Introducción a la ingeniería de software

Tabla de contenido

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:

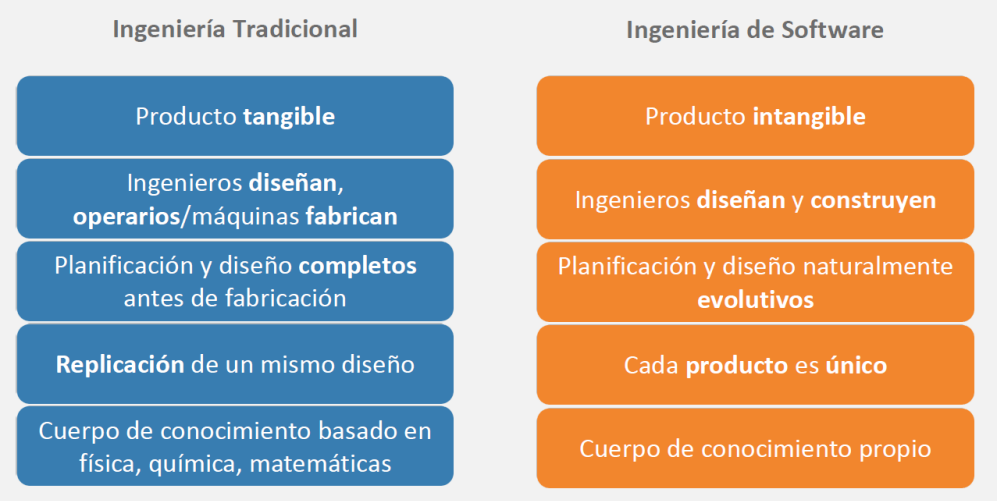
> *La **ingeniería** es la creación de soluciones costo efectivas a problemas prácticos mediante la aplicación de conocimiento codificado para construir cosas al servicio de la humanidad.*

> *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.[1]

[1] 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.[2]

Hace ya más de 30 años, Frederick Brooks analizó en un artículo famoso llamado “No silver bullet”[3] 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.

[2] La ingeniería de software NO se asemeja a la ingeniería tradicional.
[3] “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 atrás acuño 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 vídeos), 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.

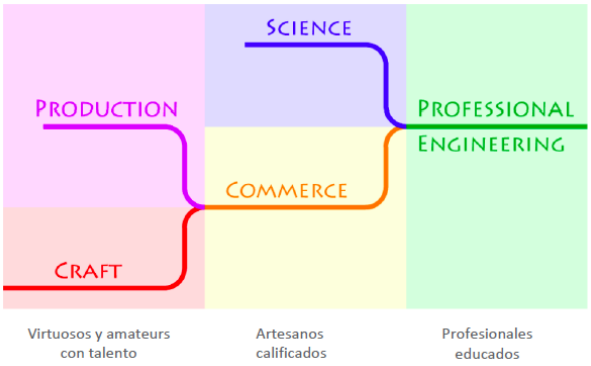
Lo que complica el panorama es que no todos los proyectos de desarrollo de software terminan bien. Se calcula que alrededor del 30% de los proyectos no cumple con alguno de sus objetivos de costo, duración y resultado. Por eso es sumamente crítico aprender a gestionar, a manejar adecuadamente los esfuerzos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

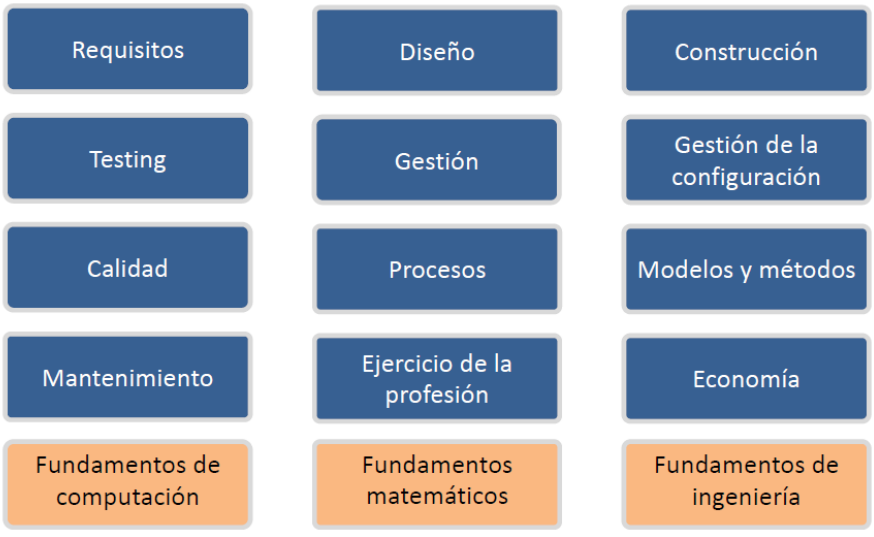


Un ejemplo interesante es el de la construcción de puentes, en el ámbito de la ingeniería civil. Al comienzo, la construcción de puentes era meramente artesanal. No se conocía prácticamente nada de las propiedades de los materiales, y mucho menos de mecánica de fluidos. Con el tiempo se fue desarrollando 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.[4]

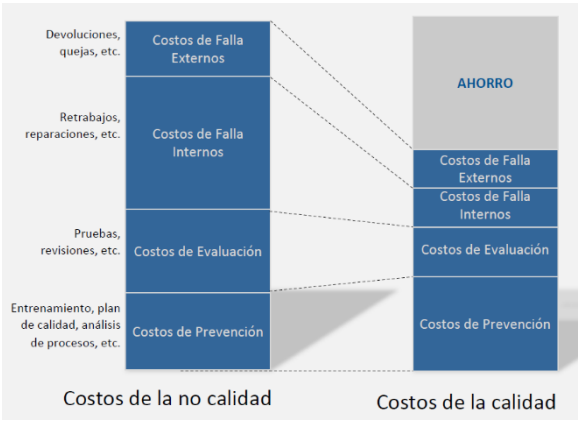
Procesos

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.

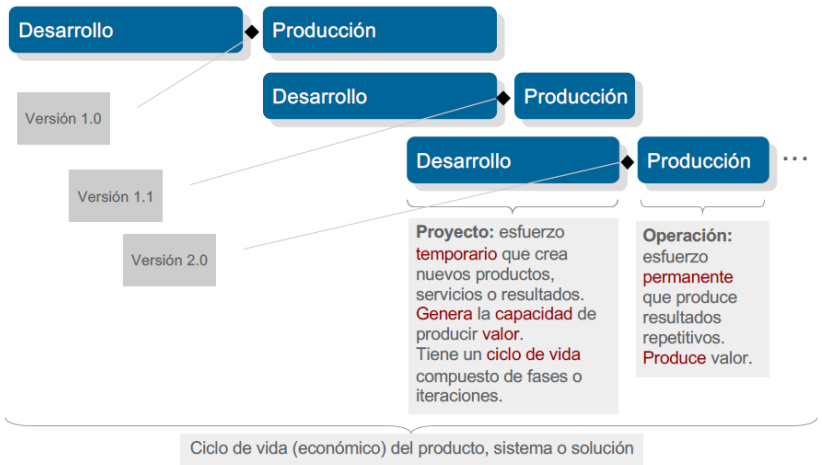
Calidad y procesos

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



[4] 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 diferenciemos 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 temporario 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.

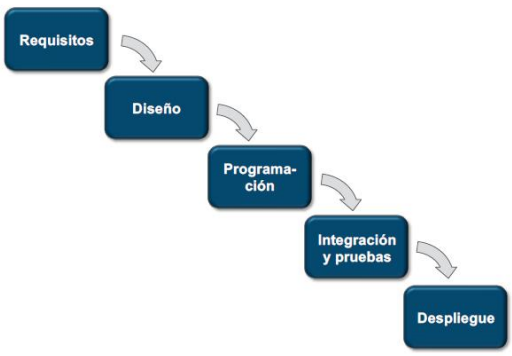
Desarrollo de software

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 imagínense 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.

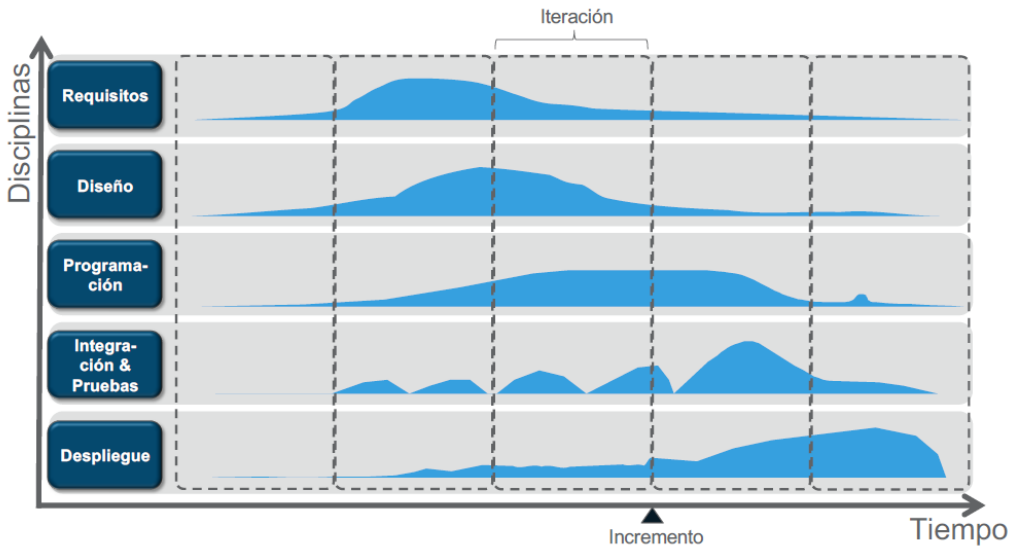
Casi todo es diseño

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 panificable, un poco caótico porque naturalmente el diseño es así.

Proceso de desarrollo de software

Disciplinas y ciclo de vida

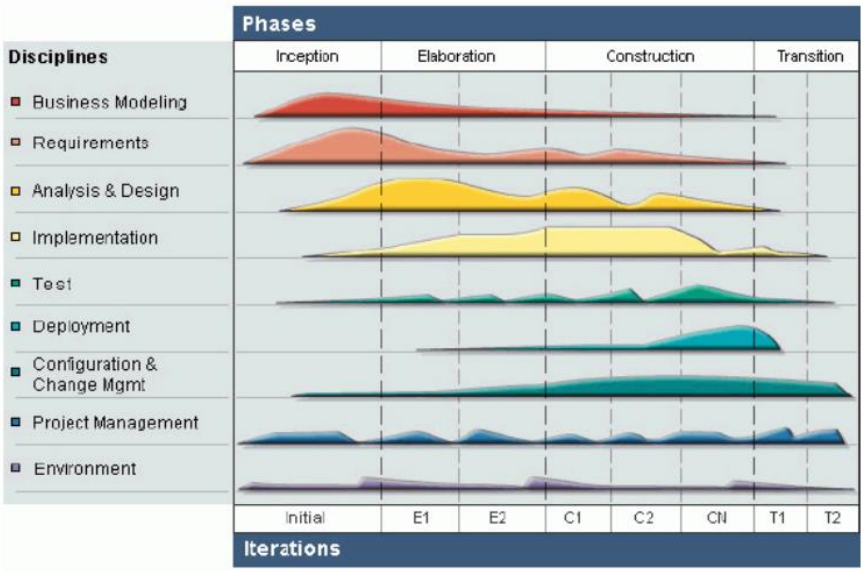
En lugar de encontrarnos con actividades secuenciales, con fases secuenciales, lo que nos vamos a encontrar en el desarrollo de software es que las actividades del cuerpo de conocimiento se ejecutan con distinto grado de intensidad a lo largo de todo el ciclo de vida.



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 folloving a plan

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

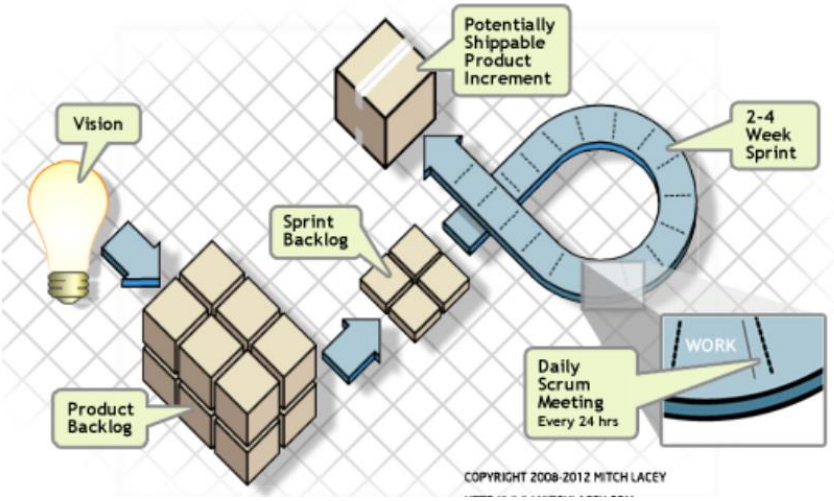
Esto se hace todavía más presente cuando, ya empezado este siglo, aparece el manifiesto ágil que da como resultado una declaración que es este famoso manifiesto en donde se declara que: “Es más importante o se valoran más los individuos y las interacciones, que los procesos y las herramientas. El software 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.

[V / **F**] El manifiesto ágil no promueve la planificación.[5]

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.

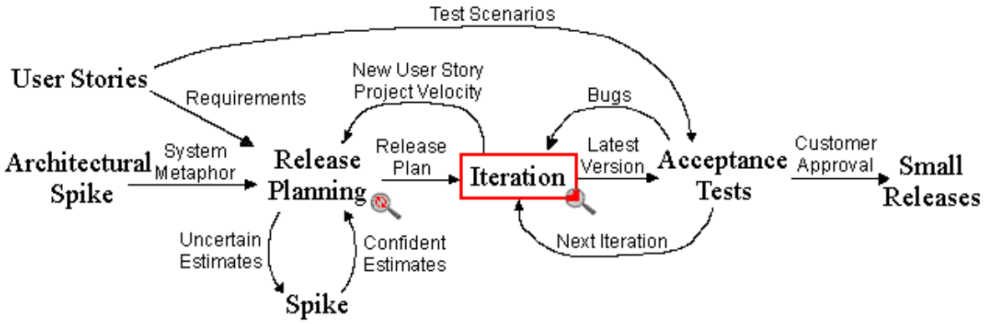
También hay una serie de actividades, rituales o ceremonias diarias. El día comienza con una reunión llamada "Daily Stand up Meeting". Es una reunión corta en donde la gente está de pie y en donde se plantea: qué fue lo que se hizo el día anterior, qué es lo que se va a hacer durante el día y se plantean 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

[5] 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.

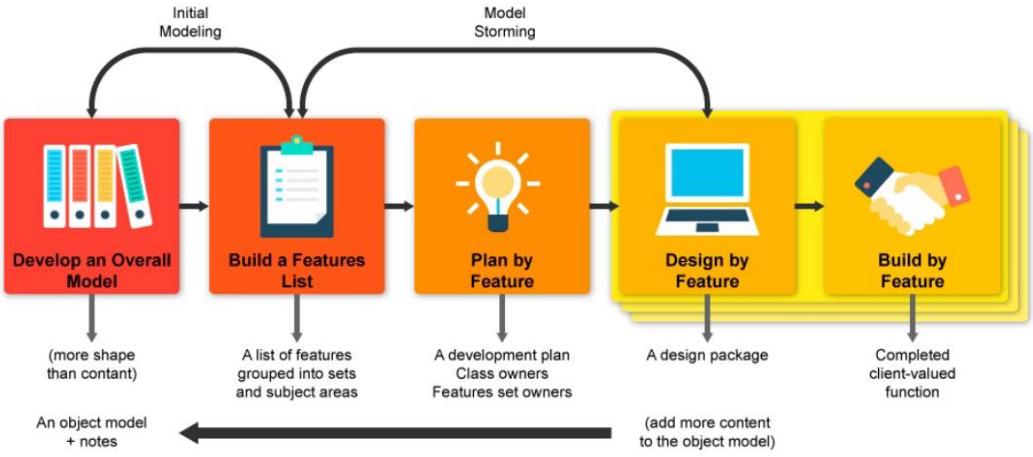
Lo otro que también propone scram es que, al final de cada sprint, se revise el sprint backlog, se hace una retrospectiva del sprint para ver qué fue lo que funcionó, qué fue lo que no funcionó, qué cosas hay que mejorar en el proceso de desarrollo, se decide cuáles son los elementos del backlog que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

Extreme Programming (XP)

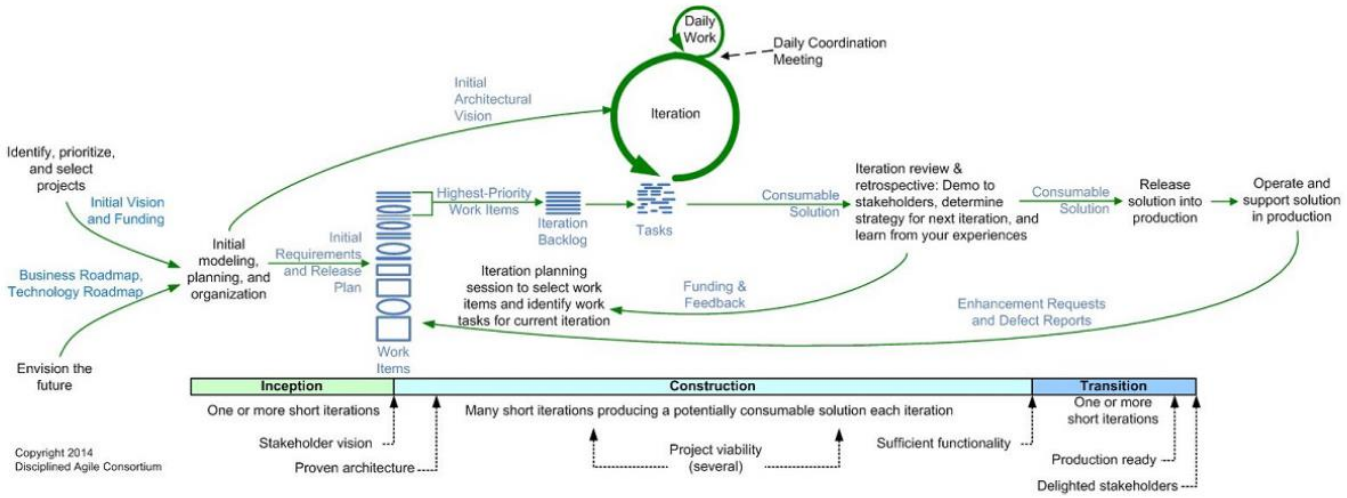


Hay otros procesos similares como XP (**Extreme Programming**) que también propone un desarrollo iterativo e incremental en donde es justamente, como el nombre se indica, se toman al extremo algunas prácticas que tienen que ver con algunas prácticas de desarrollo muy puntuales. Con extreme Programming 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 scram 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.[6]

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

- ☐ Cascada[7]
- ☑ **Scrum**
- ☑ **RUP (Rational Unified Process)**

[6] Un proceso es un conjunto de actividades que transforma una entrada en una salida y que consume recursos.
[7] Cascada es de tipo secuencial.