# Desarrollo de software

Veremos en este apartado las características propias del software y de la forma de desarrollarlo que lo hacen un elemento único, difícil de comparar con ingenierías tradicionales como la construcción.

## Software

El software es un invento muy reciente de la humanidad. Fue a mitad del siglo XX cuando se empezaron a utilizar los primeros computadores electrónicos programables en organismos oficiales y grandes empresas. Y los primeros lenguajes de programación de alto nivel (FORTRAN y Lisp) se desarrollaron a finales de los años 50 (ver detalles en https://github.com/domingogallardo/historia-computadores).

La programación, por tanto, es una profesión relativamente joven con poco más de 60 años. Todavía están vivos (¡y activos!) algunos programadores que trabajaron con los primeros computadores como Donald Knuth con el IBM 650. Comparémoslo con profesiones o disciplinas como medicina, física, derecho o arquitectura, todas ellas con cientos de años de antigüedad.

Podemos decir que el software es el invento más importante y complicado de la historia de la humanidad. El software ha conquistado el mundo y está presente en todos los aspectos de la vida actual. El entretenimiento, la medicina, la economía, etc. dependen cada vez más de complejos sistemas de software desarrollados y mantenidos por ingenieros/as como vosotros/as. Detrás de cualquier sistema de software está un conjunto de personas que ha tenido que interactuar, decidir, entender y modificar decenas de miles de líneas de código.

Conoces innumerables sistemas software de todo tipo: Instagram, Android, iOS, Netflix, Chrome, Gmail, Meet, Excel, Word, Radar COVID y un largo etcétera.

Algunos ejemplos de sistemas software que vamos a usar en algún momento en la asignatura:

- Sistema de **gestión académica** de un centro educativo: sistema con el que los estudiantes pueden matricularse, consultar horarios, consultar expediente, etc.
- Sistema de gestión de **flotas de ambulancias**: sistema con el que se puede controlar la ubicación de las ambulancias y gestionar trayectos de las mismas.
- **Quiosco de consulta** para una tienda estilo Fnac: sistemas con el que los clientes pueden consultar productos y buscar su ubicación.

Hoy en día, gracias a servicios como GitHub, podemos echar un vistazo al código de muchos de estos sistemas software y a su evolución. Algunos ejemplos:

- CartoDB. Software español para representación visual de datos geográficos.
- Guice. Framework de inyección de dependencias en Java.
- swift-nio. Framework asíncrono de entrada-salida en Swift.
- Spring Boot. Framework web en Java
- Swift. Compilador y librería stándar de Swift. Escrito en C++ y Swift.

## Metáforas

Todos conocemos el concepto de metáfora en la literatura, una figura del lenguaje en la que se utiliza un objeto o acción en otro contexto, aprovechando una comparación tácita. Por ejemplo, cuando se dice "el

otoño de la vida" o "tener los nervios de acero" se están utilizando metáforas.

Sin embargo, las metáforas son mucho más que esto. No solo se utilizan en literatura, sino que también tienen aplicación en una gran variedad de campos: economía, empresa, publicidad, política, ciencia o psicología.

Utilizamos las metáforas para explicar algo desconocido en términos de lo que ya conocemos. Por ejemplo, cuando se dice:

> "El proceso de lanzar un nuevo producto es una maratón"

Al relacionar conceptos desconocidos con situaciones usuales ya vistas estamos arrojando nueva luz sobre estos conceptos desconocidos (¡esto ha sido una metáfora!).

En la metáfora anterior lo desconocido lo comparamos con una habitación a oscuras que podemos explorar con la luz con una linterna. La metáfora nos da una indicación de en qué consiste el proceso de investigar algo. Mediante ella nos damos cuenta de la importancia de la dirección en la que estamos apuntando la linterna y de la necesidad moverla en varias direcciones. La metáfora nos lleva entonces a pensar que para conocer algo desconocido hay que estudiarlo desde distintos puntos de vista, siempre cambiando el detalle de lo que estamos estudiando, hasta poder conocerlo en su totalidad (haber iluminado toda la habitación).

La metáfora anterior puede o no ser correcta. Lo importante es que al formularla hemos tenido una representación de algo desconocido (el proceso de investigar) basada en algo conocido (explorar una habitación oscura con una linterna). Y esa representación es muy importante, porque nos lleva a tomar acciones y decisiones reales que podrían haber sido distintas si hubiéramos usado otra metáfora.

## Metáforas en la cultura de las organizaciones

Las metáforas son una parte importante de la cultura de una organización. Una cultura proporciona un conjunto de reglas invisibles que todos cumplimos de forma casi automática. Estas reglas proporcionan un contexto común y favorecen que todos nos movamos en una dirección similar.

Las metáforas, expresiones o giros del lenguaje proporcionan uno de los elementos básicos de la cultura de la organización.

Por ejemplo, una organización centrada en la competición y con una dinámica combativa con sus adversarios utilizará metáforas relacionadas con la guerra o los deportes:

> "Hemos ganado este combate"
> "Nos lo jugamos todo en esta campaña"
> "Es una oportunidad de vida o muerte"

Sin embargo, una organización orientada a la cooperación y al planteamiento de situaciones de "win-win" en las negociaciones utilizará otro tipo de metáforas:

> "Hemos coreografiado perfectamente la puesta en marcha del producto"
> "Todos debemos participar en este viaje"
> "Nuestro ecosistema premia la lealtad de nuestros clientes"

Escoger las metáforas correctas es, por tanto, fundamental para establecer una cultura, un estilo común de trabajo en el equipo o en la empresa.

## Metáforas de desarrollo de software

¿Por qué es interesante hablar de metáforas para referirnos al software? Porque podemos utilizarlas para crear una cultura, una visión, sobre nuestro trabajo, aplicable tanto a los compañeros del equipo como a los managers y directivos con los que interactuamos.

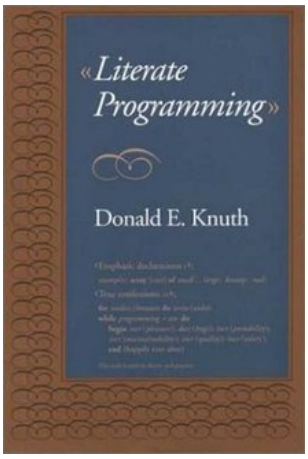
El hablar sobre metáforas del desarrollo de software nos sirve también para argumentar en contra de prejuicios o metáforas erróneas que se pueden tener a priori. Metáforas que nosotros sabemos que no son correctas, pero que están muy implantadas en la mentalidad de mucha gente.

Veamos un listado de las metáforas que se han ido utilizando para el desarrollo de software. Podemos encontrarlas en muchos sitios. En el libro de Steve McConnell *Code Complete*, se dedica un capítulo completo a hablar de ellas.

> "A confusing abundance of metaphors has grown up around software development. David Gries says writing software is a science (1981). Donald Knuth says it's an art (1998). Watts Humphrey says it's a process (1989). P. J. Plauger and Kent Beck say it's like driving a car, although they draw nearly opposite conclusions (Plauger 1993, Beck 2000). Alistair Cockburn says it's a game (2002). Eric Raymond says it's like a bazaar (2000). Andy Hunt and Dave Thomas say it's like gardening. Paul Heckel says it's like filming Snow White and the Seven Dwarfs (1994). Fred Brooks says that it's like farming, hunting werewolves, or drowning with dinosaurs in a tar pit (1995). Which are the best metaphors?"

Entre las metáforas más usadas se encuentran las siguientes.

**Escritura**




El software es algo que debe ser escrito y leído, de forma similar a como escribimos una carta, un libro o un manual de instrucciones.

Esta metáfora se lleva al extremo por los practicantes de *literate programming*, un paradigma de programación en el que el código queda embebido en una explicación en lenguaje natural del problema que se está resolviendo. Donald Knuth fue el fundador de este paradigma con su libro *Literate programming*.

Elementos positivos de esta metáfora es que resalta el que el código debe ser legible y entendible por los compañeros. El código se escribe para las personas y no sólo para los computadores. Tan importante como escribir el código es leerlo después. De hecho, para poder modificar código debemos entenderlo, y para entenderlo debemos leerlo, hablar sobre él, etc.

La metáfora no representa bien el carácter colectivo del desarrollo de software. Habitualmente la escritura es un hecho individual, mientras que el desarrollo es una labor de equipo. Tampoco comunica correctamente el carácter evolutivo y cambiante del software. Cuando un libro se termina de escribir muy raramente se modifica. Quizás sí se hace en una siguiente edición de un manual de usuario o un libro técnico, pero eso conlleva otra vez meses de esfuerzo. En el caso del software es cambio es continuo y de un día para otro. Además el software se escribe apoyándose en otro software que previamente se ha desarrollado y depende de él. Estas dependencias tampoco se recogen en la metáfora.

**Jardinería**

En su libro The Pragmatic Programmer David Thomas y Andrew Hunt repasan la metáfora de la jardinería.

> "Software is more like gardening—it is more organic than concrete."

El crecimiento y cuidado de un jardín es algo orgánico. Se plantan inicialmente todas las cosas de acuerdo a un plan. Algunas de las plantas crecen bien pero otras no, y las destinamos a compost. Podemos mover las plantas y recolocarlas, para aprovecharnos de las relaciones entre ellas, de las luces y las sombras, del viento y la lluvia. Tenemos que podar o separar las plantas que crecen demasiado. Y podemos recolocarlas también por motivos estéticos, para combinar mejor los colores. Recogemos las semillas y fertilizamos las zonas que necesitan ayuda extra. Continuamente monitorizamos la salud del jardín, y hacemos ajustes (a la tierra, las plantas, la disposición) conforme se necesita.

La metáfora recoge bien los aspectos evolutivos del software, pero no la escala temporal. El desarrollo de software es mucho más dinámico y rápido que el crecimiento del jardín.

A priori podría ser difícil encontrar una correspondencia entre la parte parte del crecimiento orgánico del jardín (semillas y plantas que crecen por si mismas, en configuraciones poco predecibles) y el desarrollo de software. El software no crece si se deja solo. ¿Y qué es el viento, la lluvia o las problemas de la naturaleza? En el desarrollo de software tenemos todo controlado (hacemos tests) y no dependemos elementos externos.

¿De verdad es así? ¿Está todo controlado? Si pensamos de esta forma nos estamos olvidando del elemento fundamental del desarrollo de software: los usuarios finales. Cuando dejamos el software desarrollado en sus manos lo van a utilizar como ellos mejor sepan. Lo van a intentar introducir en sus procesos de trabajo ya consolidados. Y el software les va a proporcionar valor o no dependiendo de lo bien que se adapte a esos procesos.

Los usuarios finales son el viento, la lluvia, las fuerzas de la naturaleza del jardín. La única forma de saber si el software va a proporcionarles valor es dejarlo crecer entre ellos. La metáfora del jardín no es tan descabellada como parecía al principio.

**Crecimiento por acreción**

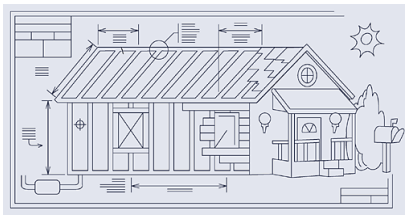
Una metáfora poco conocida, pero que da bastante juego, es representar el desarrollo de software como la formación de una perla dentro de una ostra. La perla se forma dentro de una ostra por acreción. Va creciendo a base de añadir material a un grano de arena que se introduce en la ostra. Al final se forman un conjunto de capas que van rodeando el núcleo inicial.

De forma similar, podemos ver el desarrollo de software como una tarea incremental en la que se van añadiendo funcionalidades a un núcleo central inicial. Es lo que se denomina desarrollo evolutivo o incremental. Primero se desarrolla un esqueleto que trabaja con datos de prueba y después se van añadiendo los músculos y la piel, los detalles exteriores con los que el usuario interactúa. Los datos de prueba se convierten en datos reales y ya tenemos un sistema real que puede ponerse en producción.

Un elemento que no recoge bien esta metáfora es que en un sistema software real tenemos que construir muchas "perlas". Conforme extendemos el sistema debemos ir ampliando también las funcionalidades

proporcionadas por el esqueleto, añadiendo nuevos elementos centrales que deben interactuar con los ya existentes.

**Construcción**



Una de las metáforas más usadas es la construcción. Desarrollar software es como construir una casa. Hablamos de "construir software", de "hacer una estructura resistente", de "planos con el diseño de las características" o de "arquitectos de software".

El proceso de construcción de una casa se puede simplificar en los siguientes pasos:

1. El arquitecto diseña los planos con todo detalle.
2. Los albañiles, fontaneros, electricistas, etc., dirigidos por el arquitecto técnico, construyen todos los elementos del edificio de acuerdo con los planos proporcionados.
3. Los propietarios se mudan al edificio, viven felices y, si detectan algún problema, llaman a mantenimiento para arreglarlo.

Esta es una metáfora que ha causado mucho daño al desarrollo de software. El modelo de cascada (*waterfall*) parte de esta metáfora. Según este enfoque, el desarrollo de software es como cualquier otra ingeniería tradicional en la que el esfuerzo más grande hay que hacerlo en la fase inicial de diseño. La fase de construcción del producto (implementación) se hace una vez que se ha concluido todo el diseño y consiste en seguir al pié de la letra los diagramas UML que detallan todo el diseño. El trabajo de los desarrolladores es un trabajo nada creativo porque todo el diseño ya se ha hecho de antemano. Solo tienen que pasar a código los diagramas y diseños que han realizado los "arquitectos de software". Y al final hay una fase de mantenimiento en la que se arreglan los problemas que detectan los usuarios cuando empiezan a usar el software.

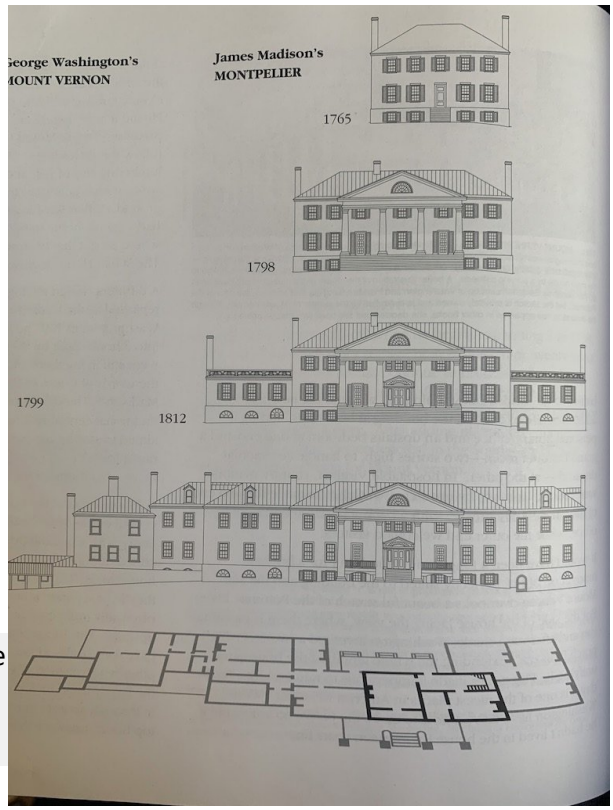
El problema de este enfoque es que la mayoría del software que se desarrolla no es así. Veremos más adelante una argumentación de Martin Fowler que critica este enfoque.

A pesar de esto, la metáfora tiene cosas buenas. Construir software conlleva realizar una planificación, una preparación y una ejecución. Similar a construir algo. También, igual que hay distintos tamaños y alcances en los sistemas de software hay distintos tamaños y alcances en las construcciones. No es lo mismo construir un edificio de 10 plantas, que una casa de campo en una parcela, que una caseta para el perro. Si nos equivocamos al construir una caseta para el perro y se nos olvida hacer la puerta, podemos rehacerlo todo sin problemas. Sólo habremos gastado una tarde. En el caso de un proyecto más grande hay que tener mucho más cuidado en que los fundamentos y estructuras sean sólidas y extensibles.

Otras cosas interesantes de la metáfora es la distinción entre estructura y forma. Un edificio tiene elementos estructurales como cimientos, vigas, pilares, muros, etc. sobre los que se construyen los aspectos más visibles como azulejos, pintura, ventanas, etc. En el caso del software, tenemos también elementos estructurales, código que no es visible para los usuarios, y elementos visibles como la interfaz de usuario o el comportamiento de las funcionalidades.

La diferencia entre construcción y software es que podemos modificar ambos tipos de elementos al mismo tiempo. En construcción una vez realizada la estructura es difícil que sea modificable. En el desarrollo de software, sin embargo, no solo podemos, sino que debemos realizar cambios en la estructura para hacerla

más flexible y para que podamos introducir más fácilmente nuevos cambios en el futuro. Es lo que denominamos refactorizar.




Una cosa poco reconocida en la arquitectura son las modificaciones continuas que se producen en los edificios. Un hilo de Twitter muy interesante de Geoffrey Litt reseña el famoso libro "How buildings learn" de Stewart Brand. Los paralelismos entre el software y la evolución de los edificios son notables. La diferencia está en la escala temporal. Vemos a la derecha la evolución de una mansión construida en 1765 y su evolución durante 50 años. Y lo ideal es que estos cambios sean determinados por el uso de los propios habitantes del edificio. Como dice Brand:

> ""Age plus adaptivity is what makes a building come to be loved. The building learns from its occupants, and they learn from it."

¿Os suena a algo?

A pesar de ser una metáfora que tiene elementos interesantes, la metáfora de la construcción falla en el aspecto de no recoger el carácter único del software como algo fácilmente modificable. El software es mucho más moldeable que cualquier otro material. Cambiar software es mucho más fácil que cambiar un tablón, o unos ladrillos, o unas tuberías. Esta flexibilidad del software es la que permite enfoques de construcción alternativos que no pueden ser usados en la construcción ni en otras ingenierías tradicionales.

**Otras metáforas**

Otras metáforas se corresponden con otros aspectos del desarrollo de software no contemplados en las anteriores.

Por ejemplo la metáfora de que hacer software es como **hacer una película de cine**. Con esta metáfora se enfatiza el aspecto creativo y multidisciplinar del desarrollo de software. Igual que para hacer una película es necesario coordinar el trabajo de múltiples profesionales técnicos y creativos (guionistas, actores, cámaras, vestuario, sonido, etc.), para hacer un buen producto software es necesario combinar también el trabajo de especialistas y diseñadores. Un problema de esta metáfora es que tiene un desarrollo similar al modelo de cascada y que cuando una película se termina no se vuelve a modificar. Sin embargo, la forma ágil de desarrollar software es entregando continuos incrementos y evolucionando el producto.

Otra metáfora es la **realización de una publicación periódica** como una revista. Cada semana o cada mes se entrega un nuevo ejemplar de la publicación, con nuevo contenido que es consumido por el lector. Podemos encontrar un parecido con las entregas que se realizan al cliente al final de cada iteración. En estas entregas se incorporan al producto nuevas funcionalidades que deben ser probadas por los usuarios, de forma similar al nuevo contenido de la revista.

Una última metáfora muy interesante es la de Alaistair Cockburn que en su libro *Agile Software Development: The Cooperative Game* define el desarrollo de software como un **juego cooperativo** dirigido a conseguir un objetivo y realizado en grupo. El juego consiste en resolver un problema de invención y comunicación. Los

desarrolladores trabajan en un problema que no entienden completamente, en el que están incluidos emociones, deseos y opiniones y que cambian conforme el juego avanza. Los jugadores deben:

- Entender el espacio del problema
- Imaginar un mecanismo que soluciona el problema en un espacio de tecnología viable
- Expresar la construcción mental de la solución en un lenguaje ejecutable y construirla en equipo

El elemento principal de este juego son las ideas de la gente y la comunicación de estas ideas entre los participantes y al computador.

Se trata de una metáfora muy interesante centrada en el proceso de entender el dominio del problema a resolver y de crear la solución poco a poco. Quizás es una metáfora algo abstracta y no tiene en cuenta muchos elementos concretos relacionados con el cómo se pone en práctica ese modelo y cómo se realizan las correcciones, modificaciones y adaptaciones. Tampoco se distingue en el juego las diferencias entre las partes estructurales de la solución y las partes más orientadas a obtener una conducta específica en esa solución.

## Ninguna metáfora es completa: el software es único

Cada metáfora resalta unas características del desarrollo de software y ninguna recoge completamente todas ellas. Por eso es importante conocerlas todas y conocer sus ventajas e inconvenientes.

También podremos conocer en qué tipo de empresa o equipo de desarrollo estamos dependiendo de las metáforas que utilizan. Y podremos ser críticos si nos damos cuenta de que se utilizan de más las metáforas incorrectas en nuestro entorno. En ese caso podremos hacerlo notar y ofrecer metáforas alternativas que sirvan para corregir posibles sesgos en la percepción del cómo debería ser nuestro trabajo como desarrolladores.

El problema en encontrar la metáfora perfecta se debe a que el software es algo único y tiene características que no son comparables a ninguna otro producto previamente inventado por la humanidad. El software es fácilmente maleable, modificable. Se construye de forma colectiva, incremental. Para su construcción se combinan elementos externos (librerías de terceras personas) con elementos estructurales creados por nosotros y elementos de conducta que definen la interacción del mismo con los usuarios. Es ejecutable y usado por personas, creando sistemas complejos de interacción entre máquinas y humanos. Y, por último, también es publicable en sistemas como GitHub, en donde podemos leer y estudiar la organización y las líneas de código del sistema.

En resumen, podríamos decir que el desarrollo de software es la actividad humana más compleja existente. Más que pilotar un avión, hacer un diagnóstico médico o diseñar un edificio. Por eso tiene esa gran cantidad de metáforas asociadas. Y seguro que seguirán planteándose nuevas.

**Abrazar el cambio**

La característica de que el software es fácil de cambiar es positiva porque permite un modelo de desarrollo único basado en iteraciones cortas, evoluciones y adaptación de cambios cuando se reciben *feedback* de usuarios.

La posibilidad de cambio ha sido abrazada desde el principio por metodologías como XP y hoy en día cada vez más herramientas asumen esta característica y proporcionan funcionalidades orientadas a gestionar correctamente la evolución del software. Por ejemplo, los sistemas de control de versiones como Git permiten desarrollar software en equipo usando *commits*, ramas, versiones, solicitud de incorporación de cambios (*pull*

*requests*), etc. O los IDEs tienen herramientas que permiten analizar qué sucede si cambiamos una declaración de una variable o una definición de una función: podemos consultar todos los sitios donde se usa esa función y sus conexiones con otros módulos.

Esta posibilidad de cambiar fácilmente una línea de código de un sistema software no nos debe llevar a engaño. El software es muy fácil de cambiar, pero ese cambio en muchas ocasiones puede conllevar problemas y errores. Puede ser que con los cambios introduzcamos regresiones (cosas que antes funcionaban dejan de funcionar después de un cambio), *bugs*, confusiones debidas a problemas de comunicación, etc.

En su charla de 2020 Continued Learning: The Beauty of Maintenance, Kent Beck repasa las características fundamentales que influyen en que el cambio de un sistema software sea más o menos fácil. Una de las más importante es el acoplamiento (*coupling*) de sus elementos.

Hablaremos más de esto cuando hablemos de SOLID, pero vamos con un avance. Dos elementos están acoplados con respecto a un cambio cuando el cambio en uno de ellos obliga al cambio del otro. El problema de un sistema con muchos elementos acoplados es que un pequeño cambio en un sitio provoca una cascada sucesiva de cambios, que a su vez provocan otros cambios, y así hasta que se para la ola.

Esta es la causa de que sea más sencillo introducir cambios al principio del desarrollo. El sistema software tiene pocos elementos, y están poco acoplados. Conforme el sistema va creciendo, si no tenemos cuidado y vamos refactorizándolo (lo veremos más adelante) se van introduciendo más elementos y se van acoplando cada vez más.

Esto es uno de los aspectos de lo que se denomina **deuda técnica**: problemas de diseño que vamos introduciendo en nuestro software y que lo hacen que cada vez menos flexible y más rígido.

Una de nuestras obligaciones como desarrolladores ágiles es conseguir que el sistema que estamos desarrollando no se vuelva excesivamente rígido y pueda ser modificado sin demasiados problemas, aunque estemos en fases avanzadas del desarrollo. Veremos que la refactorización y el diseño SOLID son técnicas fundamentales para ello.

## El desarrollo de software no es una ingeniería tradicional

Hemos visto los problemas para capturar las características especiales del software y su desarrollo. ¿Qué es lo que hace al desarrollo de software especial? ¿Por qué es tan difícil construir el producto correcto y desarrollarlo? ¿Por qué no podemos aplicar las técnicas aprendidas en ingenierías tradicionales, como la construcción? Ya hemos visto algunas ideas en los apartados anteriores. Vamos a intentar responder a continuación estas preguntas con más detalle.

En su artículo de 2005 titulado *The New Methodology* Martin Fowler analiza las características del desarrollo de software, comparándolas con otras ingenierías más tradicionales e introduce un elemento de enorme importancia: la incertidumbre del desarrollo de software hace imposible aplicar métodos predictivos basados en planes.

### Planificación en ingenierías tradicionales

Las metodologías de desarrollo de ingenierías tradicionales, como la ingeniería civil, la construcción o la ingeniería mecánica, se basan en una planificación exhaustiva de todo el desarrollo del proyecto. En esta planificación se realiza en una primera fase un diseño completo del proyecto y posteriormente, una vez validado el diseño, se pasa a su realización.

Los ingenieros responsables de estos proyectos trabajan con diagramas con los que representan las decisiones de diseño alcanzadas. El tipo de materiales a utilizar en el puente, el tipo de estructura, los pesos que va a soportar, etc. Todas esas son decisiones creativas de diseño que quedan plasmadas en los planos y diagramas.

Estos planos contienen la totalidad de detalles del proyecto y, teóricamente, es posible planificar el tiempo y el presupuesto del proyecto a partir de ellos.

En una segunda fase, los planos se pasan a otro grupo de personas que son los encargados de realizar físicamente la construcción del proyecto. Estas personas se enfrentarán a nuevos problemas y tendrán que resolverlos, pero siempre dentro de las restricciones definidas por los planos originales. Estos problemas no deberían hacer necesario un rediseño grande de esas restricciones, como mucho algún pequeño ajuste.

La segunda fase de construcción suele ser mucho más costosa y larga que la primera fase de diseño. También es realizada por un tipo de profesionales de distinto tipo que la primera. Los responsables de la fase de diseño tienen habilidades creativas e intelectuales mientras que los responsables de la segunda tienen habilidades manuales y más prácticas. La primera fase es más creativa y poco predecible. La segunda fase es más mecánica y predecible.

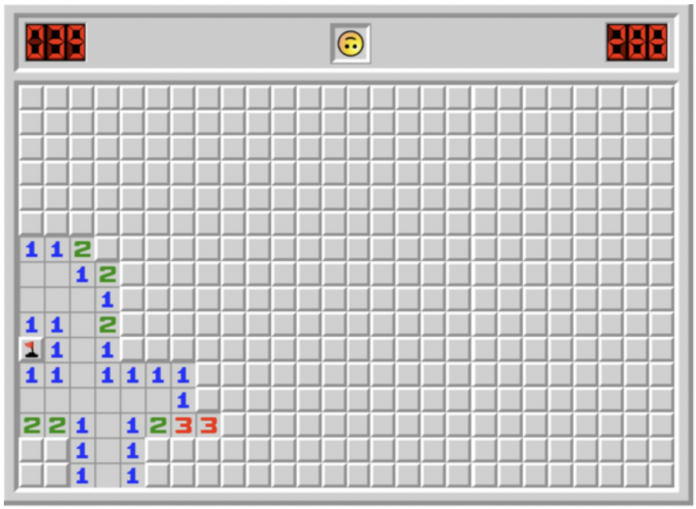
Este enfoque se ha intentado aplicar al desarrollo de software sin mucho éxito en la mayoría de los casos, utilizando el modelo denominado *waterfall* (de cascada). En un siguiente apartado explicamos que el desarrollo de software es una actividad creativa y por qué el enfoque waterfall no es posible. Algunas ideas por ahora que diferencian el software de la construcción:

- En el software la parte de construcción es muy barata.
- En el software todo el esfuerzo es diseño y requiere, por tanto, de personas creativas y con talento.
- Los procesos creativos no se pueden planificar fácilmente, por lo que la predictividad es un objetivo imposible.

## La no predictividad de los requisitos

Muchas veces los equipos de desarrollo se quejan de que "el problema es que los requisitos del proyecto siempre están cambiando". Sin embargo, esto es lo normal de los proyectos software. Lo que hay que hacer es aprender a gestionar estos cambios.

La mayoría de las veces los requisitos no cambian porque el cliente cambie de opinión, sino porque se trabaja con muchísimas incertidumbres que producen problemas y asuntos no esperados.



> "Espera, no lo sabíamos, pero resulta que ..."

> "Espera, esto hace X? Esperábamos que hiciera Y ..."

> "Espera, resulta que para conseguir X necesitamos antes hacer Y ..."

No es que haya cambiado la legislación, ni que haya aparecido una nueva tecnología a la que haya que adaptar el software. Es que, sencillamente, vamos

descubriendo (cliente y equipo de desarrollo) el problema conforme se desarrolla el software. Es como jugar al buscaminas.

Debido a la intangibilidad del software y a la incertidumbre en el conocimiento del dominio los clientes sólo pueden comprender su funcionamiento cuando lo ponen en ejecución. Por eso el valor de una característica nueva sólo se puede conocer cuando se usa de verdad. Sólo en ese momento se puede saber qué características proporcionan valor y cuáles no.

Por ello los clientes esperan poder cambiar ideas que resulta que no funcionan y poder cambiarlas por otras que se les ocurren y que pueden ser más valiosas. Los clientes saben que el software se puede modificar fácilmente.

Incluso si las funcionalidades se conocieran de antemano perfectamente, el software se ejecutará en un entorno formado por personas, restricciones, legislación, etc. que están cambiando continuamente.

Por resumir, en el desarrollo de software tenemos distintos tipos de incertidumbres y riesgos que hacen que el proceso no sea totalmente predecible ni planificable.

- **Valor del producto**: ¿estamos construyendo el producto adecuado? ¿las funcionalidades que se introducen son las que realmente necesitan los usuarios? ¿van a poder resolver sus problemas con el producto que estamos construyendo?
- **Tecnología empleada**: ¿el equipo conoce suficientemente la tecnología para construir correctamente el producto? ¿va a poder escalar suficientemente cuando se necesiten nuevos usuarios? ¿va a estar libre de bugs? ¿tiene una buena documentación?
- **Incertidumbres sociales**: los problemas que el software tratan de resolver la mayoría de las veces tienen una componente humana y social que puede afectar su desarrollo. Por ejemplo, pueden aparecer nuevas legislaciones que regulen de forma distinta el negocio, o una empresa competidora puede desarrollar un método alternativo de dar un servicio, o puede surgir una pandemia que obligue a modificar las formas de interactuar del negocio con los clientes.

Por ello, debemos asumir que no podremos predecir fielmente todo el desarrollo, debemos **matar toda esperanza de que todo va a ir bien a la primera**, debemos aprender que el cambio es una característica fundamental del desarrollo y que debemos aprender a vivir con esta característica. Debemos aprender a hacer desarrollos que sean fácilmente modificables y robustos, que no se rompan cuando se cambien los requisitos.

## Cómo controlar un proceso impredecible: iteraciones

En el artículo *The New Methodology* Martin Fowler se pregunta ¿cómo es posible enfrentarse y controlar un mundo incierto y poco predecible? Es un problema equivalente al de encontrar un camino en un entorno del que sólo conocemos mapas parciales y en el que no sabemos dónde estamos. La respuesta es que necesitamos tener un mecanismo de retroalimentación (*feedback*) con el que podamos continuamente obtener información y debemos reflexionar continuamente sobre la misma.

El mecanismo que propone Fowler para conseguir feedback en el desarrollo de un producto software es el desarrollo iterativo. La clave es producir frecuentemente versiones funcionales (*working versions*) del sistema final que tengan un subconjunto de las características requeridas. Las características ofrecidas por estos sistemas serán limitadas en cuanto a número, pero en todo lo demás estarán totalmente operativas, testeadas e integradas, de forma que funcionarán tal y como el cliente las necesita.

El objetivo de esto es proporcionar la oportunidad de que el usuario final interactúe con el sistema real. No con una documentación o con un prototipo. Es cuando la gente se pone a trabajar realmente frente al sistema cuando aparecen todos los errores, tanto en forma de *bugs* de programación como en forma de malas interpretaciones en las funcionalidades y problemas de uso.

El resultado del desarrollo mediante iteraciones es un proceso adaptativo que es capaz de gestionar tanto las incertidumbres como los cambios. En este tipo de desarrollo los planes a largo plazo son muy flexibles y los únicos planes estables son los que se realizan a corto plazo en la próxima iteración. Los planes futuros se reajustan continuamente y se basan en el resultado de las iteraciones anteriores.

## El cliente adaptativo

Otro de los puntos que trata Martin Fowler en su artículo es el cambio de relación con el cliente al que obliga este nuevo enfoque iterativo. Lo habitual cuando se contrata un desarrollo de software es acordar un coste fijo y una fecha fija. Si el proyecto no se entrega en el plazo indicado, el cliente puede no pagar el precio acordado. Este tipo de contratos requiere unos requisitos estables y por ello un proceso de desarrollo predictivo.

Un fracaso en este tipo de contratos afecta tanto al cliente como a la empresa de desarrollo. La empresa de desarrollo no cobra y pierde el dinero. Pero el cliente también resulta penalizado porque después de trabajar con la empresa de desarrollo durante un periodo de tiempo al final no obtiene nada. Además del trabajo perdido, también se esfuman todos los posibles beneficios que se hubieran obtenido si el software que se estaba desarrollando hubiera entrado en funcionamiento.

En un proceso adaptativo no se puede trabajar con esta noción tradicional de precio fijo. Esto no significa que no se pueda definir un presupuesto y un tiempo de desarrollo para el proyecto. Lo que pasa es que no puedes fijar además el alcance del proyecto. El enfoque ágil es fijar el coste y el tiempo y permitir que el alcance varíe de una forma controlada.

El cliente tiene entonces un grado de control mucho más fino en el proceso de desarrollo. En cada iteración obtiene el producto desarrollado hasta el momento y puede modificar la dirección del desarrollo. Esto lleva a una relación más cercana con los desarrolladores de software, a una relación de negocio basada en la colaboración. Este nivel de implicación no es asumible por muchas organizaciones, pero es esencial para que el proceso adaptativo funcione.

El cliente puede pagar por iteraciones entregadas. Y si en algún momento no está satisfecho con el desarrollo del proyecto suspender el contrato.

Este tipo de colaboración tiene muchas ventajas para el cliente. Desde el principio obtiene una versión usable, aunque mínima, del sistema. El proceso es mucho más transparente y visible que en un modelo de precio fijo. Los posibles retrasos y problemas se pueden visualizar mucho más pronto y es posible reaccionar y tomar decisiones cuando todavía hay tiempo de hacerlo.

Además el cliente puede también en cualquier momento cambiar las capacidades del sistema para adaptarse a cambios en el negocio.

Durante el desarrollo el cliente y los desarrolladores van aprendiendo qué elementos son valiosos y cuáles no. Muy a menudo las características más valiosas del software no se hacen obvias hasta que los clientes han tenido oportunidad de jugar con él.

Los métodos adaptativos permiten aprovecharse de esto, animando a los clientes a que aprendan sus necesidades al tiempo que el sistema se va construyendo. Y construyendo el sistema de forma que los cambios puedan incorporarse fácilmente.

Así se consigue lo que Mary Poppendieck indica con su frase:

> Un cambio de última hora en los requerimientos es una ventaja competitiva.

Esto es, el cliente puede reaccionar hasta el último momento frente cambios del entorno y ganar así a la competencia, que normalmente tendrá una capacidad de reacción mucho más lenta.

## El desarrollo de software es una actividad creativa

En 1986, hace más de 30 años, Fred Brooks escribió un artículo que se convertiría en clásico: "No Silver Bullet".

En el artículo se distinguen dos tipos de tareas en el desarrollo de software: las tareas que denomina **esenciales**, que están relacionadas con las estructuras conceptuales que están detrás del desarrollo, y otras tareas que son **accidentales** y están relacionadas con la representación de esas estructuras conceptuales en un lenguaje de programación concreto.

Muchos de los adelantos y mejoras en el desarrollo de software tienen que ver con las tareas accidentales: mejor hardware, mejores compiladores, mejores IDEs, etc. Sin embargo, el verdadero cuello de botella del desarrollo son las tareas esenciales.

### Tareas esenciales y accidentales del desarrollo de software

Brooks considera que el diseño de software es una actividad creativa que se puede dividir en dos partes: lo esencial y lo accidental. Lo esencial es la formulación mental de las construcciones conceptuales necesarias para el sistema. Lo accidental, en el sentido de secundario, es la implementación de estas construcciones conceptuales en un lenguaje de programación y hardware concreto.

Por ejemplo, en el caso de un sistema software de gestión de ambulancias un elemento central van a ser los sensores de posición que tengamos en las ambulancias. ¿Cómo van a transmitir esos sensores de posición su posición al sistema? Tenemos aquí un problema en el que queremos que múltiples sensores, quizás decenas, envíen peticiones periódicas a un servidor para actualizar un dato. ¿Las peticiones van a ser procesadas usando un sistema de eventos? ¿O serán simples peticiones a un endpoint REST? ¿Qué pasa si una petición falla? ¿Guardamos en el sensor un histórico de las últimas n posiciones y las comunicamos de golpe o las comunicamos una a una?

Todas estas son decisiones que tenemos que tomar que están relacionadas con la esencia del problema que queremos solucionar. En otro caso serán decisiones relacionadas con datos. ¿Qué datos debemos guardar para poder gestionar correctamente la información? O con lógica de negocio. Por ejemplo, en un sistema de gestión académica, cuando un estudiante se matricula ¿cómo decidir si asignar un estudiante a un grupo a otro?.

Lo accidental es la implementación de estas decisiones en un sistema software concreto. Dependiendo de si estamos usando Spring Boot con Java o .NET deberemos codificar estas decisiones en una plataforma u otra.

La parte accidental del desarrollo se puede mejorar con mejores IDEs, más capacidades de hardware o mejores lenguajes de programación. Pero estas mejoras no van a aumentar en un orden de magnitud el

desarrollo de software. No vamos a poder hacer 10 veces más software que antes porque usemos mejores IDEs o mejores lenguajes de programación. Sencillamente porque tenemos que dedicar una parte importante del tiempo a resolver problemas esenciales y no este tiempo no se puede reducir con las mejoras en los elementos accidentales.

## El cuello de botella del desarrollo de software

Brooks argumenta que el cuello de botella del desarrollo de software es el relacionado con la parte esencial, con la especificación, diseño y prueba de las construcciones esenciales del desarrollo.

Esta parte es fundamentalmente creativa y su solución correcta depende de muchos factores difícilmente optimizables como la experiencia del equipo en desarrollos anteriores, la facilidad o dificultad intrínseca del dominio, la facilidad de los clientes a la hora de definir las necesidades, etc.

Brooks enumera cuatro propiedades de los elementos esenciales de un proyecto software, que no existen en otras industrias como la arquitectura o los automóviles.

- **Complejidad**: Las entidades constitutivas de un producto software no se repiten como sucede en un coche o en un edificio. Hay mucha más cantidad de de entidades distintas que además van creciendo conforme el proyecto aumenta. Además, el número de posibles estados en los que puede encontrarse un sistema software crece de forma exponencial con el número de estados de sus elementos, por lo que es cada vez más difícil enumerarlos y, más aun, entenderlos. De esta complejidad inherente en el producto software viene la dificultad de la comunicación entre los miembros del equipo, los fallos del producto y la dificultad de extenderlo a nuevas funcionalidades sin crear efectos laterales.
- **Conformidad**: Muchas veces no es posible eliminar la complejidad anterior debido a que el software debe conectarse y estar conforme con las interfaces de otros sistemas humanos como instituciones o legislación que son complicados ya de por si. El software debe adaptarse a estas interfaces ya existentes y no podemos simplificarlo ni rediseñarlo por si solo.
- **Cambiabilidad**: Las entidades principales del software están sometidas constantemente a la presión del cambio. También sucede lo mismo con los coches o los edificios, pero en esos casos los cambios son mucho más lentos. En el caso del software, debido a su facilidad de modificación, el cambio es una constante con la que tiene que coexistir su desarrollo. El software se ejecuta en computadores, dispositivos y sistemas operativos que también están cambiando constantemente. Además, está embebido en una matriz cultural de aplicaciones, usuarios y leyes que también están en continuo cambio. Todos estos cambios fuerzan el cambio del propio software.
- **Invisibilidad**: El software es invisible e invisualizable. En el caso de otras industrias esto no es así. Los planos detallados de un edificio permiten representar todos los elementos necesarios para su construcción. Sucede igual con los diagramas y planos de diseño de un automóvil. Sin embargo, el software no está embebido en un espacio. En cuanto intentamos representar con diagramas la estructura del software nos damos cuenta de que está constituido no por uno, sino por bastantes grafos dirigidos superpuestos unos sobre otros. Estos grafos pueden representar el control del flujo, el control de los datos, patrones de dependencias, secuencias temporales o relaciones en el espacio de nombres. Al no poder visualizarlo, es muy difícil razonar sobre su estructura o comunicarnos y analizarlo entre varias personas.

Brooks argumenta que no es posible aplicar a estas propiedades las mismas optimizaciones que se han aplicado a los elementos accidentales del desarrollo. El hecho de tener lenguajes de programación de más alto nivel o mejores IDEs no mejora las características anteriores.

### Mejorar el desarrollo de los elementos esenciales

Por último, Brooks termina el artículo dando algunas ideas de cómo sería posible optimizar el desarrollo de los elementos esenciales del software y combatir las características anteriores:

- **Comprar frente a construir**. Utilizar software ya existente. En la época en la que Brooks escribió el artículo no existía la Web ni el Open Software. Por eso hablaba de "comprar" software. Hoy en día tenemos múltiples de bibliotecas pre-existentes que podemos utilizar con solo declararlas en el fichero `Pom.xml`. Sin embargo, estas librerías son todavía de bajo nivel y la mayoría de las veces sirven sólo para resolver los elementos accidentales. Existen muy pocas librerías de alto nivel que podamos usar para resolver problemas esenciales. Aunque cada vez están apareciendo más SaS (*Software as a Service*) que proporcionan APIs REST para gestionar áreas específicas de negocio. Un ejemplo podría ser los pagos con tarjeta de crédito, en el que existen soluciones cada vez más flexibles como Stripe.

- **Prototipado rápido y refinamiento de los requisitos**. De esto hablaremos más en el siguiente apartado. Pero Brooks se adelantó a todo el movimiento ágil proponiendo la entrega rápida de versiones iniciales del producto a los clientes para que éstos puedan utilizarlo y entender mejor qué es lo que necesitan. Según Brooks, la parte más complicada de la construcción de un sistema software es precisamente decidir qué construir. Los clientes no saben lo que quieren y es difícil predecir y planificar de antemano cómo va a ser la interacción entre los usuarios y el propio software y cómo esta interacción va a afectar su trabajo. Es imposible para los clientes especificar completamente todos los requisitos del software que necesitan sin haber probado alguna versión inicial del producto que están especificando. Por tanto, es necesario poder desarrollar versiones rápidas del software con las que se pueda realizar una **especificación iterativa** de sus requisitos.

- **Desarrollo incremental - crecer, no construir el software**. Con esto también se adelantó Brooks al desarrollo ágil. El enfoque para el desarrollo de software debe ser conseguir que éste crezca de forma orgánica, siguiendo las necesidades de los usuarios. La metáfora de la construcción no es correcta, hay que usar otras metáforas nuevas en las que el desarrollo de software sea más similar al empleado por la naturaleza, en la que podemos observar cómo se han creado con éxito múltiples elementos de enorme complejidad coexistiendo y colaborando en múltiples niveles. Para ello es necesario realizar un desarrollo incremental, en el que se parta de un software inicial al que se van añadiendo nuevas funcionalidades y nuevas estructuras para trabajar con nuevos datos y situaciones. De esta forma el software va creciendo de forma orgánica.

## Referencias


- Frederick Brooks (1987) *No Silver Bullet*
- Martin Fowler (2005) *The New Methodology*
- Steve McConnell (2004) *Code Complete, Capítulo 2*