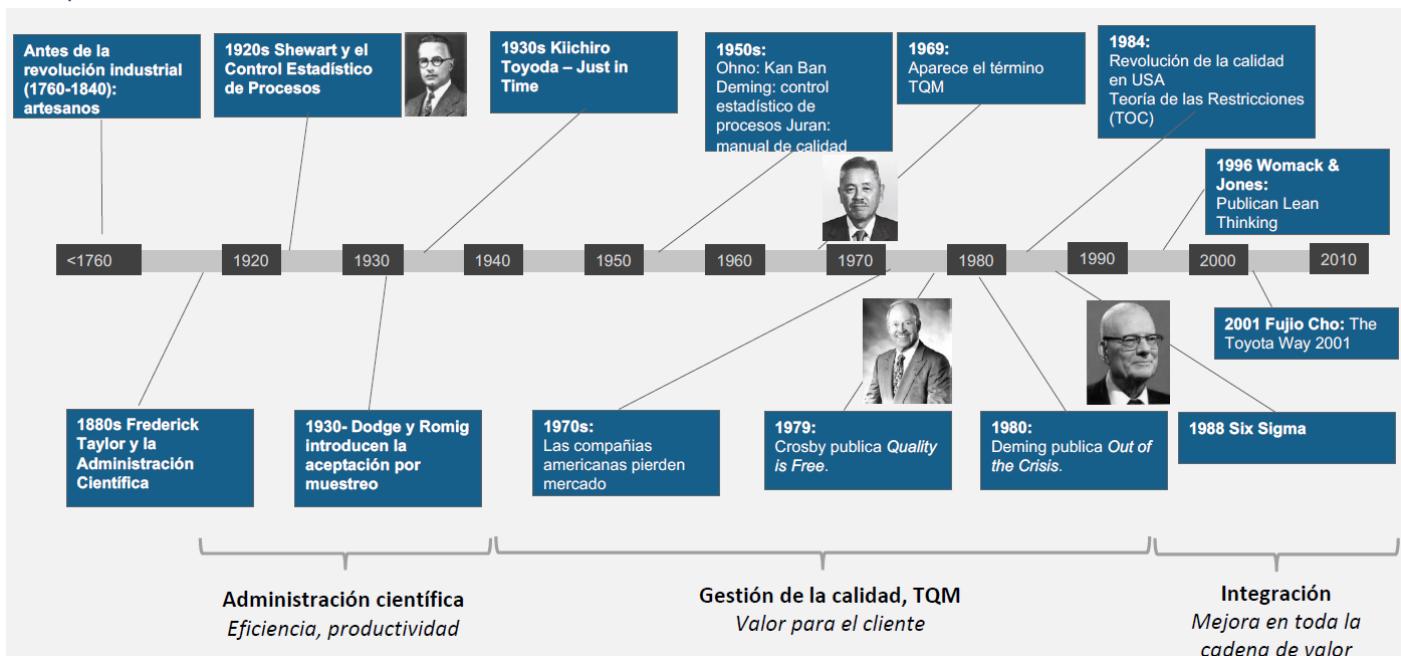
- **Control:** El control de la calidad es equiparable a lo que en nuestra profesión conocemos como testing. Son todas las actividades y técnicas que utilizamos para evaluar si un producto o servicio cumple con determinados requisitos de calidad.
- **Aseguramiento:** El aseguramiento de la calidad, busca proveer cierto nivel de confianza de que el producto o servicio satisface determinados requisitos de calidad. Parece un juego de palabras, un trabalenguas, pero en realidad, el aseguramiento de la calidad lo que busca es que estemos seguros de que hicimos las cosas de acuerdo a lo que dicen nuestros estándares, mientras que control de calidad lo que apunta es ver si realmente nuestro producto o servicio cumple con los requisitos.
- **Mejora:** Mejora de la calidad lo que busca es la mejora continua de los procesos productivos.
- **Gestión de la calidad:** Es el paraguas que le da el entorno lo más eficaz posible para que esto funcione adecuadamente, provee los aspectos de planificación, organización, etc.

[V / F] Mayor inversión en prevención y actividades de evaluación conlleva a una mayor calidad y un ahorro de costos futuros.

## Definiciones: Verificación Vs. Validación

Hay 2 términos que son parecidos pero que no son lo mismo: Verificación y validación. **Verificación** tiene que ver con la pregunta “**¿Lo hicimos bien?**”, mientras que **validación** con la pregunta “**Hace lo que tiene que hacer?**”. Para hacer el paralelo con el desarrollo de software, verificación, es equiparable al testing, mientras que validación se parece, o tiene que ver, con el test de aceptación. Esto lo que quiere decir es: el producto que desarrollamos satisface los requisitos, ¿sí o no? La validación tiene que ver con el producto que desarrollamos cumple con las necesidades y expectativas del cliente, esa es la diferencia, a veces esos términos se utilizan de manera intercambiable, con lo cual hay que tener cuidado con la bibliografía a la que uno accede porque a veces uno habla de verificación y en realidad se está refiriendo a lo que llamamos nosotros en este contexto validación.

## Un poco de historia



Desde la Revolución industrial hasta nuestros días hemos ido madurando todos estos conceptos que acabamos de presentar. La primera etapa en esta historia ha sido la administración científica que estuvo muy, muy enfocada en el trabajo mecánico de los operarios y en la producción en masas. Eso a veces incluye a lo que llamamos el Fordismo por Henry Ford. Los años 20 del siglo pasado, Walter Shewart trabajó en el control estadístico de procesos. La idea

era que la estabilidad de los procesos era muy importante para poder tener resultados predecibles. Imaginémonos que si uno fabrica un millón de unidades, no puede hacer una prueba sobre ese millón de unidades, tienen que poder tomar una muestra estadísticamente significativa de ese lote y hacer pruebas sobre esa muestra. Y hacía falta entonces tener algún tipo de aseguramiento, de garantía, de que esa muestra era estadísticamente significativa.

En Japón, mientras tanto, en esa época comienza un movimiento que llega hasta nuestros días y que hoy se conoce como "lean manufacturing". Todo esto arranca de la mano de los hermanos Toyoda fundadores y directores de la empresa Toyota. Después de la Segunda Guerra Mundial, Japón queda completamente devastado. Se decide reconstruir Japón. Los aliados ayudan en esa iniciativa. Arranca entonces para esa época el movimiento de calidad de muchos de los gurúes de Estados Unidos que viajan a Japón a ayudar a reconstruir ese país. Son enviados allá como parte del Plan Marshall para recuperar la economía japonesa. Esto, sumado a lo que ya venía pasando, hace que la industria japonesa explote, y justamente desafíe a la industria automovilística norteamericana. Para los años 80, la industria americana está en peligro porque los japoneses habían aprendido a fabricar automóviles de una manera impresionante, con mucha calidad, a costos mucho más interesantes, lo cual hace trastabillar un poco a la industria en general. Por ese entonces aparece un documental en la televisión americana que se llamó "Porque Japón pudo y nosotros no", en donde, justamente, se dan a conocer las ideas de estos gurúes americanos que a lo mejor no eran escuchados en su país natal y que sí tienen éxito apuntalando lo que se venía haciendo en Japón de la mano de Toyota y tantas otras.

Esto inicia el proceso de calidad, el movimiento de calidad en Estados Unidos que llega hasta nuestros días. Este movimiento de calidad llega hasta nuestros días con desarrollos como Six Sigma y lean, que son los que están dominando el pensamiento acerca de la calidad.

## Calidad y procesos

### PDCA

Mencionamos recién a Walter Shewhart con sus ideas del control estadístico de procesos. Shewhart también es el autor de un ciclo de mejora continua, hablar de calidad y no hablar de mejora continua es prácticamente ridículo. La calidad se debe mejorar continuamente. Encontrar mejores formas de trabajar es responsabilidad de todos, desde el primer empleado hasta el director de la compañía. Walter Shewhart en los años 20 crea este ciclo de mejora llamado



**PDCA o "Plan – Do – Check – Act"**, que fue luego muy popularizado por Deming, uno de los gurúes que va a Japón como parte del Plan Marshall que mencionábamos antes. Este ciclo de mejora continua es lo que ha inspirado muchos de los procesos de desarrollo iterativos e incrementales de desarrollo de software que hoy vemos en el mercado. Lo que propone este proceso son cuatro fases:

- 1º) **Plan:** Se establecen los objetivos y el plan para alcanzar.
- 2º) **Do:** Se ejecuta el plan, que básicamente es desarrollar y probar pequeños cambios. Imaginemos que esto lo usábamos en un proceso productivo. Durante la planificación lo que tratamos de entender es cuáles son los problemas que tiene ese proceso productivo; hacemos un pequeño cambio; lo probamos; se reúnen los datos.
- 3º) **Check:** Esa mejora que hicimos, ¿sirve para resolver el problema originalmente identificado? Eso se evalúa en la etapa de check. ¿Se mejoró?, ¿se puede mejorar más?
- 4º) **Act:** Se implementan esos cambios en la fase de act, y en función de los resultados obtenidos, se inicia un nuevo ciclo para volver a identificar nuevas oportunidades de mejora.

Es un ciclo que no termina nunca, y eso ha permeado, y está en todas las escuelas de pensamiento de calidad que vemos hoy en día.

¿Lo hicimos bien?

## Enfoques vigentes hoy

Hace un momento mencionábamos a Six Sigma y a Lean como 2 de los movimientos sobre las escuelas de pensamiento de calidad que están vigentes hoy en día. Podríamos agregar una tercera: "Teoría de las restricciones" (TOC). Las 3 están emparentadas: **Six Sigma** empieza en General Electric como una metodología para resolver problemas en los procesos. Usa herramientas de distintos tipos, algunas estadísticas están muy orientadas a reducir la variación y el énfasis está puesto justamente en resolver problemas. **Lean thinking**, por otro lado, usa muchas de las mismas herramientas, pero tiene un foco muy orientado a remover el desperdicio y a facilitar el flujo de trabajo. Y la tercera, la **teoría de restricciones** está, también muy relacionada con las otras 2, y lo que busca es identificar y resolver las restricciones que tenemos en cualquier sistema productivo, en cualquier sistema en general.

Esto viene de la mano de un señor que se llamó Eliyahu Goldratt, que escribió una serie de libros muy interesantes, entre ellos "La meta", en donde plantea, algo que, por ahí puede parecer obvio, pero a veces no tanto, y es que en cualquier sistema hay una única restricción que condiciona el funcionamiento del todo. Es como decir que en una cadena el eslabón más débil es el que termina definiendo la fortaleza de la cadena. O lo mismo que, cuando salimos a correr en grupo, y queremos todos correr al mismo ritmo, el grupo va a terminar corriendo al ritmo del que corre más lento. Lo interesante de la teoría de restricciones es que la mejora tiene que estar orientada a resolver esa restricción, a resolver esa máquina, a resolver esos problemas que tenemos con esa máquina que no está funcionando bien. Todas las mejoras que hagamos alrededor, son simplemente cosméticas. Si no nos focalizamos en resolver el verdadero problema que es esa restricción única que es la que condiciona el funcionamiento de todo el sistema.

## Mejora continua

Lean, por otro lado, está muy enfocado en la mejora continua, en poder eliminar esos tiempos de espera. En cualquier proceso nos vamos a encontrar que la mayoría de las actividades se quedan esperando muchas veces a que haya un OK, a que esté disponible algún tipo de recurso. En general, todas las metodologías que hemos mencionado apuntan a mejorar la calidad de forma integral, en particular lean, lo que busca es remover el desperdicio, remover esos tramos grises que están ahí para poder mejorar la calidad, acortar el tiempo de ciclo, remover las restricciones, aumentar la colaboración. Y esto que se dice fácil en la práctica es bastante complicado de implementar y la mayoría de las filosofías que hoy tenemos disponibles en el mundo del software están sumamente inspiradas, sobre todo en estas últimas 3 que hemos mencionado y en particular en lean.

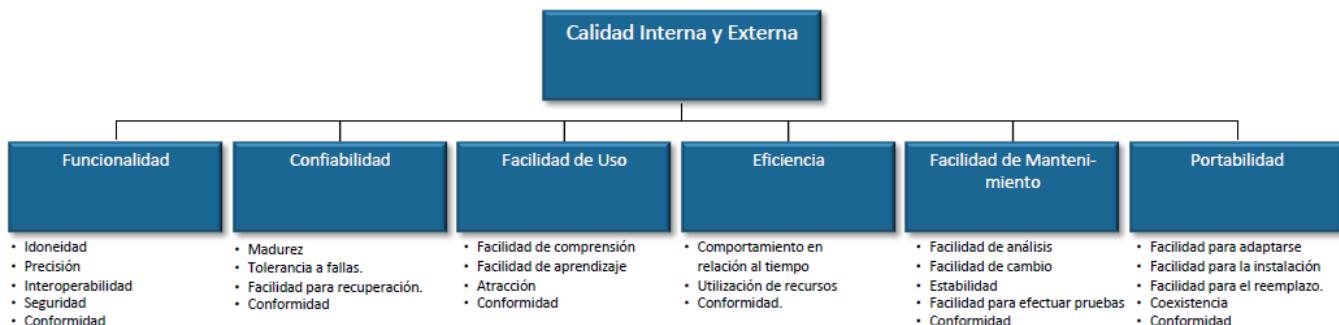
## Calidad del software

En el ámbito del software, la calidad ha sido, por supuesto, una preocupación permanente. Sabemos que el software tiene un impacto altísimo en el funcionamiento de la economía moderna. Sabemos el impacto que tiene el software cuando no funciona bien. Hay 2 grandes grupos de factores de calidad de software:

- ◆ Un primer grupo, que tiene que ver con factores que son **observables durante su uso**:



- ◆ Otro grupo que son más bien **internos y externos**:



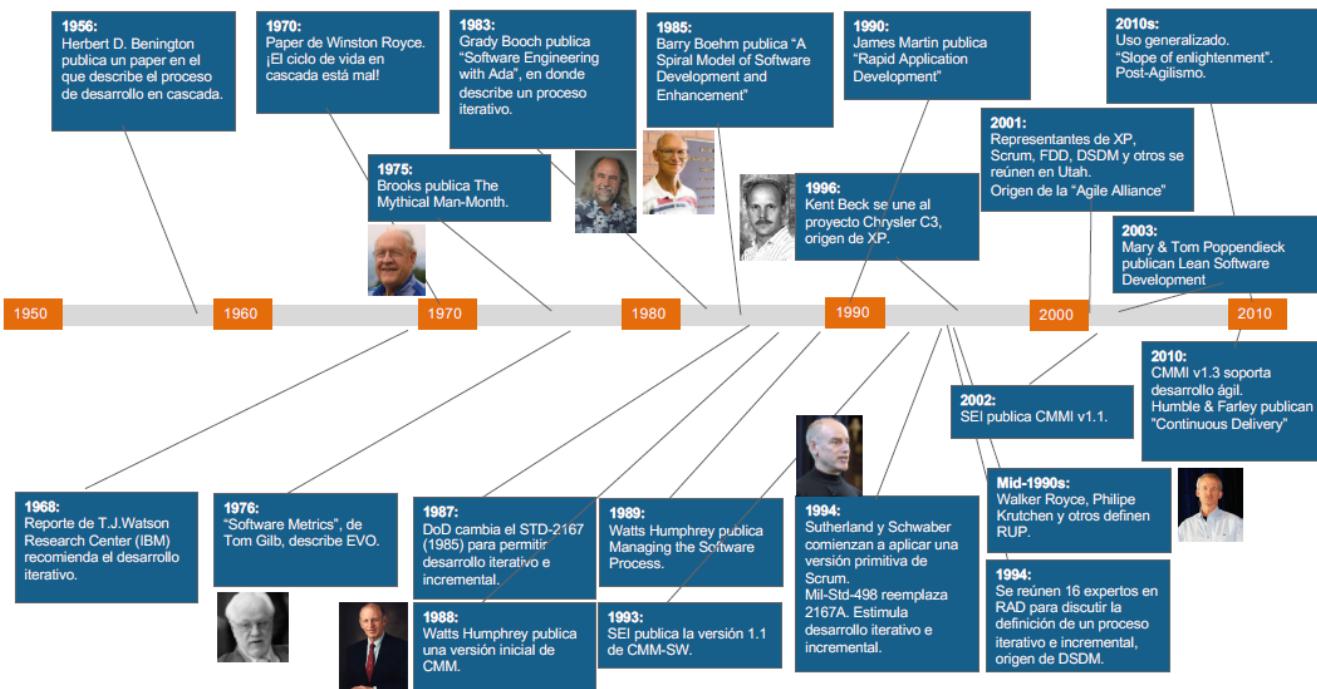
A los usuarios por supuesto, le va a interesar que el software haga correctamente lo que se supone que debe hacer, es decir, que cumpla bien los requisitos. Mientras que a los desarrolladores les va a interesar que sea fácil de mantener, que sea fácil de entender. A la gente del data center le va a interesar que hagan un buen uso de los recursos.

El *testing* está asociado con...

- Validación  
 Verificación

## Calidad y procesos en la industria del software

### Perspectiva histórica



Prácticamente desde el inicio del software, la preocupación por la calidad ha sido algo sistemático. Al igual que pasó en otras industrias, se buscó permanentemente nuevos métodos, nuevos enfoques. Se llegó a la conclusión, obviamente que, hacía falta tener gente capacitada, buenas herramientas, pero también, buenos procesos, procesos eficientes, procesos que permitieran producir software de calidad. En el año 56 había aparecido un paper inicial, en donde se proponía desarrollar software de la misma manera que se desarrollaba hardware. Rápidamente se llegó a la conclusión de que el software era un animal distinto, que había que enfocar el desarrollo de otra manera. Ya en el año 70, Winston Royce decía que el ciclo de vida en cascada, heredado de la ingeniería de hardware, era incorrecto. En ese ciclo de vida lo que se planteaba era una idea de desarrollo secuencial, en donde primero había que entender cuál era el problema, después desarrollar la solución y después probarla, y hoy sabemos que las actividades de desarrollo de software se van ejecutando de manera no necesariamente secuencial. Sabemos que entendemos un

poco el problema, formulamos una solución, ese entendimiento de la solución me permite reformular y entender un poco mejor el problema y eso ha dado origen a infinidad de propuestas que apuntan a que el desarrollo de software es una actividad iterativa e incremental inspirados en modelos como PDCA propuesto por Shewart hace casi 100 años. Hoy contamos con infinidad de modelos de proceso o modelos de referencia que lo que proponen justamente son ciclos, ciclos continuos de desarrollo en donde vamos refinando paulatinamente nuestro entendimiento del problema y nuestra formulación de la solución.

A lo largo de estos 50 años hemos entendido claramente que el software es una entidad diferente, maleable, intangible, que requiere un proceso distinto. Ya a mediados de los 70, Tom Gilb presentaba, probablemente uno de los primeros ejemplos de propuesta de ciclo de vida iterativo e incremental. Ya antes, en el año 68, un reporte de IBM decía, advertía y sugería que el desarrollo de software no podía ser secuencial, no podía inspirarse demasiado en la ingeniería tradicional, sino que hacía falta un enfoque distinto. Hubo mucha gente que estudió la mejor forma de producir software, entre ellos Watts Humphrey, que fue uno de los pioneros del estudio de la calidad en nuestra industria. Humphrey escribió un libro famoso, publicado a fines de los 80, llamado “Managing the Software Process”, que planteaba un esquema a seguir para mejorar progresivamente la calidad del proceso de desarrollo de software a través de los llamados niveles de madurez. Hoy en día esos 5 niveles de madurez son ampliamente conocidos y se utilizan en otros en otros ámbitos. También en los 80 Barry Boehm hablaba del ciclo de vida en espiral. A mediados de los 90 ya comienzan los primeros intentos con Extreme Programming y Scrum que son, digamos, enfoques claramente inspirados en el ciclo PDCA, y llegamos a nuestros días en donde ya los métodos, los llamados métodos ágiles que están muy inspirados en PDCA, en la filosofía Lean y en tantos otros movimientos de calidad están ampliamente difundidos. Hoy la preocupación más grande pasa por ver de qué manera conseguimos llevar la agilidad más allá del ámbito de desarrollo de software, hablamos de la integración entre desarrollo y operaciones (DevOps), hablamos de integración entre desarrollo de operaciones y seguridad informática (DevSecOps), hablamos de transformación digital, hablamos de integrar no solamente desarrollo de operaciones y seguridad, sino también las áreas de negocio con los grupos de desarrollo, operaciones y seguridad.

Un largo camino se ha recorrido. La preocupación por la calidad del software sigue vigente. Hoy sabemos mucho más, entendemos mucho más. Todavía nos queda un montón por aprender y por entender. Está claro que la tarea de software no depende únicamente de tener la última herramienta de moda, de saber el último lenguaje de programación, de saber utilizar el último framework, sino que necesitamos gente capacitada, gente motivada y un muy buen proceso; un proceso que nos oriente, nos organice, no necesariamente un proceso que nos agobie, sino un proceso que nos permita ser creativos y usar esa creatividad en producir software de calidad en forma económica.

¿Cuál o cuáles de los siguientes son enfoques de calidad vigentes?

- JIT
- Lean
- ToC
- PDCA

# 1.2 - Modelos en la ingeniería de software

## ¿Qué es un modelo?

Antes que nada es importante que recordemos qué es un modelo. Un **modelo** es una **simplificación de la realidad**, es una representación de algún aspecto de la realidad que nos interesa. En el contexto del modelado de sistemas, desarrollamos modelos con el **propósito de entender mejor** el software que queremos construir y muchas veces también para entender el software que ya está construido y que necesitamos entender para poder mantenerlo, para poder darle evolución. Es muy difícil entender la totalidad de los aspectos de un sistema, por eso vamos a encontrarnos con que muchas veces vamos a tener que desarrollar varios y diferentes modos.

Brooks, en “No Silver Bullet”, nos decía que al no existir el software en el espacio, es muy difícil poder representarlo mediante modelos, mediante diagramas. La ingeniería tradicional, la arquitectura, tienen la suerte, si se quiere, de poder utilizar planos, poder utilizar diagramas que no son ni más ni menos que abstracciones geométricas de *realidades* geometría. Cuando queremos llevar esa misma experiencia, esa misma práctica, al ámbito de la ingeniería de software, nos encontramos con que no hay un diagrama que sirva para mostrar todos los aspectos que nos interesan, terminamos utilizando varios. Y al no existir en el espacio es medio difícil hacer un mapeo entre lo que representan esos modelos y los que es el software. Por otra parte, recordemos que el código es, ni más ni menos, que otro modelo. Es una representación del comportamiento que tiene el software. Entonces, usualmente, cuando pretendemos modelar un sistema, nos vamos a encontrar con un montón de diagramas que representan en algunos casos la base de datos o el modelo conceptual de datos; nos vamos a encontrar con diagramas que representan la arquitectura o que representan el flujo de datos; o que representan el control. Entre todos esos tipos de diagramas, podemos terminar de entender qué es lo que el sistema hace o tiene que hacer.

## Modelos y métodos

### Una breve (e incompleta) perspectiva histórica



~1967: Programación Estructurada.	
~1973: SADT/IDEFO (Ross)	Structured Analysis and Design Technique
~1975: Diseño Estructurado (Constantine & Yourdon)	Módulos, cohesión, acople. Diagrama de estructuras.
1976: Diagrama de Entidad-Relación (Chen)	Modelado de datos
~1978/9: Análisis Estructurado (Gane & Sarson, DeMarco)	Funciones, datos. Descomposición funcional. Diagramas de flujos de datos.
1980: Smalltalk.	
1981/3: Ingeniería de la Información (Finkelstein/Martin)	Estrategia y análisis del negocio, entidades, procesos
1982: Diseño orientado a objetos [OOD] (Booch)	
1984: Essential Systems Analysis (McMenamin & Palmer)	Partición por eventos. Esencia vs. Implementación.
1988: Object-Oriented System Analysis (Schlaer/Mellor)	
1989: Modern Structured Analysis (Yourdon)	
1990/1: Object Oriented Analysis; Object oriented Design (Coad/Yourdon)	
1991: Rapid Application Development (Martin)	
1991: Object-Oriented Modeling and Design (Rumbaugh); Object-Oriented Design With Applications (Booch)	Casos de uso. Modelos de análisis y diseño.
1992: Object-Oriented Software Engineering (Jacobson)	
1995: Architectural Blueprints: The “4+1” View Model of Software Development (Kruchten)	
1997: UML v1.0	Unificación de las notaciones propuestas por Booch, Rumbaugh y Jacobson.
1998: Extreme Programming.	User stories
1999: Proceso Unificado.	
2001: Agile Manifesto	Fases, iteraciones, modelos del sistema basados en “4+1”
2005: UML 2.0	

La preocupación por modelar sistemas nace prácticamente con el nacimiento de la ingeniería de software. En los años 70, mucho del desarrollo vino de la mano de la programación estructurada. Al surgir la programación estructurada, aparece la necesidad de tener un diseño estructurado, y al aparecer un diseño estructurado, aparece el análisis estructurado también. En los 90 del siglo pasado aparece la orientación a objetos muy fuertemente y, de

vuelta, se repite el mismo patrón: aparecen distintas metodologías para realizar y representar diseño orientado a objetos y análisis orientado a objetos.

[V / F] Dentro de la ingeniería de software los modelos son relativamente simples de desarrollar.<sup>8</sup>

## Modelos

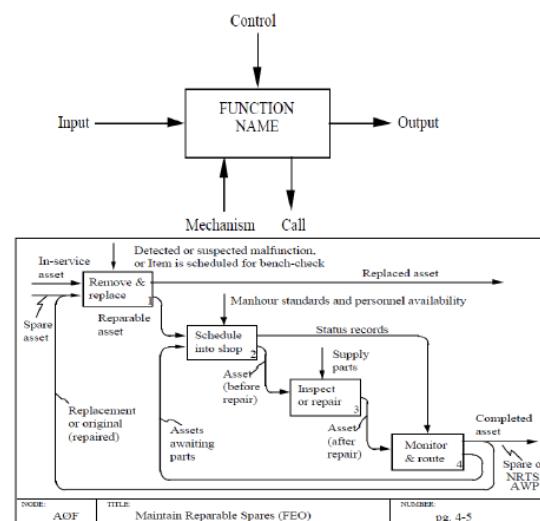
### La influencia del lenguaje de programación

El protagonista exclusivo a fines de los 90 era UML. UML tuvo un peso muy importante. Aparecieron también otras notaciones complementarias como BPMN, más recientemente aparece C4. Lo que es importante destacar es que estos lenguajes de modelado, estas notaciones, están muy influenciados por los paradigmas de programación. En principio podemos hablar de que existen como 2 grandes grupos de notaciones, algunas más orientadas a la descripción de la implementación y otras que tienen más que ver con el modelo conceptual del sistema. Los modelos y las notaciones más relacionadas con el diseño obviamente están mucho más cercanas al lenguaje de programación empleado. Los que tienen más que ver con el análisis de requisitos, con la ingeniería de requisitos, son modelos que describen lo que el sistema tiene que hacer, independientemente de cómo lo hagamos. Entonces se utilizan abstracciones del tipo actividades, a veces almacenamientos, a veces objetos; pero claro, tienen esos nombres, actividades, objetos, almacenamientos, pero no necesariamente son programas, aplicaciones, tablas u objetos del lenguaje orientado a objetos que se emplee. Entonces, lamentablemente, lo que termina pasando es que el lenguaje de programación influencia obviamente, en el lenguaje utilizado en el diseño, y esa anotación de diseño, termina influenciando lo que se utiliza para describir al sistema en términos tecnológicamente neutros, en términos abstractos. Vamos a verlo a continuación, con un ejemplo.

### Perspectiva histórica

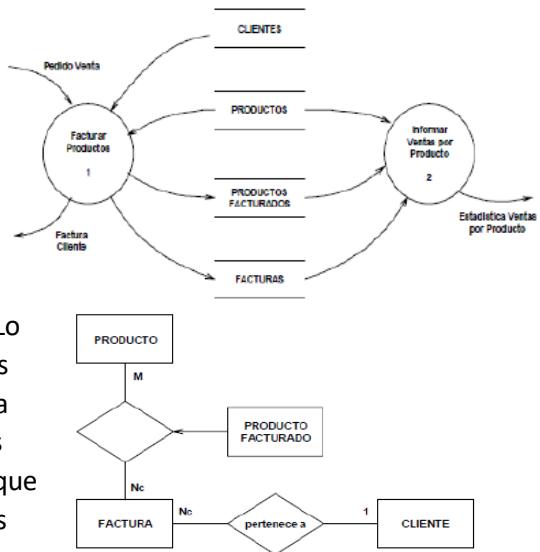
#### IDEFO

Uno de los primeros desarrollos que hubo en los 70, fue una técnica llamada SADT (Structured Analysis and Design Technique), que fue propietario durante mucho tiempo (la solicitó la fuerza aérea). Esta metodología, derivó en una notación llamada IDEF0 (modelado funcional) continuada por varios sucesores (IDEF1X: modelado de la información; IDEF3: modelado de procesos; IDEF4: modelado orientado a objetos; IDEF5: ontología). Lo que plantea es descomponer una función, una actividad en subfunciones. Esto supuestamente es una descripción abstracta de lo que tiene que hacer un sistema. Imaginémonos que en un primer nivel tenemos un único diagrama que representa todo el sistema, con los inputs con los outputs; si hacemos doble clic en esa primer cajita nos deberíamos encontrar con una cajita de menor nivel que representa las grandes actividades en las cuales se descompone esa función (subfunciones), y obviamente, el flujo de datos que hay entre cada una de esas situaciones. Eso se parece mucho a descomponer un programa de subrutinas. Y esta es una muestra clara de como la tecnología de implementación termina influenciando en la notación empleada para representar el análisis de requisitos, el análisis o la descripción del sistema en términos tecnológicamente neutros.



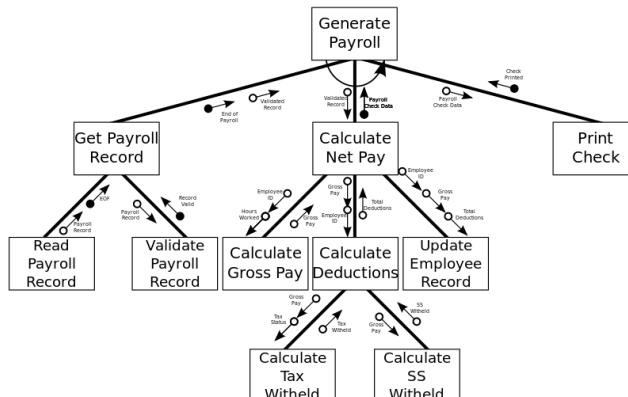
## Análisis estructurado (DFD, DER, DD)

Otro ejemplo similar, es el del análisis y el diseño estructurado en versión Yourdon, Larry Constantine, Tom DeMarco. En esta notación, que estuvo lejos de estar estandarizada, un sistema se descompone en actividades y en subactividades (igual que en el caso anterior). Las actividades están representadas por burbujas. En este diagrama vemos un diagrama en el cual tenemos 2 burbujas y varios almacenamientos. Esas burbujas se pueden descomponer a su vez en subburbujas, en subactividades, es decir, que podemos ir desde lo general a lo particular, es una estructura jerárquica. Lo diferente de este tipo de diagramas es que aparecen los almacenamientos. Los almacenamientos representan, de alguna manera, los objetos de ese diagrama de entidad, relación que vemos más abajo. En principio, son almacenamientos esenciales, es decir, que no son archivos ni son tablas, sino que representa lo que la terminología del análisis estructurado llamaba almacenamiento esencial, cosas que el sistema tiene que recordar. Esto obviamente da lugar a un posterior diseño de una base de datos o un sistema de archivo. Pero nuevamente acá el problema que tenemos es que estamos fuertemente influenciados por la tecnología de implementación, aunque estemos usando una abstracción (programas y archivos).



## Diseño estructurado (DE)

En la misma línea, está el diagrama de estructuras, el diseño estructurado, en ese momento también planteado por Larry Constantine y Ed Yourdon. Se descompone un programa en módulos. Entre los módulos se intercambian parámetros. Y esto también está muy asociado a los lenguajes de programación que se utilizaban en ese entonces, que eran muy relacionados con el paradigma estructurado.



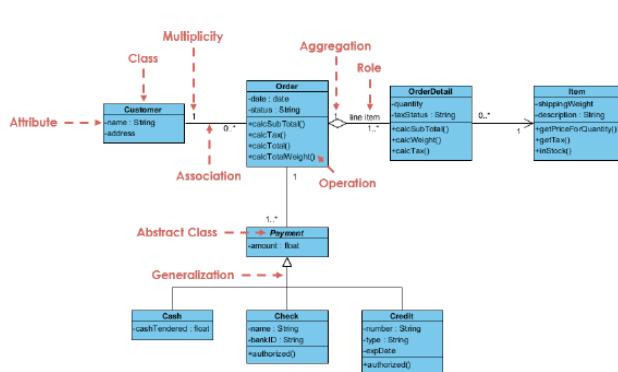
## Unified Modeling Language (UML)

Ya más cerca de los 90 aparece UML. UML es un lenguaje gráfico que se usa para especificar, visualizar, también para construir, documentar, sistemas basados en software. UML fue el resultado de fusionar las notaciones más populares que había en la década del 90, las de Jacobson, Rumbaugh y Booch. Es un estándar del Object Management Group (OMG) desde el año 97 (actualmente está en la versión 2.5.1). En principio es independiente de la metodología o del proceso de desarrollo y plantea una serie de diagramas.

## UML

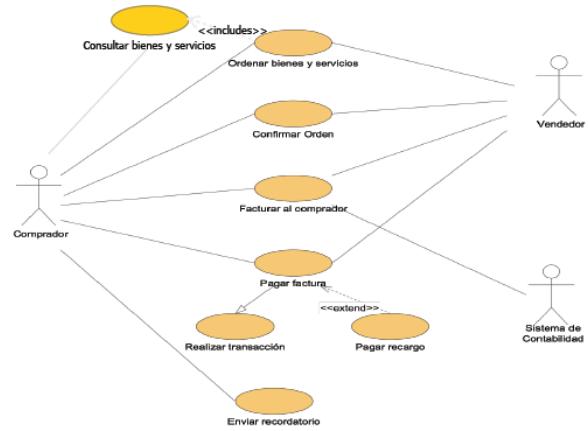
### Diagrama de clases

El diagrama de clases describe la estructura interna de un sistema en términos de clases, asociaciones; entre esas clases colaboraciones, interfaces. Es el diagrama que quizás se utiliza más, y muestra una estructura estática. Este diagrama puede utilizarse para modelar aspectos de diseño más cercano a la implementación y, con algunas particularidades, con algunas limitaciones, para modelar también aspectos conceptuales que no tienen que ver con la implementación (puntualmente vamos a utilizar una variante de este diagrama para desarrollar algo que llamamos modelo de dominio).



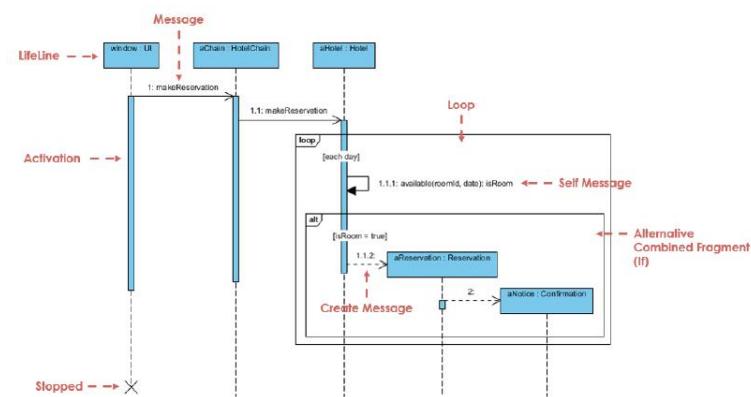
## Diagrama de casos de uso

Un modelo más cercano al mundo del análisis de requisitos es el casos de uso. Cada una de esas elipses que vemos ahí está representando un caso de uso. Un caso de uso es un paquete de funcionalidad que nos ofrece el sistema. El diagrama de caso de uso además incluye actores que son las personas que interactúan con el sistema. Es un diagrama estático, es un diagrama que no ofrece mucha información, por eso, más adelante vamos a ver que para cada uno de estos casos de uso vamos a necesitar algún tipo de especificación en formato texto para poder dar más información.



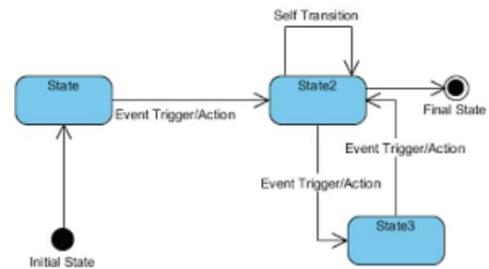
## Diagrama de secuencia

El diagrama de secuencia nos muestra la interacción entre una serie de clases para completar una acción determinada. Normalmente para un caso de uso, probablemente necesitemos implementar esa funcionalidad que está representada en ese caso de uso, mediante una serie de objetos o de clases que van a colaborar entre sí, probablemente entonces utilicemos este tipo de diagrama para representar esa colaboración. Lo que muestra el diagrama es el paso del tiempo de izquierda a derecha y el intercambio de mensajes entre las clases. Esos mensajes, por supuesto, invocan a los distintos métodos que ofrecen cada una de esas clases.



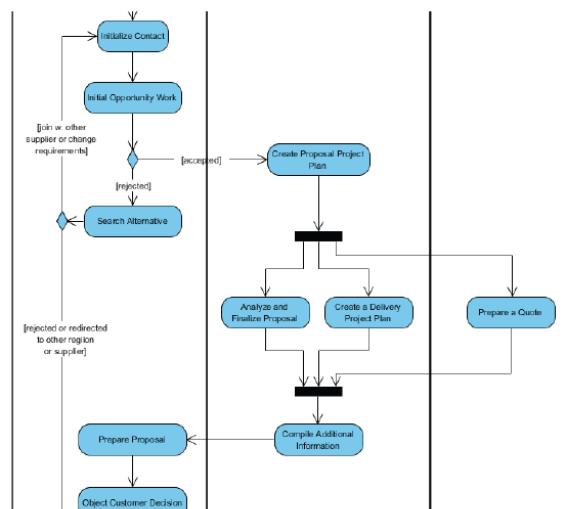
## Diagrama de estado

Otro diagrama muy utilizado es el diagrama de estado. Básicamente es una máquina de estados que permite ilustrar el ciclo de vida de una clase o de una colaboración. La notación es muy sencilla, hay un nodo de inicio, un nodo de fin, hay estados y hay transiciones entre los estados que son representados mediante rectángulos y mediante flechas respectivamente.



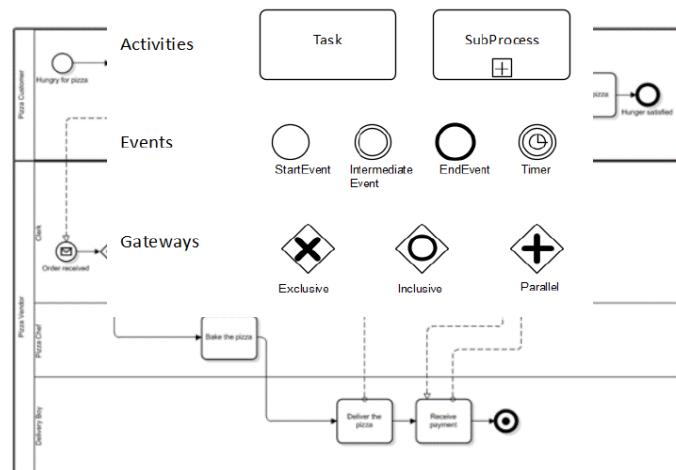
## Diagramas de actividad

UML también nos ofrece un diagrama de actividad. Esto se puede utilizar para representar el flujo de actividades de un ciclo de trabajo, para representar lo que pasa dentro de una función, dentro de un método, tiene múltiples aplicaciones. Nosotros podemos utilizar, por ejemplo, para representar un proceso de negocio que hace falta automatizar mediante algún tipo de software. Es muy sencillo, simplemente lo que hay son andariveles que representan distintos roles; hay actividades, hay transiciones entre las actividades, hay puntos de decisión. Es una especie de cursograma en una notación un poco más moderna. Esto es muy útil, obviamente para representar ciertos aspectos relacionados con el análisis de requisitos.

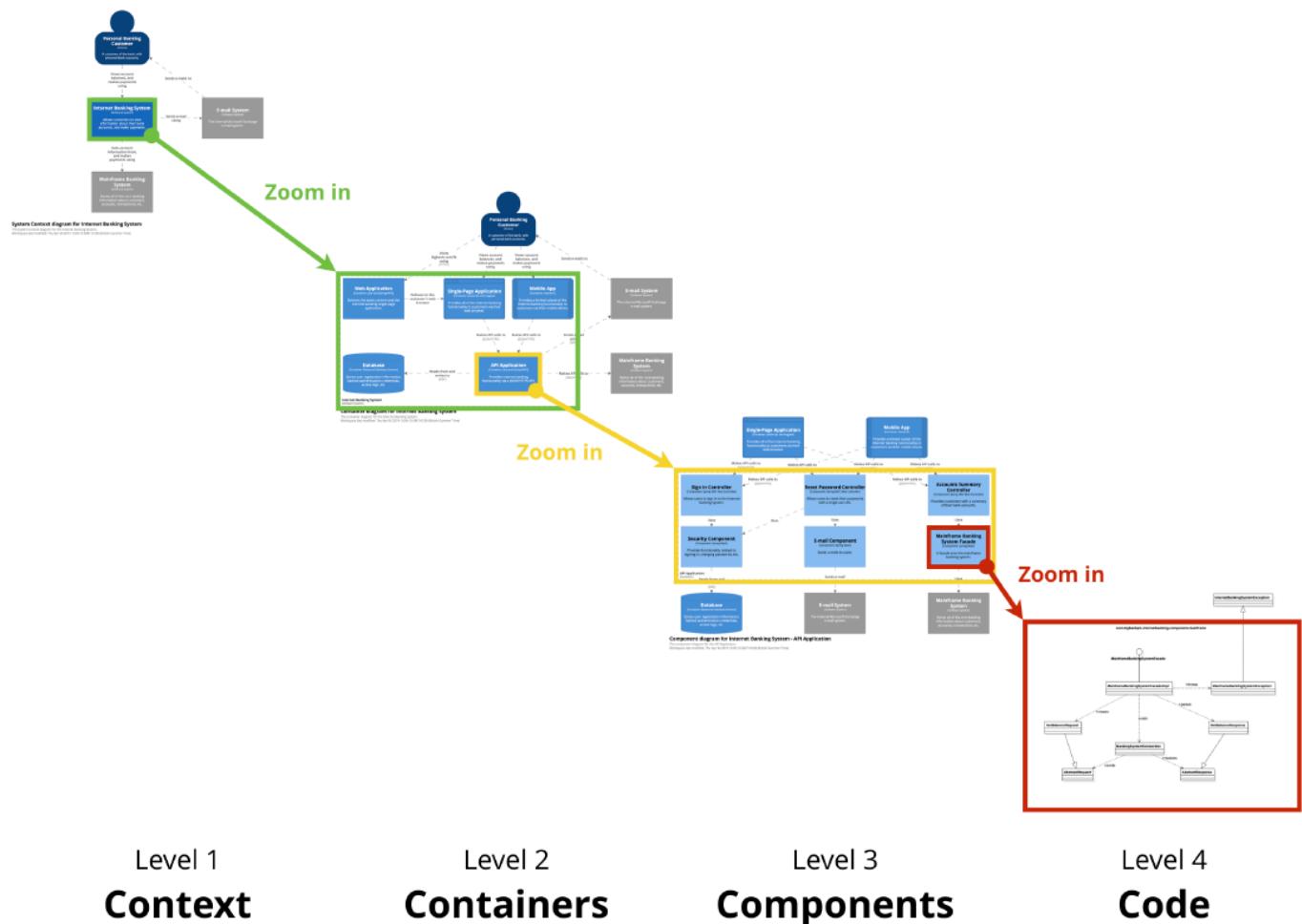


## Business Process Modeling and Notation (BPMN)

Algo que también se puede utilizar y que tiene similares características es BPMN. BPMN son las siglas de Business Process Modeling and Notation. Es un estándar de la OMG desde 2005 también. Lo que busca representar es un proceso de negocio, un proceso de manera similar al diagrama de actividades de UML que representábamos recién. La notación es sumamente sencilla, hay un nodo de inicio, hay un nodo de fin, hay andariveles, hay tareas; las tareas tienen transiciones entre sí. Podríamos representar acá, por ejemplo, el proceso de ventas, el proceso de incorporación de un nuevo cliente a una aplicación, las aplicaciones son infinitas. Esto está muy pensado no sólo para describir procesos, sino también para automatizarlos mediante herramientas. También vamos a ver que es muy útil para entender. De repente cuando uno va a una organización y necesita entender cuál es el proceso antes de automatizarlo, usualmente puede utilizar este tipo de diagramas. Lo mismo con el diagrama de actividades de UML. Muy sencillo, muy útil, hay que tenerlo en la caja de herramientas.



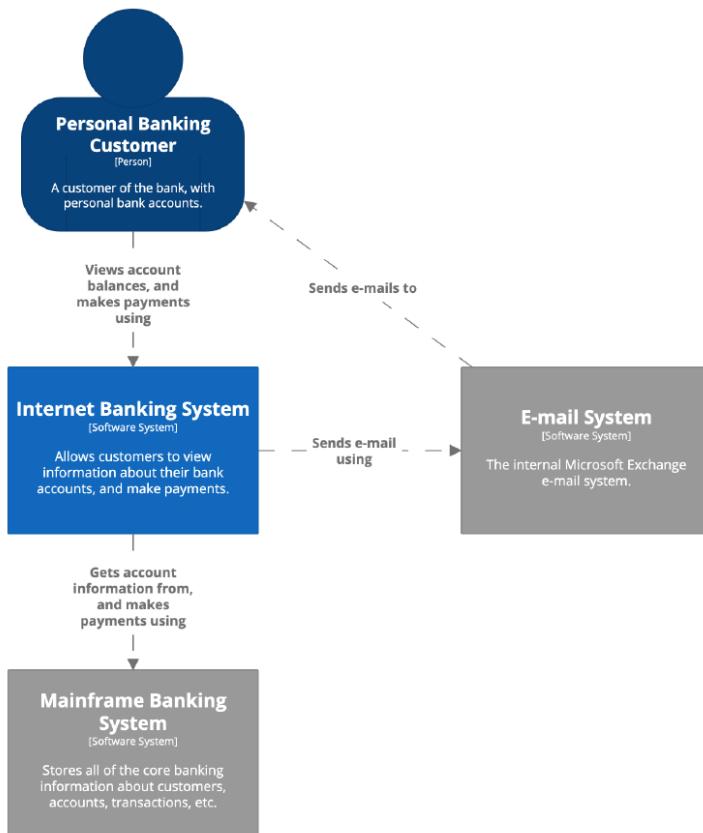
[V / F] BPMN está orientado a modelar procesos de negocio.



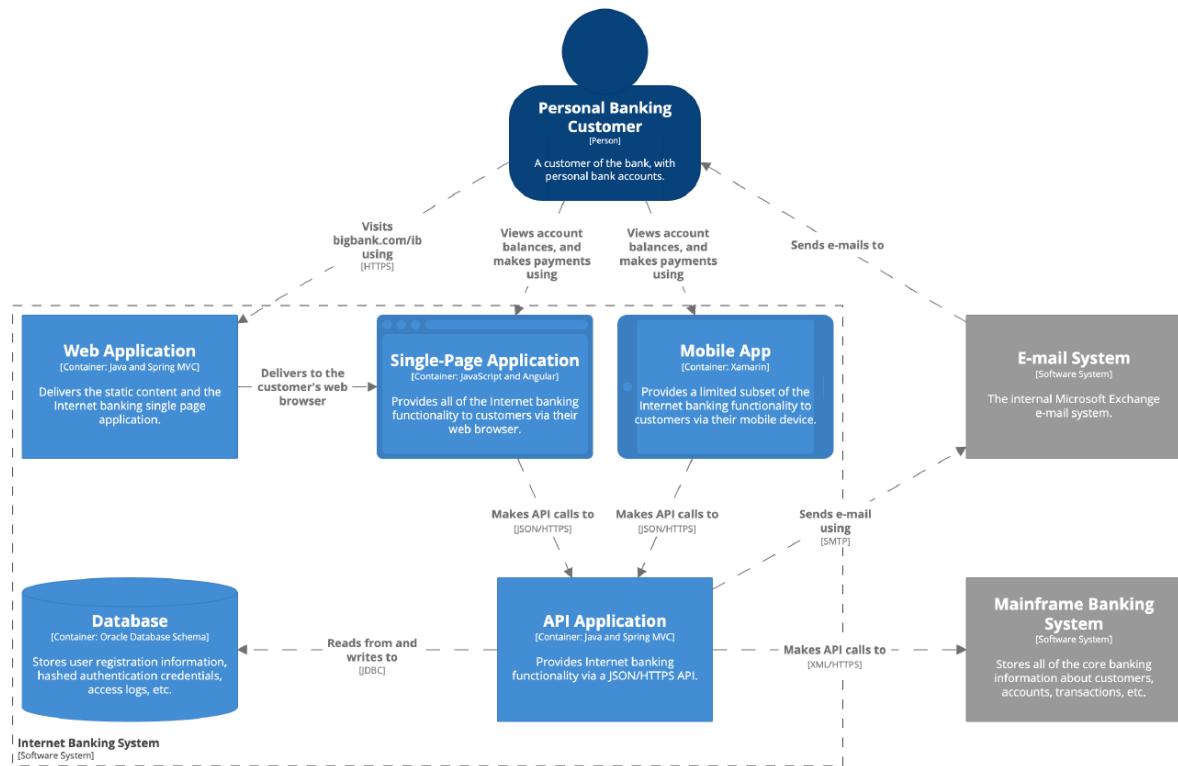
C4 tiene como propósito describir la arquitectura de software. ¿Qué es la arquitectura? La descripción de los grandes elementos que forman parte del software. Lo que propone C4 es una estructura de diagramas en niveles como muchas de las otras notaciones que hemos visto antes.

## Nivel 1: Diagrama de contexto

Estos cuatro niveles arrancan con un diagrama de contexto que nos muestra el medio ambiente en el cual va a estar funcionando el sistema que queremos construir. Es un diagrama que tranquilamente se puede utilizar en el mundo de la ingeniería de requisitos. Muy sencillo, nos está mostrando (con una caja azul en este caso) el **sistema** en cuestión, el **usuario** (con ese ícono en una azul un poco más oscuro), con cajas grises los **sistemas que soportan** el sistema que estamos modelando y las relaciones entre ellos. Fíjense que es un diagrama muy sencillo, que tranquilamente uno podría incluir en un documento de visión o en una presentación ejecutiva para describir el contexto en el cual funciona el sistema.



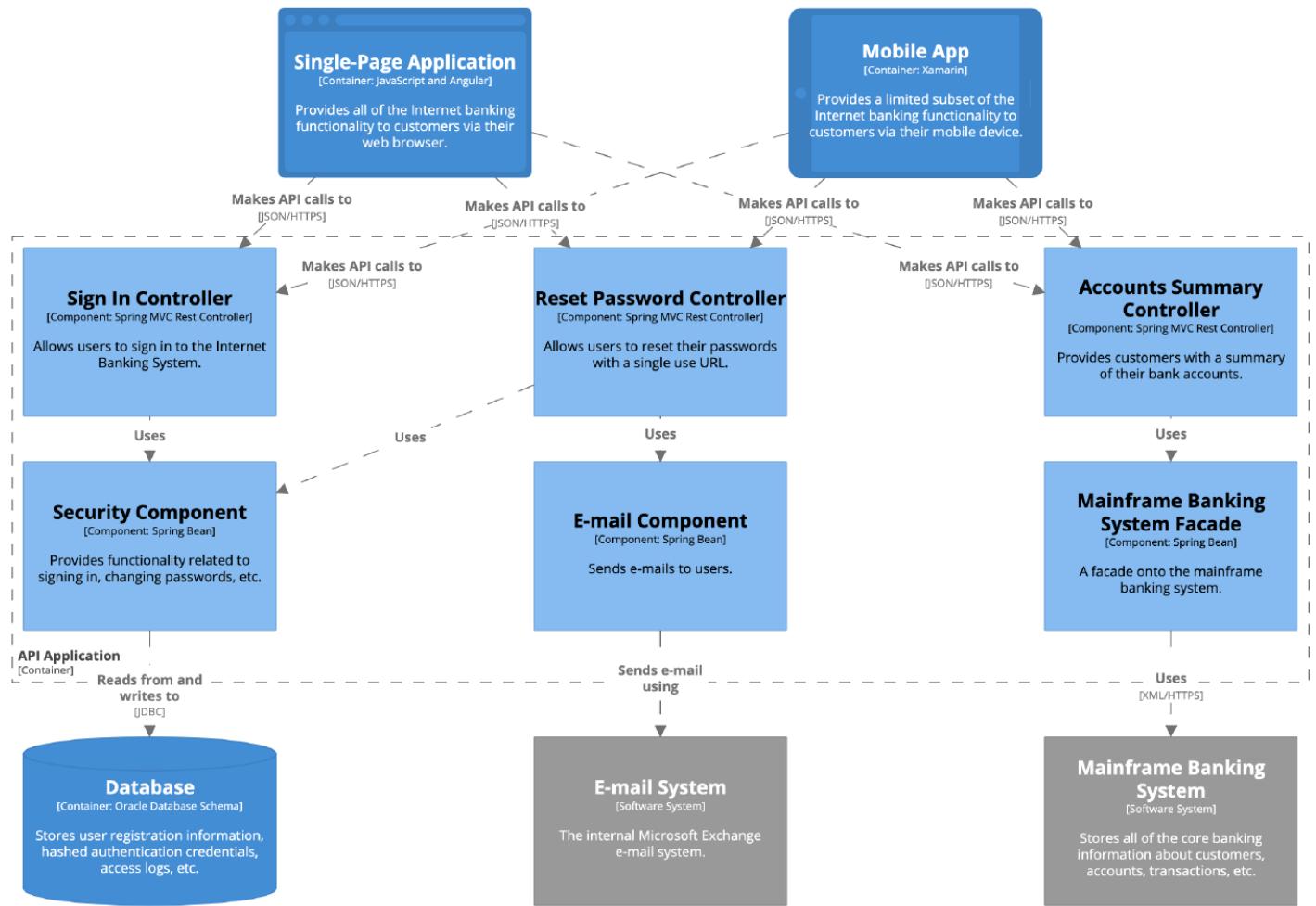
## Nivel 2: Diagrama de contenedores



El siguiente nivel, el nivel 2, es el diagrama de contenedores. Lo que estamos haciendo en definitiva acá es un zoom, un doble clic en el sistema, qué es lo que hay ahí adentro. Y nos encontramos con esa abstracción que C4 llama "**Containers**", que no tienen absolutamente nada que ver con los containers de los que estamos hablando hoy. Simplemente es un mecanismo que lamentablemente tiene el mismo nombre, pero es un mecanismo de abstracción que lo que busca representar son aplicaciones, almacenamiento de datos, microservicios, base de datos. En este caso en particular, nos encontramos con que el sistema en cuestión está compuesto por una aplicación web, una aplicación móvil, una aplicación que tiene que ver con la API, interactúa con el sistema de email, eso está fuera del

del sistema, pero ahí se muestra interacción. Hay una base de datos. Es un diagrama más orientado a la implementación.

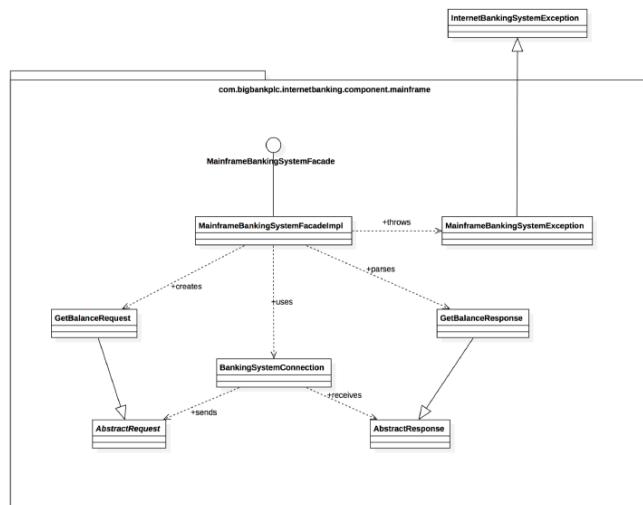
### Nivel 3: Diagrama de componentes



Si hace falta podemos hacer un doble clic en un container y utilizar un diagrama de componentes para describir qué es lo que está pasando ahí adentro, qué es lo que hay adentro de ese contenedor. Aquí ya estamos empezando a hablar de **componentes**. Esos componentes tienen que mapear contra alguna cosa real del entorno de implementación. En este caso hay un par de controladores, hay componentes de seguridad, hay una interfaz con un sistema en un equipo mainframe. Acá ya estamos más cerca de la implementación, más cerca del código.

### Nivel 4: Diagrama de código

Y si hace falta, podemos bajar un nivel más y tener un diagrama de código que no deja de ser más que un diagrama de clases o un diagrama de colaboración que lo que busca es representar el código que forma parte de ese componente.



[V / F] A medida que se avanza de niveles en C4 los diagramas se acercan menos a la implementación.<sup>9</sup>

## Resumen

Hemos visto distintas notaciones. UML probablemente sea la más conocida, la más importante; BPMN no tanto; y C4 probablemente sea la menos conocida de todas. UML tiene la ventaja de que se enseña en los cursos de programación, con lo cual, muchos de ustedes probablemente estén muy familiarizados, al menos con los diagramas de clase, los diagramas de secuencia. Hay que tener mucho cuidado cuando utilizamos UML fuera de lo que es diseño, cuando queremos utilizar UML más para los que son los modelos abstractos, conceptuales del análisis de requisitos, ahí hay que tomar en cuenta algunas consideraciones. BPMN es muy bueno para modelar procesos de negocio, con lo cual, si necesitamos entender un proceso antes de pensar en automatizarlo, es una muy buena herramienta. Y C4 nos ofrece una muy buena alternativa porque combina una notación muy sencilla con UML en el último nivel. C4 surge como alternativa a los diagramas de arquitectura que propone UML, que tienden a ser muy complejos. Entonces, con esos diagramas muy, muy elementales, muy sencillos, nos podemos comunicar y evitamos ese modelado informal que, como muchos autores llaman de “cajitas y flechitas” para representar arquitecturas, para representar las grandes abstracciones que describen a al sistema que estamos pretendiendo modelar.

Por supuesto que el tema no se agota acá, simplemente es un panorama general. Wiegers en su libro de ingeniería de requisitos nos propone una descripción muy permeabilizada de los tipos de modelos y diagramas que se pueden utilizar en el ámbito de la ingeniería de requisitos. Los libros de Booch o de Larman nos plantean los diagramas que se pueden utilizar más para los aspectos de diseño. Ahí uno tiene un gran abanico de alternativas de opciones para elegir.

## Escenarios de utilización

Lo que siempre hay que tener presente es que hay que tener mucho cuidado con el escenario que nos toque, y elegir correctamente las herramientas y las notaciones en función de esos escenarios. Si vamos a construir una casa para una mascota, probablemente no haga falta hacer un diagrama, probablemente el modelo lo tengamos en la cabeza y simplemente lo que hacemos es ir, conseguir un poco de madera, un par de clavos, algunas herramientas y trasladar ese diseño que tenemos en la mente a la cosa concreta. Si lo que vamos a construir es una casa, probablemente necesitemos modelos un poco más detallados, porque vamos a necesitar emplear materiales más caros, vamos a tener que hacer cálculos de estructuras, vamos a tener que compartir la información con distintos especialistas, el proceso de construcción tiene que estar bien definido para optimizar costos, de manera que en ingeniería de software nos vamos a encontrar con la misma situación. Quizás, hay casos en donde el modelado puede ser muy informal, alcanza con dibujar un diagrama de clases en un pizarrón, en otros casos probablemente, en vez de dejar ese diagrama en un pizarrón, quizás sea conveniente hacerlo en una herramienta. Lo que siempre tenemos que sopesar es el costo de construir el modelo y mantenerlo versus la utilidad que nos provee ese diagrama. Siempre vamos a tener que balancear entre esos 2 aspectos. En algunos segmentos de la industria del desarrollo de software, hay algo llamado Modeling Driven Design. En ese entorno, los modelos son muy, muy formales y se pueden ir construyendo y transformando en modelos de menor nivel hasta llegar al código. Las herramientas Case de los años 90 prometían eso y, de hecho muchas lo consiguieron. Hay algunos artículos muy interesantes al respecto, que podemos compartir con ustedes, en donde en segmentos muy particulares de la industria de la ingeniería de software ese modelado, esa estrategia de modelado, tiene mucho éxito. En otro tipo de aplicaciones, el modelado es muy muy informal, de hecho, hay toda una escuela de Agile Modeling que provee unas perspectivas sumamente interesantes de cómo enfocar el ejercicio de modelado.

En definitiva y, como siempre, qué herramientas utilizar, qué notación utilizar, qué modelos construir y qué modelos no, va a depender de nuestro criterio profesional.

<sup>9</sup> A medida que se avanza de niveles en C4 los diagramas se acercan MÁS a la implementación

El diagrama de clases es un diagrama...

- estático**
- dinámico
- de contexto

# 1.1 - INCOSE

An approach which “crosses many disciplinary boundaries to create a **holistic approach**”



## Cross-disciplinary

Focuses on working across multiple disciplines while allowing each discipline to apply their own methods and approaches

**Systems Engineering** is a **transdisciplinary** and **integrative** approach to enable the successful realization, use, and retirement of **engineered systems** using **systems principles and concepts**, and scientific, technological, and management methods.

The ways that systems thinking and the systems sciences infuse systems engineering

- Mental models
- System archetypes
- Holistic thinking
- Separation of concerns
- Abstraction
- Modularity & encapsulation
- Causal loop diagrams
- Systems mapping

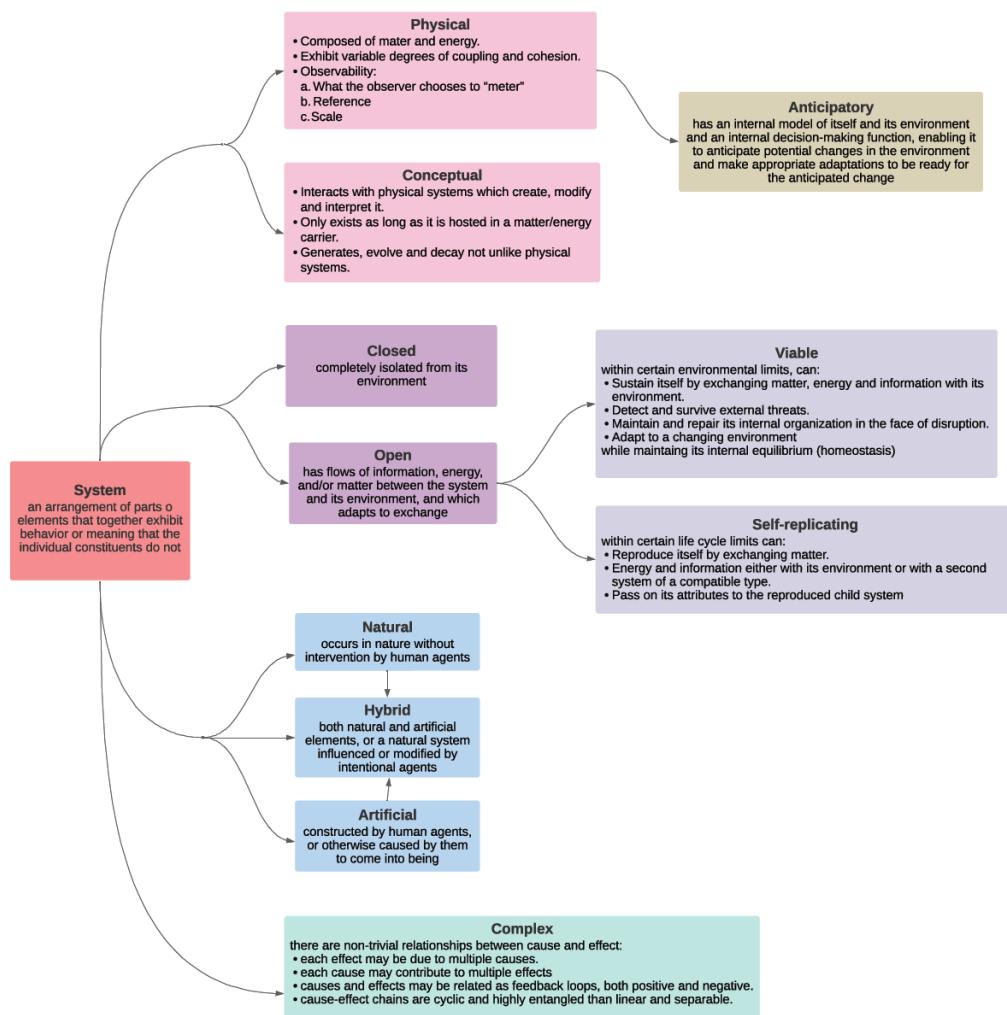
- A **system** designed or adapted to interact with an anticipated operational environment to achieve one or more intended purposes while complying with applicable constraints
- A composite of people, products, services, information, and processes (and possible natural components) that provides a capability that satisfies a stated customer need or objective
- Includes the sub-categories of products, **services** and **enterprises**

A large undertaking, especially one of large scope, complication and risk

Actions taken to satisfy needs of individuals or organizations

Usually depend on technological products

An arrangement of parts o elements that together exhibit behavior or meaning that the individual constituents do not



[V / F] El software es un sistema del tipo conceptual que no necesita de un sistema físico para su transporte o almacenamiento.

Ejemplos válidos de sistemas diseñados (engineered systems) son los siguientes:

- El sistema circulatorio
- El servicio de atención al cliente de un hotel**
- Una aplicación móvil**
- La atención a estudiantes en una universidad**
- Un banco**

Un sistema es...

- un conjunto de elementos que exhiben comportamiento
- un conjunto de elementos que exhiben significado
- un conjunto de elementos
- un grupo de partes o elementos que en conjunto exhiben un comportamiento o significado que los elementos individuales no tienen**

## 1.2 - Videos 1, 1.1 y 1.2 y Bourque

Pruebas y revisiones de software son actividades relacionadas con los

- costos de prevención de calidad
- costos del desarrollo de software
- costos de falla externos
- costos de falla internos
- costos de evaluación de la calidad**
- costos de aseguramiento de la calidad

Ingeniería es

- la creación de soluciones
- la creación de soluciones costo efectivas a problemas prácticos mediante la aplicación de conocimiento codificado para construir cosas al servicio de la humanidad**
- la aplicación de un enfoque sistemático al desarrollo de soluciones a problemas complejos

Indicar a qué área de conocimiento del SWEBOK [Bourque] es MÁS PROBABLE que pertenezcan las actividades que realiza una persona que desarrolla software en la compañía PSA

	Requisitos	Diseño	Construcción	Prueba	G. Configuración	G. Ingeniería de software
Llega 8:45 a la oficina. Se sirve un café y se sienta al escritorio a revisar su correo y a repasar el estado de las tareas a su cargo.						✓
A las 9:15 se reúne con su equipo de proyecto para la Daily Stand Up Meeting. En ella, cada uno de los participantes expone brevemente qué tareas realizó el día anterior, cuáles tiene planificadas para el día y si hay algún obstáculo o problema que deba ser abordado.						✓

A las 10:00 mantiene, junto a su equipo de proyecto, una reunión de trabajo con el usuario que cubre el rol de product owner para clarificar ciertos aspectos de la funcionalidad de la aplicación que están desarrollando.	✓					
Luego de la reunión retorna a su escritorio y se dispone a modificar unas componentes que desarrolló el día anterior. Trabaja con ellas por espacio de una hora, modificando varias líneas de código y haciendo algunas pruebas manuales.			✓			
Antes de salir a almorzar, ejecuta el script para compilar las componentes y desplegarlas en el ambiente de pruebas.					✓	

Un proceso es

- un enfoque sistemático para la especificación, diseño, construcción, prueba y verificación de software
- un mecanismo para entender, modelar y comunicar
- un conjunto de actividades que transforma una entrada en una salida y que consume recursos**

La ingeniería de software es

- una ingeniería como las restantes, sin nada en particular que la diferencia del resto
- la creación de soluciones basadas en software
- la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software**

## 1.3 - No silver bullet

[V / F] La parte más compleja en la construcción de software es llevar adelante la ejecución de lo que se decidió construir

Un banco necesita manejar altos estándares de seguridad en todas sus transacciones. ¿Qué le recomendaría Brooks?

- Desarrollar su propio sistema encriptado, para tener todo bajo control
- Contratar más gente para que el conocimiento sea distribuido
- Comprar a terceros el sistema de encriptado**

[V / F] Una metodología que se concentre durante una década en atacar las problemáticas esenciales del software permitiría mejorar en un orden de magnitud la productividad en el desarrollo de software.

Las características esenciales del software (complejidad, maleabilidad, conformidad, invisibilidad) se pueden resolver plenamente mediante:

- Ambientes de desarrollo más poderosos, con ayuda de herramientas de inteligencia
- Programación basada en modelos
- Ninguna de las opciones**
- Lenguajes de programación más poderosos
- MBSE (Model Based Software Engineering)
- Todas las opciones
- Mayor capacitación del personal involucrado en el desarrollo

[V / F] No existen ni existieron balas de plata en el desarrollo de *hardware* que permitan obtener mejoras en un orden de magnitud en una década en lo que refiere a la productividad, simplicidad y confiabilidad.

[V / F] El desarrollo de lenguajes de alto nivel fue uno de los avances más importantes en lo que refiere al trabajo sobre las dificultades accidentales del software.

## 1.4 - Boehm y Royce

Según [Boehm], las funciones primarias de un modelo de proceso de desarrollo de software son:

- Describir cuáles son los criterios de transición de una etapa a otra**
- Describir los entregables
- Describir cuáles son los criterios de transición de una actividad a otra
- Determinar el orden de las etapas**
- Determinar el orden de las actividades

Según [Royce], el mayor problema del ciclo de vida en cascada es su elevado:

- Tiempo
- Costo
- Riesgo**

Indicar cuál de las siguientes afirmaciones acerca del modelo del espiral propuesto en [Boehm] es correcto:

- La dimensión radial indica cuánto se ha progresado en una fase
- La incertidumbre es mayor en el inicio de los proyectos. Por esta razón, en los ciclos iniciales se debe hacer un análisis de riesgo que deja de ser necesario en los ciclos finales
- Si es falsa la hipótesis del espiral en un determinado momento, se debe dar por concluido el ciclo actual y comenzar uno nuevo
- El costo acumulado está representado por la dimensión vertical
- En cada comienzo del ciclo se debe identificar un objetivo que cada ciclo que pasa debe ser más específico**
- Ninguna de las anteriores es correcta

Según [Boehm], los principales problemas del modelo Code & Fix son:

- Dificultades para identificar casos de prueba
- Mala estructura del código**
- Dificultades para estimar el esfuerzo necesario
- Alto costo de arreglos debido a la poca preparación para pruebas y modificaciones**
- Falta de alineamiento con las necesidades del usuario**

¿Qué propone [Royce] para mitigar los riesgos de desarrollar software siguiendo el ciclo de cascada?

- Estimar cada tarea antes de llevarla adelante
- Probar cada secuencia de pasos lógica del sistema construido**
- Agregar una etapa de diseño preliminar antes de llevar adelante el análisis**
- Construir un prototipo antes de comenzar el análisis del sistema
- Documentar claramente el equipo de trabajo que construirá la solución
- Elegir una metodología ágil que se enfoque en maximizar el valor otorgado al cliente

## 1.5 - Fowler

De acuerdo a [Fowler], indicar cuál o cuáles de las siguientes afirmaciones son verdaderas en relación a la predictibilidad del software

- Existen proyectos que por su naturaleza nos permiten utilizar una metodología predictiva**
- La importancia de utilizar iteraciones es mayor en metodologías adaptativas que en predictivas**
- Los cambios en los requisitos de un programa son consecuencia de un mal trabajo en la ingeniería de requisitos
- Incorporar cambios tardíos en los requisitos es un problema para los clientes

Según [Fowler] ¿cuál de las siguientes actividades se ajusta mejor a la tarea de programar el software?

- Diseño**
- Construcción
- Verificación
- Validación

## 2 - Introducción a la ingeniería de requisitos

### Introducción

Una de las dificultades más grandes que enfrentamos en el desarrollo de software es decidir qué es lo que se va a desarrollar. El proceso de entender cuáles son las necesidades del usuario, cuáles son sus expectativas y definir qué es lo que debe hacer el producto para satisfacerlas, está plagado de infinidad de problemas: Falla en la comunicación, diferencias culturales, poder, política. Alan David señala que entre el 40 y el 50% de los defectos encontrados en el software tienen su origen durante estas actividades. Para resolver estos problemas dentro de la ingeniería de software contamos con lo que se conoce como ingeniería de requisitos.

[V / F] Dentro de lo más difícil en el desarrollo del software es decidir el qué se debe desarrollar.

En una entrevista con un usuario podríamos escuchar algo como esto: “Nuestros clientes deben poder retirar efectivo de cualquiera de los cajeros de nuestra red, hasta un monto de \$5.000 diarios”. A partir de esta simple declaración que aparentemente es clarísima, surgen un montón de interrogantes, por ejemplo: ¿Quiénes son los clientes? ¿Qué información tenemos de ellos? ¿Qué implica que haya una red propia? ¿El monto es fijo, podrá variar en el futuro? ¿De qué manera medimos los días? ¿Contamos 24 horas?

Para ayudarnos a entender qué es lo que los usuarios necesitan y transformar esto que acabamos de ver en una definición del comportamiento del sistema a construir, contamos justamente con la ingeniería de requisitos.

¿Qué es formalmente un **requisito**? Lo que un producto o sistema debe hacer para satisfacer las necesidades y expectativas de los usuarios, independientemente de cómo lo implementemos. Hay otras definiciones, por supuesto, cada uno de los autores que ha escrito acerca del tema ha dado la propia. Quizás la más completa, sean la que dan Sommerville y Sawyer:

*Los requisitos son una especificación de lo que se debe implementar. Constituyen una descripción de cómo el sistema se deberá comportar o de una propiedad o atributo que deberá poseer.*

### Requerimientos y requisitos

#### Definiciones

Es importante aclarar que la palabra en inglés “requirement” se traduce muchas veces al español como “requerimiento”. Pero lo correcto, sería utilizar la palabra “requisito”. En castellano “requerir” implica “intimar con la autoridad pública o solicitar algo de alguien”. “Requisito” parece ser la palabra más adecuada para traducir requerimiento, al menos en la mayoría de los casos, ya que “requisito” es “una condición necesaria para algo”. De todas maneras, no hay que perder de vista que en este contexto requisito tiene un significado muy especial que es el que acabamos de explicar anteriormente y que no necesariamente sigue al pie de la letra a la definición del diccionario.

### Requisitos

#### Distintos tipos de requisitos

Hay 2 grandes tipos de requisitos:

- Los **funcionales** que describen lo que necesitamos que el sistema haga, independientemente de cómo lo implementemos. En el ejemplo, necesitamos que el sistema nos permita registrar los pedidos de los clientes informando costos e impuestos aplicables, que nos permita mantener información de los clientes, razón social, número de CUIT, dirección; que nos permita mantener actualizado un catálogo de productos. Este sistema lo va a hacer independientemente de cómo lo implementemos, por eso algunos llaman a estos requisitos funcionales también, **requisitos esenciales o requisitos de comportamiento**.

- La otra gran categoría son los requisitos **no funcionales**, llamados también **calidad de servicio**, que describen **determinadas propiedades que el sistema deberá tener o restricciones con las que deberá cumplir**. En el ejemplo, hablamos de disponibilidad, de tiempo de respuesta, de sistemas operativos sobre los que deberá poder operar.

Lo que se puede ver de los ejemplos es que, independientemente del lenguaje que empleemos o del método o la notación que utilicemos para la especificación, los requisitos:

- Definen un **objeto**, una **función** o un **estado**.
- **Limitan** o controlan las **acciones** asociadas con un objeto, una función o un estado.
- Definen **relaciones** entre esos objetos, funciones y estados.

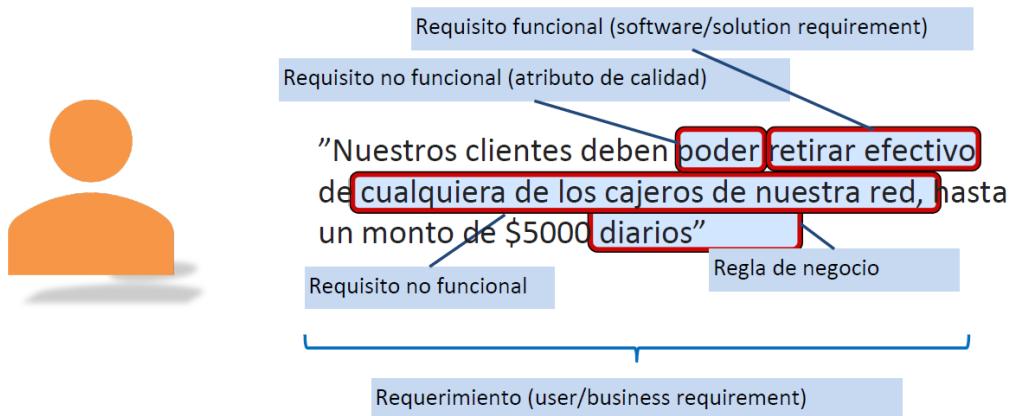
Es importante aclarar que aquí estamos hablando de objetos, que no tienen absolutamente nada que ver con los objetos de la orientación objetos, sino que son abstracciones de una entidad del mundo real que es importante para la discusión de los requisitos.

## Distintas perspectivas

También podemos clasificar a los requisitos según distintos puntos de vista:

- Los objetivos y las necesidades del negocio o de la organización que dan origen a los requisitos de los usuarios, son llamados **requerimientos de negocio** o **requisitos de negocio**.
- A partir de esos requisitos podemos derivar los **requisitos del usuario** que describen lo que necesita cada uno de los usuarios con respecto al sistema a construir.
- Y finalmente, hay una perspectiva que es la que les va a interesar a los constructores del sistema: Los **requisitos de la solución** o **requisitos del software**.

[V / F] La ingeniería de requisitos ayuda a traducir las necesidades y expectativas de los interesados y transformarlas en requisitos del sistema o solución a construir.



Volviendo a nuestro ejemplo inicial, a partir de esta necesidad manifestada por nuestro usuario, se pueden衍生 varios requisitos: Por un lado, la declaración podría ser clasificada como un requerimiento de negocio o como un requisito de usuario. Si examinamos más detalladamente nos vamos a encontrar con un requisito funcional que es “poder retirar dinero de la red”. Esto describe lo que el software tiene que hacer, lo que el sistema tiene que hacer independientemente de cómo lo implementemos. También aparece un requisito no funcional: ¿qué significa “poder” en este contexto? ¿Las 24 horas del día? ¿O hay ventanas de mantenimiento que nos limitan la posibilidad de retirar efectivo? Hay otro requisito funcional que es, hay una “red de cajeros”, ¿a dónde está esa red de cajeros? ¿Dónde se encuentran distribuidos esos cajeros? ¿Están siempre disponibles: están adentro de una sucursal bancaria, están en la vía pública? Ahí normalmente tenemos que indagar, tenemos que volver a entrevistar,

probablemente a esta gente para poder profundizar estos detalles. Y también aparece una regla de negocio que es la posibilidad de retirar esos \$ 5.000 diarios. ¿Esta regla de negocio podría llegar a cambiar? ¿Tiene algo que ver con la categoría del cliente? Es muy interesante esto porque a partir de una declaración relativamente sencilla aparecen y se derivan un montón de interrogantes. Adicionalmente, hay un problema grave en todo esto: el usuario está asumiendo que la única manera de obtener dinero en efectivo será mediante un cajero automático. Entonces quizás debamos plantear a los interesados otras alternativas, como por ejemplo retirar dinero en efectivo de la caja de supermercado. Lamentablemente, al enunciar una necesidad, muchas veces los usuarios, los stakeholders o interesados, están influenciados por soluciones preconcebidas. Nuestra tarea es llegar al fondo de la cuestión e identificar el verdadero problema que se debe resolver independientemente de estas ideas iniciales o preconcebidas que puedan llegar a tener a nuestros interlocutores.

Los requisitos son una pieza fundamental en el desarrollo de software porque constituyen la base que vamos a emplear para, entre otras cosas, diseñar y construir la solución, definir los casos de prueba y planificar y organizar las actividades de desarrollo de software.

## Ingeniería de requisitos

### ¿Qué tipos de actividades incluye?

La ingeniería de requisitos incluye varias actividades:

- ◆ Por un lado, las relacionadas con el **desarrollo de requisitos** que van a incluir:
  - Aquellas que tienen que ver con el [descubrimiento](#), es decir, determinar cuáles son los requisitos.
  - Las relacionadas con el [análisis](#), qué tienen que ver un poco con lo que hemos hecho recién, detectar y resolver conflictos entre los requisitos, derivar nuevos requisitos, identificar la interacción con el medio ambiente.
  - La [especificación](#) que consiste en documentar los requisitos para facilitar su revisión, su evaluación, su aprobación, para facilitar compartir esa definición entre los distintos interesados.
  - La [verificación](#) que consiste en evaluar si escribimos correctamente los requisitos.
  - La [validación](#) que consiste en evaluar si entendemos correctamente los requisitos que nos han formulado.
- ◆ La otra gran actividad, es la **administración de los requisitos**. Inevitablemente, los requisitos van a cambiar a lo largo del desarrollo del producto. No está necesariamente mal eso, pero tenemos que administrarlo, tenemos que estar seguros de estar trabajando sobre la versión actualizada de los requisitos.

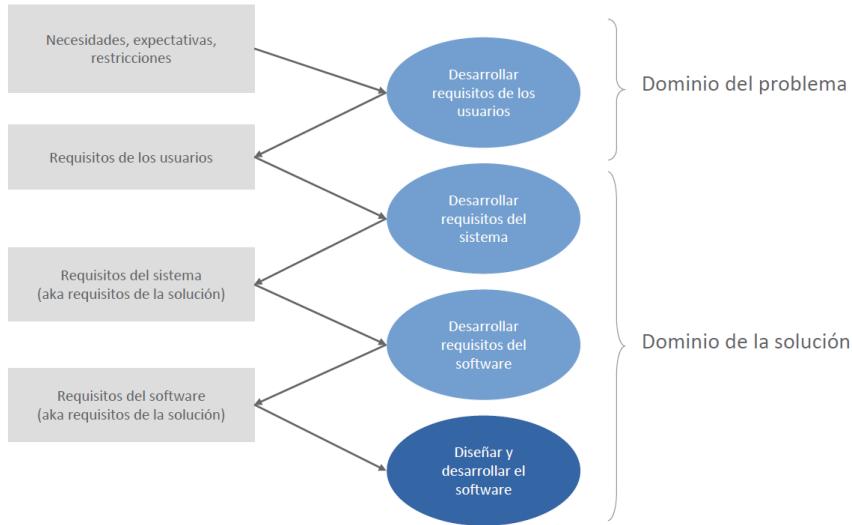
Los requisitos se emplean para...

- Definir el equipo de trabajo
- Organizar el trabajo**
- Diseñar las interfaces
- Identificar casos de prueba**

Podemos ver a la ingeniería de requisitos como una actividad que a través de sucesivas transformaciones refina las necesidades y expectativas de los usuarios e interesados en una definición de los requisitos que deberá satisfacer el software a desarrollar y que van a constituir la base para el diseño y el desarrollo del producto. Una parte del trabajo tiene que ver mucho con el dominio del problema y otra buena parte del trabajo tiene que ver con el dominio de la solución.

### ¿Qué implica?

Estas actividades que acabamos de anunciar no son disjuntas. No suceden, tampoco en forma secuencial, sino que se superponen con variados grados de intensidad a lo largo de todo el ciclo de vida, se realimentan entre sí. Es claramente un proceso iterativo e incremental.



### Los requisitos y el ciclo de vida del proyecto

Normalmente en los proyectos de desarrollo de software, el entendimiento de los requisitos es general al comienzo y se va refinando conforme se avanza en el entendimiento del problema y en el desarrollo del producto. Es usual, como decíamos anteriormente, que haya cambios en los requisitos por las más diversas causas. A veces cambia el contexto del negocio, se entienden mejor las necesidades, aparecen regulaciones, todo eso de alguna manera se tiene que reflejar en el proceso de ingeniería de requisitos.

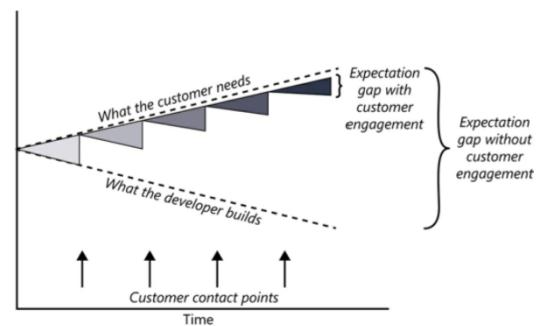
El proceso de ingeniería de requisitos es muy sensible a aspectos humanos, a aspectos relacionados con la comunicación, con el aprendizaje, la política, el poder.

### Complicaciones

- 👉 *Falta de conocimiento nuestro acerca del dominio del problema que estamos atacando.*
- 👉 *Dificultades para consensuar qué es lo que se quiere*, ya que hay distintos actores, distintos interesados, distintos usuarios con objetivos, muchas veces contrapuestos con respecto a la solución que hay que desarrollar.
- 👉 *Es difícil involucrar activamente a los interesados.*
- 👉 *Omitimos algún interesado clave*, simplemente por desconocimiento.
- 👉 Es complicado consensuar las agendas, lo que lleva a tener *dificultades en la planificación* de las actividades de ingeniería de requisitos. Recordemos que hay que entrevistar a usuarios potenciales del producto a desarrollar para entender qué es lo que necesitan, entender cuál es el problema que necesitamos resolver.
- 👉 *Los requisitos* son naturalmente *ambiguos*, hay muchos problemas en cuanto al vocabulario del dominio del problema.
- 👉 *Los requisitos* presentan un fenómeno *inflacionario*, tienden a crecer, tienden a multiplicarse, y de alguna manera hay que acotar ese fenómeno para poder entregar el producto deseado.
- 👉 Hay *dificultades de comunicación*. Algunos de los problemas de comunicación tienen que ver con el diferente vocabulario, el diferente lenguaje. La otra es no saber bien qué preguntar. Justamente por ignorancia acerca del área de aplicación.
- 👉 Por supuesto que hay *temas culturales* y ni que hablar de los *temas políticos*. Detrás de cada usuario, de cada interesado que tiene un objetivo contrapuesto con respecto al software a desarrollar, hay muchas veces intereses políticos, intereses que tienen que ver con espacios de poder, intereses que tienen que ver con manejo presupuestario, a veces con el manejo de la agenda estratégica.
- 👉 *La incorrecta gestión de los cambios*. **Los cambios en los requisitos son inevitables**. En países como Argentina muchas veces hay cambios regulatorios de la noche a la mañana y eso tiene que estar reflejado en la definición de los requisitos que va a tener que implementar el software.

## Expectativas e involucramiento

Es muy importante entonces, manejar adecuadamente las expectativas y el involucramiento de todos los interesados. Cuanto más contacto tengamos con los clientes, con los usuarios y con otros interesados, tanto mejor. Si mantenemos distancia, si no nos reunimos frecuentemente vamos a empezar a tener una brecha importante entre lo que nosotros percibimos que el cliente necesita y lo que terminamos construyendo. Por eso es importante tener un proceso de desarrollo de software iterativo e incremental que haga énfasis en este feedback lo más frecuente posible.



## Ventajas

La ingeniería de requisitos correctamente realizada tiene múltiples ventajas:

- 👉 Tenemos *menos defectos en el producto entregado*. Sabemos que detectar efectos en los requisitos nos produce una economía de escala muy importante.
- 👉 Nos permite *reducir el retrabajo*.
- 👉 Nos permite *desarrollar y entregar software de manera más rápida*.
- 👉 Nos permite *acotar la funcionalidad* que desarrollamos si es que *deja de ser necesaria*.
- 👉 Genera *menos costo de mantenimiento/evolución*.
- 👉 Hay *menos malentendidos*. Esto que mencionábamos antes de los contactos frecuentes, ayudan a que vayamos entendiendo progresivamente el vocabulario del problema y vayamos ofreciendo y validando soluciones posibles.
- 👉 Se acota el tema de los cambios indiscriminados a los requisitos.
- 👉 Hay *menos caos* en los proyectos.
- 👉 Los *clientes* están más *contentos*. Los miembros del *equipo* están más *satisfechos*.
- 👉 Nos ayuda a tener productos que hacen lo que se supone que deben hacer.

## ¿Quién hace ingeniería de requisitos?

Un rol, muchos roles, diferentes nombres

A veces es un rol de dedicación exclusiva, a veces es un sombrero diferente que usa alguien que tiene otro rol. En algunos casos, en algunas organizaciones, nos vamos a encontrar con que hay roles con el título de analista funcional, analista de negocios, analista de requisitos, más recientemente, jefe de producto o dueño de producto. Lo importante es que este rol es un traductor que debe entender lo que el usuario final está diciendo para transformarlo en una especificación de lo que el sistema, la solución o el producto a construir va a tener que hacer. Tienen que ser un buen observador, tiene que entender e interpretar lo que el usuario hace o quiere hacer. Tiene que tener una vocación para **idear mejores maneras** de hacer lo que el usuario está haciendo. Debe **registrar**, de alguna manera y con cierto nivel de detalle, los resultados obtenidos mediante una especificación de requisitos o algún tipo de modelo.

## Habilidades necesarias

Más allá de las habilidades técnicas, tiene que tener muchas **habilidades blandas**:

- ✍️ Saber escuchar
- ✍️ Saber entrevistar
- ✍️ Saber preguntar activamente
- ✍️ Ser analítico en su forma de pensar

<sup>10</sup> Inevitablemente, los requisitos van a cambiar a lo largo del desarrollo del producto

- Pensar sistémicamente: tiene que poder ver el bosque y si hace falta también ver el detalle
- Capacidad y voluntad de aprender
- Tener muy buenas habilidades para facilitar reuniones y talleres: ustedes saben que las reuniones suelen ser una de las actividades que más tiempo consume en las organizaciones y que, en general, traen muchos dolores de cabeza si no se manejan adecuadamente porque se quema el tiempo y se queman los recursos en reuniones mal organizadas y mal realizadas.
- Poder liderar estas actividades
- Capacidad de observación
- Muy buena habilidad para comunicarse y para organizarse (FUNDAMENTAL).
- Saber crear y preparar modelos.

[V / F] Detectar tempranamente defectos en los requisitos reduce el trabajo.

## ¿Cuán formal debe ser?

El proceso de ingeniería de requisitos puede ser muy formal, muy informal, o algo en el medio. Muy informalmente nos vamos a encontrar con organizaciones en las que por ahí no hay ningún proceso, no hay estándares, no se documenta. En la otro apunta, nos vamos a encontrar con procesos muy burocráticos, muy pesados, con estándares críticos, con requisitos de documentación que hay que generar, con revisiones rigurosas. En definitiva, el punto medio es el mejor en donde vamos a encontrar el punto de equilibrio que obviamente va a depender muchísimo del tipo de software que estemos desarrollando, el tipo de cliente que tengamos, el tipo de interesados que tengamos.

## Diferentes contextos

También nos vamos a encontrar en situaciones que van a tener que ver con el tipo de involucramiento del cliente:

- ◆ En algunos casos **el cliente/usuario no va a formar parte del equipo** entonces se suceden situaciones como las que mencionábamos antes:
  - Nos es difícil alcanzar el consenso.
  - Los requisitos de muy alto nivel que nos formulan estos usuarios hay que bajarlos a nivel de detalle como para poder transmitírselo a un desarrollador, lo cual implica que haya una posibilidad ahí de error de interpretaciones de traducción.
  - A veces la organización que formula la solicitud o el pedido, el requerimiento, lo hace y la organización que desarrolla es otra en otra entidad, lo cual implica la necesidad de formular a lo mejor algún tipo de contrato, algún tipo de documento formal y todo eso complica obviamente las cosas.
- ◆ Si tenemos la suerte de **contar con un representante del usuario**, con un representante del cliente en el equipo:
  - Las cosas son mucho más fáciles porque ese experto está disponible para que lo consultemos si tenemos alguna duda, está ahí con nosotros la mayoría del tiempo.
  - Se evita la necesidad de comunicarse formalmente, disminuye la necesidad de armar documentos demasiado detallados
  - El feedback es mucho más fluido, y de alguna manera forma parte del equipo.
  - Por supuesto que tiene esto algunas contraindicaciones: si este usuario representante no es realmente de peso y no representa adecuadamente a los usuarios, estamos en problemas; también es cierto que tener a alguien experto en el negocio sentado en el equipo de desarrollo es una especie de claudicación, de entender el problema por parte de la gente del equipo de desarrollo. En definitiva, hay que buscar ahí un escenario intermedio.

También es cierto que el tipo de software que desarrollemos va a tener una profunda influencia, como decíamos antes, en el proceso. No es lo mismo desarrollar software en forma interna, en un área de sistemas en donde mis clientes pertenecen a la misma organización, que desarrollar software a medida cumpliendo a lo mejor con los requerimientos explicitados en una licitación pública. A veces estamos desarrollando un producto de software que va a impactar en el mercado masivo, a veces estamos desarrollando software que va a ser usado internamente, claramente ahí hay un tema importante de escala. A veces estamos desarrollando un servicio que va a estar basado en software. A veces vamos a estar desarrollando software para un producto sobreembebido o software en tiempo

real que va a estar embebido en un producto, lo cual complica muchísimo la definición de los requisitos y el proceso de desarrollo en general, porque una vez que ese software está incluido en el dispositivo, va a ser probablemente muy difícil de ajustar y de cambiar, con lo cual es crítico entender cuáles son los requisitos.

## Actividades y técnicas

Hay múltiples técnicas que vamos a ir viendo en futuros vídeos aquí simplemente vamos a presentar algunas de ellas.

- **Descubrimiento:** Algunos autores hablan de obtención de requisitos, como si los requisitos estuvieran ahí y los pudiéramos ir a recoger. Lo cierto es que los requisitos se van descubriendo paulatinamente. ¿Cómo?
  - Entrevistas a los usuarios.
  - Talleres de requisitos. Los talleres son reuniones un poco más estructuradas en donde hay un facilitador que organiza las actividades en donde hay exposición de algunos expertos en algunos casos.
  - Encuestas que vayan indagando acerca de cuáles son los problemas actuales, que vayan indagando acerca de las complicaciones actuales con los procesos internos y que nos permitan identificar posibles mejoras.
  - Sesiones de brainstorming. Ante un problema, formular soluciones. Sesiones de brainstorming y los Focus Group son muy adecuados para productos de consumo masivo. Los Focus Group son reuniones de trabajo con posibles usuarios del producto que queremos desarrollar. En el caso para el mercado masivo son muy útiles.
  - Observación. Ir a ver qué es lo que está haciendo el usuario con las herramientas actuales, también nos permite entender cuáles son sus problemas y cuáles son las posibles soluciones. En definitiva, nos permite derivar cuáles son los requisitos que va a tener que resolver el software. Una versión extrema de la observación es sentarse y hacer el trabajo uno mismo en lugar del usuario para entender realmente cuáles son sus problemas.
  - Analizar documentos actuales, analizar sistemas actuales en el caso de que tengamos que reemplazar algún sistema ya existente.
  - Plantear escenarios de utilización.
  - Presentar prototipos que ayudan a discutir y a entender si descubrimos bien los requisitos.

[V / F] Los *focus groups* son adecuados para descubrir requisitos en productos de consumo masivo.

- Una vez que tenemos entendido cuáles son los requisitos, tenemos que analizarlos. **Análisis y especificación usualmente se realiza en simultáneo**. El **análisis** se enfoca más bien en identificar y resolver conflictos entre los requisitos, descubrir los límites del sistema y la interacción entre el sistema y el medio ambiente, y también, el análisis se encarga de derivar requisitos del sistema a desarrollar. No todo lo que nos dicen los usuarios van a automáticamente transformarse en requisitos. Van a surgir otra vez probablemente requisitos. Al usuario final probablemente no se le ocurran temas que tengan que ver con las regulaciones del mercado en el que operan, o temas de compliance con algún estándar. Por eso son requisitos con los que vamos a tener que cumplir, y que probablemente no nos lo digan en forma explícita. El modelado forma parte del análisis, tiene como propósito producir modelos abstractos del sistema del software o de la solución a desarrollar. No incluyen aspectos de diseño esos modelos. Normalmente vamos a encontrarnos con que va a hacer falta desarrollar más de un modelo. Vamos a hablar, por ejemplo, de modelos de dominio, de modelo de casos de uso, de historias de usuario, etc. La **especificación** consiste en producir uno o varios documentos, no necesariamente físicos, pueden ser obviamente electrónicos, de formalidad variable, con diferentes niveles de detalle, con la descripción del comportamiento que debe tener la solución a desarrollar. Normalmente no es algo que tenga que ver con el diseño, sino que va a tener que ver con el

comportamiento, con la esencia, con lo que el sistema tiene que ser, independientemente de cómo lo implementemos.

Algunas de las técnicas de análisis-especificación, son las siguientes:

- La clasificación de requisitos. No todos los requisitos son igualmente importantes, deberíamos poder estar en condiciones de priorizarlas, priorizarlas de acuerdo a por ejemplo, “esto tiene que estar sí o sí”, “esto podría estar” y “esto definitivamente podría esperar para más adelante”. Para poder hacer esa clasificación y esa priorización, necesitamos poder estar en condiciones de negociar.
- Negociar no es imponer una visión, sino llegar a un acuerdo entre las partes acerca de un tema determinado.
- El modelado de escenarios, el modelado de dominio, el prototipado, la especificación utilizando lenguaje natural o algún otro tipo de notación, también son técnicas aplicables al análisis y a la especificación de requisitos.
- Utilizar reglas de negocio que son mecanismos claros para especificar políticas, para especificar determinado tipo de conductas en la organización también son muy útiles.
- Criterios de aceptación y casos de prueba. Definir bajo qué condiciones vamos a dar por satisfechas las necesidades y expectativas del usuario que hemos especificado mediante requisitos es un mecanismo espectacular para asegurarnos de haber entendido qué es lo que tenemos que hacer, qué es lo que tenemos que construir. Si podemos definir casos de prueba, o criterios de aceptación, en conjunto con los interesados con los usuarios de alguna manera vamos a estar definiendo bajo qué condiciones el sistema va a tener que operar y de alguna manera eso también constituye una especificación de cómo se tiene que comportar el sistema.

[V / F] El análisis y la especificación de requisitos generalmente se realizan secuencialmente en forma independiente.<sup>11</sup>

- Finalmente vamos a necesitar **verificar** los requisitos. Verificar implica asegurarnos de que hemos descripto correctamente los requisitos y para eso vamos a utilizar:
  - Revisiones.
  - Inspecciones.
  - Prototipos.
  - Los mismos criterios de aceptación y los mismos casos de prueba también nos van a servir para chequear si hemos descripto correctamente los requisitos.
- Por supuesto que las mismas técnicas las vamos a poder utilizar para la **validación**. ¿Entendimos correctamente los requisitos? Esto es claramente una actividad que vamos a realizar junto con los usuarios, con los interesados. Si le podemos mostrar un prototipo, de alguna manera vamos a estar validando que entendimos correctamente sus necesidades, que entendimos correctamente los requisitos.

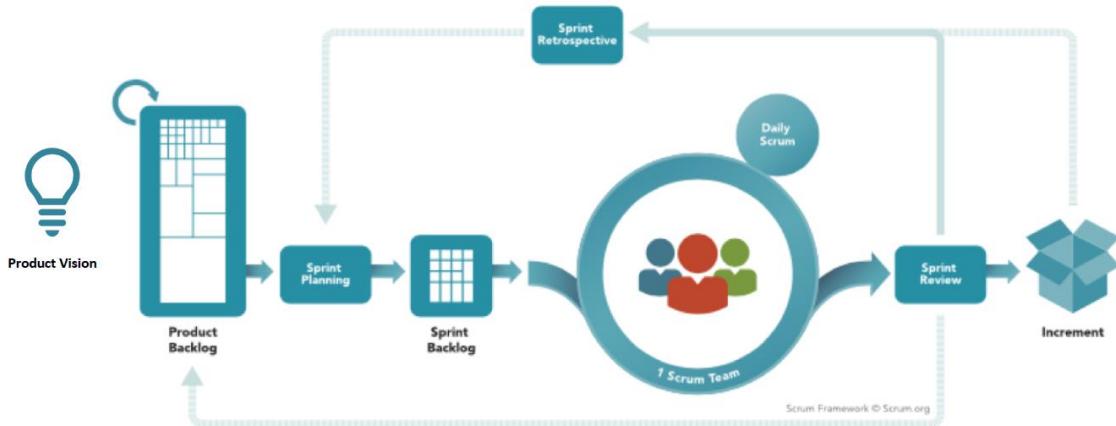
## Ingeniería de requisitos en el ciclo de desarrollo

Ahora bien, ¿qué es lo que pasa con el proceso de ingeniería de requisitos en el contexto del ciclo de desarrollo? Bueno hay varias alternativas, hay varios ejemplos que podemos tomar. En el pasado, estamos hablando hace 40, 50 años atrás, algunos sosténían la visión de que el software había que desarrollarlo de manera similar a como se desarrollaba el hardware. Entonces sería importante tener la definición completa de lo que el producto tenía que hacer antes de proceder al desarrollo. Entonces usualmente con ese tipo de ciclo de vida, con ese tipo de ciclo de desarrollo, todas las actividades de ingeniería de requisitos estaban concentradas al inicio de los proyectos, al inicio del ciclo de desarrollo. En los ciclos de vida un poco más modernos, las actividades de ingeniería de requisitos, como decíamos al principio, van sucediendo en forma iterativa e incremental a lo largo de todo el ciclo de vida.

---

<sup>11</sup> Análisis y especificación usualmente se realiza en simultáneo.

## Un ejemplo: Scrum



Un ejemplo de ciclo de vida iterativo e incremental es Scrum, que ya hemos mencionado en un momento con anterioridad. Scrum divide al ciclo de vida del producto en etapas, en releases. Cada release es una versión que sale del producto. Para preparar una versión, se utiliza a su vez una serie de interacciones llamadas Sprints. Un Sprint es una iteración que dura entre 2 y 4 semanas. Durante ese sprint el equipo está abocado a refinar la definición del producto y a desarrollar y probar el producto. Al final de cada sprint el equipo entrega una versión potencialmente operativa del producto final. No necesariamente la versión que se vaya a entregar, pero tiene que ser una versión que esté probada y que esté estable, no necesariamente que sea completa. Entonces el proceso arranca con una definición de la visión del producto, esto puede estar plasmado en un documento, en una presentación. Esa visión del producto, la establece el product owner. El product owner es, como su nombre lo indica, el dueño del producto. Es un representante del usuario que trabaja junto al equipo de desarrollo y que es el encargado de definir qué es lo que el sistema tiene que hacer. Ese documento de visiones, esa definición de visión es una descripción de muy alto nivel de cuáles son los problemas que busca resolver el producto a desarrollar, cuáles son sus objetivos, cuáles son potencialmente sus usuarios y en todo caso, grandes funcionalidades que tiene que incluir. Esa definición de visión permite establecer el Backlog del producto. ¿Qué es el Backlog del producto? Una lista priorizada de las funcionalidades que hay que desarrollar. No es una lista estática, sino que se va a ir refinando a medida que vayamos avanzando en el desarrollo. Entonces, el equipo de trabajo, al comienzo de cada sprint, se reúne con el product owner, discute cuáles son las features, las funcionalidades prioritarias a desarrollar, decide, de común acuerdo, cuáles son las features, las funcionalidades que se pueden tomar para desarrollar en el sprint, esto en función de la capacidad operativa del equipo, previamente, ha habido una estimación de esfuerzo, duración, costo de desarrollar cada una de esas funcionalidades, esas funciones, en definitiva, son el equivalente a nuestros requisitos. Y con esa información, se prepara la planificación del sprint, en definitiva, el Sprint Backlog es otro Backlog que lo que tiene es una definición de las actividades que hay que llevar a cabo para implementar cada una de las funcionalidades que se han tomado para este Sprint. El equipo inicia su trabajo, hay una dinámica diaria, hay una reunión que se llama Daily Stand up Meeting, en donde el equipo en su totalidad se reúne, revisa qué fue lo que hizo el día anterior, revisa o define qué es lo que va a hacer durante el día vigente, el día de la fecha, se plantean algunos obstáculos, se plantean problemas, no se discuten soluciones, simplemente se define el plan de acción a seguir. Entonces tenemos como una mini iteración, que es la iteración diaria, una iteración más grande que es la iteración del sprint y una todavía más grande que es la del release. En definitiva, son todos los loops que estamos viendo en el diagrama. Entonces, al final de cada sprint, lo que vamos a tener es una versión del producto probada, estable, que cumple con los requisitos, con las funcionalidades comprometidas, y que potencialmente, se podría entregar si fuera el caso, por supuesto, que poder arribar a una definición completa del producto, va a llevar varios Sprints. Fíjese que al final de cada sprint lo que tenemos es una retrospectiva. En esa retrospectiva, lo que realiza el equipo es qué funcionó en el equipo, qué es lo que no funcionó, qué obstáculos tuvieron. De alguna manera, estamos implementando ahí un feedback que nos permite refinar la planificación y nos permite a su vez mejorar el proceso de desarrollo. También, al final de cada sprint, estamos en condiciones de refinar el product Backlog. Una de las cosas que hay que hacer con el product Backlog, es decidir si la lista de funcionalidades, en definitiva, la lista de requisitos que tenemos ahí cambia de prioridad, sigue siendo la misma o si hay que hacer algo sobre esa lista.

[V / F] El product owner es un representante del usuario que trabaja junto al equipo de desarrollo. Se encarga de definir qué es el producto o servicio a desarrollar.

Scrum no necesariamente utiliza una técnica llamada User Stories. User Stories es una manera de describir qué es lo que el sistema necesita con respecto al sistema. Qué es lo que necesita que el sistema le ayuda a resolver. No es un requisito en el sentido absolutamente formal del término, pero sí es un puntero a una charla que hay que tener con el especialista, en este caso el product owner. Por ejemplo, como feature yo puedo tener una definición de una historia de usuario que nos diga como alumno quiero poder inscribirme en un curso de una materia, esa puede ser una definición de una user story. Claramente, para que sea un requisito en el sentido clásico que hemos descripto antes, hace falta alguna información adicional. En el caso de la User Stories, esa información adicional lo dan los criterios de aceptación asociados a esa historia de usuario. Vamos a ver un poco más de eso más adelante. Pero bien, este es un ejemplo de qué es lo que pasa con los requisitos a lo largo de un ciclo de desarrollo que sigue un modelo de proceso iterativo e incremental. Vamos a profundizar en próximos vídeos, cada una de las técnicas que hemos presentado y vamos a profundizar qué lo que pasa con cada una de ellas en función de la metodología o estilo de desarrollo elegido.

## Disciplinas relacionadas

Hay 2 disciplinas relacionadas con la ingeniería de requisitos que es importante mencionar:

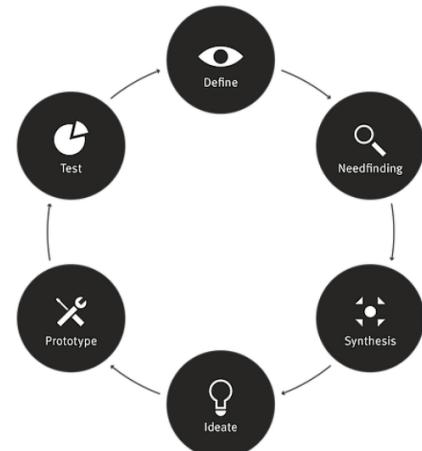
### Business Analysis

Una es el business analysis. Esta disciplina busca aplicar conocimiento, técnicas y herramientas para identificar problemas, determinar necesidades, y formular y recomendar soluciones para satisfacer dichas necesidad. No es necesariamente la solución es software, sino que puede implicar cambiar un proceso de negocio, transformar una estructura organizacional, cambiar el rumbo estático de la compañía. Es una disciplina si se quiere más general y que está muy alineada con la ingeniería de requisitos porque siempre terminamos identificando problemas que no necesariamente se resuelven con software. Y de hecho, si recordamos lo que hemos dicho con anterioridad, el software siempre forma parte de un sistema, con lo cual, es entendible que cuando estemos analizando los requisitos de una pieza de software de una futura aplicación nos encontramos con que hay otros problemas para mejorarlo.



### Design Thinking

La otra disciplina relacionada es Design Thinking. Design Thinking viene del lado del diseño de producto, no necesariamente del diseño de productos de software. Es un método centrado en las personas que tienen como propósito resolver problemas difíciles y mal definidos. Sigue un ciclo iterativo e incremental. Se fusiona muy bien con Scrum, con las metodologías ágiles, es perfectamente aplicable en dichos contextos. De hecho, en la Universidad de San Gallen hay un cuerpo de docentes que está trabajando, investigadores que vienen trabajando desde hace tiempo en la fusión entre ambas disciplinas.



[V / F] Dentro de la ingeniería de requisitos es poco de importante contar con habilidades sociales ya que existen herramientas que simplifican el trabajo.<sup>12</sup>

---

<sup>12</sup> Más allá de las habilidades técnicas, tiene que tener muchas habilidades blandas.

## 2.1 - Design Thinking

### Design Thinking

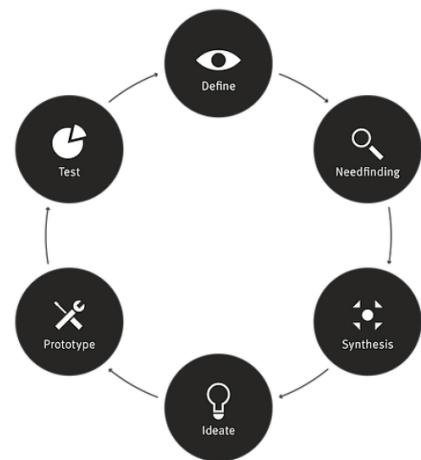
#### Definición

Ya sabemos que el desarrollo de software exige un enfoque cada vez más ágil, **centrado en las personas**, con el propósito de descubrir y satisfacer las necesidades de los distintos sectores, de las distintas partes interesadas. La mayoría de esas veces esas necesidades son difusas, o están en conflicto entre sí. Design thinking es un enfoque que se ha convertido en uno de los métodos que se pueden utilizar **para abordar problemas que se consideran difíciles por su propia naturaleza**. Por ejemplo, aquellos en donde los requisitos son muy volátiles o están poco claros. En lugar de concentrarse prematuramente en soluciones tecnológicas, design thinking propone una inmersión profunda en el dominio del problema. Proporciona **un enfoque estructurado de resolución de problemas** que se basa en gran medida en la exploración de necesidades, en la creación rápida de prototipos no tecnológicos (básicamente en papel) y el trabajo de equipos multidisciplinarios.

#### Etapas

Design thinking propone un proceso iterativo, exploratorio, compuesto de varias fases, varias etapas:

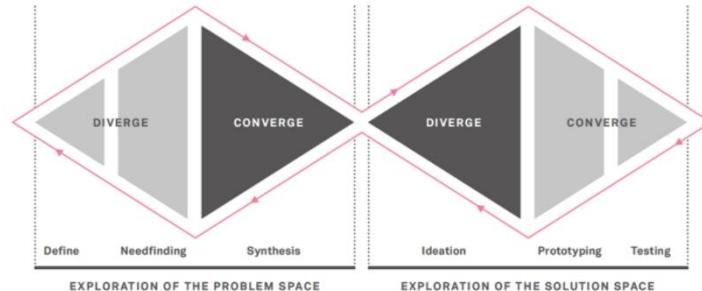
- En la etapa de “**Define**” se identifica cuál es el problema de resolver, se lo define, se entiende cuál es el contexto del problema.
- La siguiente etapa de “**Needfinding**” se identifican las necesidades de los usuarios y de los clientes y seguramente de otros stakeholders, a través de técnicas muy similares a las que se utilizan en la ingeniería de requisitos (entrevistas, observación, encuestas, etcétera).
- En la siguiente etapa, con toda esa información recabada, se procede a analizar y estructurar los resultados (“**Synthesis**”).
- Eso va a permitir en la siguiente etapa, “**Ideate**”, empezar a explorar una serie de posibles soluciones a los problemas identificados, soluciones que van a hacer prototipadas en la siguiente etapa (“**Prototype**”) y van a ser evaluadas por los usuarios, clientes y demás interesados en la siguiente (“**Test**”).



Design Thinking es un enfoque de trabajo centrado en:

- el producto
- el negocio
- las personas

Como puede verse, hay 2 grandes períodos que permiten agrupar estas etapas. Las primeras 3 etapas están más bien orientadas a analizar el espacio del problema, mientras que las segundas 3, están más orientadas a explorar el espacio de la solución. Se puede observar que tanto la exploración del espacio del problema como la exploración del espacio de la solución, las primeras etapas tienden a diverger, ya que se analizan múltiples alternativas, múltiples escenarios.



Posteriormente tenemos las etapas finales que tienden a converger en el caso de la exploración del espacio de las soluciones, en la síntesis de lo que hemos entendido y en el caso del espacio de la solución, en la síntesis de la definición de una solución posible a los problemas que hemos identificado.

# Design Thinking for Requirements Engineering (DT4RE)

## Métodos y herramientas (algunos)

Las técnicas que se utilizan son varias:

- Una de ellas es la **Stakeholder Map**, que describe a todos los interesados y a sus conexiones.
- **Storytelling**, que consiste en que el entrevistador que estuvo trabajando con los usuarios y clientes, recrea las entrevistas usando herramientas visuales.
- El **benchmarking** que permite analizar qué es lo que están haciendo los competidores, qué es lo que han hecho otras empresas, otras organizaciones que han resuelto problemas similares.
- La técnica de **Persona** que permite definir un arquetipo de los usuarios que han sido entrevistados, siempre hablando de un mismo grupo en la misma categoría de usuarios.
- El **Customer Journey**, que es un **relato de cómo uno de los posibles usuarios interactuaría con el producto o servicio o solución que vamos a definir**.
- Los **prototipos**, prototipos en papel. Vamos a ver que se propone utilizar prototipos de baja tecnología.
- Y por último, los **Wireframes**, que ya son prototipos un poquito más avanzados, pero que de alguna manera también permiten describir la interacción de los posibles usuarios con el producto.

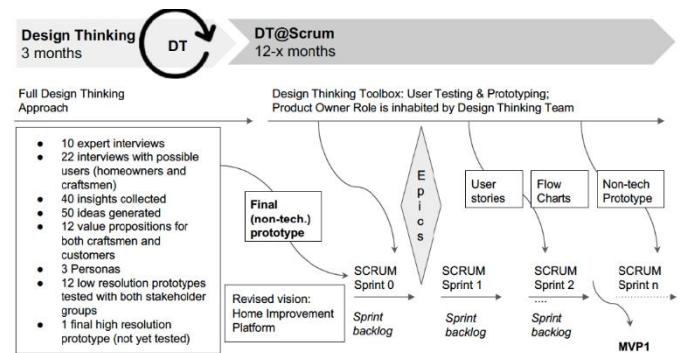
[V / F] *Customer journey* es un relato de como un usuario interactúa con el producto o servicio a definir.

## Design Thinking & Scrum: ejemplo de aplicación

Hay 3 estrategias básicas para emplear design thinking en el contexto de la ingeniería de requisitos:

1. La primera es utilizar design thinking en estado puro.
2. La otra es utilizar design thinking para una etapa previa al desarrollo, un “upfront” design thinking.
3. La tercera opción es incluir alguna de las herramientas o actividades de design thinking en el proceso de ingeniería de requisitos. Esta última opción es de integración de ambas disciplinas.

Lo que vemos en el diagrama es un ejemplo de aplicación de design thinking en un contexto de Scrum. Lo que se hizo en este caso fue ejecutar una etapa previa de design thinking, en la cual se analizó el problema a resolver y se definió la solución. Esa definición, en definitiva, permitió establecer la visión del producto y el product backlog, elementos que se utilizaron para realizar el desarrollo utilizando Scrum. Por otra parte, se enriqueció Scrum con alguna de las técnicas y herramientas propuestas por design thinking.



Claramente, design thinking y las metodologías ágiles se mezclan muy, muy bien. La naturaleza, iterativa, incremental y exploratoria de ambos enfoques trabajan muy bien en conjunto. Hay un equipo de investigación, que ha puesto un montón de información en la web que ampliamente recomendamos que consulten:

<https://www.dt4re.org>

Cuáles de las siguientes son características del enfoque Design Thinking

- Es un enfoque estructurado de resolución de problemas**
- Aborda problemas que por su naturaleza son difíciles**
- Se concentra en soluciones tecnológicas
- Posee un proceso iterativo, exploratorio y compuesto por fases o etapas.**

## 2.1 - Videos 2 y 2.1

[V / F] En los desarrollos predictivos los requisitos se van elaborando progresivamente a lo largo del ciclo de vida.

[V / F] Los requisitos son un diseño del sistema a construir

[V / F] En la exploración del espacio de la solución solamente hay convergencia.

[V / F] Business Analysis y Design Thinking son nombres alternativos para la ingeniería de requisitos.

## 2.2 - Wiegers capítulos 1 y 2

[V / F] El objetivo del desarrollo de requisitos es acumular entendimiento y conocimiento compartido que sea lo suficientemente bueno como para permitir la construcción de la siguiente porción del producto en un nivel de riesgo aceptable.

Marque todas aquellas opciones que demuestran un bajo nivel de aplicación de buenas prácticas de requisitos en una organización

- El equipo de desarrollo interactúa en forma constante con los usuarios representativos para entender sus necesidades
- Existen funcionalidades ya desarrolladas en el software que nadie aprovecha o utiliza**
- Al momento de programar los desarrolladores encontraron ambigüedades y falta de información por lo que tuvieron que adivinar y hacer suposiciones**
- Los usuarios se concentran en las pantallas o características de la interfaz del usuario y no en lo que ellos necesitan que cumpla el software**

[V / F] Es suficiente con que algunos de los interesados se comprometan a aplicar las mejores prácticas en el desarrollo y gestión de requisitos para producir productos de calidad superior

Seleccione qué término corresponde con cada una de las opciones:

	Req. No funcional	Req. funcional	Funcionalidad (feature)	Req. del usuario	Req. del sistema
Portabilidad	✓				
Pagar una factura				✓	
Localización	✓				
Se deben poder recolectar los datos del sensor de temperatura					✓
Actualización automática del listado de firmas de virus en un producto antimalware			✓		
El sistema pedirá confirmación al usuario ante la solicitud de imprimir más de 10 copias		✓			

[V / F] Un interesado es un individuo u organización que obtiene un beneficio directo o indirecto de un producto

Seleccione la o las disciplinas que componen el desarrollo de requisitos:

- Validación**
- Obtención / Descubrimiento (Elicitation)**
- Evaluación
- Especificación**
- Documentación
- Análisis**
- Exploración
- Confirmación

[V / F] Un requisito es una descripción de cómo el sistema deberá comportarse, o una propiedad o atributo que deberá poseer

[V / F] Planificar iteraciones de exploración de requisitos, refinándolos progresivamente desde un alto nivel a uno con más precisión y detalle, y confirmando las modificaciones con los usuarios, es una de las claves para el éxito del desarrollo de requisitos

Seleccionar qué definición corresponde a cada uno de los términos:

	Requerimiento de negocio	Requisito de usuario	Requisito funcional	Requisito del sistema	Feature (funcionalidad)
Una o más capacidades del sistema, lógicamente relacionadas, que proporcionan valor a un usuario y se describen mediante un conjunto de requisitos funcionales.					✓
Una descripción del comportamiento que un sistema exhibirá bajo condiciones específicas.			✓		
Un objetivo o tarea que clases específicas de usuarios deben poder realizar con un sistema o atributo de producto deseado.		✓			
Un objetivo de negocio de alto nivel que tiene la organización que construye un producto o un cliente que lo adquiere.	✓				

Un requisito de nivel superior para un producto que contiene múltiples subsistemas, que podría ser todo software o software y hardware.					✓	
---	--	--	--	--	---	--

[V / F] Las reglas de negocio son en sí mismas requisitos porque existen más allá de los límites de cualquier aplicación de software

## 2.3 - Wiegers capítulo 14

[V / F] La integridad trata de prevenir la pérdida de información.

[V / F] Aumentar la usabilidad generalmente impacta en forma negativa la portabilidad

[V / F] La portabilidad es un atributo de calidad externa.

[V / F] Los requisitos de seguridad (safety) buscan evitar daños a los datos de las personas.

[V / F] La robustez es un atributo que indica cuán confiable es un sistema.

[V / F] El desempeño (performance) y la eficiencia son atributos que no están relacionados entre sí.

# 3 - Técnicas de descubrimiento de requisitos

## Ingeniería de requisitos

### Actividades

Recordemos que esta actividad, que forma parte del grupo de actividades de desarrollo de requisitos, junto al análisis, la especificación, la verificación y la validación, tiene como propósito determinar cuáles son los requisitos que deberá satisfacer el sistema que tenemos que construir. Utilizamos la palabra descubrimiento en lugar de obtención, captura, ordenamiento, por el simple hecho de que los requisitos no están en la mente de los usuarios o de los clientes a la espera de que vayamos a cosecharlos o recolectarlos. Como bien decía Joseph Goguen los requisitos en realidad están en el sistema social de los usuarios y deben ser ideados, descubiertos a través de un esfuerzo cooperativo entre desarrolladores, clientes, usuarios y otras partes interesadas. **Por eso decimos que las dificultades más grandes que debemos enfrentar no son tanto técnicas, sino más bien sociales, políticas y culturales.**

## Descubrir requisitos

### Complicaciones

Estas dificultades tienen múltiples orígenes: Por un lado, los interesados no siempre saben qué es lo que quieren, excepto en términos muy muy general. Cuando saben que es lo que quieren muchas veces, no están en condiciones de articularlo adecuadamente. Los usuarios tienden a comunicarse utilizando su propio vocabulario y dando por sentado el conocimiento implícito de su propio trabajo, que nosotros no necesariamente tenemos por qué conocer. Adicionalmente, nos vamos a encontrar con distintos usuarios, con diferentes puntos de vista acerca de un mismo tema. Nuestra falta de experiencia en el dominio del problema como desarrolladores, como analistas, agregan un grado extra de complejidad. Si no es suficiente, se nos va a dificultar aún más entender correctamente las necesidades y las expectativas de los usuarios. Finalmente, no se puede soslayar el papel que juegan los factores organizacionales y políticos. Las agendas de los diferentes grupos de usuarios e interesados muchas veces son opuestas, son contrapuestas. Lo que convierte a este proceso de descubrimiento y al proceso de ingeniería de requisitos en general, en un permanente juego de negociación política. Y los cambios legales y regulatorios, que suelen aparecer de la noche a la mañana, que son tan comunes en nuestro entorno, le agregan todavía una dimensión extra.

Las mayores dificultades para descubrir requisitos están asociados a temas

- Culturales**
- Técnicos
- Sociales**
- Del dominio o el contexto del problema a resolver
- Políticos**

# Partes interesadas

## ¿Quiénes son?

Por esto que es tan importante identificar tan temprano como se pueda, quienes son las personas, grupos, organizaciones que tienen algún tipo de interés en el sistema que debemos desarrollar. Alexander y Robertson proponen un modelo muy útil llamado simplemente “**The onion**” por las capas que se van dibujando alrededor de ese sistema, que es el centro. En cada una de esas capas podemos ubicar a los diferentes interesados:



- 1º) En la capa más cercana del producto encontraremos a los **usuarios finales**, quienes van a interactuar directamente con él. Para estos autores, el producto o aplicación junto a los usuarios finales constituyen lo que llaman un sistema.
- 2º) En la siguiente capa, nos vamos a encontrar con los **beneficiarios económicos**, los **responsables de mantener en funcionamiento la solución y otros sistemas que actuarán como interfaces con la última capa**, el medio ambiente.
- 3º) En este **medio ambiente** es donde encontraremos a los **beneficiarios funcionales, a los auspiciantes económicos, a los entes reguladores** (AFIP, ARBA, el Banco Central o cualquier otro organismo que plantea algún tipo de regulación), **asociaciones profesionales y el público en general**. También vamos a encontrar en esta última capa a **los interesados negativos**, que son aquellos que activa o veladamente, atacan la idea del nuevo sistema por las más diversas razones, quizás por intereses económicos o porque prefieren una solución alternativa o porque el sistema que se va a desarrollar afecta su actual forma de trabajar.

En el modelo de interesados propuesto por Alexander (Onion model)...

- El modelo busca determinar las interacciones entre los roles.**
- El modelo busca determinar solamente los roles ubicados en la segunda capa ya que son los más relevantes para el desarrollo.
- En la primera capa del modelo se ubica el sponsor del proyecto.
- La última capa del modelo representa el ambiente.**
- La primera capa del modelo está compuesta por los usuarios que van a operar el sistema o servicio.**
- La segunda capa del modelo contiene a los beneficiarios funcionales del sistema.**<sup>13</sup>

## ¿Cómo descubrir requisitos?

### Técnicas

Una vez identificados los interesados y sus respectivos objetivos, vamos a estar en condiciones de iniciar la actividad de descubrimiento de requisitos propiamente dicho. Hay varias técnicas que se pueden utilizar, vamos a revisar algunas de las más relevantes:

#### Entrevistas

La técnica más conocida y relativamente fácil de aplicar es la de **entrevistas**. Una entrevista es básicamente una reunión con uno o varios usuarios en donde hay un diálogo que tiene como objetivo identificar requisitos. Pueden ser estructuradas, cuando se elabora una lista de preguntas con anticipación, o desestructuradas. Por supuesto que también existe un enfoque mixto. Es una técnica relativamente fácil de usar, pero que tiene algunas desventajas. Entre ellas, los limitados puntos de vista que se obtienen, lo que nos deja algunos conflictos por resolver, y el conocimiento tácito de los entrevistados. También puede ser difícil que los entrevistados imaginen nuevas formas de hacer las cosas. Para que la entrevista sea efectiva hay que convocar a los interesados adecuados, definir la agenda

<sup>13</sup> En el video dice que están en la tercera capa, pero en el texto original, dice que los beneficiarios funcionales (functional beneficiary) están en la segunda capa.

con anticipación, saber preguntar, saber escuchar activamente y documentar las respuestas, ya sea en forma escrita o mediante audio o vídeo.

#### Para que una entrevista sea efectiva

- Tiene que ser estructurada
- No es necesario escuchar activamente ya que se puede documentar el audio.
- Es necesario que los entrevistadores propongan alternativas de solución.
- Se debe convocar a los interesados adecuados**
- El entrevistador debe tener conocimiento del contexto
- Es adecuado definir la agenda previamente.**
- Es mejor saber preguntar.**

### Observación

La **observación**, es una técnica que consiste en estudiar cómo la gente hace su trabajo. Puede ser pasiva (solo se observa, no se hacen preguntas) o activa. Una variante es la de hacer uno mismo el trabajo bajo la supervisión del especialista. Al igual que con las entrevistas, hay que saber preguntar. En el caso de que se trate de una observación activa y saber escuchar. También hay que saber documentar las observaciones realizadas. Nuevamente, podemos usar audio, video o simplemente tomar nota. Es una técnica fácil de utilizar, pero que tiene algunas limitaciones. Una de ellas es que puede no alcanzar para identificar los requisitos, ya que simplemente se observa el comportamiento actual. También es cierto que puede no detectar eventos inusuales, situaciones poco frecuentes.

### Talleres (de requisitos)

Los **talleres o “Workshops”** de requisitos son una de las técnicas más efectivas. Un taller consiste en **reunir a un grupo de interesados y expertos** con el objetivo común de encontrar una solución a un problema compartido por todos los presentes, sobre el cual probablemente se tengan distintas visiones, distintos puntos de vistas. Los talleres son actividad muy productivas, muy intensas, de duración limitada, que **requieren la coordinación de un especialista experimentado**, preparado y neutral. Es una actividad que, si no está correctamente administrado fácilmente, se puede salir de causa.

### Encuestas

Las **encuestas** tienen como propósito obtener información de mucha gente durante un período acotado. Es una técnica muy útil cuando hay una gran cantidad de usuarios potenciales. Las preguntas del cuestionario pueden ser cerradas o pueden ser abiertas lo que usualmente requiere un mayor análisis.

### Grupos de discusión (Focus Groups)

Un **“Focus group” o grupo de discusión** es una **discusión moderada por un coordinador** en la que típicamente participan entre 5 y 10 personas, que tiene como propósito obtener ideas, percepciones, actitudes acerca de un producto, de un servicio, de un concepto o de una oportunidad. Es un tipo de investigación cualitativa muy empleado en marketing y publicidad. Es una técnica muy útil, sobre todo si el software que queremos desarrollar apunta al mercado masivo, es decir, que no tenemos un cliente concreto. La calidad de los resultados depende, por supuesto, de la experiencia y preparación del moderador, y de la predisposición de los participantes a hablar u opinar.

#### En que técnica de descubrimiento de requisitos es necesario que el moderador esté capacitado

- Talleres de requisitos**
- Entrevistas
- FODA
- Focus group**
- Encuestas

## Análisis de reglas de negocio

Las **reglas de negocio** son directivas específicas, procesables, verificables que describen cómo opera una organización. Por ejemplo, los solicitantes de tarjeta de crédito deben ser mayores de 16 años y, en ese caso, si tienen menos de 16 años, tienen que ser autorizados por sus padres o tutores. Eso es una regla de negocio. Hay distintos tipos de reglas de negocio, cálculos, restricciones, hechos, facilitadores de acciones, etcétera, que están muchas veces embebidos en procesos, en documentos, en aplicaciones y por supuesto en el conocimiento tácito de los usuarios. El desafío es poder identificarlas, extraerlas, catalogarlas y documentarlas. Es una técnica muy útil que requiere en muchos casos un trabajo prácticamente arqueológico.

## Brainstorming

Una técnica general de creatividad que se puede utilizar para descubrir requisitos es la de **tormenta de ideas** o “**Brainstorming**”, que consiste en reunir a un grupo de personas para resolver un problema, y generar una gran cantidad de posibles soluciones, con la particularidad de que no se censura. La discusión de la viabilidad de las ideas generadas queda para una etapa posterior.

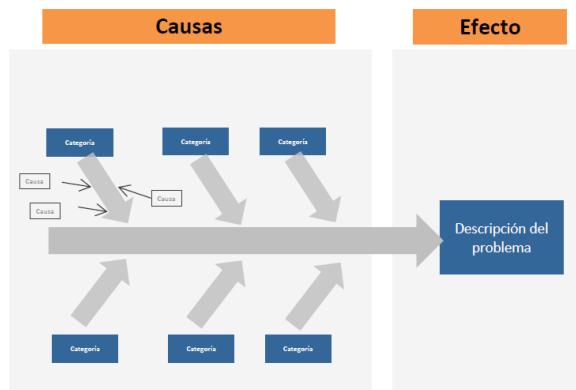
## Análisis de documentos

Otra técnica que se puede utilizar es el **análisis de documentos**. Esta técnica consiste en examinar la documentación existente con el propósito de identificar información relevante. Se pueden revisar, por ejemplo, planes de negocio, análisis de mercado, contratos, estudios de productos, documentos de productos de la competencia, procesos de negocio, etc. Es una técnica que puede resultar tediosa y puede consumir mucho tiempo, y lo que suele ser más grave, nos podemos llegar a encontrar con que la documentación está desactualizada.

[V / F] En la técnica de *brainstorming* resulta igual de importante que no exista censura sobre las ideas como así se desarrolle el debate de las mismas.

## Análisis de causa raíz

El **análisis de causa raíz**, conocido también como diagrama de Ishikawa o de espina de pescado, es una de las herramientas clásicas del movimiento de calidad japonés del siglo pasado. Tiene como propósito determinar cuáles son las causas que originan un problema determinado. Como ilustra la figura, se ubica en estos diagramas la descripción del problema a la derecha, mientras que a la izquierda se van describiendo las causas y sus causas agrupadas en categorías. Alguna de las categorías que se utilizan son personas, procesos, herramientas, métricas, pero no necesariamente estamos limitados a estas categorías. Es una herramienta muy útil para guiar discusiones, para utilizar en combinación con otras técnicas.



## Prototipado

El **prototipado** es una técnica que permite dar un paso tentativo en el espacio de la solución a través del modelado de las posibles interfaces del producto. Los prototipos ayudan a hacer un poco más tangibles los requisitos. Facilitan la verificación y también el descubrimiento de nuevos requisitos que se podrían haber omitido con anterioridad. Es una herramienta sumamente útil que se puede hacer simplemente con papel y lápiz, no hace falta aquí adoptar un enfoque demasiado tecnológico, que debe ser utilizado con mucho cuidado porque el objetivo que no hay que perder de vista es clarificar y encontrar requisitos, no discutir aspectos relacionados con el diseño.

## Open Space (Technology)

Una técnica general que se puede utilizar para realizar actividades de descubrimiento es “**open Space**” u “**Open Space Technology**”, como también es conocida. Es una técnica originalmente pensada para organizar dinámicamente reuniones o congresos con una gran cantidad de asistentes. El proceso a grandes rasgos es el siguiente: Se convoca a una reunión en un mismo lugar a la gente interesada en el tema que se va a tratar. A continuación, se define la agenda de la reunión en forma colaborativa. Aquellos interesados en un tema en particular lo manifiestan públicamente y lo escriben en una pizarra o en un elemento similar. Posteriormente, aquellos que estén interesados

en alguno de los temas planteados se reúnen, y en simultáneo con el resto de los grupos, dialogan entre sí acerca del tema planteado.

[V / F] El diagrama de espina de pescado se suele utilizar en forma individual o combinado con otra técnica para descubrir requisitos.

## Documentación de los resultados

Como ya hemos dicho, más allá de la técnica utilizada, es fundamental que registremos la valiosa información que vamos recogiendo durante las actividades de descubrimiento. Se puede tomar notas durante las entrevistas y sesiones de trabajo, pero lo ideal sería complementarlo con herramientas más modernas como audio o video.

### Minuta de reunión

En el caso particular de las entrevistas, se estila documentar lo conversado en una minuta de reunión. Este documento electrónico tiene como propósito registrar los principales temas tratados, quienes han sido los asistentes, los temas que han quedado abiertos para futuras reuniones y los próximos pasos que se han acordado dar. **Usualmente se distribuye el tiempo después de la reunión con el propósito de que sea validada por todos los asistentes.** Por supuesto que hay que evitar utilizar la minuta como un instrumento de coerción para evitar futuros cambios a los requisitos. Los requisitos van a cambiar, eso es inevitable. Lo que no hay que perder de vista es que éste es un mecanismo, la minuta, que tiene como propósito simplemente registrar lo conversado y confirmar si hemos entendido bien. Dicho esto, está claro que no solo las entrevistas deberían ser documentadas de esta manera. Cualquier reunión que tengamos relevante con cualquier Stakeholder debería tener su correspondiente minuta.

## Otras técnicas

Además de las técnicas que hemos visto, hay otra series de técnicas que se pueden utilizar, cuya lectura recomendamos como el modelado de metas y objetivos, el desarrollo de un mapa del ecosistema, el análisis de Fortalezas, Oportunidades, Debilidades y Amenazas (FODA), los cinco porqués, el diagrama de afinidad, el diagrama de impacto o “impact map” y el diagrama de historias de usuario. Estas técnicas se pueden utilizar, al igual que el resto, de manera combinada, dependiendo de la situación por supuesto, y de nuestra propia experiencia con respecto a su utilización.

[V / F] La minuta de reunión suele completarse durante la reunión para que sea validada por los participantes al final de la misma.

## Recomendaciones

Antes de terminar, van algunas recomendaciones. Como mencionábamos al comienzo, la actividad de ingeniería de requisitos en general y en particular el descubrimiento, está muy influenciada por aspectos políticos, organizacionales y culturales. Tenemos que tener mucho tacto y movernos con cuidado. Por eso es muy importante procurar identificar claramente a todos aquellos que deban participar en el proceso de descubrimiento o que puedan influenciarlo de alguna manera. Especial atención tenemos que poner en aquellos interesados negativos, quienes, por acción deliberada, o no, podrían ser una dificultad para el normal desenvolvimiento de las actividades. En estos casos tenemos que tener un plan de acción claro y concreto. No hay que olvidarse, por otra parte, de mantener cierto nivel de trazabilidad entre los requisitos y sus fuentes. Es crucial entender cuál es el origen de cada requisito, la persona que lo formuló o la sesión de trabajo en la que se acordó. También es importante registrar, cuando corresponda, la fundamentación de los requisitos. ¿Por qué es que se arribó a esa definición? ¿Cuáles son los justificativos? En términos generales, las actividades de descubrimiento deberían estar siempre guiadas por las inquietudes, preocupaciones de la organización que nos está requiriendo el desarrollo de la solución. Siempre nos vamos a encontrar que hay sospechas de cuáles son los problemas a resolver, o se sabe con cierto nivel de certeza cuáles son las necesidades concretas del mercado. Hay que tomar siempre esta información como punto de partida.

Otra fuente muy importante de requisitos que no hay que perder de vista son las leyes, regulaciones, normas y estándares que las organizaciones y las personas deben cumplir. De alguna u otra manera todas las ramas de la actividad humana tienen algún tipo de disposición que regula su funcionamiento, por ejemplo, los bancos y las empresas fintech están regimentadas por las circulares del Banco Central de la República Argentina. Los pagos con tarjetas de crédito están regulados por un estándar internacional que ha sido promulgado por la PSI que es la Asociación Internacional de Medios de Pago. AFIP, ARBA y otras agencias recaudadoras del gobierno regulan el funcionamiento de múltiples actividades económicas. Hay leyes que limitan la utilización de los datos personales. Todo esto muchas veces forma parte del conocimiento implícito de los usuarios. Seguramente van a omitir mencionarlo porque van a dar por sabido que nosotros estamos al tanto. Por eso es tan importante que los tengamos en el radar, que los tengamos bien identificados. Inevitablemente, durante las actividades de descubrimiento de requisitos nos vamos a encontrar con distintos puntos de vista acerca de un mismo tema. Los usuarios pueden tener visiones diferentes acerca de un proceso de negocio o distintas áreas pueden plantear prioridades diferentes. El management inclusive puede tener visiones divergentes acerca de cuál es la solución, o cuál es el producto que se debe desarrollar. Es crucial que busquemos la manera de llegar a un consenso a través del diálogo y la negociación. El prototipado nos puede ayudar a clarificar estos puntos de vista divergentes. Ante requisitos no del todo bien entendidos o que implican cierto nivel de riesgo, lo mejor es preparar un prototipo para facilitar el diálogo y la validación. Adicionalmente, no hay que perder de vista el interés de quienes deberán operar y mantener la solución una vez desarrollada. Ahí tenemos una fuente para múltiples requisitos en su mayoría no funcionales. Y si tenemos experiencia en el dominio, ya hemos desarrollado una solución similar a la que nos están pidiendo ahora, es muy válido reutilizar esa experiencia, reutilizar esos requisitos.

[V / F] El mayor desafío al momento de descubrir requisitos es consensuar con los interesados cuál es el problema que se quiere resolver.

## Conclusiones

### Resumen

Para cerrar entonces, un rápido resumen.

- ❖ Hemos visto que hay múltiples factores que pueden afectar el proceso de descubrimiento de requisitos:
  - Nuestro conocimiento del dominio.
  - Nuestra capacidad para comunicarnos.
  - La capacidad de los usuarios, clientes y otros interesados para comunicarse.
  - La política, la cultura de la organización del cliente y por qué no también la cultura y la organización del propio equipo de desarrollo.
  - Nuestra habilidad con las técnicas, nuestra habilidad con las herramientas que hemos planteado.
  - El medio ambiente en general.
- ❖ Es crucial entonces, manejar adecuadamente las técnicas y esforzarse en entender el dominio del problema. No hay que perder de vista que la ingeniería en requisitos es un grupo de actividades que permiten traducir necesidades y expectativas en una definición conceptual de cómo el sistema a desarrollar deberá comportarse, material que es imprescindible para las actividades técnicas, por eso es tan importante manejar el vocabulario, entender el dominio del problema. Si no entendemos el vocabulario que utilizan los usuarios, si no entendemos el dominio del problema, probablemente no entendamos cuáles son los requisitos.
- ❖ También hemos hecho mucho énfasis en identificar interesados, en identificar cuáles son sus objetivos, esto es fundamental para organizar adecuadamente las actividades de descubrimiento.
- ❖ También hemos visto que la técnica de entrevistas es la más sencilla de utilizar, pero la principal desventaja que tiene es que impide que apreciemos varios puntos de vista.
- ❖ Por esta razón es que en la mayoría de las situaciones, los talleres de requisitos van a ser la técnica más viable, sobre todo para conseguir consenso acerca del temas conflictivos. Lamentablemente consumen mucho tiempo de mucha gente, con lo cual hay que tener mucho cuidado a quién invitar y cómo manejar la sesión de trabajo para que sea productiva. Es muy interesante utilizar los talleres para revisar prototipos, o

utilizar estos talleres para desarrollar cooperativamente mapas de impacto o mapas de historias de usuario (dos técnicas de las que vamos a hablar un poco más adelante).

- ❖ Probablemente ante problemas difíciles, la técnica más viable sea brainstorming, y cuando tenemos productos que no tienen un cliente concreto, sino que estamos desarrollando software para un segmento del mercado, quizás lo único que podamos utilizar como técnica sea Focus group.
- ❖ Para finalizar, hemos visto que el prototipado es una técnica excelente, pero advertimos acerca de la importancia de no perder de vista su propósito en este contexto que tiene que ver, no con discutir temas de diseño. Será crucial que estemos en condiciones de manejar la ansiedad de los interesados. Lo que estamos viendo se parece al producto que tenemos que desarrollar, pero no lo es. Ese es un tema fundamental y que vamos a necesitar transmitir adecuadamente a los usuarios.

Al momento de descubrir requisitos...

- es muy importante la capacidad de comunicarse**
- se aprende del dominio o del contexto**
- es muy importante registrar quién dijo qué**
- es clave identificar los problemas del negocio**
- hay que tener cuidado qué se pregunta y a quién se pregunta**

## 3.1 - IIBA y PMI

¿Cuál técnica es más probable que se utilice en cada uno de los siguientes escenarios?

	<p><i>Una empresa de aviación planea el desarrollo de una nueva aplicación para el entretenimiento de sus pasajeros abordo.</i></p>	<p><i>Una empresa especializada en software para empresas está analizando agregar a su suite de gestión un nuevo sistema de CRM (Customer Relationship Management). Para ello, planea convocar a expertos en la materia y a los responsables del área de Marketing.</i></p>	<p><i>Una startup de cuatro socios con un producto orientado al público masivo detecta que el crecimiento de usuarios empieza a declinar, incluso a generar una contracción. Uno de los socios sugiere analizar qué funcionalidades se pueden agregar para revertir la situación.</i></p>	<p><i>El gerente comercial de una empresa convoca a un analista del área de sistemas para comenzar a conversar acerca del desarrollo de una nueva aplicación.</i></p>
Entrevista				✓
Observación				
Taller de requisitos				
Encuestas	✓			
Grupos de discusión / Focus group		✓		
Análisis de documentos				
Análisis de reglas de negocio				
Análisis de causa raíz				
Tormenta de ideas			✓	
Prototipado				
Open space				

### 3.3 - IIBA y PMI

Indicar cuál o cuáles de los siguientes opciones referidas a técnicas para descubrir requerimientos son correctas

- En los talleres de requerimientos todos los participantes pertenecen a un único sector. Precisamente el sector sobre el que se desean descubrir los requerimientos.
- La técnica de brainstorming, a diferencia de los focus groups y talleres de requerimientos, no requiere de un facilitador o moderador
- El facilitador en un focus group y en un taller de requerimientos debe tomar acciones concretas (si es necesario) para asegurarse que cada participante tenga su espacio para opinar**
- La técnica utilizada para recibir feedback de prototipos suele ser el taller de requerimientos.
- Una de las ventajas de la técnica de observación es que las personas realizan sus tareas correctamente cuando son observadas

¿Qué riesgo o riesgos asumimos al aplicar técnicas como talleres de requerimientos o focus group sin haber hecho previamente un análisis de interesados o stakeholders?

- Invitar a personas que pueden intentar sabotear la actividad**
- Ofender a interesados que no fueron invitados a la actividad
- No incluir a stakeholders cuya experiencia y conocimiento del problema puede llevar a la solución de un problema**

Son dueños de una organización que se dedica a exportar software. Construyen un análisis FODA. ¿Cuál o cuáles de las siguientes opciones se deben incluir dentro de las Oportunidades?

- Se espera un aumento considerable del dólar para el siguiente año**
- El principal competidor del mercado decidió especializarse en el mercado interno**
- El sector de I+D (investigación y desarrollo) diseñó un proyecto base que aumentará la velocidad de inicio de cada proyecto
- Contrataron a una persona de ventas que tiene contactos con importantes firmas del exterior

Una técnica utilizada para identificar reglas de negocio es:

- Encuestas
- Prototipado
- Brainstorming
- La construcción de árboles de decisión**

¿Cuál o cuáles de las siguientes afirmaciones del diagrama de espina de pescado son correctas?

- Cada causa a su vez puede tener otras causas**
- El diagrama de espina de pescado se focaliza en resolver el problema identificado

Se comienza identificando el problema a tratar (efecto)

## 3.4 - Bartyzel

[V / F] Para poder encontrar las principales necesidades y obtener información relevante las preguntas son la herramienta adecuada

[V / F] Las personas entrevistadas responden a las consultas sobre procesos indicando cómo funciona en la actualidad y no como debería funcionar con cambios o mejoras

[V / F] El contexto asumido en una conversación es clave al momento de indagar detalles asociados a la realidad

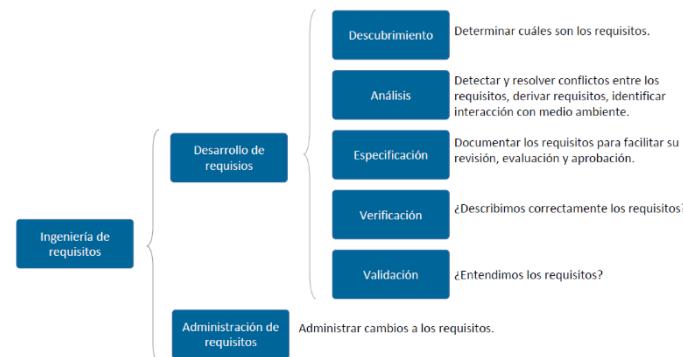
[V / F] La información detallada es consecuencia de que la información sea concreta.

# 4 - Técnicas de análisis, modelado y especificaciones de requisitos

## Introducción

Recordemos que el desarrollo de requisitos incluye actividades relacionadas con el descubrimiento, el análisis, la especificación, la verificación y la validación de requisitos:

- El análisis se enfoca en identificar y resolver conflictos entre los requisitos, en determinar cuáles son los límites del sistema y cómo es la interacción entre sistema y su medio ambiente. Como consecuencia del análisis es usual que aparezcan nuevos requisitos: requisitos derivados. El análisis implicará producir modelos, modelos *abstractos* del sistema, software o solución que vamos a desarrollar. Decimos que son abstractos o conceptuales porque estos modelos no incluyen aspectos de diseño.
- La especificación, por otro lado, consiste en producir uno o varios documentos de formalidad variable con diferente nivel de detalle con la descripción del sistema, software o solución a desarrollar. Hay que tener siempre presente que en condiciones normales, la especificación y el análisis se realizan en simultáneo.



Identifique de las siguientes sentencias cuál o cuáles son verdaderas

- A través del análisis y la especificación se definen las necesidades de los interesados
- En análisis de requisitos los modelos se consideran conceptuales o abstractos porque no incluyen aspectos de diseño
- Especificación y análisis se realizan en simultáneo
- El análisis de requisitos solamente se enfoca en identificar y resolver conflictos entre requisitos
- La especificación consiste en producir uno o varios documentos de formalidad variable con diferente nivel de detalle

## Requisitos

### Distintos interesados, distintos puntos de vista

También es importante que tengamos presente que la ingeniería de requisitos es un proceso que, partiendo de las necesidades y expectativas de los usuarios, progresivamente va elaborando una definición de los requisitos que deberá satisfacer el software que debemos desarrollar. Estos requisitos los podremos describir desde distintos puntos de vista:

- Uno de los puntos de vista es el de la organización que necesita el software. Estos tipos de requisitos son normalmente conocidos como **requisitos o requerimientos de negocio**.
- Otro punto de vista es el de los usuarios que van a utilizar el software. Normalmente conocidos como **requisitos de usuario**.
- Y finalmente nos encontraremos con el punto de vista de los desarrolladores que deberán implementar y mantener el software. A estos requisitos se los conoce como **requisitos de software**.
- En algunos casos, puede ser que haya una capa superior. En ese caso, serán los **requisitos del sistema**.

Para cada uno de estos puntos de vista usualmente se producirán distintos documentos llamados genéricamente *especificaciones de requisitos*. Los nombres de estos documentos variarán dependiendo de la audiencia y del alcance, pero su propósito colectivo es invariable: acordar entre las partes cuáles son los requisitos del software a desarrollar.

Por ejemplo, para registrar los requisitos de la organización que nos ha encargado el producto, utilizaremos artefactos tales como el documento de visión, el ConOps (Concepto de Operación), o la especificación de requisitos de negocio (BRD).

Para describir los requisitos desde el punto de vista del usuario (URD), usualmente utilizaremos una estrategia de caja negra. Es decir, vamos a describir únicamente la interacción de los actores de los usuarios con el sistema, no los detalles internos. Para esto hay múltiples alternativas, desde emplear el lenguaje natural hasta técnicas más elaboradas, como casos de uso o historias de usuarios.

Finalmente, para el punto de vista del equipo de desarrollo aparece la especificación de requisitos de software (SRS). Esta sí es una visión interna de comportamiento que deberá tener el software a desarrollar. No es un documento de diseño, sino una descripción abstracta o conceptual de lo que el software deberá hacer, independientemente de cómo lo implementemos.

[V / F] El análisis de requisitos se enfoca en identificar y resolver conflictos entre los requisitos.

## Alternativas de representación

Los requisitos se pueden representar de varias formas:

- El **lenguaje natural** es una opción, pero debe tenerse cuidado con la ambigüedad. Estos requisitos normalmente van a ser enunciados con los formatos: “el usuario debe...” o “el sistema debe...”.
- También se pueden utilizar **modelos visuales** para ilustrar procesos, para describir estados del sistema y cambios entre ellos, para describir relaciones de datos, grupos lógicos de trabajo, etc.
- Una tercera opción, menos frecuente en aplicaciones convencionales, son las **especificaciones formales**. En este caso, los requisitos se describen mediante el uso de lenguajes matemáticos muy precisos. Normalmente la mayoría de los casos utilizaremos una combinación de las primeras 2 opciones.

## Visión (visión y alcance)

Revisemos ahora en profundidad algunos de estos artefactos que acabamos de mencionar. Un documento muy habitual es el *documento de visión*, llamado en ciertos ámbitos también, de visión y alcance. Normalmente se produce en la primera fase o en la primera iteración de los proyectos de desarrollo. Define el sistema que tenemos que construir desde el punto de vista de los interesados de negocio. Es la base para las especificaciones más detalladas que se desarrollarán posteriormente y suele servir muchas veces como base para la elaboración de un contrato. El contenido puede variar, pero usualmente incluye una definición del problema a resolver o de la oportunidad de negocio a aprovechar, una descripción de los usuarios o interesados, una descripción de las prestaciones del producto (a muy alto nivel), riesgos, valor para el negocio, supuestos, restricciones, etc.

## SRS (Especificación de Requisitos de Software)

La especificación de requisitos de software, por otro lado, es un documento o grupo de documentos electrónicos, usualmente escrito en lenguaje natural y a veces complementado con gráficos y descripciones semi formales, como por ejemplo caso de uso o reglas de negocio. Se utiliza en los más diversos ámbitos, aunque históricamente está más ligado a ciclos de desarrollo más clásicos, más secuenciales, o en algunos entornos en donde hay contratos de por medio. Para evitar malos entendidos, la prosa debe ser clara, tiene que estar bien estructurado y los términos poco frecuentes deberían estar incluidos en un glosario. Hay varios estándares y varias recomendaciones acerca de cómo estructurar y escribir este tipo de documentos. Hay uno muy conocido que se llama EARS, del cual vamos a hablar un poco más adelante.

## Desarrollo ágil

En entornos ágiles es más difícil, sino improbable, encontrar especificaciones como las que acabamos de ver. En general, nos vamos a encontrar con historias de usuario que suelen estar complementadas con criterios de aceptación. Desde el punto de vista estricto para el mundo ágil, las historias de usuario no son requisitos en el sentido clásico, sino que son recordatorios de una conversación que hay que tener con el usuario o con el experto en

el dominio. Recordemos que en los desarrollos ágiles, el equipo no solamente está conformado por desarrolladores, sino que tenemos especialistas en el negocio (especialistas o usuarios capacitados) trabajando junto a nosotros.

[V / F] En los desarrollos ágiles hay especialistas del negocio o usuarios capacitados que trabajan a la par del equipo, por ende las historias de usuario no son consideradas requisitos puros sino que son un recordatorio de una conversación con dicho especialistas o usuarios.

## Atributos de una especificación bien escrita

Más allá del formato que utilicemos, una especificación bien escrita debe reunir las siguientes características:

- 👉 **Correcta:** Debe reflejar las verdaderas necesidades de los usuarios. No debe tener ambigüedades, es decir, que debe haber una única interpretación posible para cada uno de los requisitos.
- 👉 **Completa:** Esto ocurre cuando se cumplen las siguientes cualidades:
  - 📝 Todo lo que se supone que el software debe hacer está incluido en la especificación.
  - 📝 Todas las respuestas a todas las entradas al sistema en todas las situaciones posibles están incluidas en la especificación.
  - 📝 Todas las páginas del documento, todas las figuras, todas las tablas están numeradas con nombre y referenciadas correctamente.
  - 📝 No hay secciones vacías o que se vayan a completar posteriormente.
- 👉 **Verificable:** Es decir, que hay criterios para determinar si el software, una vez desarrollado, satisface o no los requisitos.
- 👉 **Consistente:** Es decir, no hay requisitos en la especificación que estén en conflicto entre sí, o que estén en conflicto con otros documentos, con otros artefactos.
- 👉 **Entendible:** La especificación puede ser entendida por terceros. Los requisitos se van a escribir normalmente una sola vez, pero van a ser leído muchas veces, por eso es muy importante tener una prosa clara y bien entendida.
- 👉 **Modificable:** La estructura, el estilo de la especificación, tienen que permitir que se pueda hacer una modificación de manera relativamente fácil ante un cambio en alguno de los requisitos.
- 👉 **Independiente del diseño:** Recordemos que la especificación describe el comportamiento del sistema independientemente de las decisiones que tomemos para implementarlo. Con lo cual, no implica ningún tipo de arquitectura en particular, ningún tipo de interfaz de usuario en particular, de ningún tipo de algoritmo o pieza de software.
- 👉 **Concisa:** No debe ser innecesariamente extensa.
- 👉 **Organizada:** Los requisitos tienen que ser fáciles de localizar en el documento.
- 👉 **Trazable:** Se puede identificar el origen de cada requisito.
- 👉 **Priorizada:** Los requisitos deben tener una prioridad.

Indique cuál o cuáles de las siguientes oraciones son correctas:

- Una característica para considerar una especificación completa es cuando no falta completar secciones**
- Que una especificación posea estructura facilita que sea entendible
- Los requisitos se escriben una vez pero se leen muchas veces**

## Análisis, modelado, especificación

Analizar, muchas veces, va a implicar construir modelos del sistema que tenemos que desarrollar. Estos modelos reflejan las abstracciones del dominio del problema y decimos que son abstractos o conceptuales porque describen el sistema en términos tecnológicamente neutros. Estos modelos no describen el diseño del producto. Y aquí hay 2 enfoques que se pueden adoptar: Uno más formal, como por ejemplo ocurre en el ámbito del Model Driven Design; o más informal, como pasa en el desarrollo ágil.

# Cómo analizar, modelar y especificar requisitos

## Técnicas

A continuación, vamos a presentar las principales técnicas que se pueden utilizar para analizar, modelar y especificar:

### Organización

Una técnica fundamental en la **organización** de requisitos. Esta técnica nos permite establecer categorías para identificar y ubicar más rápidamente los requisitos. Hay varias clasificaciones posibles:

- Binaria: funcional, no funcional.
- FURPS+: Clasifica a los requisitos en términos de funcionalidad, usabilidad, confiabilidad, desempeño, facilidad de soporte, otros.
- Por rol del usuario.
- Por área o proceso de negocio.
- Por evento, caso de uso o escenario.

Hay algunos aspectos que hay que considerar al organizar los requisitos. Por un lado debería haber un repositorio. Cualquier desarrollo no trivial implicará una cantidad significativa de requisitos. Es importante entonces establecer algún tipo de repositorio mediante el cual se puedan compartir entre todos los actores relevantes los requisitos. En las metodologías ágiles es muy usual, la utilización de notas adhesivas que se pegan en algún tipo de tablero físico en el lugar de trabajo. Eso, por supuesto que es totalmente viable, pero en algún momento va a ser importante que haya también una versión electrónica de ese tablero y de esas notas.

Otro tema importante es la **gestión de cambios**. Ya hemos mencionado que los requisitos inevitablemente van a cambiar a lo largo del proyecto. Hay que definir de qué manera vamos a reflejar esos cambios en el repositorio y en los demás mecanismos que vamos a utilizar para registrar los requisitos. Uno de los problemas más graves que podemos llegar a enfrentar es no saber cuál es la línea base oficial de los requisitos que el software tiene que satisfacer.

También es muy importante mantener cierto nivel de **trazabilidad**. Es importante entender la relación que tienen cada uno de los requisitos con el resto de los artefactos del ciclo de vida. Por ejemplo, un requisito, le pueden corresponder una serie de criterios de aceptación o de casos de prueba. También es importante saber cuál es el origen de cada requisito, por ejemplo, siempre viene de una conversación que se tuvo con el usuario de una reunión, una sesión de trabajo. Es muy importante saber quién dijo qué, en cuál reunión, en qué contexto. Esto es sumamente importante porque nos permite hacer análisis de impacto: Ante un cambio, qué otros artefactos vamos a tener que revisar y ajustar.

También necesitaremos registrar algunos **atributos** para cada requisito. Cada requisito debería tener, por ejemplo, una identificación única (aunque esto en las metodologías ágiles es un poco resistido), debería tener una prioridad, puede tener riesgos asociados, puede tener una complejidad. Puede también tener una definición de cuál es el esfuerzo que implicaría implementar ese requisito en el software.

Estos atributos, estos aspectos, es importante que los definamos, independientemente del tipo de herramientas que utilicemos. Como siempre, es mucho más fácil organizar las cosas de entrada y no tener que estar después sufriendo por la falta de estructura en la información.

Identifique de las siguientes sentencias cuál o cuáles son verdaderas respecto a la organización de requisitos:

- La organización de requisitos permite establecer categorías para identificar y ubicar fácilmente a los requisitos**
- Al organizar los requisitos contar con una buena trazabilidad de los mismos suele ser opcional
- Gestionar de los cambios en los requisitos ayuda a conocer la línea base de los requisitos**
- Es poco importante contar con un repositorio de requisitos
- Es necesario registrar atributos asociados a los requisitos como su identificación, prioridad, riesgos, complejidad o esfuerzo estimado**

## Priorización

La priorización nos permite asegurar que el esfuerzo del equipo esté dirigido hacia el análisis y la implementación de los requisitos más críticos para todos los interesados. Son varios los **criterios** que se pueden utilizar para establecer la prioridad:

- El valor para el negocio, el valor que genera para el negocio.
- Los riesgos técnicos o los riesgos de negocio.
- La dificultad que implica la implementación.
- Regulaciones o las políticas organizacionales.

Hay varias **técnicas** que se pueden utilizar:

- MoSCoW: Must / Should / Could / Won't. Se clasifican los requisitos en categorías: Obligatorios, probablemente necesarios, deseados, innecesarios o postergables. Otra alternativa posible, necesario deseados, o nice to have.
- Votación.
- Time boxing: Se decide la prioridad de los requisitos en función del tiempo de desarrollo disponible.
- Por el resultado de un análisis de riesgo.
- Por el resultado de un análisis de decisiones, a través de mecanismos más específicos de toma de decisiones.

**[V / F]** Para garantizar que el esfuerzo del equipo esté dirigido hacia el análisis y la implementación de los requisitos más críticos para todos los interesados es necesaria la priorización de los requisitos.

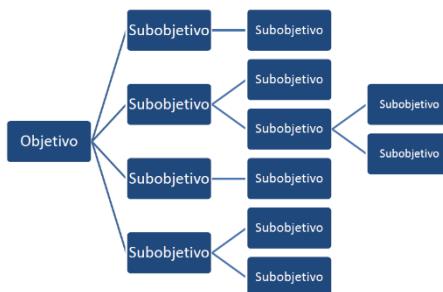
## Escenarios

### *Una familia de técnicas*

Vamos a hablar a continuación de la especificación de requisitos mediante escenarios. ¿Qué son los escenarios? Bueno, con este nombre se conoce a una familia de técnicas que están orientadas a contar historias acerca de qué es lo que hace la gente con los sistemas. ¿Cómo es que los utiliza?, o ¿cómo es que piensa que los van a utilizar? Nuestro cerebro social está especialmente adaptado para entender historias, es la forma en la que el hombre primitivo transmitía conocimiento de una generación a la otra, con lo cual este grupo de técnicas nos resulta muy familiar, resulta muy eficiente. Hay varias técnicas incluidas en esta familia, las más conocidas y de las que vamos a hablar a continuación son: casos de uso, historias de usuario y storyboards o guiones.

### *Usuarios y objetivos*

Todas estas técnicas comparten una característica esencial: Para identificar a los escenarios, proponen partir de los objetivos que las distintas categorías de usuarios tienen con respecto a la utilización del sistema. A partir de estos objetivos, se van identificando objetivos de menor nivel. Los objetivos de menor nivel se satisfarán mediante un escenario, un caso de uso, una historia de usuario, un storyboard.



### *Casos de uso*

Un **caso de uso** es una secuencia de acciones realizadas por un sistema que generan un resultado observable y de valor para un actor en particular. Un actor en este contexto es un tipo de usuario que va a utilizar el sistema. A cada acción de un actor, dentro de un caso de uso, le corresponde una respuesta del sistema. Los casos de uso en general son cajas negras, es decir, describen qué es lo que responde el sistema, pero no cómo el sistema elabora esa respuesta.

### *Historias de usuario*

Otra técnica muy popular, es la de **historias de usuarios**. Una historia de usuario es una descripción de la funcionalidad esperada de un sistema desde el punto de vista de un usuario. En sentido estricto según algunos autores, las historias de usuario no son requisitos, son simplemente un recordatorio de que se debe mantener una conversación con el usuario para profundizar los detalles de la funcionalidad asociada a la historia. Son más bien

requisitos de usuario con poca granularidad, si se quiere. Sin embargo, cuando las historias de usuario van acompañadas de criterios de aceptación, constituyen una descripción bastante completa de lo que se espera del comportamiento del sistema.

### Guiones (Storyboards)

Y por último, los **storyboards**: Los storyboards o guiones son pequeños relatos en prosa o descritos en forma gráfica, que nos cuentan cómo usar el sistema de determinadas circunstancias.

[V / F] Los casos de uso y las historias de usuario son herramientas complementarias.

## Modelado

### Lenguajes y notaciones

Vamos a continuar a hablar de algunas de las notaciones de lenguaje de modelado más importantes en el ámbito del análisis de requisitos. Como ya hemos mencionado en el video anterior, la preocupación por modelar sistemas viene desde los primeros años de la ingeniería de software. En la década de 1970 y de 1980 eran muy populares distintos enfoques basados en análisis estructurado. La notación IDEF (Integration DEFinition) es de esa época, como así también los diagramas de flujo de datos (DFD: Data Flow Diagram) y los diagramas de identidad-relación (ERD/ERM: Entity-Relationship Diagram/Model). Más recientemente, se han popularizado otros lenguajes de modelado, como por ejemplo UML (Unified Modeling Language), BPMN (Business Process Model and Notation) y SysML (System Modeling Language).

### Modelos y diagramas

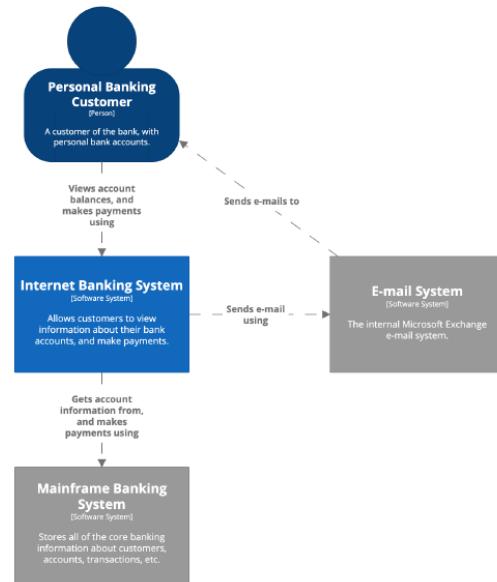
Vamos a presentar a continuación algunos de los modelos de análisis más populares:

#### Diagrama de contexto

C4 es una anotación para describir arquitectura de software. Si bien la arquitectura de software es un tema que no tiene que ver directamente con el análisis de requisitos, esta notación nos ofrece un diagrama muy útil para describir la vinculación del sistema con su medio ambiente. Es el llamado, muy adecuadamente, diagrama de contexto.

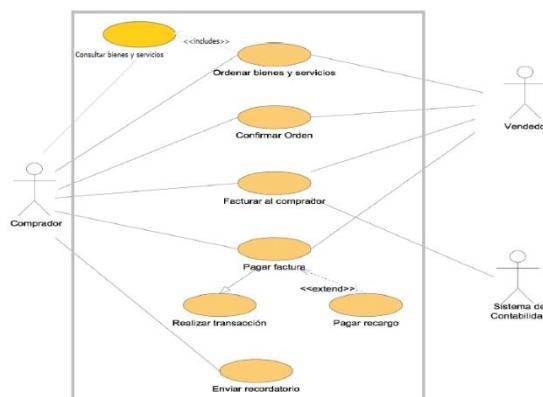
Como se puede ver en la figura, el diagrama incluye un icono para representar al usuario, un rectángulo (en este caso celeste) que representa el sistema a desarrollar y (en este ejemplo en particular) 2 rectángulos grises que representan los sistemas con los que, el sistema en cuestión, deberá interactuar.

Como puede verse, es un diagrama muy sencillo, tecnológicamente neutro, que es muy útil para describir el medio ambiente en el que se encuentra el sistema y los principales vínculos que mantiene este con las entidades que lo rodean.



#### Modelo de casos de uso

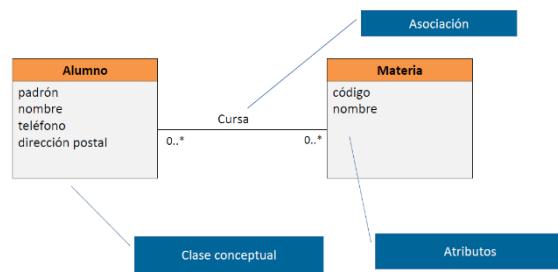
Un diagrama que tiene un propósito similar es el de **caso de uso**. Este es un diagrama que está incluido en el estándar UML, cuyo objetivo es representar los casos de uso del sistema y los actores primarios y secundarios con lo que el sistema interactúa. Estos actores representan tipos de usuarios que utilizan el sistema y también pueden representar otros sistemas que proveen o reciben información de este. Es un diagrama muy sencillo que cuando se utiliza en combinación con las especificaciones del caso de uso, forman parte del modelo de casos de uso.



## Modelo de dominio

Un modelo de dominio es una representación visual de los objetos o clases conceptuales del dominio del problema y de las asociaciones que éstos tienen entre sí. En este contexto, los objetos de dominio representan cosas que el sistema en cuestión necesita manejar o conocer (ejemplo: objetos manipulados en una organización: contratos, facturas, pedidos), objetos y conceptos del mundo real que el sistema necesita conocer o monitorear (ejemplo: avión: trayectoria, misil), eventos pasados o futuros (arriba, partida, pago), personas, roles u organizaciones (clientes, socios, alumnos). Construimos este tipo de modelos fundamentalmente para entender y analizar mejor el contexto en el que operará el sistema que tenemos que desarrollar. Usualmente lo utilizaremos en combinación con otras técnicas como casos de uso o historias de usuario.

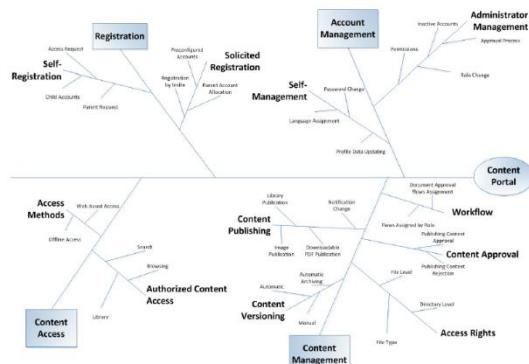
Para representar gráficamente los modelos de dominio, empleamos un diagrama de clases con la notación UML. La única salvedad que tenemos que hacer es que se deben incluir únicamente clases conceptuales, atributos y asociaciones. Estas clases no representan objetos del lenguaje de programación, no representan clases del lenguaje de programación, sino que representan objetos del negocio. Con lo cual, todos los aspectos que tengan que ver con la implementación deben ser removidos del diagrama. Recordemos, es un modelo de análisis y no un modelo de diseño de software.



[V / F] En pos de entender y comprender mejor el contexto del sistema a desarrollar el modelo de dominio es una técnica de análisis adecuada.

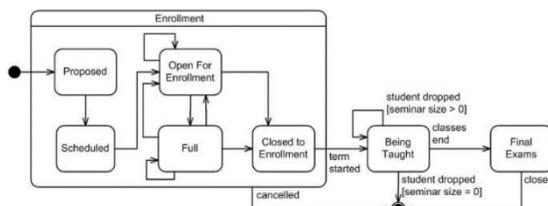
## Árbol de funcionalidades (prestaciones) [feature tree]

Otro modelo de análisis muy útil es el **árbol de funcionalidades o feature tree**. En el ejemplo lo vemos representado como un diagrama, similar al de causa efecto o Ishikawa. El objetivo es mostrar cuáles son las prestaciones o funcionalidades que componen el sistema y cómo éstas se pueden desglosar en prestaciones más pequeñas, de menor nivel. La estructura de descomposición típica sería: prestación, subprestación (o varios niveles de subprestación), requisitos de usuario, requisitos de software.



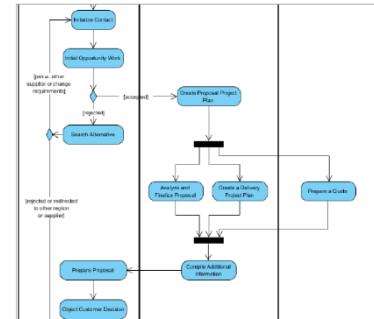
## Diagrama de Estados

El **diagrama de Estados** o de estado-transición, es un diagrama que permite representar los diferentes estados, por lo que puede pasar un objeto de dominio o un sistema. El software no solamente implica funcionalidad, sino que también implica manipulación de datos y también cambios de estado. Representar estos cambios de estado mediante lenguaje natural puede ser bastante problemático, por la ambigüedad que plantea, por eso esta herramienta es sumamente útil. Por otro lado, en sistemas de tiempo real, es decir, aquellos que ante un momento determinado en producir una respuesta dentro de un período específico, prácticamente es imprescindible desarrollar un diagrama de estas características.



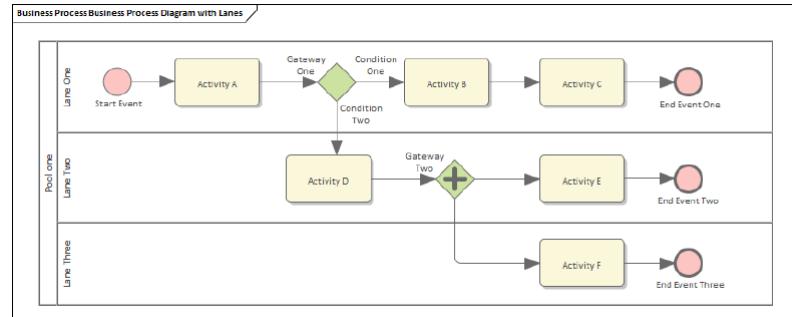
## Diagrama de actividades (UML)

Permite especificar las actividades o pasos dentro de un flujo de trabajo, dentro de un proceso, e inclusive, dentro de un caso de uso. Cada rectángulo en el diagrama representa una actividad o paso, las flechas representan el paso de control de una actividad a otra, los rombos representan punto de decisión, y las columnas verticales pueden representar actores, roles o sistemas.



## Flujo de actividades en un proceso de negocio (BPMN)

Un modelo similar al anterior es el **flujo de actividades** propuesto por la notación **BPMN** (Business Process Modeling and Notation). Este es un estándar de la OMG (Object Management Group). Este diagrama está orientado exclusivamente a representar procesos de negocio, procesos organizacionales. Nuevamente aquí los rectángulos son actividades, las flechas representan la secuencia de ejecución y los andariveles (swimlanes) representan roles o áreas organizacionales. Además de ser una herramienta para modelar procesos, la idea detrás de esta narración también es que sirva para generar un proceso que pueda ser ejecutado por herramientas especializadas, es decir, que pueda generar un esquema ejecutable del proceso. La notación es relativamente sencilla y muy fácil de aprender.



## Prototipado

Nuevamente, los **prototipos** aparecen como una muy buena herramienta, en este caso no sólo para descubrir requisitos, sino también para analizarlos. La idea con los prototipo de las interfaces de usuario, es explorar cómo podrían ser dichas interfaces, con el propósito fundamental de entender mejor el comportamiento del sistema a construir. Nuevamente, el tema aquí no es discutir aspectos de diseño, sino tratar de entender un poco más acerca de cómo el usuario entiende que utilizará el sistema.

## EARS (Easy Approach to Requirements Syntax)

Si lo que estamos haciendo es especificar el lenguaje natural, hay varios lineamientos que se pueden utilizar para minimizar los problemas derivados de la ambigüedad. Uno de los lineamientos se llama EARS. Propone formatos muy específicos para describir requisitos de distinto tipo: Ubicuos, es decir, generales, que aplican a todo el sistema; relacionado con eventos; relacionados con estados; relacionados con comportamiento opcional o indeseado. En la siguiente tabla podemos ver los formatos que propone:

Ubícuo	Algo que el sistema debe hacer incondicionalmente.	N/A	“El sistema deberá cumplir con el estándar IRAM 34882”
Basado en eventos	Algo que el sistema debe hacer en respuesta a un evento disparador	Cuando	“Cuando arribe un mensaje, el sistema deberá emitir un alerta de notificación”
Basado en estados	Algo que se activa mientras se permanece en un estado determinado	Mientras	“Mientras el vehículo esté en movimiento, el sistema mantendrá”
Opcional	Algo necesario sólo bajo determinadas circunstancias	Si	“Si se abre la puerta del vehículo con el motor en marcha, el sistema emitirá un alerta de peligro”
Comportamiento indeseado	Una respuesta del sistema a eventos no deseados	Si...entonces	

## Criterios de aceptación

Una estrategia muy útil para completar la especificación es identificar criterios de aceptación. La idea detrás de esto es que el simple acto de pensar las condiciones bajo las cuales los requisitos van a ser dados por satisfechos (esos son los criterios de aceptación), ayuda a encontrar problemas en la especificación mucho antes de comenzar con el desarrollo del producto. Por otro lado, en escenario donde se utilizan historias de usuario, los criterios de aceptación, sirven para completar la especificación. Recordemos que las historias de usuarios no son requisitos de software en el sentido estricto de la definición, sino que son más bien recordatorios de que se debe mantener una conversación. Agregarles criterio de aceptación a estos requisitos de usuario, los acercan un poco más a la definición clásica de requisitos de software, o sea que a través de ellos empezamos a detallar un poco más cuál es el comportamiento esperado, qué es lo que pasa en distintos escenarios de utilización, en distintos escenarios o variantes de esa historia de usuario.

[V / F] Los criterios de aceptación ayudan a clarificar el alcance de la historia de usuario y permiten derivar casos de prueba.

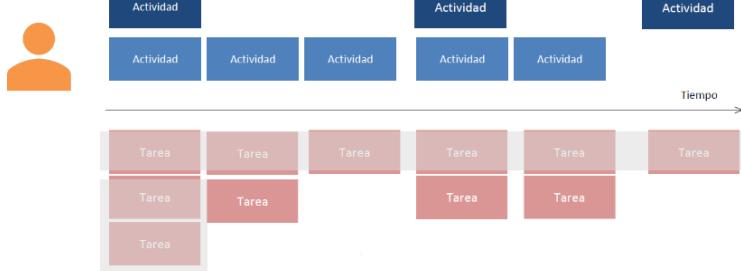
## Reglas de negocio

También podemos utilizar reglas de negocio. Todas las organizaciones están sujetas a regulaciones de algún tipo, a leyes, estándares, a políticas. Estos principios de operación y de control son colectivamente conocidos como reglas de negocio. Por ejemplo, el IVA es del 21%, las facturas se deben emitir el último día hábil del mes o los usuarios menores de 18 años deben ser autorizados por sus padres, tutores o encargados. Las reglas de negocio pueden describir entonces cálculos complejos, restricciones, inferencias o simplemente hechos.

Hay 2 técnicas que vamos a mencionar y que vamos a explicar en detalle más adelante:

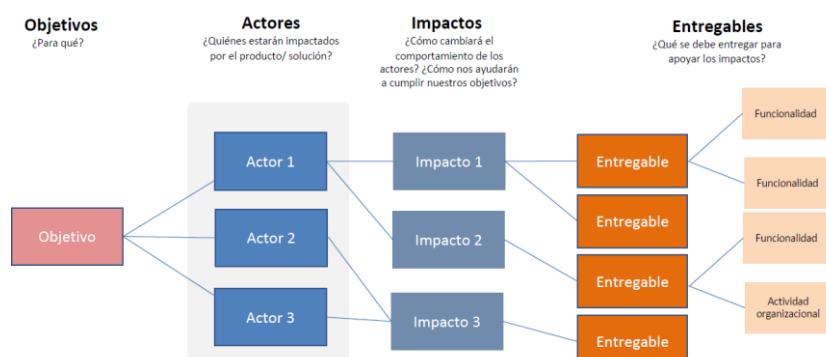
### User Story Mapping

Es un modelo que describe las actividades que realizan a lo largo del tiempo los usuarios de una aplicación y cómo esas actividades se descomponen en subactividades y tareas. A partir de las tareas de último nivel podemos identificar historias de usuario. El diagrama se lee de izquierda a derecha.



### Impact mapping

Mapa de impacto o impact map, es un diagrama que tiene como propósito alinear los equipos de trabajo con los objetivos de la organización. Este diagrama también se lee de izquierda a derecha. Primero se identifican los objetivos que se buscan satisfacer con la solución, o producto desarrollar. Luego, se identifican los actores que serán impactados por la solución. Y finalmente, qué es lo que se deberá hacer para producir esos impactos en esos actores, es decir, qué entregables tenemos que producir, qué productos, qué servicios, qué funcionalidades. Si seguimos descomponiendo esa estructura, en el último nivel vamos a encontrar realmente, historias de usuario. Es una herramienta muy útil para identificar requisitos y, obviamente, para alinear esos requisitos con los objetivos que se persiguen.



## Recomendaciones

Para ir finalizando, van algunas recomendaciones:

### Generales

- ★ Es importantísimo **definir cuáles son los límites** del sistema que estamos analizando, si no lo hacemos, vamos a estar profundizando en temas que a lo mejor no nos corresponde hacer.
- ★ Hay que estar **preparado para gestionar** los inevitables conflictos que van a surgir. No todo el mundo tendrá la misma visión, no todos los requisitos serán consistentes desde el comienzo.
- ★ Tampoco todos los requisitos serán igualmente importantes, los recursos y el tiempo son limitados, con lo cual es imprescindible que **clasifiquemos y prioricemos**.
- ★ Ya lo hemos mencionado antes al hablar de las actividades de descubrimiento, es importante **mantener la trazabilidad vertical y horizontal**. Necesitamos saber de dónde vienen los requisitos y también necesitamos saber qué artefactos están relacionados con ellos. Por ejemplo, si hay un cambio en un requisito determinado, qué otros requisitos se podrían ver afectados, qué impacto tiene ese cambio en los criterios de aceptación, en los casos de prueba, en las estimaciones, en los planes; qué cambios hay que hacer en el modelo de dominio; cómo afecta el cambio al diseño y al código. Si nos organizamos bien de entrada, las respuestas a todas estas preguntas deberían ser sencillas de elaborar, en caso contrario, estaremos desperdiando mucho tiempo (que por otro lado es valioso) y que deberíamos distraer de otras actividades.
- ★ También mencionamos con anterioridad la importancia de **evaluar riesgos**. Aquellos requisitos que impliquen un nivel de riesgo por encima de determinado umbral, ya sea por su complejidad o por lo difícil de trasladar al código o por cualquier otra razón, merecen tener una prioridad más alta, y probablemente, merezcan ser analizados con mayor nivel de detalle, probablemente a través de algún prototipo.

### Modelado

- ★ Tenemos que definir **cuáles son los modelos que nos conviene** desarrollar de acuerdo al escenario en el que estemos. No todos los modelos son útiles en todas las situaciones. También es importante que los modelos guarden consistencia entre sí. Por ejemplo, los objetos de dominio deberían estar mencionados en algún caso de uso, y los objetos de dominio que mencionemos en los casos de uso o en las historia de usuarios deberían estar incluidos en el modelo de dominio.
- ★ También es importante decidir **qué modelos serán necesario formalizar**. Una gran parte del modelado será informal, como propone Agile Modeling, pero probablemente algunos modelos merezcan tener una versión en algún medio electrónico, que, por otra parte, nos va a permitir compartir.
- ★ Al modelar, tenemos que procurar hacerlo desde **varios puntos de vista**. Para el aspecto objeto de dominio, o datos tenemos el modelo de dominio; para el aspecto funcional, podemos utilizar el modelo del caso de uso o User Story Map; para el aspecto control, que es muy importante en sistemas de tiempo real, podemos emplear diagramas de estado.
- ★ Y, por supuesto, el **prototipado** es una técnica muy adecuada para, como ya dijimos, explorar requisitos poco claros o riesgosos.

### Especificación

- ★ Es muy importante **determinar el nivel de formalidad**. Si el equipo de desarrollo es relativamente pequeño y se encuentra físicamente en el mismo lugar, probablemente alcance con historias de usuario y con criterios de aceptación.
- ★ Ahora bien, si el equipo es más grande o hay dispersión geográfica o hay una relación contractual entre la organización que produce el software y la organización que lo está solicitando, probablemente sea necesario **desarrollar estándares para una especificación** un poco más formal.
- ★ En todos los casos es importante **utilizar un lenguaje claro, consistente y conciso**. Esto es independiente de la técnica que utilicemos. Siempre hay que recordar *que un requisito se escribe una vez pero se lee muchas veces*.
- ★ Por eso es muy importante empezar los requisitos en forma **cuantitativa** (siempre que se pueda).
- ★ Utilizar un **glosario o diccionario de datos** para aclarar palabras o expresiones particulares del dominio del problema.

## Conclusiones

### Para recordar

Recordemos entonces que, **análisis y especificación usualmente se realizan en simultáneo**. Recordemos también que analizar no consiste solamente en desarrollar modelos. El objetivo es **encontrar conflictos, encontrar inconsistencias** y resolverlos. Ya hemos también mencionado la importancia que tiene que definir cómo se van a **organizar y gestionar los requisitos**: ¿Los vamos a poner en un repositorio compartido?, ¿los vamos a pegar en un pizarrón?, ¿los vamos a pegar en una pared? Nuevamente no hay que perder de vista que los modelos ayudan a entender el problema y a definir el sistema deseado pero, hay que analizar **seriamente cuáles son los modelos que nos conviene construir**, para qué, con qué grado de formalidad. En algún entorno muy, muy específico, los modelos de análisis, a través de sucesivas transformaciones, pueden transformarse en modelos de diseño y luego, en código fuente. Esto es lo que pasa en Model Driven Design, por supuesto, no es la generalidad de los casos, por eso es importante determinar qué nos va a convenir desarrollar y qué modelo no nos va a convenir desarrollar. Lo mismo ocurre con las especificaciones. Tenemos que analizar cuidadosamente qué tipo de especificaciones producir, para quienes, para qué.

Y finalmente, lo que tenemos que evitar en todos los casos es la parálisis del análisis. Esa tendencia a sobre analizar y a no poder avanzar. En algún momento la actividad de análisis tiene que darse por concluida para poder seguir con el desarrollo.

[V / F] Generalmente los casos de uso y las historias de usuario se van identificando y refinando progresivamente a medida que avanza el proyecto.

## 4.3 - Modelado de dominio

### Introducción

#### Modelos

Ya hemos visto la importancia que tiene el modelado en las actividades de ingeniería de requerimientos, hemos visto que un modelo es una simplificación de la realidad. El modelo no es la realidad, sino que nos da una perspectiva de lo que queremos construir o de lo que hemos construido. Los sistemas que nosotros desarrollamos en general son complejos y necesitamos utilizar los modelos para poder visualizarlos, para poder especificarlos, para guiar la construcción, para de alguna manera documentar las decisiones que tomamos.

#### Modelado del dominio

Vimos que hay distintos tipos de modelos, algunos más relacionados con el análisis de los requerimientos, con el modelado del problema, otros más orientados a un modelado de la solución, al diseño. En particular estamos hablando hoy del modelo de dominio. Un **modelo de dominio** es una representación visual del vocabulario del problema, del vocabulario del dominio del problema. Los objetos del modelo de dominio tienen que ver con ese vocabulario que emplea el usuario para contarnos cuál es el problema que tiene, para describirnos como es su operatoria, para describirnos qué es lo que hace. Si estamos pensando en un sistema bancario, probablemente los objetos de dominio o clases conceptuales más importantes sean el cliente, la cuenta corriente, la caja de ahorro, el movimiento, el saldo. Si pensamos en función del sistema de inscripciones de una facultad, los objetos de dominio más importantes van a ser las materias, los cursos, los alumnos, la inscripción. Construimos este tipo de modelos para entender mejor, y analizar mejor, el contexto en el que va a operar el sistema que tenemos que desarrollar.

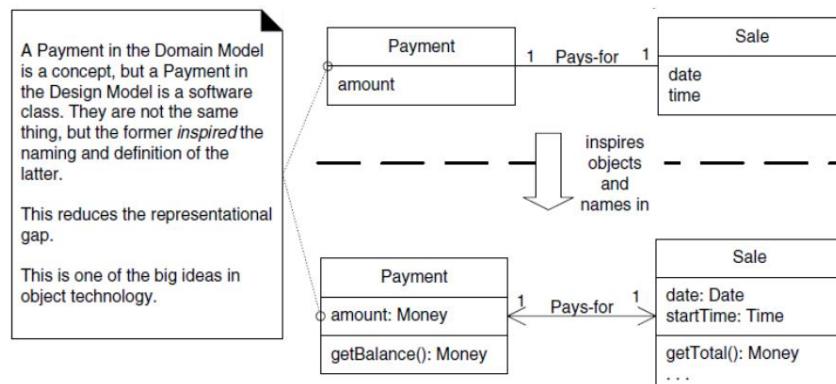
En general, vamos a encontrarnos con que los objetos de dominio pertenecen a alguna de las siguientes grandes familias:

- **Cosas** manipuladas en una organización: Qué maneja la organización. Por ejemplo: contratos, facturas, pedidos.
- **Objetos** y conceptos del mundo real que el sistema necesita conocer y/o monitorear. Por ejemplo: un avión, una trayectoria, un misil.
- **Eventos** pasados o futuros. Por ejemplo: Un pago de una factura, la programación de una función de cine.
- **Personas o roles** de esas personas en organización o en el mundo del dominio, **u organizaciones**, por ejemplo, el cliente, el socio, el alumno.

[V / F] Un modelo de dominio es una representación visual del vocabulario del dominio del problema.

Lo que hay que tomar en cuenta es que estos objetos de dominio no son los objetos de diseño. Eso es muy importante. Nosotros podríamos decir que un cliente se caracteriza por tener un CUIT, un nombre, una dirección y no va a faltar que un día, bueno, pero un nombre es de clase string, dirección también... esas son clases que tiene que ver más con la implementación. Cuando

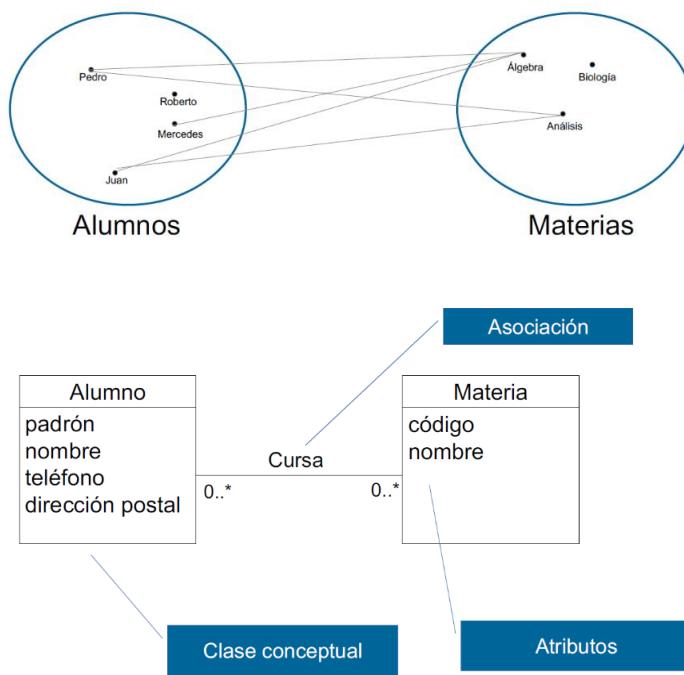
hablamos de objetos de diseño estamos hablando de los grandes elementos que forman parte del vocabulario: El cliente, la factura, el ítem de la factura, la transacción, la operación, no son objetos de diseño. Por supuesto que uno de los conceptos fundamentales de la orientación a objetos, es minimizar la brecha entre el vocabulario del problema y la solución, pero, a no confundirse, **no todo objeto de diseño existe en el dominio del problema**. Hay que tener mucho, mucho cuidado, con esto.



## Modelado del dominio: objetos, asociaciones y atribuciones

El modelo de dominio está profundamente basado en teoría de conjuntos.

Aquí tenemos un conjunto de alumnos, un conjunto de materias, y vemos que hay instancias en ambos conjuntos. Que a su vez se asocian unos con otros, por ejemplo, Pedro está cursando álgebra y análisis; Roberto no está cursando ninguna materia. Podríamos presentar esto a través de un diagrama que utiliza notación UML, y vamos a ver que tenemos una clase alumno, una clase materia que están representando objetos del dominio. Vemos que esas 2 clases conceptuales se relacionan entre sí mediante una asociación, en este caso alumno cursa materia (la asociación es cursa), y vemos también que cada uno de estos objetos, cada una de estas clases, tienen atributos. Esos atributos son propiedades que caracterizan a cada uno de esos objetos. Esto va a formar la base, como decímos antes, del diseño. Seguramente en nuestro diseño vamos a tener un objeto alumno (vamos a tener una clase alumno para ser más preciso), una clase de materia, pero fíjense que acá no estamos hablando de aspectos de implementación, no estamos hablando de métodos ni de operación, estamos hablando de clases conceptuales, objetos del dominio y propiedades o atributos.



[V / F] Todo objeto del diseño existe también como entidad del modelo de dominio del problema.

## Modelo del dominio

### ¿Cómo se construye?

¿Cómo hacemos para construir un modelo de dominio? Bueno, no hay una receta, pero en general podríamos decir que el primer paso es identificar las clases conceptuales utilizando un par de técnicas que vamos a mencionar: Se puede dibujar, hacer un diagrama, utilizando la notación de UML con la salvedad de que no son objetos de diseño, no son objetos de implementación, sino que son abstracciones. Podemos agregar detalles tales como asociaciones, atributos, generalizaciones y especializaciones, composiciones, agregaciones. En general este es un proceso iterativo incremental. Puede ser el modelo de dominio algo muy informal, dibujado en una pizarra; puede ser algo más formal dibujado a través de alguna herramienta. Lo importante es, no tanto es el resultado, sino pensar y entender bien cuáles son los grandes elementos, cuáles son los grandes objetos que forman parte o los objetos más relevantes más que grandes, los objetos relevantes del dominio y cómo se asocian entre sí. ¿Por qué? Porque esa es la información que el sistema va a tener que manejar independientemente de cómo lo implementemos. El sistema va a tener que recordar que el alumno Juan Pérez se anotó para cursar la materia de análisis matemático. El sistema va a tener que recordar que Juan aprobó análisis matemático, en tal fecha con tal nota, entonces eso lo tenemos que de alguna manera, representar en el modelo de dominio.

### ¿Cómo encontrar clases conceptuales?

Hay 3 estrategias básicas para encontrar clases conceptuales: Una es aplicar el análisis lingüístico, la otra es utilizar algún tipo de listas tramada de categoría y otra es reutilizar algún modelo ya existente.

## Análisis lingüístico

"Necesito que el nuevo sistema permita generar las **facturas** -a partir de los **pedidos** de los **clientes**- más rápidamente que el sistema actual. También necesito que el inventario de cada **producto** se mantenga actualizado en forma permanente."

Necesidades, expectativas.

Generación de **facturas**: las facturas deben ser generadas a partir de los **pedidos** de los **clientes** previamente ingresados. Una factura podrá contener **ítems** incluidos en diferentes **pedidos**. Actualización de **inventario**: el inventario debe estar actualizado permanentemente...

Requerimientos del usuario (necesidades y expectativas "pasadas en limpio").  
Conflictos e inconsistencias identificados y resueltos. Prioridades.

CU 10.1: Ingresar el **pedido** de un cliente

...

CU 10.2 Facturar **pedidos** de **clientes**

...

CU 10.3 Actualizar **clientes**

...

CU 10.4 Conciliar inventario de **productos**

...

Requerimientos de la solución, derivados y analizados. Prioridades (*triage*)

El análisis lingüístico es bastante simple de explicar. Buscamos en nuestras minutas de reunión, en nuestras especificaciones, en nuestros casos de uso, buscamos sustantivos o frases que incluyan sustantivos para identificar candidatos a posibles objetos. Por ejemplo, acá tenemos facturas, pedidos, clientes, productos. Hay que tener cuidado porque también van a aparecer atributos o propiedades de otros objetos (que también son sustantivos), con lo cual hay que andar con un poquito de cuidado.

Lo que hay que tener, sí, mucho, mucho cuidado, es con aplicar este enfoque de manera mecánica. Si todos los sustantivos que están en la especificación van a ser objetos, seguramente nos vamos a encontrar con que va a haber un objeto que es dirección del cliente, o dirección, o código postal, que son sustantivos pero no son objetos, en este caso. Con lo cual, ahí hay que tener cuidado para justamente evitar esta tentación de darle un grado de granularidad muy, muy, alto y caer en la tentación de meter temas que tienen que ver más con la implementación y no con el aspecto más conceptual de lo que estamos tratando de definir. Con lo cual es conveniente aplicar esta técnica combinada con la siguiente que es la lista de categorías.

### Lista de categorías

Categoría	Ejemplos
Transacciones	Venta, reservación, pago.
Ítems de una transacción	Ítem vendido, ítem facturado
Producto o servicio relacionado con una transacción o con un ítem de una transacción	Artículo, vuelo, butaca.
Dónde se registra la transacción	Factura, asiento contable, manifiesto de carga.
Roles u organizaciones relacionadas con una transacción	Cajero, sucursal, cliente.
Lugar de la transacción o de la provisión del servicio.	Sucursal, aeropuerto, vuelo.
Eventos	Venta, pago.
Objetos físicos	Avión, automóvil, cronómetro.

Descripción de cosas	Especificación de producto, descripción del vuelo.
Catálogos	Catálogo de productos, lista de precios.
Contendedores de cosas	Depósito, almacén.
Cosas en el contenedor	Ítems, pasajeros.
Sistemas externos que colaboran	Sistema de autorización, control de tráfico.
Registros financieros o de trabajos realizados, contratos, etc.	Recibo, bitácora de cambios.
Instrumentos financieros	Efectivo, tarjeta de crédito.
Cronogramas, manuales, etc.	Calendario de reparación, cronograma de proyecto.

En la lista de categorías, aquí lo que tenemos son algunas categorías de tipos de objetos que nos pueden orientar en la búsqueda de objetos. Por ejemplo, transacciones, ítems de una transacción, productos o servicios, los lugares en donde se registran las transacciones. Pensemos, por ejemplo, cuando vamos a un comercio a comprar algún tipo de artículo y nos emiten una factura. Pensemos ahí, cuáles son los objetos que están relacionados: Estoy yo como cliente, está la empresa probablemente como proveedora (tema que debemos discutir después), está el objeto que esté comprando (el producto que estoy comprando). El producto tendrá sus características: tendrá un precio, tendrá una descripción. También está la transacción en sí misma que queda reflejada a través de una factura, por ejemplo, que va a tener una serie de atributos que tendremos que analizar. O, por ejemplo, hacemos una reserva para ir a ver una película. Fíjense los objetos de dominio que podrían aparecer ahí: Nosotros como clientes, la película que vamos a ver, en qué sala la vamos a ver, la transacción propiamente de compra que nos va a indicar: compraste una entrada para tal película para tal fecha en tal asiento, etc. Así que aquí tenemos una lista de categorías también, por ejemplo, roles, lugares en donde se provee servicio o tiene lugar la transacción; objetos físicos, a veces necesitamos tener información de objetos físicos. Por ejemplo, en algunos casos es importante que identifiquemos cada uno de los productos que hemos fabricado, por ejemplo, tal auto con tal número de serie, con tal número de chasis.

[V / F] Al momento de identificar las clases conceptuales utilizando el análisis lingüístico generalmente todos los sustantivos se representan como objetos.

## Atributos

Habíamos dicho que las clases conceptuales o modelos del dominio tienen atributos. Un atributo es un dato lógico acerca de un objeto. Por ejemplo, acerca de un cliente, nos interesaría conocer su clave única de identificación tributaria o CUIT, su razón social, su dirección, su teléfono. En este caso teléfono, es un atributo repetitivo, es decir, tenemos varias instancias de ese atributo. Lo que no tenemos que caer nuevamente en la tentación, es decir que el CUIT es una clase en sí misma, la razón social es una clase que pertenece a tipo string, etc. Estamos hablando de vuelta de, en este caso, datos primitivos. Si es que nos interesa a esta altura decir que algo es numérico o no. Quizá en muchos casos va a valer la pena y en otros mejor no conviene hacer esa profundización.

En el comienzo del análisis orientado a objetos, cada instancia de una clase conceptual, existía por el medio hecho de existir, y no necesitaba ningún tipo de identificador para diferenciar una instancia de la otra porque se asumía que cada instancia de la otra se podía autoidentificar de alguna forma que no era relevante para el análisis del problema. Pero de todas maneras, hay casos en donde hay identificadores naturales, por ejemplo, la clave única de identificación tributaria o CUIT es un identificador natural. Entonces, podemos poner un estereotipo como el que figura ahí entre llaves y “*id*” para indicar que es un identificador natural. El DNI es un identificador natural, el número de padrón es un identificador natural, con lo cual conviene indicarlo ahí. Ya en tiempo de implementación, cuando transformamos esto, por ejemplo, en una tabla de clientes, la clave primaria será el CUIT o el

Cliente
CUIT {id}
RazónSocial
Dirección
Teléfono: [1..*]
En el ejemplo, un cliente puede tener uno o varios teléfonos.
Teléfono es un atributo que puede tener múltiples instancias.

DNI, o el número de padrón, pero eso es algo de implementación, no es un problema que nos preocupa acá, pero podemos, de todas maneras, si ir identificando que hay determinados atributos que nos permiten individualizar cada una de las instancias de un determinado objeto de valor.

También es importante que algunas veces mostremos atributos que son derivados a partir de otros. En el ejemplo, TotalVenta es un atributo derivado que resulta de la suma entre TotalNeto y TotalImpuestos. Esto lo podemos poner siempre y cuando sea relevante para entender el dominio. No tenemos que poner atributos ahí que tengan que ver más con un tema de implementación. Digamos, si yo pongo TotalVenta porque digo “no porque desde un punto de vista de implementación va a ser más rápido, en lugar de tener que hacer el cálculo”, o puedo decir lo contrario “no, no pongo TotalVenta que es un atributo significativo desde el punto de vista del dominio porque me va a ocupar mucho espacio en la base de datos”, no, es ridículo ese planteo porque acá de lo que estamos hablando del dominio del problema. Con lo cual restricciones de implementación acá no aplica. Estamos en un nivel de abstracción muy alto como para ponernos a discutir de esos temas.

Venta
NúmeroVenta {id}
FechaVenta
TotalNeto
TotalImpuestos
/TotalVenta

[V / F] Un atributo referenciado como “Teléfono: [1..\*]” indica que existe una relación con la entidad Teléfono de entre 1 y muchos.

## Asociaciones

El otro tema sumamente importante son las asociaciones. Las asociaciones representan una relación entre instancias. Al igual que los atributos, que tienen que ser recordados por el sistema, las asociaciones también tienen que ser recordadas por el sistema. Cada asociación tiene un nombre y tiene una multiplicidad. En el ejemplo, un alumno puede estar anotado en cero o en muchas materias, y en una materia puede haber anotados muchos o ningún alumno. Por ejemplo, si la materia es nueva, todavía no va a tener alumnos inscriptos. Lo que hay que tomar en cuenta es que la multiplicidad nos está indicando una regla de negocio. En este caso, la regla de negocios es que un alumno puede cursar cero, una o muchas materias y que puede haber casos en donde las materias no tengan alumnos porque todavía no se están dictando o porque todavía no es el momento de inscribirse o porque no se anotó nadie para cursarla. Hay que tener cuidado, una cosa es una asociación y otra cosa es un atributo. No tenemos que usar, reitero, **no tenemos que usar atributos que referencien a otros objetos**. Yo podría decir ahí que, por ejemplo, ¿podría tener algún tipo de puntero desde materia a los alumnos inscriptos y que eso fuera un atributo que estuviera incluido en materia? No. Eso es un tema de implementación. ¿Ok? Sí, cuando tengamos que implementar esto en una base de datos, probablemente tengamos una tabla de alumnos, una tabla de materias y tengamos una tabla de correlación que implemente la relación entre alumnos y materia, pero, ojo con eso, acá estamos hablando de asociaciones que son conceptuales que no implican ningún tipo de atributo referencial. No tenemos que poner atributos que referencien a otros objetos, porque en realidad, para eso, para mostrar eso, tenemos las asociaciones.

### Asociaciones binarias

Las **asociaciones** típicas son **binarias** entre 2 objetos. Puede haber más de una asociación entre un par de objetos. Tranquilamente en este caso, un alumno puede cursar una materia y puede tener aprobadas otras materias. En viaje siempre tenemos una ciudad de destino y una ciudad de llegada. Eso es perfectamente normal. Recuerden que las asociaciones relacionan distancias de las clases conceptuales que forman parte de la relación.

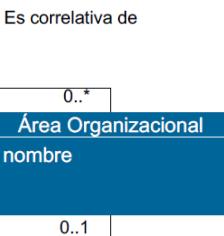
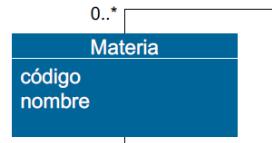
Alumno	Cursa ▶	Materia
padrón	0..*	0..*
nombre		
teléfono	Tiene aprobadas ▶	
dirección postal	0..*	0..*

Viaje	Sale de ▶	Ciudad
DiáHoraSalida	0..*	1
DiáHoraLlegada	Llega a ▶	
	0..*	1

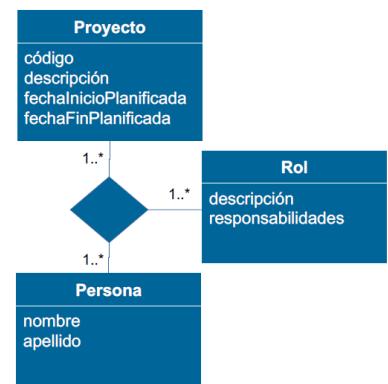
## Asociaciones unarias

Típicamente las asociaciones son binarias, pero también puede haber **asociaciones unarias**. Por ejemplo, una materia puede tener como correlativas otras materias y una materia es correlativa de otra. Lo que hay que tener en cuenta ahí, en este tipo de relaciones jerárquicas, es que no todas las materias tienen materias que les sigan, ni todas las materias tienen materias anteriores (las materias del CBC no tienen materias anteriores). Para cursar introducción al conocimiento científico probablemente no necesiten tener una materia aprobada desde antes. Siempre que uno representa organizaciones o estructuras jerárquicas hay que tener cuidado con los nodos y las hojas. Por ejemplo, área organizacional puede tener, ahí lo que representa, imagínese un organigrama. Entonces, la raíz del organigrama: del gerente general dependen las gerencias; de las gerencias depende las jefaturas; de las jefaturas dependen los empleados. Ahora hay empleados que no tienen ningún tipo de estructura jerárquica por debajo, con lo cual ahí, el árbol se termina, con lo cual hay que tener cuidado con la cardinalidad, por eso está el cero a muchos, el cero a uno. Hay que mirarlo con detalle.



## Asociaciones ternarias

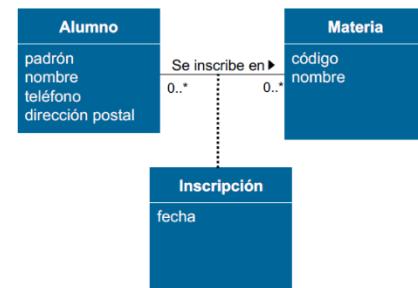
Si bien son extrañas, pueden aparecer **asociaciones** de grado mayor a 2. Las más comunes son las **ternarias** que puede haber de grado 4, 5, pero ya son más raras de ver. En este ejemplo en particular, lo que estamos viendo es que una persona puede tener más de un rol en un proyecto determinado. Eso es lo que está representando esa asociación. Recuerden que en este caso, cada instancia de la asociación está representando una relación entre persona, rol y proyecto.



En el ejemplo, una persona puede tener más de un rol en un proyecto determinado.

## Clases asociativas

En algunos casos nos vamos a encontrar con unas asociaciones medio extrañas que nos vamos a dar cuenta que, además, tienen atributos que son propios. En ese caso no vamos a estar ante una asociación, sino que vamos a estar ante lo que se llama un **tipo de objeto asociativo o clase asociativa**. La clase asociativo tiene la particularidad de *comportarse como una clase y como una asociación al mismo tiempo*. Entonces, en este ejemplo, lo que vemos es que un alumno se inscribe en una materia. ¿Cuándo se inscribe? En una fecha determinada. Entonces ahí lo que necesitamos es un objeto asociativo “inscripción” con el dato “fecha”. No podemos tener fecha como un atributo de una asociación porque eso, conceptualmente, no es viable.



## Composición y agregación

Hay un tipo particular de asociaciones que son las que permiten representar las relaciones de todo y parte. Por ejemplo, un automóvil tiene ruedas. En este ejemplo, tiene cuatro. Pero las ruedas son también elementos que son independientes (las puedo sacar).

En cambio, en un proyecto, yo tengo varias tareas; pero esas tareas, si las saco del proyecto, no existen, no tiene existencia por la independencia. Entonces, lo que propone UML es este tipo de asociaciones que llaman **composición y agregación**. Cuando tenemos una colección de elementos que no depende del ciclo de vida del contenedor (el caso del automóvil y la rueda), estamos ante una agregación y se representa con ese diamante sin rellenar. Ahora, cuando los elementos de la



colección dependen del contenedor (tarea con respecto a proyecto), estamos ante una composición. ¿Cuándo es conveniente utilizar este tipo de representación o de asociación? ¿Cuándo es conveniente utilizar una asociación común? Porque esto mismo que estamos viendo acá, se podría representar utilizando una asociación de las que vimos antes. La verdad es que el criterio que debería primar es el de la claridad en la representación. Digamos, si todos entienden de lo que estamos hablando, probablemente convenga utilizar este tipo de estructura.

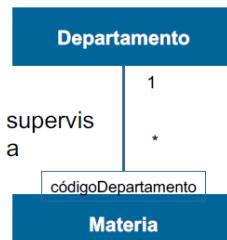


¿Cuáles de los siguientes son buenos nombres de asociaciones?

- Registra**
- Cursa**
- Tiene
- Utiliza
- Es correlativa de**
- Escribe**

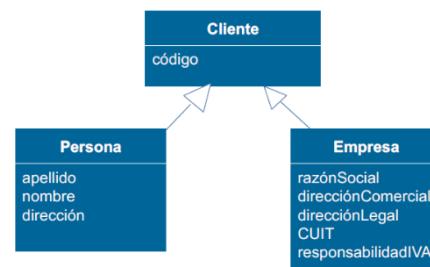
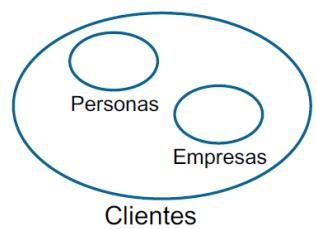
## Asociaciones calificadas

Hay un tipo de **asociaciones** que se llaman **calificadas**. Supongamos por un momento que el departamento de computación aglutinara a todas sus materias en “75algo” y que el departamento de Computación sea el departamento 75. Entonces está la materia 7501, la materia 7509 (como la nuestra), etc. Entonces la materia queda identificada a partir de su propio código (09) más el código del departamento (75). Entonces, en este caso, hablamos de una asociación calificada porque *distinguimos un grupo de distancias en uno de los extremos mediante un valor calificador*, que en este caso es el código del departamento. Entonces, si hicieramos doble click en “materia”, nos encontraríamos que materia tiene como atributos código de materia y el nombre. Pero para poder identificarse necesita del código del departamento al que pertenece, porque, por ejemplo, podría haber una materia 09 en otro departamento. ¿Ok? Esto es algo muy, muy importante y muy útil para el modelado.



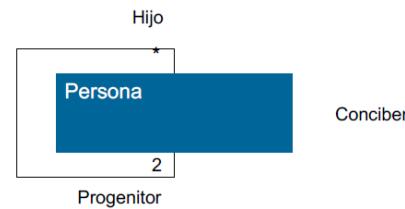
## Generalización y especialización

También hay situaciones en donde vamos a necesitar hacer algo que es clásico de la orientación a objetos, que es establecer **generalizaciones y especializaciones**. En el ejemplo estamos viendo que para nosotros es importante el cliente, pero tenemos dos tipos de clientes, que tienen atributos propios. Por ejemplo, tenemos clientes que son personas físicas, de las cuales nos interesa su apellido, su nombre, su dirección. En cambio, hay otros clientes que son empresas, y de esos en particular, lo que nos interesa son la razón social, su dirección comercial, su dirección legal, su CUIT. Entonces probablemente necesitemos modelar esto como una clase padre, que tiene un atributo que si nos interesa, que es el código del cliente, y 2 clases hijas que heredan parte de la definición de cliente pero tienen sus propias particularidades. ¿Cuándo hacemos esto? Cuando realmente nos interesa mostrarlo porque necesitamos mostrar atributos que son diferentes, porque hay asociaciones distintas entre los hijos (por ejemplo: personas se puede asociar con determinado tipo de objetos y empresa con otros). De vuelta, el criterio que tiene que considerarse aquí es claridad en la representación, claridad en la comunicación es lo más importante.



## Roles en una asociación

También puede haber **roles en una relación**. Eso *ayuda a clarificar el papel que juega cada uno de los objetos en la relación*. En este caso, vemos que una persona es concebida como resultado de la interacción de 2 personas. En un extremo tenemos el rol de una persona que juega el rol de hijo, y en el otro extremo, la persona juega el rol de progenitor (como nos pasa a todos). No es obligatorio incluir el tema de los roles, pero sí ayuda a mejorar la relación entre objetos.



## ¿Cómo encontrar asociaciones?

Nuevamente para encontrar asociaciones podemos utilizar categorías:

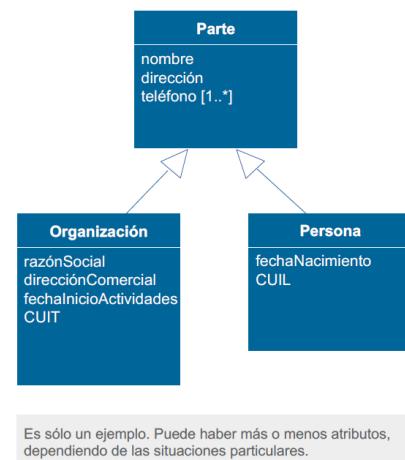
Categoría	Ejemplos
Una transacción relacionada con otra.	Venta (o factura) - Pago.
Un ítem de una transacción.	Ítem factura - Factura.
Un producto o servicio incluido en una transacción (o en un ítem)	Vuelo – Reserva; Compra – Ítem Comprado
Un elemento forma parte de otro (física o lógicamente)	Sala – Butaca Computadora – Procesador
Roles en una transacción.	Cliente (en una compra o factura) Comercio (Cupón Tarjeta Crédito)
Descripción de algo.	Especificación (de un producto) Descripción de vuelo.
Registración, conocimiento.	Reserva (de una habitación)
Algo que es miembro de otra cosa.	Cajas (en un supermercado). Socios (de un club). Empleados (de una organización)
Una subunidad organizacional.	Departamento (dentro de una gerencia)
Algo administrado o usado.	Chofer (conduce un micro) Complejo de cines (administra salas)

## Patrones

Vamos a revisar rápidamente algunos patrones que habitualmente nos vamos a encontrar. Patrones que tienen que ver con el modelado de dominio. Vamos a repasar algunos de estos:

### Personas y organizaciones

Algo que es típico es el de personas y organización. Medio parecido a lo que veíamos hace un minuto. Si agarramos la agenda de nuestro teléfono, vamos a encontrar con que hay personas pero también hay organizaciones. Entonces... ¿cómo hacemos? ¿No hay atributos comunes entre ellos? ¿No hay atributos particulares para la persona y atributos para la organización? Entonces algunos autores proponen un patrón de modelado que lo llaman “parte” (sea la traducción de party en el libro de Fowler). Sinónimos: Persona física/jurídica, tercero, etc. Fíjense que acá el objeto de dominio padre “parte” tiene nombre, dirección, teléfono, atributos que son comunes a una persona o a una organización. En particular, organización tiene razón social, dirección comercial, fecha de inicio de actividades, CUIT. Mientras que persona, tiene fecha de nacimiento, CUIL. Nuevamente, es sólo un ejemplo, puede haber más o menos atributos y a lo mejor puede pasar que no nos interese realmente modelar esto de esta manera porque no es relevante para el dominio.



<b>Empresa</b>	1
	*
<b>División</b>	1
	*
<b>Región</b>	1
	*
<b>Dirección</b>	1
	*
<b>Gerencia</b>	1
	*
<b>Jefatura</b>	1
	*

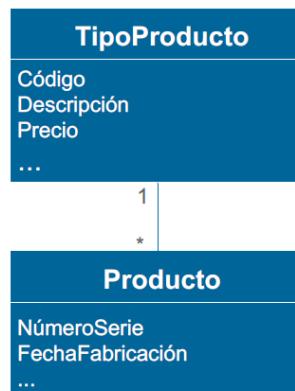
El tema de las **estructuras jerárquicas** también es un tema importante. Podríamos modelar la estructura de una empresa con este modelo de dominio que estamos viendo ahí (que es espantoso) o podríamos hacer algo un poco más inteligente, y utilizar este tipo de estructuras, en donde hay una unidad organizacional, que puede tener cero, una o muchas unidades organizacionales que las reporten. Y podríamos decir que cada unidad organizacional pertenece a un tipo de unidad organizacional determinada. Por ejemplo: departamento, gerencia, dirección. Nuevamente, fíjense que hay unidades organizacionales que son hojas en un árbol y no tienen unidades organizacionales hijas y hay nodos en el organigrama (la gerencia general) no tiene alguien que esté por arriba. Con lo cual, por eso es importante entender la cardinalidad de la asociación: 0...1, 0...\*(muchos), etcétera. La pregunta que les dejo para que analicen es: ¿soporta este modelo una estructura matricial?, ¿soporta este modelo posibles cambios a la estructura organizacional? Por ejemplo, que se fusionen las gerencias o que desaparezcan las gerencias y desaparezca un nivel en el organigrama.



## Productos y especificaciones

Producto
Código
Descripción
Precio
StockActual
...

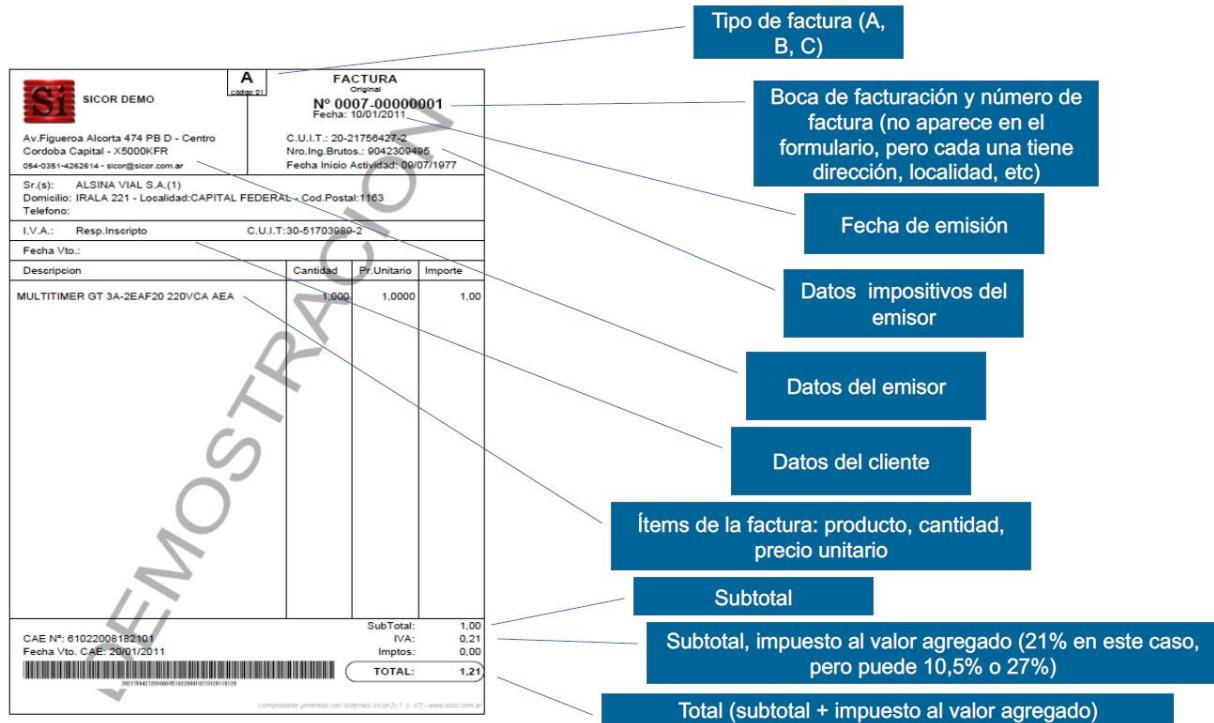
Producto	Una de las cosas que los sistemas, en general, siempre necesitan conocer es qué tipo de productos se fabrican en la organización, qué stock hay de esos productos. Y acá hay 2 alternativas importantes: A veces no nos interesa identificar cada unidad de los productos que fabricamos. Por ejemplo, si producimos frascos de mermelada probablemente nos interese saber que tenemos en stock 20 unidades de mermelada de naranja. A lo mejor no nos interesa obtener información de CADA frasco de mermelada de naranja. Si fabricamos autos, probablemente nos interese saber el número de serie de cada auto. Entonces acá, hay que analizar muy cuidadosamente si nos interesa tener información de cada producto fabricado en forma individual. Entonces, en esos casos, en los casos que sí nos interese manejar cada uno de los productos en forma individual, probablemente tengamos que tener un tipo de producto que nos describa las cosas comunes. Por ejemplo: Mermelada de naranja que se vende a \$ 60 el frasco; de las estancias de esos frascos de naranja, por ejemplo, hay un frasco de naranja, número de serie: 300.223 que está en tal negocio, pertenece a tal lote de producción, que se fabricó tal día y salió a tal hora de la fábrica. Depende nuevamente, del dominio del problema que estemos analizando.
Código	
Descripción	
Precio	
StockActual	
...	



## Facturas

La factura es un objeto que uno lo ve a simple vista y parece sencillo: Información de quién es el cliente, de quién es el que está emitiendo la factura, hay un número de factura, hay un detalle de lo que estoy comprando, etc. La factura es un documento que refleja la información de una operación de venta (es la que nos dan cuando vamos a cualquier comercio). Cada país tiene regulaciones muy particulares. En Argentina, ustedes han visto que las facturas pueden ser A, B o C dependiendo de distintas particularidades. No solamente hay que mantener la copia impresa, sino que también hay que mantener la información electrónica de esa factura. Actualmente se están imprimiendo lo que llamamos la factura electrónica, con lo cual uno genera un archivo PDF con la imagen.

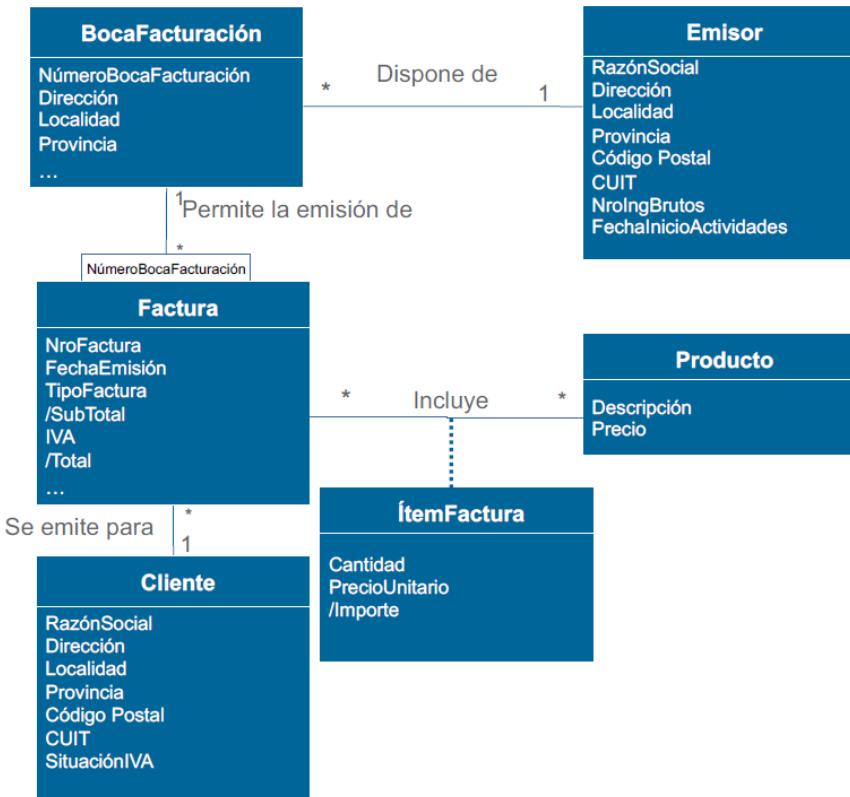
Ahora, qué información tenemos acá:



Vamos a encontrarnos con que hay un tipo de factura (A, B, C), que depende de si la factura que estoy emitiendo es para alguien que necesita que le discrimine el IVA o no. Después hay un número de factura que incluye la boca de facturación (que son esos primeros cuatro dígitos más un número correlativo). Por ejemplo: ¿qué es una boca de facturación? Y... yo puedo tener distintas sucursales. Entonces la sucursal número uno o la sucursal de Barracas tendrá la boca de facturación 0001 y emitirá facturas que van a empezar con el número 0001 y va a seguir después con 7 u 8 números secuenciales que tienen la información del número de factura propiamente dicho. O, en un mismo lugar, puedo tener más de una boca de facturación. Quiero decir, puede tener más de un "talonario" la factura. También hay información con respecto a la fecha de emisión de la factura, los datos impositivos de quien está emitiendo la factura (del vendedor), los datos del cliente (dirección, teléfono), la condición frente al IVA del cliente/destinatario de la factura. Después están los ítems de la factura, que es el detalle de los productos y servicios que estoy comprando, la cantidad, el precio unitario, el importe total. Al final vamos a tener un subtotal. Vamos a tener un subtotal con el IVA (que en algunos casos es el 21% sobre el precio neto o precio unitario, en algunos otros es el 10,5% en otros el 27%, eso varía). Y el total, que va a ser en definitiva el total neto (sin los supuestos) más el IVA. Esto es una recontra simplificación porque también podrían haber impuestos internos incluidos aquí, pero mantengamos el ejemplo sencillo. Entonces: ¿cómo hacemos?

Acá hay varias cosas: Si se fijan candidatos a objetos de dominio hay montones. Pensemos por un momento: Esta la factura propiamente dicha, pero la factura tal cual la tenemos representada acá, en realidad está asociada con otros objetos si yo tuviera que representar esto como un objeto de dominio: El cliente, el emisor, los ítems, etc.

Vamos a ensayar una posible representación de esto:

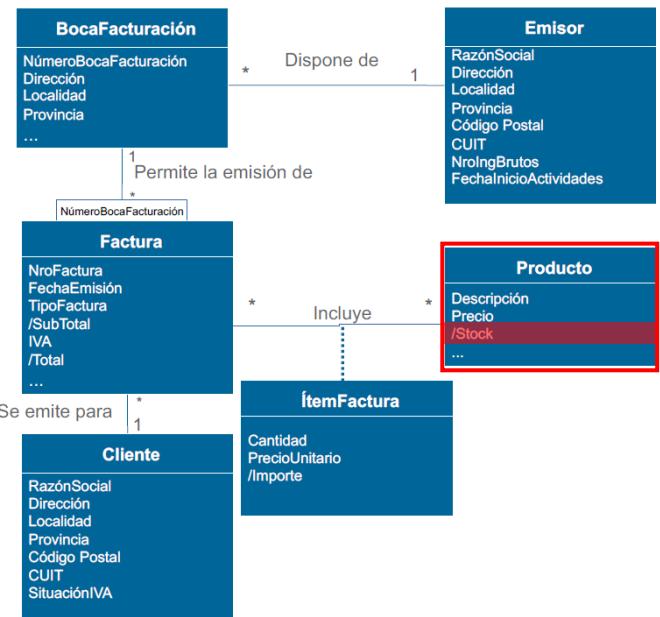


#### Ejemplo 1

Ahí tenemos un posible ejemplo. Tenemos el emisor que es el que emite la factura; dispone de una o varias (el \* representa: 0, 1 o varias. A veces UML nos propone que pongamos explícitamente si es cero o muchos) bocas de facturación. Esa boca de facturación tiene una dirección, una localidad, una provincia. Después tenemos la factura, que en realidad la factura lo que estamos incluyendo acá son los datos que son comunes a las facturas: el número, la fecha de emisión. Fíjense que factura tiene una relación ahí calificada con respecto a boca de facturación por el tema de la identificación que habíamos comentado antes. Pero también lo que vemos es que cada factura incluye varios ítems. Entonces, lo que estamos viendo acá en esta relación, en esta asociación “incluye” es este detalle, que es el que normalmente tenemos acá: “itemFactura”: Cantidad, precio unitario, importe. Pero cada ítem referencia a un producto y obviamente está asociado con la factura. La factura es un objeto compuesto por varios ítems de factura asociados a un emisor, a una boca de facturación, a un cliente. La pregunta que yo les haría acá es por qué hay un precio unitario en el ítem de factura. ¿Por qué, por ejemplo, ese precio también está en el producto? Bueno, la explicación es sencilla: porque los precios varían y varían en el tiempo. Una cosa es el precio que tiene hoy el producto y otra cosa es el precio al cual yo lo vendí. Con lo cual, a veces parece que hay redundancia de información, pero a veces no. No es redundancia, sino que tiene que ver con la definición misma del dominio del problema.

## Facturas: otros escenarios

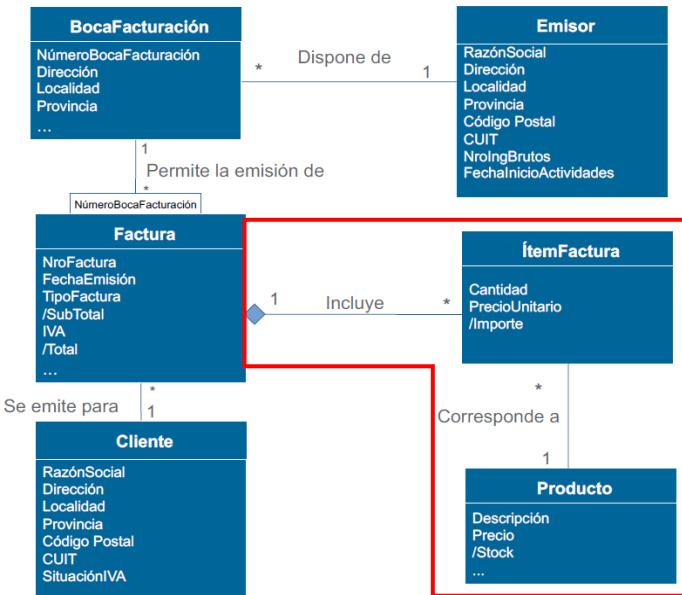
Preguntas: ¿Qué pasa si yo quiero repetir como ítem más de una vez el mismo producto? ¿Lo soporta el modelo? ¿Qué ocurriría si fuese necesario incluir el número de serie de un producto en particular? Acá (*Ejemplo 3*) me daría la sensación que producto ahí está representando un tipo de producto en realidad. La otra pregunta es si necesitamos siempre la clase cliente. A lo mejor podemos omitirla. ¿Harían falta las clases localidad y provincia? Posiblemente son atributos de cliente.



*Ejemplo 2*

## Facturas: (algunos) otros escenarios

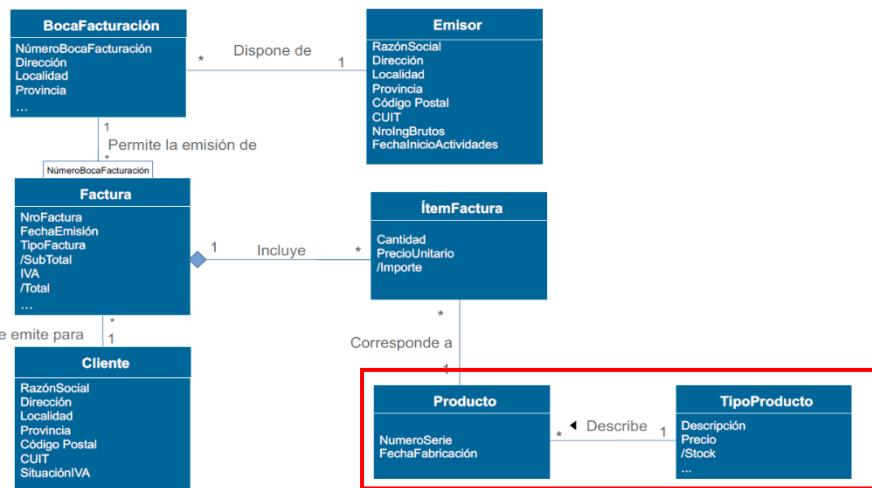
Vemos algunos otros escenarios:



*Ejemplo 3*

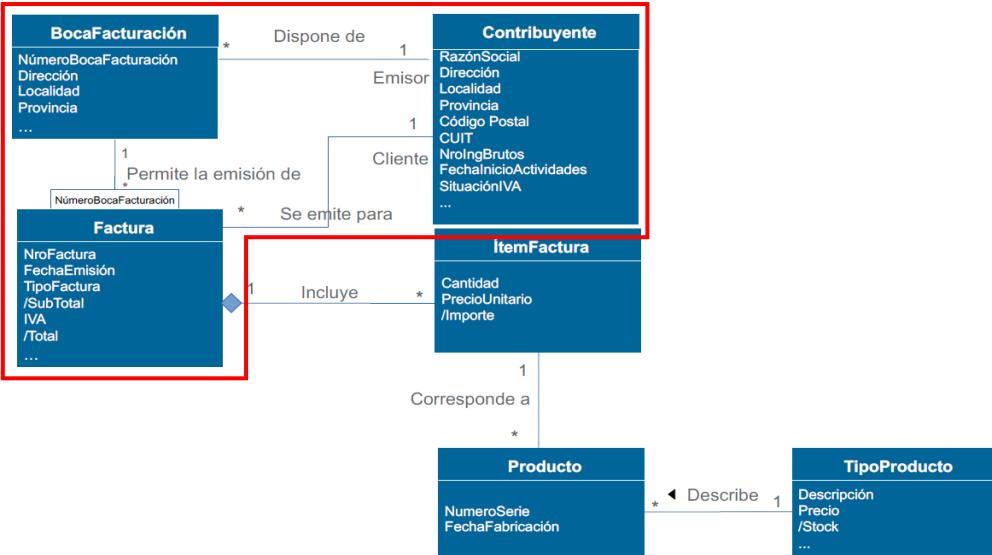
Acá (*Ejemplo 4*) modificamos el modelo para poder vender ítems o productos diferenciados, es decir, productos que necesitamos incluir el número de serie. Entonces, en este caso producto ya no representa una familia o un tipo de producto determinado, es el caso de que necesitamos vender en vez de la mermelada, el frasco de mermelada, número de serie 33.323

Acá (*Ejemplo 3*) lo que vemos es que factura incluye varios ítems de factura. Hay una relación de agregación. Y cada ítem de factura referencia a un producto. Con esto resolveríamos en parte, el tema de poder repetir el mismo tipo de producto más de una vez en la factura.



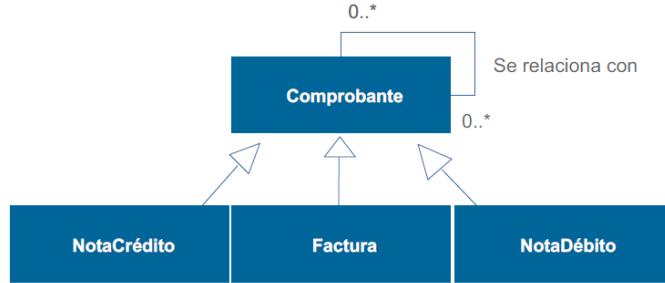
*Ejemplo 4*

La AFIP tiene un sitio web en el cual uno ingresa y genera facturas electrónicas. Y ahí, todos somos contribuyente. Nosotros (los que emitimos facturas) y los clientes, con lo cual, ya dejaría de existir el objeto cliente diferenciado del objeto emisor. Tendríamos un objeto contribuyente que puede cumplir el rol de emisor o de cliente dependiendo de la situación.



Ejemplo 5

Las facturas no están solas en la vida: hay notas de crédito, notas de débito. Las notas de crédito y notas débito son muy parecidas. Tienen un número, una boca de facturación, número de nota de crédito, de nota de débito. Pueden ser A, B o C. Representan cosas ligeramente diferentes: Las notas de crédito son saldos a favor del cliente, y las notas de débito son a favor del vendedor. Por ejemplo: Me devolviste un producto, yo te hago una nota de crédito por el monto del producto que me devolviste. Una nota de débito podría ser, me pagaste con retraso en la factura, te genero una nota de débito para que te quede a vos una deuda y a mí me quede esa deuda registrada. Esto ya es entrar en demasiado detalle, pero tranquilamente podríamos tener que representar eso de esta manera. Y comprobante se sucedía con emisor, con el cliente, podrían incluir distintos ítems, etc.



## Inventario y contabilidad

Otro patrón interesante son los que tienen que ver con **inventario y contabilidad**. Muchas veces es necesario registrar los movimientos y los saldos de bienes y valores. Por ejemplo, cuando uno agarra la cuenta de la caja de ahorro o el saldo de la caja de ahorro uno se encuentra con que la cuenta tiene un saldo que es derivado de una serie de movimientos. Cuando uno ve su resumen de cuenta bancaria, se encuentra que para llegar al saldo actual en la fecha en la cual se emite el resumen hubo toda una serie de movimientos que sumaron y restaron valores y que nos permitieron llegar al número actual.

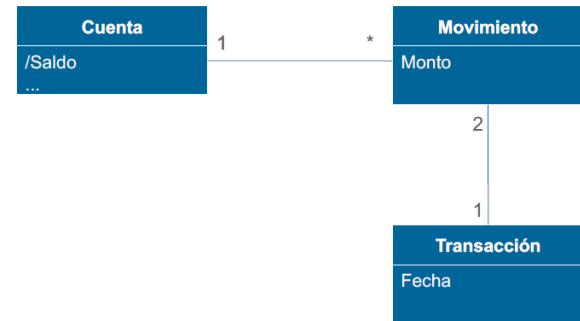
Lo mismo pasa con los productos. Hay movimientos de stock. Si yo necesito reflejar que hubo salidas de tanto frascos de mermelada, tendré que registrar esos movimientos de stock, en qué fecha fueron, qué cantidad. Eso se representa de la manera en la cual lo estamos viendo acá.



A veces también es necesario registrar el movimiento entre cuentas. Por ejemplo, transfiero plata de mi cuenta corriente a mi caja de ahorro o viceversa. Entonces eso en general queda representado por una transacción que tiene una fecha, podría también tener una identificación única; con 2 movimientos asociados, uno para debitar de la cuenta, es decir, para sacar por ejemplo, de la caja de ahorro \$ 100, y otro para acredecir, en la otra cuenta, por ejemplo la cuenta corriente, es decir para sumar, \$ 100. Eso se representaría de esta manera. Por supuesto, esto es una recontra simplificación, se puede complicar muchísimo más.

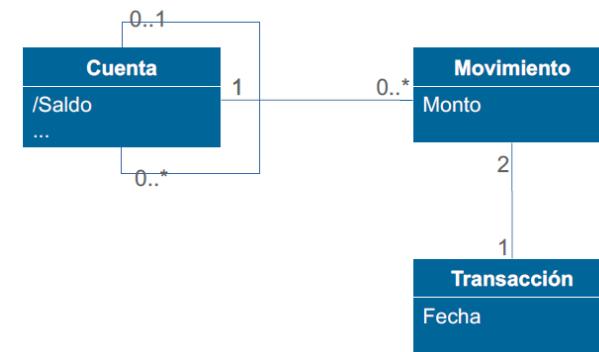
La vida real es mucho más complicada que este modelito, pero en lo conceptual, cuando necesitamos identificar cómo llegamos a un número a partir de una serie de movimientos, las estructuras que se utilizan, en general son estas.

Lo mismo con los esquemas contables. En la contabilidad hay movimientos de fondos entre cuentas de distintas jerarquías. Algunas de esas cuentas tienen subcuentas. Y ahí estamos, representándola, a través de esa estructura jerárquica. Y obviamente, saldo es un atributo derivado de los distintos movimientos que se hicieron sobre esa cuenta (o sobre las cuentas hijas). Probablemente haya cuentas de una determinada jerarquía que no tengan movimientos porque solamente suman cuentas contables que si tienen movimientos.



## Mediciones

El tema de **mediciones** también es muy interesante. A veces necesitamos, por ejemplo, registrar que Roberto mide 1,72 M entonces lo que estamos representando en este modelo es que hay una persona que tiene un nombre, que tiene una medición que es una cantidad y que tiene que ver con un tipo de fenómeno que es la descripción, que en este caso es la altura. También podríamos refinar esto indicando la unidad de medición.



Ejemplos de patrones de modelado de dominio son

- Organizaciones y personas**
- Productos y especificaciones**
- Mediciones**
- Facturas**
- Inventario y contabilidad**

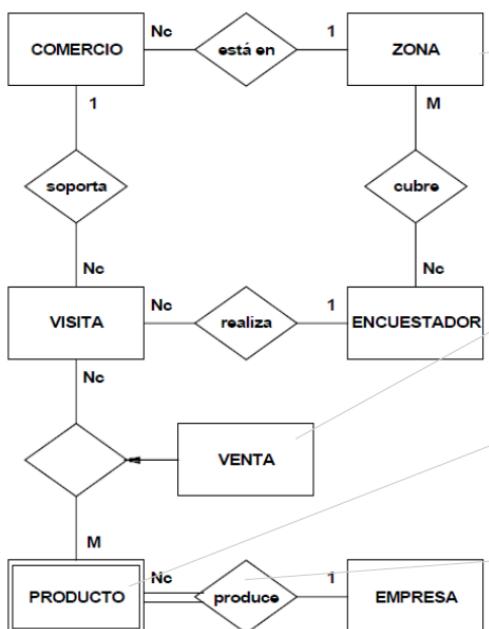
# Modelado de datos

## Métodos y modelos: una breve (e incompleta) perspectiva histórica

~1967: Programación Estructurada.	
1970: A Relational Model of Data for Large Shared Data Banks (Codd)	
~1973: SADT/IDEF0 (Ross)	
~1975: Diseño Estructurado (Constantine & Yourdon)	Módulos, cohesión, acople
1976: Diagrama de Entidad-Relación (Chen)	<b>Modelado conceptual de datos</b>
~1978/9: Análisis Estructurado (Gane & Sarson, DeMarco)	Funciones, datos. Descomposición funcional.
1980: Smalltalk.	
1981/3: Ingeniería de la Información (Finkelstein/Martin)	<b>Modelo de información</b>
1982: Diseño orientado a objetos [OOD] (Booch)	
1984: <i>Essential Systems Analysis</i> (McMenamin & Palmer)	<b>Memoria esencial.</b>
1987: <i>A Framework for Information Systems Architecture</i> (Zachman)	<b>Arquitectura organizacional</b>
1988: <i>Object-Oriented System Analysis</i> (Schlaer/Mellor)	<b>Análisis orientado a objetos</b>
1989: <i>Modern Structured Analysis</i> (Yourdon)	
1990/1: <i>Object Oriented Analysis; Object oriented Design</i> (Coad/Yourdon)	<b>Análisis orientado a objetos</b>
1991: <i>Object-Oriented Modeling and Design</i> (Rumbaugh); <i>Object-Oriented Design With Applications</i> (Booch)	
1992: <i>Object-Oriented Software Engineering</i> (Jacobson)	<b>Modelo de análisis, modelo de casos de uso</b>
1995: <i>Architectural Blueprints: The “4+1” View Model of Software Development</i> (Kruchten)	
1997: UML v1.0	Unificación de las notaciones propuestas por Booch, Rumbaugh y Jacobson.
1998: Extreme Programming.	User stories
1999: Proceso Unificado.	Fases, iteraciones, modelos del sistema basados en “4+1”
2001: Agile Manifesto	
2005: UML 2.0	

El modelado de datos se inició prácticamente con el desarrollo de software. Hay como 2 grandes escuelas con respecto al tema de desarrollo de software. Por un lado, algunos especializados, mucho, en que lo más importante es la funcionalidad y las funciones, los datos vienen después. Y otros que decían, no, primero pensemos bien los datos y luego las funciones. Y hay infinitas combinaciones entre los 2 enfoques. El tema de modelado de actos arranca, probablemente, con el paper de Chen y su planteo del diagrama Entidad-Relación. Eso evolucionó con los años y yo diría que, la evolución que ha llegado a nuestros días es el modelado de dominio.

## Diagrama de Entidad-Relación (ERD o DER)



Cada entidad y cada relación tiene una entrada en el diccionario de datos. Por ejemplo:  
Comercio = @CUITComercio + RazonSocialComercio + TitularComercio + DomicilioComercio + TelefonoComercio  
Zona = @IDZona + DescripcionZona + TotalHabitantesZona  
Está en = {@CUITComercio} + @IDZona

Las entidades asociativas se comportan también como relaciones. Además de los atributos de la relación, tiene atributos propios:

Venta = @NroVisita + @IDEmpresa + @CodProducto + CantidadVendida

Las entidades débiles (o atributivas) tienen una dependencia de existencia con otra entidad (su identificación es una concatenación de el identificador del tipo de objeto fuerte con el que está vinculada y su propio identificador).

Producto = @IDEmpresa + @CodProducto + DescripcionProducto + PresentacionProducto

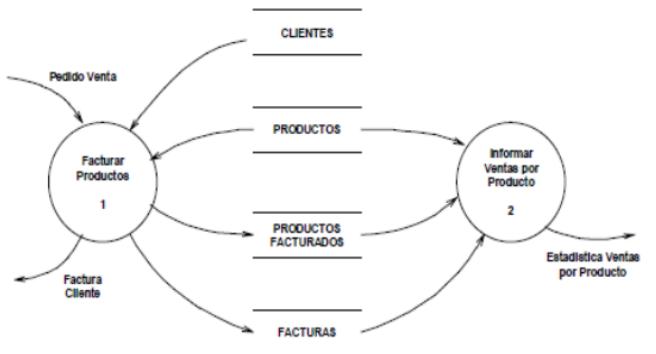
Las relaciones tienen cardinalidad (cuántas entidades participan una relación) y condicionalidad (obligatoriedad o no). Por ejemplo:

En una zona puede haber cero, uno o muchos comercios, pero un comercio pertenece solamente a una zona (Nc – 1).

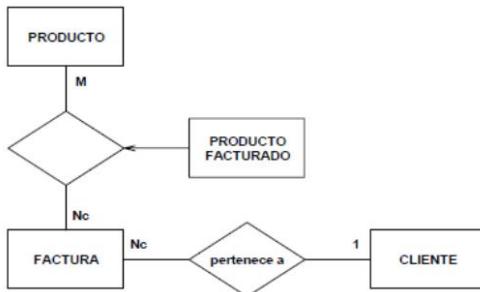
El modelo conceptual de datos, básicamente, lo que proponía era modelar entidades y relaciones. Esto es lo mismo que cuando estamos hablando de objetos y asociaciones entre objetos. Cada entidad es una abstracción, esto también está basado en teoría de conjuntos. La particularidad que plantea el diagrama de entidad-relaciones es que los objetos tienen que ser siempre instanciables, no puede ser que haya una sola instancia de un objeto, sino en ese caso no sería un objeto, una entidad en este modelo. Eso en el modelo de dominio no es algo que sea importante. Pero la verdad es que existen también los objetos débiles, que en definitiva son equivalentes a lo que representamos en el modelo de dominio con una asociación calificada. Cada entidad tiene atributos, algunos atributos son identificadores, y cada entidad debe tener siempre un identificador. Esa restricción en el modo dominio no existe.

## Funciones y datos

Y después, sí empiezan los problemas, porque en el análisis estructural esta entidad-relación tenía que balancear con lo que se representa en los diagrama de flujo de datos como almacenamientos. Y acá es donde algunos autores no nos terminan de cerrar del todo mucho el tema. Hay 2 autores muy importantes que vamos a nombrar más adelante, que son McMenamin & Palmer, y en realidad los datastore que aparecen en DFD<sup>14</sup> corresponden a los objetos del DER<sup>15</sup>, no hay otra cosa para hacer ahí.



NOMBRE	DESCRIPCIÓN
CLIENTE	= @Nro Cliente + Razon Social Cliente + Direccion Cliente
ESTADISTICA VENTAS POR PRODUCTO	= Fecha Informe + {Codigo Producto + {Mes + Cantidad}}
FACTURA	= @Nro Factura + Fecha Emision Factura
FACTURA CLIENTE	= Nro Factura + Fecha Emision Factura + Razon Social Cliente + Direccion Cliente + {Codigo Producto + Cantidad Venta + Precio} + Total
PEDIDO VENTA	= Nro Cliente + {Codigo Producto + Cantidad}
PRODUCTO	= @Codigo Producto + Descripcion Producto + Stock Actual Producto + Precio Producto
PRODUCTO FACTURADO	= @Factura-ref-Nc + @Producto-ref-M + Cantidad Venta + Precio Venta

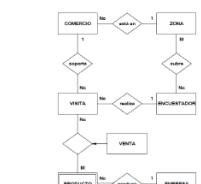
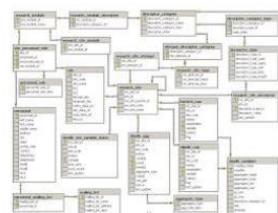


## Modelado de datos y modelado de objetos

Tres comunidades relacionadas, pero con perspectivas diferentes

El tema se complica porque hay distintas comunidades:

- Por un lado está la gente que está más cerca de la **base de datos** que lo que le interesa más es trabajar en el espacio de la solución, con lo cual se empezó a utilizar en algunos casos una especie de entidad-relación pero orientado a la representación de tablas. Con lo cual dejó de usarse con el sentido que proponía Chen que era un modelo conceptual.
- Despues está la comunidad que tiene que ver más con el **modelado conceptual de datos**, que le interesa trabajar en el modelado del espacio del problema, que cree que los datos son activos de organización.
- Por último, está la comunidad de **objetos**, que está más cerca de la solución y que para ellos, todo es un objeto.



<sup>14</sup> Data Flow Diagram

<sup>15</sup> Diagrama Entidad – Relación

Entonces hay 3 grandes tribus hablando de lo mismo o de cosas parecidas, y acá es muy difícil ponerse de acuerdo.

## Conciliación de perspectivas

En realidad lo que dicen los autores es que UML se puede utilizar para modelar datos conceptuales (sería modelado de dominio), para modelar las base de datos (sería modelo lógico de la base de datos) , y para modelar objetos más cerca de lo que sería el diseño, en terminología del diseño de la solución. Hay que tener cuidado con las particularidades, pero se puede utilizar UML para las 3 tribus, pero hay que ponerse de acuerdo en algunas cosas básicas: Cuando estamos hablando de modelado de conceptos, estamos hablando de modelo de dominio, ahí no hay temas de implementación. Eso puede servir para que diseñemos una base de datos, puede servir para que diseñemos los objetos de diseño, los objetos de la solución pero, hay que entender que son vocabularios diferentes.

## Conclusiones

### Resumen

Los modelos entonces nos ayudan a entender el problema, a definir el sistema deseado, pero hay que tener cuidado con qué modelos tenemos que construir. No podemos avanzar ciegamente y construir todos los modelos que muchas veces nos proponen las diversas metodologías que andan dando vueltas, porque no siempre va a ser necesario. Hay infinitas variables que van a influenciar: la complejidad del problema, la complejidad de la solución, el tipo de proyecto, la cantidad de gente involucrada, la complejidad técnica, la complejidad de gestión, etc.

En particular, el modelado de dominio ya sea que lo hagamos del otro lado de una servilleta, en un pizarrón, o que lo hagamos formalmente en una herramienta de modelado, es fundamental para entender el vocabulario del dominio, para entender el contexto en el que va a operar el futuro sistema y para derivar una buena solución del diseño.

## 4.1- Video 4

Clasificar los requisitos que se detallan a continuación.			
Requisito	Funcional	No funcional	No es requisito
Para validar la transacción de pago con tarjeta de crédito, el sistema solicitará la autorización de la operación al gateway de pagos.	✓		
Al final del día, el sistema deberá enviar al gateway de pago las transacciones realizadas	✓		
La interfaz del usuario debe ser fácil de usar.			✓
Un usuario deberá poder hacer la instalación del producto en 15 minutos o menos.		✓	
Para cada producto, el sistema deberá informar código, descripción, precio y cantidad en stock.	✓		
El desarrollo deberá estar terminado en el primer trimestre del año.			✓
El número de tarjeta de crédito no debe mostrarse en ninguna pantalla. Solamente deben exhibirse los últimos cuatro dígitos.		✓	
Con el nuevo sistema, se espera reducir los costos de operación en un 10%			✓
Cada cajero automático podrá interrumpir su servicio por tareas de mantenimiento no más de quince minutos por día.		✓	
Diariamente, el sistema recibirá las novedades (nuevos productos, baja de productos, modificaciones de productos) del sistema de inventario	✓		

Los requisitos pueden ser especificados desde al menos tres puntos de vista diferentes. Cada punto de vista describe:			
	Requisito de usuario	Requisito de software	Requisito de negocio
Qué es lo que el usuario debe poder hacer con el sistema / software	✓		
Por qué la organización necesita el sistema / software			✓
Qué es lo que deben implementar los desarrolladores para que los usuarios puedan hacer sus tareas		✓	

## 4.4- Larman

¿Cuáles de los siguientes son buenos nombres de asociaciones?

- En general, los atributos de las entidades son tipos de datos primitivos, pero pueden crearse nuevos tipos de datos, como por ejemplo “Dirección”.**
- La asociación entre dos entidades conceptuales nos permite representar si la navegación es unidireccional o bidireccional.
- La multiplicidad en una asociación aplica a un determinado momento y no a un periodo de tiempo. Por ejemplo, un jugador JUEGA en UN equipo de fútbol. No importa que en el futuro pueda llegar a jugar en otros.**

En un modelo de dominio se deben indicar:

- Asociaciones entre entidades conceptuales**
- Operaciones
- Clases de diseño
- Atributos**

[V / F] El análisis de requisitos se enfoca en identificar y resolver conflictos entre los requisitos.

[V / F] Un modelo de dominio, luego de sucesivas iteraciones, debe incluir detalles de los patrones de diseño utilizados.

## 4.2 - Historias de usuario

### Historias de usuario

#### Definición

Una **historia de usuario** formalmente es una descripción de la funcionalidad esperada de parte de un sistema. Esta descripción está siempre expresada desde el punto de vista del usuario del sistema.

#### Formato

Si bien el trabajo original con el formato propuesto era más libre, simplemente un título, en la actualidad, el formato más aceptado es el que llamamos: “¿Quién, qué, por qué?”.

Como <actor>, quiero poder <función> de modo que/para así  
<razón>

#### Ejemplo

En un sistema de ventas, una posible historia de usuario podría ser:

Como **cliente**, quiero poder consultar los **productos disponibles**  
para así poder tomar una **decisión de compra**.

Como si fuera a ver las historias de usuarios simplemente describen qué es lo que los usuarios desean del sistema. **No profundizan en el detalle ni dan información adicional respecto a la funcionalidad esperada**. En este ejemplo no nos indica qué datos de los productos se van a informar. Tampoco se aclara qué pasa si no hay productos en el sistema.

### Historias de usuario y requisitos

Es importante entonces destacar que las historias de usuario no son requisitos en el sentido estricto o tradicional del término. Algunos autores dicen que las historias de usuario son más bien punteros, o recordatorios de que se debe mantener una conversación con el usuario acerca de las funcionalidad deseada descripta en la historia.

[V / F] Las historias de usuario son útiles para definir el detalle de la funcionalidad esperada.

### Historias de usuario

#### 3 Cs: Card, Conversation, Confirmation

Esto nos lleva a un concepto llamado de las **3 Cs**, por las siglas en inglés de **Card, Conversation** y **Confirmation**. Al desarrollar las historias de usuario, se debe tener presente que hay 3 elementos que las componen:

- **Card:** La descripción de la historia propiamente dicha, que pone de manifiesto cuál es la intención del usuario con respecto a la utilización del sistema, que normalmente se registra en una ficha o en algún tipo de herramienta.
- **Conversation:** La conversación que debe mantenerse con el usuario para profundizar los detalles del comportamiento esperado.
- **Confirmation:** La confirmación que describe los criterios de aceptación que permitirán garantizar que se han cubierto todos los aspectos de la historia.

### Criterios de aceptación

#### Formato

Los criterios de aceptación se especifican siguiendo el formato:

Dado que <contexto>, cuando <acción> entonces  
<consecuencias>

## Ejemplo

Un posible criterio de aceptación para nuestra historia de usuario de consulta de productos disponibles podría ser:

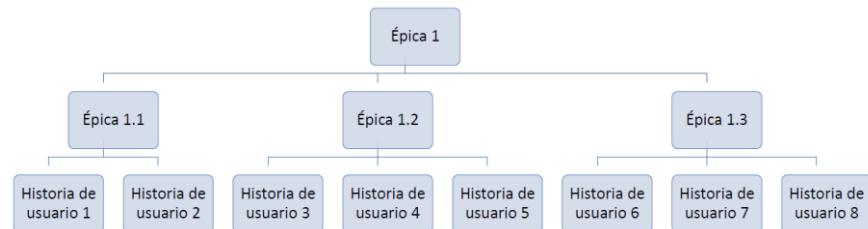
Dado que soy cliente  
Cuando consulto productos disponibles  
Entonces se me informa descripción, precio e inventario de los productos disponibles

Claramente se puede apreciar que los criterios de aceptación crearán posteriormente origen al caso de prueba que se podrían especificar mediante, por ejemplo, el lenguaje Gherkin. Sirve para completar la especificación del comportamiento esperado del sistema. Desde ese punto de vista, forman parte de la especificación de requisitos de software o son equivalentes en especificación de software en un esquema más tradicional por supuesto.

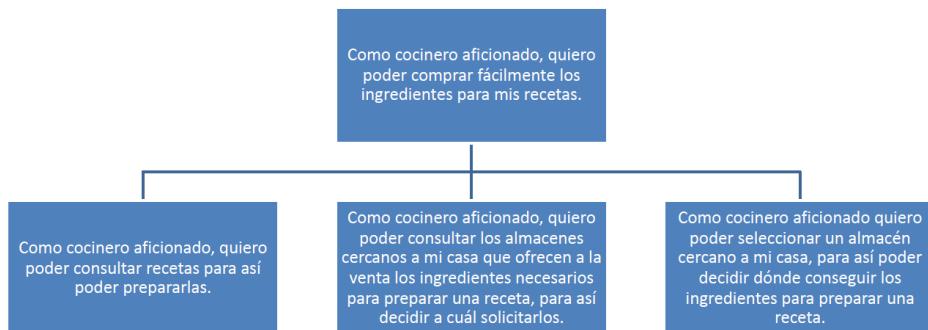
[V / F] Los criterios de aceptación ayudan a definir la funcionalidad.

## Epopeyas

Ahora viene un problema que suele aparecer en cualquier proyecto de un tamaño no trivial, que es el manejo de numerosas historias de usuario. Para resolverlo, aparece el concepto de **épica o epopeya**. La idea es agrupar historias en relatos de mayor nivel, llamadas justamente épicas.

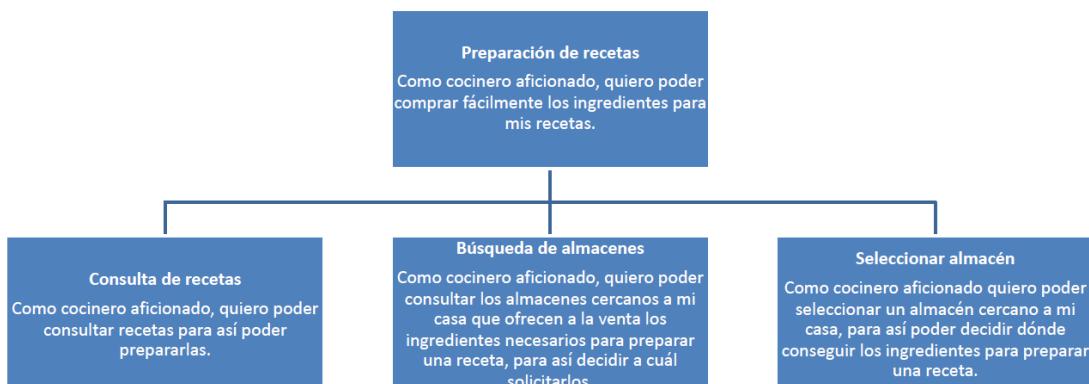


Por ejemplo, en este caso, una historia de usuario ha sido desagregada en 3 historias con mayor nivel de detalle:



## Nombres

Lamentablemente, no hay convenciones acerca de cómo debe estar estructurada una épica, algunos pueden sugerir utilizar el mismo formato que se utiliza para las historias o simplemente describirlas con un título. También se le puede poner un título a las historias para simplificar la forma en la que nos referimos a ellas durante el desarrollo.



No es lo mismo hablar de la historia “consulta de recetas” que referirse a ella como la historia “Como cocinero aficionado, quiero poder consultar recetas para así prepararlas”.

## Épicas e historias de usuario

### Formatos

El gráfico anterior era simplemente con fines ilustrativos. Nadie construye un gráfico de ese tipo. Usualmente nos vamos a encontrar con una estructura como esta:

- **Gestión de compras**
  - **Consulta de productos**
    - Como cliente, quiero poder consultar los productos disponibles ordenándolos por categoría, para así decidir cuáles agregar a mi pedido.
  - **Selección de productos**
    - Como cliente, quiero poder agregar a mi pedido una o varias unidades de un producto mientras los estoy consultando, para así satisfacer mis necesidades de compra.
  - **Confirmación del pedido**
    - Como cliente, quiero poder confirmar mi pedido para así recibir los productos seleccionados en mi domicilio.

Inclusive hay herramientas que proponen agrupar historias de usuario y mostrarlas de manera similar a este formato que estamos viendo acá.

[V / F] Para solucionar el problema en la organización de una gran cantidad de historias de usuario se utilizan épicas.

## Calidad de las historias de usuario

### Regla INVEST

I	independiente	(de otras historias)
N	negociable	(una declaración flexible, no un contrato)
V	valiosa	(aporta valor al cliente/usuario)
E	estimable	(se puede estimar; negociable)
S	pequeña	(se puede desarrollar dentro de una iteración)
T	verificable	(se entiende lo suficiente como para ser probada)

A lo largo del tiempo han surgido criterios para determinar si una historia de usuario es correcta o no. Una de ellas es **INVEST**, que propone que cada historia debe ser **Independiente** de otras historias; **Negociable** ya que debe ser una declaración flexible, no un contrato solamente por como debe tener lugar en una conversación, hay cosas que por ahí tienen que ajustarse; **Valiosa** o valorable, tiene que aportar valor al cliente o al usuario; **Estimable**, es decir, que tiene un tamaño suficiente como para poder ser estimada; **Pequeña** (Small) que se puede desarrollar dentro de una iteración, esto tiene que ver mucho con el desarrollo iterativo e incremental o tiene que ver mejor dicho, con el desarrollo iterativo incremental; y **Verificable** o testeable, se entiende lo suficiente como para poder ser probado.

### QUS Framework

Otro marco de referencia es el **Quality User Story (QUIS)** que propone 13 atributos de calidad, agrupados en 3 categorías:

<b>Calidad sintáctica</b> <i>Estructura, sin considerar contenido</i>	Bien formada Atómica Mínima
<b>Calidad semántica</b> <i>Relación y significado de los elementos</i>	Conceptualmente acertada Orientada al problema No ambigua Libre de conflictos
<b>Calidad pragmática</b> <i>Interpretación subjetiva de la audiencia</i>	Oración completa Estimable Única Uniforme Independiente Completa

Algunos de estos criterios se aplican a historias individuales, y otros se aplican a un conjunto de historias de usuario.

Criterio	Descripción	Individual/grupo
Bien formada	Debe incluir al menos un rol y una acción	Individual
Atómica	Debe expresar una única funcionalidad	Individual
Mínima	Debe contener solamente un rol, una acción y un fin	Individual
Conceptualmente acertada	La acción expresa una funcionalidad y el fin una razón	Individual
Orientada al problema	No debe especificar la solución	Individual
No ambigua	Debe evitar el uso de términos que puedan llevar a múltiples interpretaciones	Individual
Libre de conflictos	Debe ser consistente con otras historias de usuario	Grupo
Oración completa	Debe ser una oración bien formada	Individual
Estimable	Debe poder ser estimada y planificada	Individual
Única	No debe haber duplicados	Grupo
Uniforme	Todas las historias de usuario emplean el mismo formato	Grupo
Independiente	No debe depender de otras historias; debe ser autocontenido	Grupo
Completa	No faltan pasos	Grupo

[V / F] Según el criterio INVEST que una historia de usuario sea pequeña facilita su estimación.

Veamos a continuación algunos ejemplo de aplicación:

Ejemplo 1

Quiero consultar los **productos disponibles**

Bueno, esta historia NO CUMPLE con el criterio de bien formada: Faltan el rol y el fin. Lo correcto habría sido expresarla siguiendo el formato: “quién, qué, para qué” que vimos anteriormente.

Una aclaración al respecto, este framework considera como opcional la indicación del por qué. Nosotros consideramos que, sin embargo, que el uso es obligatorio de el por qué, ya que nos permite justificar para qué es necesaria la funcionalidad deseada, para qué la necesitamos, cuál es el objetivo detrás de esa historia de usuario.

Ejemplo 2

Como **cliente**, quiero poder clickear en el botón “Consulta” para ver los **productos disponibles**

En este otro ejemplo se incumple el criterio de orientada al problema, ya que se dan detalles de implementación que son completamente irrelevantes en este contexto. Es más, nos está forzando quizás de manera demasiado anticipada, un tipo de solución posible. Yo en lugar de un botón podría decidir tener un link. Esa es una discusión que no debería tener lugar en este contexto.

## Ejemplo 3

Como cliente, quiero poder consultar los **productos disponibles** para así luego **ingresar mi pedido**.

En este otro caso, se incurre en un error bastante común en la práctica. Se hace referencia incorrectamente a otra historia de usuario, en este caso la correspondiente al ingreso de pedidos. No hace falta hacer eso. Lo importante es la razón por la cual queremos consultar los productos en este caso probablemente para tomar una decisión de compra. **Cuando decimos para qué tenemos que especificar cuál es el valor que esperamos conseguir de esa historia de usuario.**

## Historias de usuario

### ¿Cómo encontrarlas?

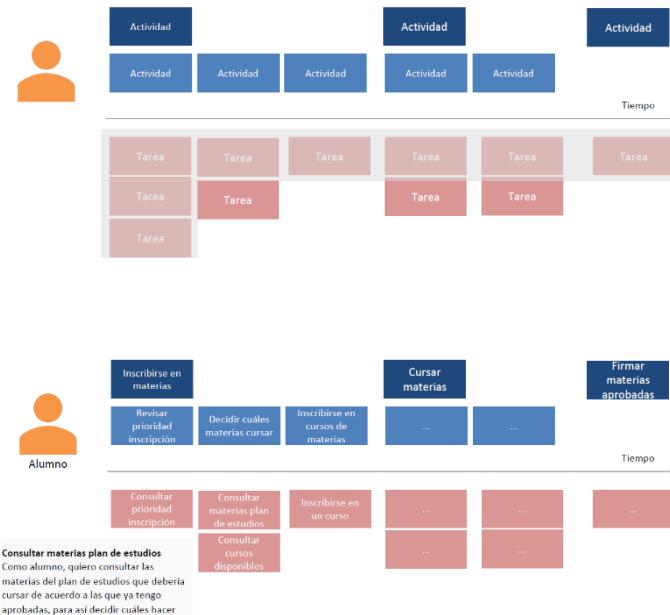
Uno de los problemas habituales que enfrentamos es cómo identificar historias de usuario. No siempre es fácil hacerlo, aún conociendo los conceptos teóricos. Lo que siempre hay que tener presente es que primero debemos entender cuáles son los límites del sistema que estamos analizando; y entendido este punto vamos a poder determinar quiénes son los que utilizarán el sistema. Luego, para cada uno de estos usuarios debemos identificar cuáles son sus objetivos, qué es lo que quieren lograr utilizar del sistema. Usualmente, los objetivos pueden incluir una jerarquía de subobjetivos. Al llegar al final de la jerarquía deberíamos estar en condiciones de encontrar una historia de usuario para cada subobjetivo.

[V / F] El **para qué** especifica cuál es el valor que se espera de la historia de usuario.

## User Story Mapping

Un posible enfoque para descubrir historias de usuarios es el **User Story Mapping**. Un User Story Map es un modelo que describe las actividades que realizan a lo largo del tiempo los usuarios de una aplicación y su descomposición en subactividades y tareas. A partir de estas tareas podemos identificar las historias de usuario.

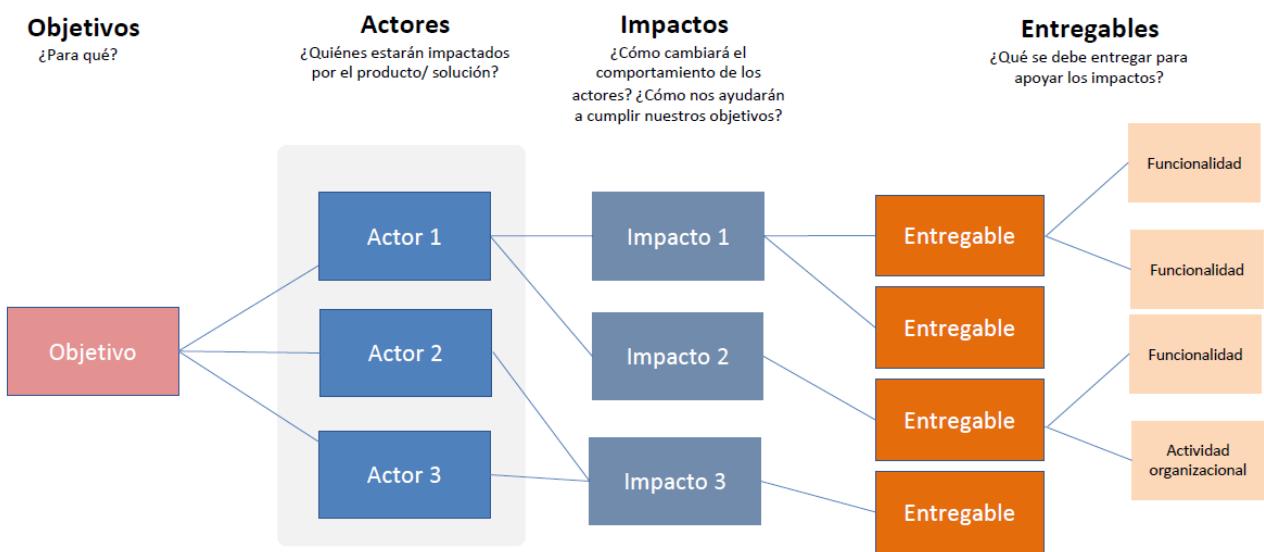
Por ejemplo, un alumno de una universidad podrá inscribirse en una materia, cursarla, firmarla, etc. Para inscribirse una materia revisará su prioridad de inscripción, decidirá cuáles materias cursar y luego se inscribirá en los cursos de las materias que haya seleccionado. **Para cada una de estas actividades habrá una tarea, como mínimo, que el sistema le ayudará a llevar a cabo.** En el ejemplo, consultar el plan de estudios, es una historia que contribuye a la actividad de decidir cuáles materias cursar.



[V / F] En la técnica *User Story Mapping* para cada una de las actividades hay al menos una tarea.

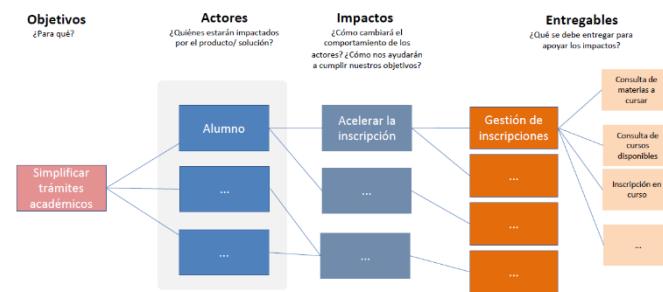
## Impact Mapping

Otra alternativa, es emplear una técnica de planificación llamada **Impact Mapping**. Impact Mapping tiene como propósito alinear los equipos de trabajo con los objetivos de la organización.



Como se puede observar en el gráfico, a partir de la identificación de los objetivos (y estamos leyendo el gráfico de izquierda a derecha) que se definen para la solución a desarrollar, se identifican los actores afectados, cómo se pretende afectar el comportamiento de cada uno de esos actores, y finalmente, qué es lo que se deberá hacer para producir esos impactos (en el gráfico los entregables), que a su vez se pueden descomponer en productos, servicios funcionalidades.

En este ejemplo, el objetivo identificado para el sistema de gestión universitaria es simplificar los trámites académicos. Entre los actores impactados vemos al alumno. Uno de los impactos que se desea tener en los alumnos es acelerar la inscripción. Para ello, se entregará una funcionalidad de gestión de inscripciones que incluirá varias historias de usuario, entre ellas la consulta de materias a cursar, la consulta de materias disponibles y la inscripción en el curso.

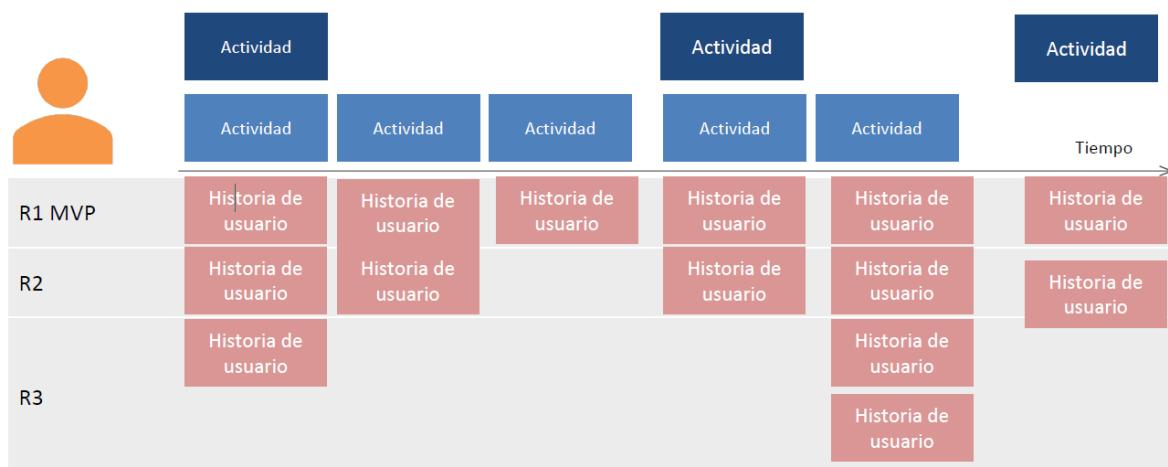


Este enfoque anterior **nos permite ir de lo general a lo particular, pero también recorren el camino inverso**. Muchas veces nos vamos a encontrarnos en esa situación. Esta técnica en particular tiene la ventaja de alinear funcionalidades con objetivos de la organización o con objetivos del proyecto, algo que muchas veces se pierde de vista.

[V / F] La técnica de *Impact Mapping* permite ir de lo general a lo particular y viceversa.

## En el proceso de desarrollo

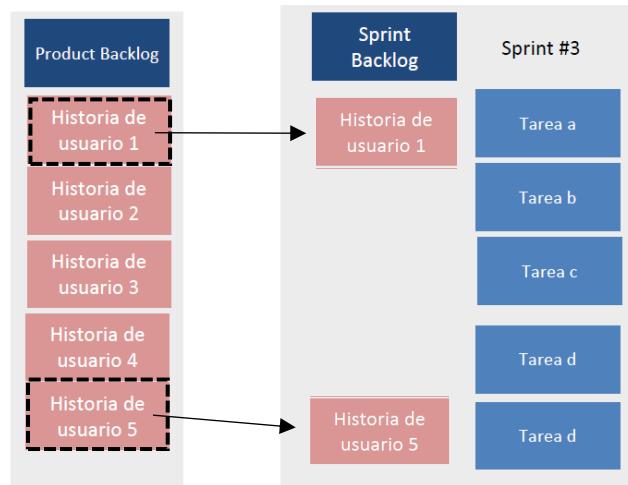
Volviendo a User Story Mapping, es importante destacar que, además de lo que ya hemos comentado, nos ayuda a realizar la planificación, nos ayuda a determinar qué historias de usuario vamos a asignar a cada uno de los releases.



Lo que vemos en el ejemplo es cómo se han analizado las historias en torno a cada uno de los releases que se han identificado, que en este caso son 3. Son esas cajas grises que están en forma horizontal. El primer release (R1), también llamado MVP o Producto Mínimo Viable, el release 2 (R2) y el release 3 (R3).

## Product & Sprint Backlogs

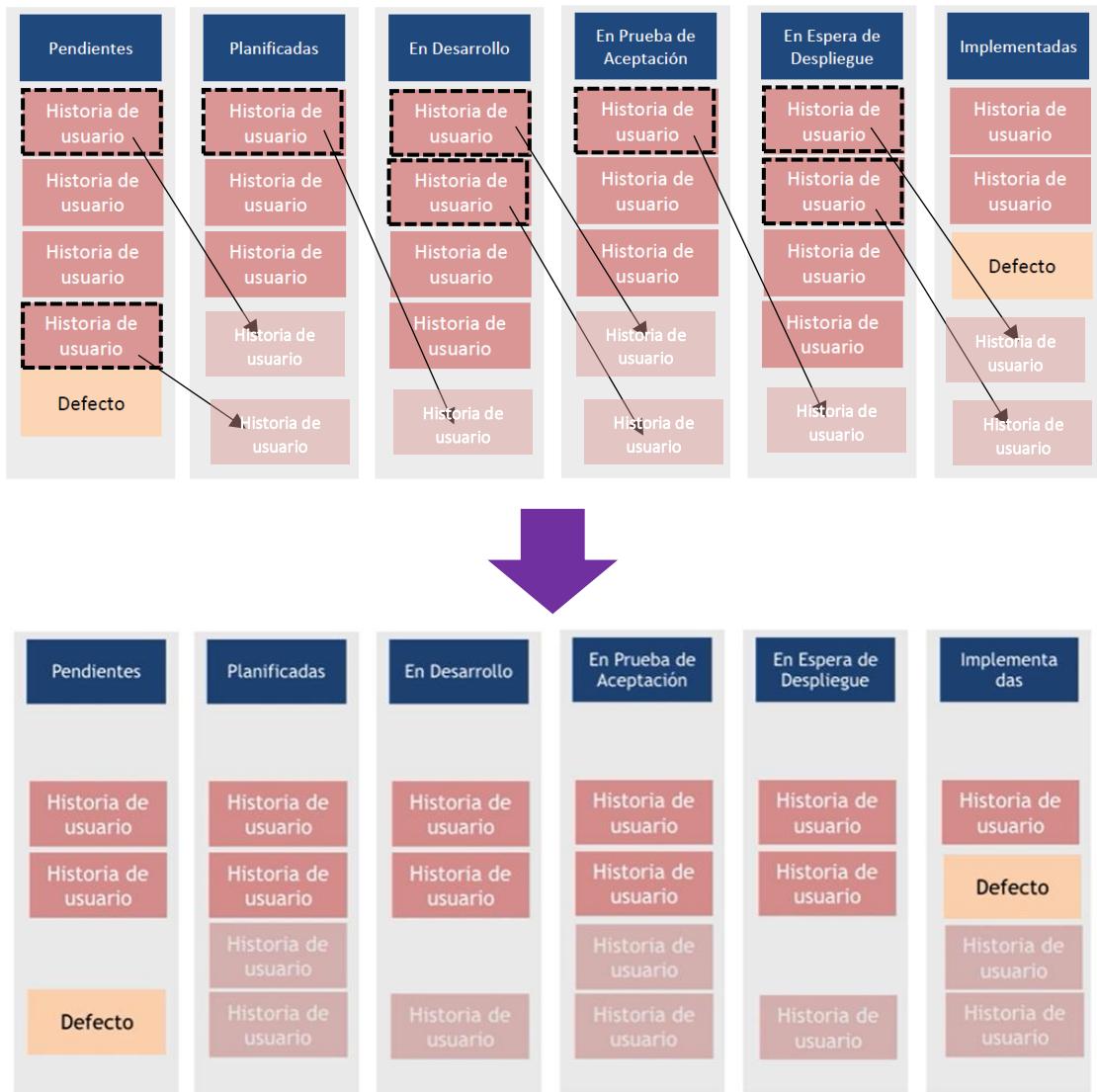
En Scrum, una metodología ágil que ya hemos mencionado, las historias de usuario se pueden incluir en un **Product Backlog**. Cada historia se estima empleando una técnica llamada user story points, que nos da una medida relativa de la complejidad que puede llegar a implicar, desarrollar y probar la funcionalidad asociada a cada una de las historias de usuario. En función de la capacidad disponible, y también con respecto al plan de entregas, el equipo asignado va a decidir qué historias de usuario desarrollar en cada iteración, que en Scrum recordemos que se llaman **Sprints**. A partir de ellas, se determinan las tareas que se deben ejecutar para implementar. Esta lista de tareas es el **Sprint Backlog**.



[V / F] La técnica de *Impact Mapping* facilita la planificación.

## Kanban

Otro mecanismo muy interesante para organizar el trabajo que está basado en historias de usuario, es Kanban. Kanban es un sistema de planificación y organización del trabajo utilizado originalmente en las plantas de fabricación de Toyota y adaptado desde ya un tiempo al desarrollo de software. Su objetivo es ayudar a darle visibilidad al trabajo, a limitar el working progress o trabajo en simultáneo y mejorar el flujo de trabajo.



En el ejemplo vemos una tarjeta por cada historia de usuario y por cada defecto encontrado en el software. Esto es utilizado obviamente en el contexto de un esfuerzo de desarrollo de software, un proyecto de desarrollo de software. **Cada columna representa el estado en el cual se encuentran las historias y los defectos.** La idea es que la cantidad de tarjetas que puedan estar en cada estado esté limitada por las restricciones propias del equipo y la infraestructura disponible. A medida que cambiamos el estado de las historias, las vamos moviendo por el tablero.

[V / F] En el sistema Kanban cada columna representa un estado.

## 4.3 - Lucassen

Para cada una de las historias de usuario, determinar cuál es el criterio de calidad MÁS GRAVE que incumple ([Lucassen])

	<i>Como sistema, quiero enviar las transacciones de pago al sistema de tarjetas, para que se procesen.</i>	<i>Como sistema quiero instalar los nuevos comercios en los servidores propios para dar acceso a nuevos compradores.</i>	<i>Como alumno, quiero presionar un botón en la pantalla para inscribirme a una materia para poder cursarla.</i>	<i>Como comprador quiero consultar los productos ofrecidos y preparar el pedido sin necesidad de registrarse.</i>	<i>Como gateway de pago quiero recibir todas las transacciones para poder proceder con las mismas.</i>	<i>Como comprador quiero poder agregar o quitar productos de la consulta así como variar la cantidad de los mismos para tener más información.</i>
Atómica						✓
Mínima						
Bien formada	✓				✓	
Libre de conflictos						
Conceptualmente razonable		✓				
Orientada al problema			✓			
No ambigua						
Dependencias explícitas						
Oración completa						
Independiente						
Escalable						
Uniforme						
Única						

# 4.1 - Casos de uso

## Casos de uso

### Definición (informal)

Informalmente, podemos decir que un caso de uso es **una historia acerca de cómo un actor (el usuario), utiliza un sistema para alcanzar sus objetivos**. Por ejemplo, al especificar el sistema de inscripciones de una universidad, probablemente nos encontraremos con un caso de uso como este:

#### Consultar cursos disponibles

El alumno consulta los cursos disponibles para una materia en la que se quiere inscribir. Para cada curso disponible, el sistema le informa días y horarios, docentes y cantidad de vacantes.

Que describe un escenario de utilización del sistema en particular y correspondiente a la consulta de los cursos disponibles.

### Definición (formal)

Más formalmente, un caso de uso es **una secuencia de acciones realizadas por un sistema que generan un resultado observable de valor para un actor en particular**.

## Actor

### Definición

En este contexto, un actor es **cualquier entidad con comportamiento que interactúa con el sistema: personas, organizaciones, e inclusive otros sistemas**. Volviendo al ejemplo, un alumno es en este caso un actor que interactúa con el sistema de inscripciones.

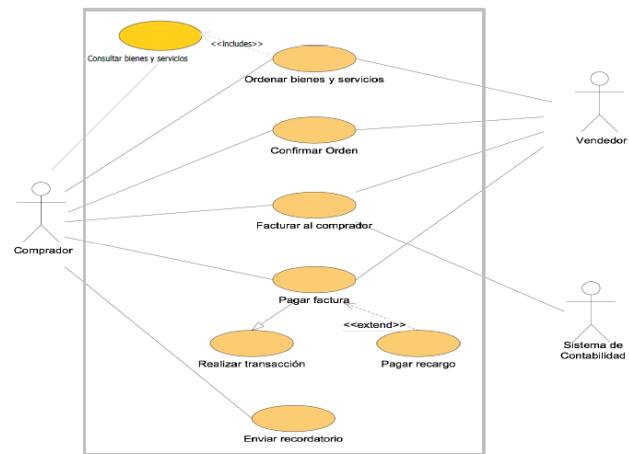
## Casos de uso

### Tecnológicamente neutros

Es importante destacar que **los casos de uso son tecnológicamente neutros**, es decir, que son esenciales, no describen la solución, no describen la interfaz del usuario. Describen cuáles son las acciones del usuario, cuáles son las respuestas del sistema, pero lo hacen en términos tecnológicamente neutros, es decir, independientemente de los aspectos de implementación.

### El modelo visual: diagramas de casos de uso en UML

Los casos de uso también se pueden representar en forma visual. Esto se hace utilizando la notación propuesta por UML. En el gráfico podemos apreciar los actores, los casos de uso y los límites del sistema analizado, que se representa a través de un rectángulo. Es muy importante destacar que este diagrama no reemplaza, de manera alguna, a las descripciones, a las especificaciones de los casos de uso en formato texto. Como se puede ver, el diagrama aporta muy poca información. No podemos sacar ninguna conclusión acerca de cuáles son los objetivos de cada caso de uso, de cuáles son los objetivos de cada actor, del comportamiento que implica cada caso de uso, de las acciones que hace el actor. Todo este detalle es lo que vamos a encontrar en la especificación de cada caso de uso.



Un caso de uso...

- es una secuencia de acciones que realiza un actor que otorgan un tipo de resultado
- detalla claramente cómo el sistema elabora las respuestas
- Es tecnológicamente neutro

## Escenario

### Definición

Los casos de uso están compuestos de escenarios. Un escenario es **una secuencia particular de acciones e interacciones dentro de un caso de uso**. Normalmente nos vamos a encontrar con un **escenario principal**, con lo que muchos llaman “el camino feliz”, o sea que, muestra las acciones del actor y las respuestas del sistema sin indicar situaciones particulares, excepciones u errores; y uno o varios **escenarios alternativos o secundarios** que describen situaciones particulares que se pueden dar bajo determinadas circunstancias.

#### Consultar cursos disponibles

##### *Escenario principal*

El **alumno** consulta los cursos disponibles para una materia en la que se quiere inscribir. Para cada curso disponible para la materia, el sistema le informa días y horarios, docentes y cantidad de vacantes.

##### *Escenario alternativo*

De no existir cursos disponibles, el sistema le informa al alumno “No hay cursos disponibles para la materia ingresada”

Por ejemplo, en este caso de uso, nos encontramos con que hay un escenario alternativo que describe qué es lo que pasa cuando no hay cursos disponibles para una materia que el alumno ha consultado.

## Casos de uso

### Un contrato

Resumiendo:

- Un caso de uso describe un **contrato** entre las partes interesadas acerca del **comportamiento** de un **sistema**.
- Dicho comportamiento describe qué responde el sistema a las acciones de una de las partes interesadas en particular llamada **actor primario**.
- El actor primario inicia una **interacción** con el sistema para **lograr** algún tipo de **objetivo**. El sistema responde protegiendo los intereses de todas las partes interesadas.
- El caso de uso recoge, incluye, varios **escenarios** que son el resultado de distintas condiciones y acciones de los actores.

## Formato breve

Para describir un caso de uso hay varios formatos posibles. En el llamado **formato breve**, simplemente hay un título que siempre es un verbo en voz activa más un objeto, y una descripción resumida del comportamiento.

#### Consultar productos disponibles

El cliente consulta los productos disponibles, filtrándolos por categoría, descripción o código de producto.

## Formato secuencia de acciones

En este otro formato, se describen las acciones del actor y las respuestas del sistema uno por uno. Es importante destacar que el sistema se escribe como una **caja negra**, **no hay detalles de cómo el sistema elabora las respuestas a cada una de las acciones del actor**.

### Consultar productos disponibles

El cliente consulta los productos disponibles, filtrándolos por categoría, descripción o código de producto.

- 1) El cliente ingresa una categoría de producto, un código de producto o una descripción de producto (total o parcial)
- 2) El sistema muestra el código, la descripción, la categoría y el stock disponible del producto o de los productos que cumplan el criterio de búsqueda.

En el ejemplo vemos que un cliente ingresa una categoría de producto y que el sistema muestra el código, la descripción, la categoría, etc. El caso de uso no describe cómo el sistema elabora esa respuesta, más allá de los aspectos de implementación que claramente están totalmente prohibidos en el caso de uso.

### Formato de dos columnas o diálogo

En este otro formato, llamado de dos columnas o diálogos se opta por separar las acciones y respuestas en 2 columnas separadas.

Consultar productos disponibles	
El cliente consulta los productos disponibles, filtrándolos por categoría, descripción o código de producto.	
Actor	Sistema
Ingresar una categoría de producto, un código de producto o una descripción de producto (total o parcial)	Informa el código, la descripción, la categoría y el stock disponible del producto o de los productos que cumplan con el criterio de búsqueda.

### Más completo: precondiciones, postcondiciones, escenarios alternativos

Por supuesto, hay otros campos que se pueden agregar y que se deben agregar al caso de uso:

- ◆ Uno de esos campos son las **precondiciones**. Las precondiciones son prerequisitos para que el caso de uso se pueda ejecutar, y que por supuesto valga la pena contarle al lector. Precondiciones puede ser el estado inicial del sistema o la ejecución de otro caso de uso anterior. Las precondiciones no describen la intención del usuario ni de obviedad del tipo “el sistema está disponible” o “el usuario está logueado”. Nuevamente tenemos que pensar en quién es el que va a leer, qué información necesita para entender lo que pasa en el caso de uso.
- ◆ Las **postcondiciones**, describen qué es lo que debe ser cierto para cumplir con el objetivo del actor con respecto al caso de uso. Puede ser algo observable, o no; un cambio en el estado del sistema. Por ejemplo, objetos, asociaciones del modelo de dominio que se han creado, actualizado o eliminado.
- ◆ También se puede incluir un **disparador**, que es un **trigger**.
- ◆ Los requisitos no funcionales asociados a el caso de uso.

Agregar producto a un pedido de compra	
El cliente agrega un producto a su pedido de compra.	
Precondiciones	
	El cliente ya consultó los detalles del producto.
Actor	Sistema
1. Ingres el código de producto que desea y la cantidad	
	2. Informa la descripción, el precio por unidad, el precio total del ítem
Postcondiciones	
El producto y la cantidad de unidades ingresadas han sido agregadas al pedido de compra.	
Escenarios alternativos	
2.1 Si no hay stock del producto ingresado, el sistema informa "No hay actualmente unidades disponibles del producto seleccionado. La entrega de este producto podrá verse demorada"	

[V / F] Que un sistema se describa como una caja negra significa que no hay detalles de cómo el sistema elabora las respuestas.

## ¿Cómo encontrarlos?

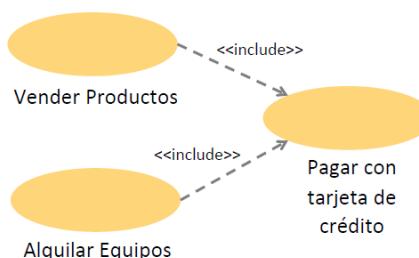
¿Cómo hacemos para encontrar los casos de uso? Bueno es importante:

1. Establecer, entender cuáles son los **límites del sistema** que es lo que de alguna manera vamos a incluir y qué es lo que vamos a dejar afuera, a quiénes son los actores que van a interactuar con el sistema.
2. Tenemos que identificar cuáles son los **actores** primarios y cuáles son sus **objetivos**.
3. **Para cada uno de los objetivos**, en general de cada uno de los actores, en general, vamos a encontrar un caso de uso.
4. Para cada uno de los casos de uso, lo que vamos a hacer es identificar las pre y las post condiciones.
5. Después vamos a describir los escenarios principales de cada uno de los casos de uso.
6. Buscaremos y describiremos los escenarios alternativos de cada uno de los casos de uso.
7. Revisaremos, refinaremos y ajustaremos: Buscaremos relaciones de inclusión, relaciones de extensión y especializaciones.

## Casos de uso

### Relaciones de inclusión

Muchas veces nos vamos a encontrar con que hay comportamiento que se repite en varios casos de uso. Para evitar duplicar el texto una y otra vez, lo que se puede hacer es crear un nuevo caso de uso y poner en ese caso uso el comportamiento compartido que hasta ahora hemos visto repetido en más de un caso de uso, e invocarlo, a ese nuevo caso de uso, desde los casos de uso que lo necesiten. Esos casos de uso son llamados casos de uso **base**, y esta relación que tenemos entre los casos de uso base y ese caso de uso nuevo que hemos creado, son **relaciones de inclusión**.



#### CU010: Vender Productos

- Escenario principal:
  - 1) El cliente arriba al punto de venta con los productos para pagar.
  - 2) ...
  - 6) El cliente paga su compra ([CU020 Pagar con tarjeta de crédito](#)).

#### CU011: Alquilar equipos

- Escenario principal:
  - ...
  - 3.a) El cliente paga el alquiler ([CU20 Pagar con tarjeta de crédito](#)).

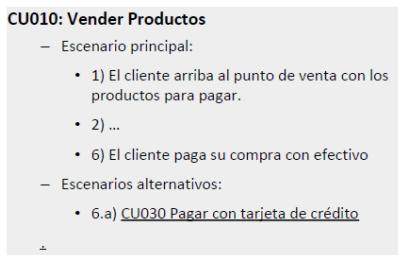
#### CU20: Pagar con tarjeta de crédito

- Escenario principal:
  - ...

En este ejemplo vemos que se ha decidido factorizar el comportamiento relacionado con el pago mediante tarjeta de crédito, que se encuentra repetido en 2 casos de uso, lo hemos puesto en un caso de uso aparte que es invocado desde “Vender Productos” y “Alquilar equipos”. La relación se dibuja como una flecha con línea punteada que va desde el caso de uso base al caso de uso incluido, es decir, el caso de uso que necesita al caso de uso que provee esa funcionalidad.

## Relaciones de extensión

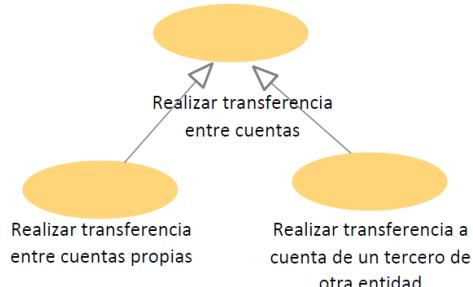
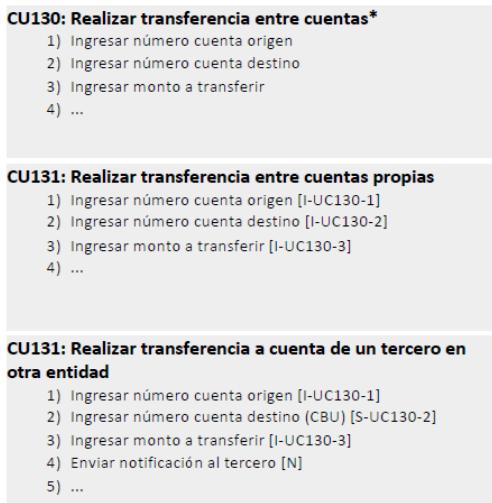
Cuando en un caso uso encontramos comportamiento adicional u opcional, también podemos crear un nuevo caso de uso para ubicar esas acciones y así simplificar la lectura del caso de uso base.



En este ejemplo, los usuarios normalmente pagan en efectivo, pero en determinadas circunstancias pueden hacerlo mediante tarjetas de crédito. Aquí se ha decidido extraer ese comportamiento y ubicarlo en un nuevo caso de uso, informalmente llamado **extendido**. Hay que tener cuidado porque aquí el sentido de la flecha es al revés que en la inclusión. “Pagar con Tarjeta de Crédito” extiende el comportamiento de “Vender Productos”, por eso se pone esa relación en ese sentido.

## Relaciones de generalización

También es posible encontrar situaciones en donde es necesario especializar un caso de uso por alguna circunstancia en particular.



Lo que vemos aquí es que hay un caso de uso **padre** con el comportamiento compartido entre 2 casos de uso **hijos**. No es algo que se utilice mucho, pero la posibilidad está disponible. Lamentablemente, no hay consenso acerca de cómo especificar correctamente este tipo de casos de uso. Una alternativa es indicar en los casos de uso hijos, cuál es el comportamiento que se hereda sin cambiar, cuál se hereda pero se especializa y cuál comportamiento es nuevo propio del caso de uso hijo.

[V / F] La *inclusión* en casos de uso se refiere a la *repetición* de comportamiento *común* entre los casos de uso.

[V / F] La *extensión* en casos de uso se refiere a la *adición* de comportamiento entre los casos de uso.

## Adicionales

Algunos temas adicionales:

- A partir de cada caso de uso, se pueden identificar **casos de prueba**: Qué es lo que se debe probar en cada escenario, bajo qué condiciones, con qué datos, etc. Si leemos detenidamente el caso de uso nos vamos a encontrar con que hay distintos caminos, distintas combinaciones. Básicamente el escenario principal nos va a determinar un camino de prueba posible, pero la combinación de ese escenario principal con los distintos escenarios alternativos nos da otros casos de prueba posible.
- También vamos a encontrarnos con que en los casos de uso se mencionan objetos de negocio u objetos de dominio, asociaciones, atributos o propiedades. Todos ellos que se mencionan en los casos de uso, deberían existir o **deben existir** en el modelo de dominio o en el modelo que se esté utilizando para representar ya sea el dominio o los datos.

## Casos de uso CRUD / ABMC

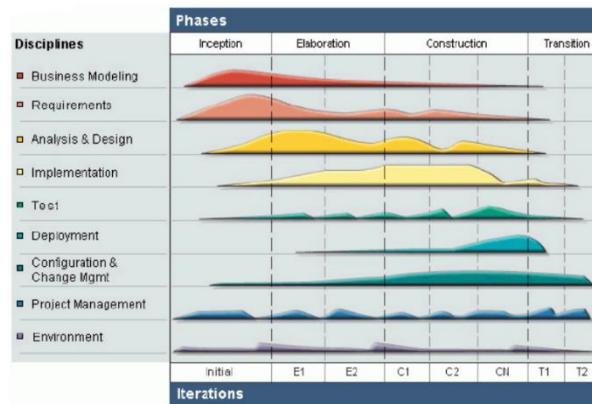
Otro tema bastante particular es el de los casos de uso CRUD (por Create, Read, Update, Delete), o en castellano, como se suele llamar ABM o ABMC que es Altas, Bajas, Multiplicaciones y Consultas. No hay consenso acerca de si debe haber un único caso de uso para hacer altas, bajas, modificaciones de una determinada entidad, o si va a haber un caso de uso para el alta, otro para la baja, otra para modificaciones, otra para la consulta. Ahí depende, claramente, del contexto y de cuánto se complique el modelo. Lo que sí es importante es que debe haber casos de uso que permitan custodiar, mantener, las entidades del modelo de dominio o las entidades del modelo de datos, dependiendo que otro modelo hayamos construido para representar los objetos del negocio.

[V / F] Todas las entidades y asociaciones referenciados en los casos de uso deben aparecer en el modelo de dominio o modelo de datos.

## Casos de uso en el proceso de desarrollo

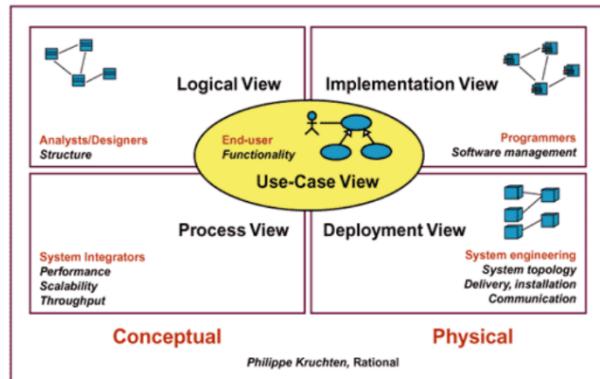
### Proceso unificado

Bueno hablaremos un poco de la historia de los casos de uso. Los casos de uso se popularizan a comienzos de la década de 1990 de la mano de Ivar Jacobson. Ivar Jacobson había escrito un libro muy conocido en esa época llamado “Object-Oriented Software Engineering” en el que además de presentar esta herramienta como una pieza central, describía una serie de disciplinas de ingeniería de software que, con el tiempo, constituirían un modelo de proceso de desarrollo de software llamado en esa época “objectory”, qué más tarde se incorporó al proceso unificado de desarrollo de software desarrollado por Booch, por Rumbagh, por Jacobson, los 3 también autores de UML. Este proceso unificado tenía o tiene una versión comercial llamada Rational Unified Process y actualmente IBM Rational Unified Process.



## Proceso unificado

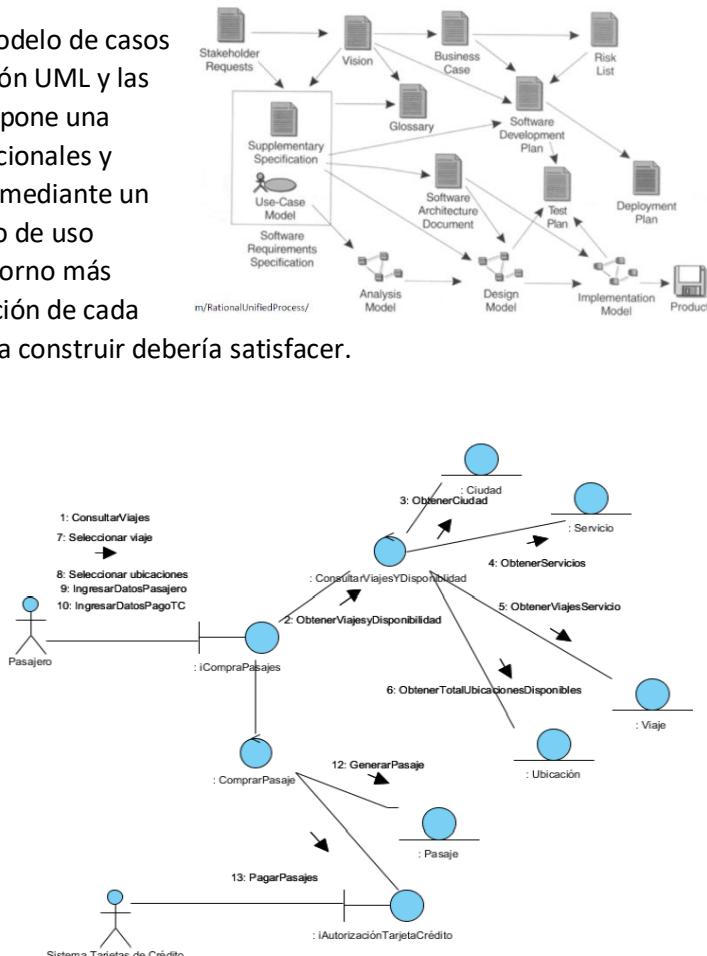
En el proceso unificado claramente los casos de uso juegan un papel fundamental, ya que son la piedra angular alrededor de la cual se organizan las diversas actividades y modelo. Philippe Krutchen, miembro del equipo de desarrollo del Unified Process, publicó un famoso paper acerca del modelo de arquitectura de software, conocido popularmente como “4+1”. En este paper, Krutchen sostiene que para poder representar adecuadamente los distintos aspectos de la arquitectura de un producto de software, es necesario emplear más de una vista, más de una perspectiva. Cada una de estas perspectivas está enfocada en un aspecto particular. La vista funcional, la vista central o la que identificamos como la “+1” en el modelo “4+1”, es representada en Unified Process mediante casos de uso.



Esta perspectiva se presenta en el Unified Process mediante el modelo de casos de uso que está compuesto por uno o varios diagramas en notación UML y las especificaciones de cada uno de los casos de uso. También se propone una especificación suplementaria para poner ahí los requisitos no funcionales y quizás algún otro requisito funcional que no se pueda especificar mediante un caso de uso. Esa especificación suplementaria y el modelo de caso de uso constituyen la especificación de requisitos de software. En un entorno más tradicional, esto sería simplemente un documento con la descripción de cada uno de los requisitos funcionales y no funcionales que el sistema a construir debe

## Modelo de análisis en el proceso unificado

Con esta información del Unified Process, propone desarrollar el **modelo de análisis que es una descripción de lo que ellos llaman la realización de cada caso de eso**. Una realización es la descripción del comportamiento interno del sistema en términos tecnológicamente neutros, necesario para elaborar las respuestas a las acciones de los actores que se describen en los casos de uso. En este modelo, el comportamiento se describe mediante lo que en Unified Process se llama objetos de análisis, que son abstracciones que tienen propiedades y comportamiento. Colaboran entre sí para elaborar las respuestas que necesitan los actores. Estos objetos de análisis pueden ser: entidades, equivalentes a nuestros objetos de dominio o entidades de datos, pero con el agregado de comportamiento; objetos de interacción, que son las interfaces con el mundo exterior; y objetos de control.



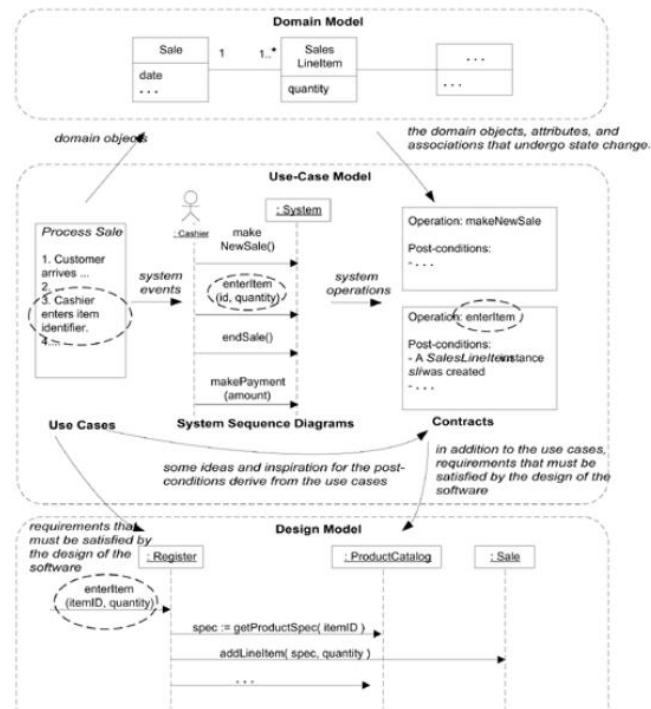
Lo que vemos en el gráfico es, para un caso de uso, cómo colaboran una serie de clases de análisis para poder elaborar las respuestas a cada una de las acciones del actor. A partir de este modelo de análisis se puede pasar al modelo de diseño en donde lo que vamos a encontrar son equivalentes, clases de diseño equivalentes, a estas clases de análisis que estamos viendo en el gráfico.

[V / F] El modelo de análisis representa una realización de un caso de uso.

# Modelo de casos de uso

à la Larman

Una alternativa a este enfoque es el que propone Craig Larman, un autor muy reconocido y autor de varios libros de textos clásicos respecto al tema de procesos unificado y UML. Larman propone desarrollar, en lugar de un modelo de análisis como el propuesto por el Unified Process, **un modelo de caso de uso más completo que, además de los casos de uso propiamente dicho, contiene contratos y operaciones**. Todo esto constituye luego el input para las actividades de diseño. En esta propuesta nos encontraremos con diagramas de secuencia a nivel de sistema y contratos para cada una de las operaciones.



## Contratos y operaciones

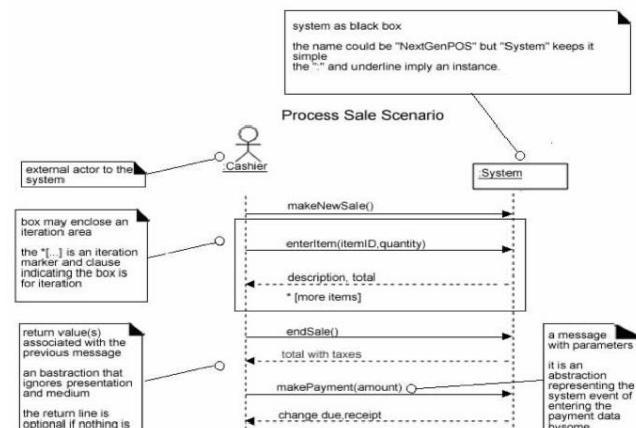
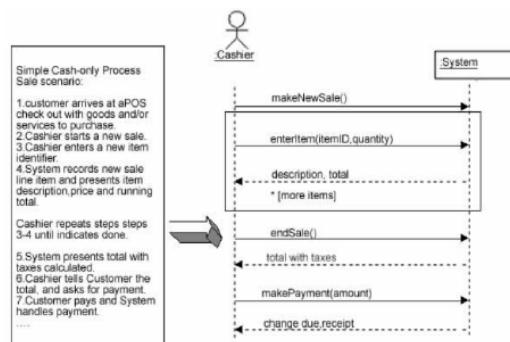
Entonces para cada caso de uso nos vamos a encontrar con lo siguiente:

- ❖ Una operación se va a disparar como resultado de un **evento**, es decir, de una acción del actor. Cuando vemos una acción del actor en el caso de uso, en definitiva, para este modelo vamos a encontrarnos con un evento.
- ❖ Como resultado de la operación se pueden producir **cambios** en los objetos de **dominio**. Nuevamente, el modelo de dominio es una representación de los objetos y asociaciones del dominio del problema no es un modelo de base de datos. Entonces, cada vez que hay un evento, esos cambios pueden impactar en la creación de nuevas instancias o la creación de nuevas asociaciones entre objetos. Por ejemplo, si estamos describiendo el caso de uso correspondiente a la inscripción de un alumno en un curso, probablemente "Alumno", "Curso" sean entidades del modelo de dominio y, probablemente, en ese caso de uso, se cree una instancia de un objeto de dominio llamado "Inscripción".
- ❖ Cada operación, al menos las más complejas, debe ser definida mediante un **contrato**.
- ❖ Lo que siempre tenemos que acordarnos, es que estamos hablando de la **esencia** del sistema, no estamos hablando de aspectos de implementación.

## Diagrama de secuencia del sistema

Aquí podemos ver el ejemplo un poco más detallado. El sistema nuevamente es una caja negra, no describe cómo se elabora la respuesta. Fijémonos que en este diagrama de secuencia lo único que tenemos es el actor y el sistema. Las acciones del actor se representan como un mensaje que se envía del actor al sistema, las respuestas se dibujan como una flecha del sistema hacia el actor,

como un mensaje del sistema hacia al actor (ahí cuando hace falta hacerlo, sino la respuesta está implícita).



Aquí tenemos otro ejemplo, tomado del libro del Larman. A la izquierda vemos el caso de uso, y para cada acción, vemos a la derecha como se dibujan los eventos del sistema.

## Operaciones del sistema

Decíamos antes que cada evento va a disparar una operación. Esa operación es una operación si se quiere abstracta, conceptual. La vamos a necesitar especificar mediante un contrato. El contrato de operación va a incluir:

- Nombre de la operación y parámetros.
  - Responsabilidades
  - Precondiciones: estado del sistema o de los objetos del dominio antes de la ejecución de la operación.
  - Postcondiciones: estado de los objetos en el modelo de dominio luego de hacerse ejecutado la operación (nuevas instancias o asociaciones creadas, cambios a los atributos, instancias o asociaciones destruidas).
- |                   |   |
|-------------------|---|
| Contract Name:    | enterStudentIdentifier<br>(studentIdentifier)   |
| Responsibilities: | Accept and validate the Student's identifier.<br>Verify that the Student is eligible to register.   |
| Type:             | System  |
| Exceptions:       | If the student identifier is not valid, indicate that it was an error.<br>If the Student is not eligible to register, inform the Student. |
| Preconditions:    | Student is known to the system.   |
| Postconditions:   | None  |

Repetimos que esta no es una descripción funcional, es una descripción esencial. Entonces, con esta descripción lo que estamos haciendo es complementando un poco qué es lo que hace el sistema. Convengamos que el modelo de casos de uso, los casos de uso en general, al tratar a los sistemas como una caja negra, no nos cuentan que es lo que debería estar pasando del lado de adentro del sistema, independientemente de cómo lo implementemos. Hay un comportamiento que el sistema tiene, para elaborar esas respuestas, que no tiene que ver con la implementación, tiene que ver con la lógica del sistema. Entonces si yo consulto los cursos disponibles, y el sistema me responde cuáles son los cursos disponibles, ¿cómo hago yo para elaborar esa respuesta? Bueno, el sistema se tendrá que fijar cuáles son los cursos que están dentro de la oferta académica, cuál es la materia que yo estoy consultando, cuál es la materia de los cursos que quiero conocer y voy a tener que ver cuáles son los cursos que están habilitados, cuáles son los horarios, etc. Bueno, todo eso, si yo lo veo en un caso de uso, no me describe cómo pasa eso dentro del sistema, entonces con este esquema que estamos presentando, con este tema de las operaciones, o con el modelo de análisis que proponen en Unified Process, estamos de alguna manera, construyendo una especificación de lo que debe hacer el software, mientras que con los casos de uso estamos, en general, planteando una visión más desde el punto de vista del usuario.

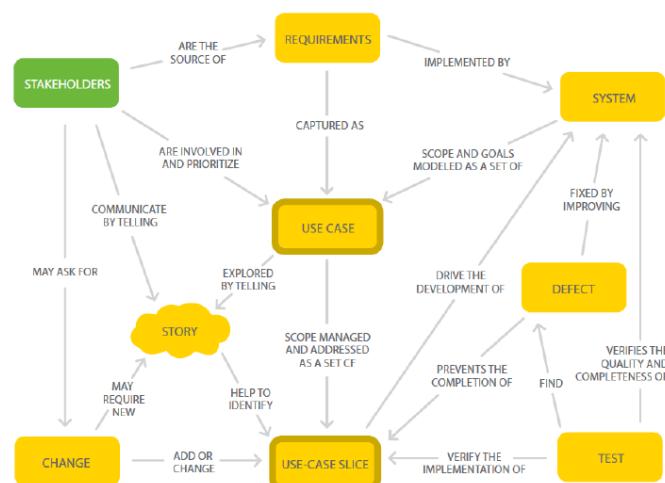
[V / F] En el modelo propuesto por Larman cada operación abstracta se especifica a través de un contrato que es una descripción esencial.

## Casos de uso

### 2.0

Si bien han pasado más de 20 años desde la invención de los casos de uso, aún se siguen produciendo desarrollos novedosos.

Recientemente Jacobson y su equipo, han impulsado la adopción de lo que han de llamar casos de uso 2.0. La novedad más importante es que los casos de uso en esta nueva versión ya no son un conjunto de escenarios, sino un conjunto de historias, como las historias de usuario, que ayudan a organizar las porciones o slice en las cuales se puede dividir un caso de uso para poder organizar la implementación. Todavía es bastante prematuro hablar del impacto que podría llegar a tener esta nueva versión. Claramente vale la pena profundizar un poco en los detalles.



## Conclusión

Bien para finalizar, hemos dado un panorama general de los casos de uso. Si bien parecen haber perdido protagonismo, hoy en día, todavía tienen su lugar en situaciones muy particulares. Por ejemplo, en aquellos casos en donde no se puede contar con un representante del usuario para que se siente junto al equipo de desarrollo. También, en situaciones contractuales; también en situaciones en donde los equipos están geográficamente dispersos. Nuevamente hay que utilizar muy cuidadosamente los casos de uso. No todos los casos de uso merecen ser especificados con un gran nivel de detalle. Hay que tomar decisiones con respecto cuáles valen la pena profundizar y cuáles no. Sí es importante identificar la mayor cantidad de casos de uso posibles, identificar las relaciones entre esos casos de uso y que los casos de uso guarden consistencia con el resto de los modelos que construyamos.

[V / F] La novedad de los Casos de Uso 2.0 es la incorporación de *historias* y *slices*.

## 4.2- IIBA y PMI

[V / F] Al momento de llevar adelante el análisis de riesgos, las organizaciones deben contar con una estrategia que minimice el riesgo.

[V / F] La técnica de budgeting consiste en priorizar requerimientos cuando el tiempo es un recurso fijo

Según la técnica de MoSCoW,

- Los requerimientos de tipo "Could" son necesarios en la solución final
- Un requerimiento considerado crítico para el usuario puede ser catalogado bajo la categoría "Should"**
- Un requerimiento catalogado bajo la categoría "Won't" es un requerimiento que se decidió que satisfacerlo traería más problemas que beneficios.
- Todos los requerimientos catalogados como "Should" deben estar satisfechos al momento de salir a producción.

[V / F] Una técnica para priorizar requisitos consiste en asignar un peso a una serie de criterios (por ejemplo usabilidad, riesgo, valor para el usuario) y luego evaluar cada uno de los requisitos para terminar seleccionando los que mayor valor suman.

Una estimación arrojó que cumplir con el requisito A lleva 5 semanas, con el requisito B (el más valioso) lleva 7 semanas y con el requisito C se necesitan 4 semanas. Para todos los casos el tiempo es del equipo completo. El equipo escuchó de una técnica llamada "Timeboxing All in" y desean aplicarla ya que sólo cuentan con 15 semanas para llevar adelante el proyecto. Por lo tanto:

- Incluirán los requisitos A, B, C dentro de las tareas a realizar. Como el total excede el tiempo con el que contaban, solicitarán una prórroga de 1 semana adicional.
- Incluirán los requisitos A, B, C dentro de las tareas a realizar. Luego de ver que el total excede a la capacidad, definirán si se saca del alcance inicial el requisito A o el C.**
- Incluirán el requisito B que es el más valioso dentro de las tareas a realizar. Como todavía les queda capacidad, elegirán el requisito A. Luego, como no queda capacidad adicional, dejarán el requisito C para una iteración futura.

## 4.11- Reglas de negocio

Indicar de qué tipo son las siguientes reglas de negocio:							
	Hecho	Restricción	Habilitador de acción	Inferencia	Cálculo	No es regla de negocio	Puntuación
Si no se recibe el pago a los 30 días corridos de emitida la factura, el cliente entra en mora				✓			
Todas las facturas deben discriminar los impuestos aplicables	✓						
La aplicación debe estar disponible 7x24							✓
Solamente los analistas de mesa de ayuda pueden crear incidentes		✓					
Superada la velocidad de 10 km/h, se debe activar la alarma si el conductor no tiene colocado el cinturón de seguridad.			✓				
El impuesto a los ingresos brutos se calcula como el 3.5% del importe neto de cada factura.						✓	