

Leyes de Lehman

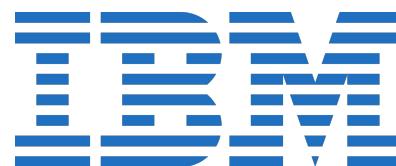
Fausto Panello 2025

Meir M. Lehman

Meir Lehman era un profesor en la Escuela de Ciencias Computacionales de Middlesex, Londres.

Estudió matemática en el Colegio Imperial de Londres, y en los 70s se fue a trabajar a Nueva York, en la división de descubrimiento de IBM.

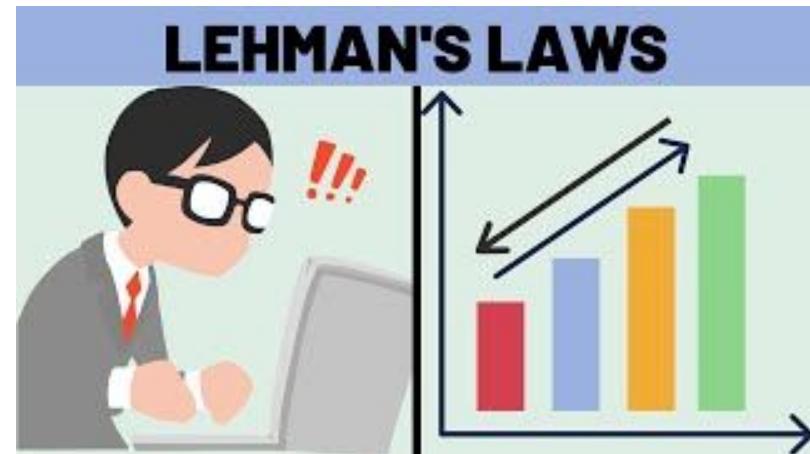
El estudio del proceso de programación de IBM, dió las fundaciones para que Lehman desarrolle las leyes de evolución de software, a principios de la década de los 80.



¿Por qué diseñó leyes?

A principios de los 80, justo cuando Lehman entró a IBM, estaban en pleno auge las metodologías ágiles (espiral, incremental, etc.).

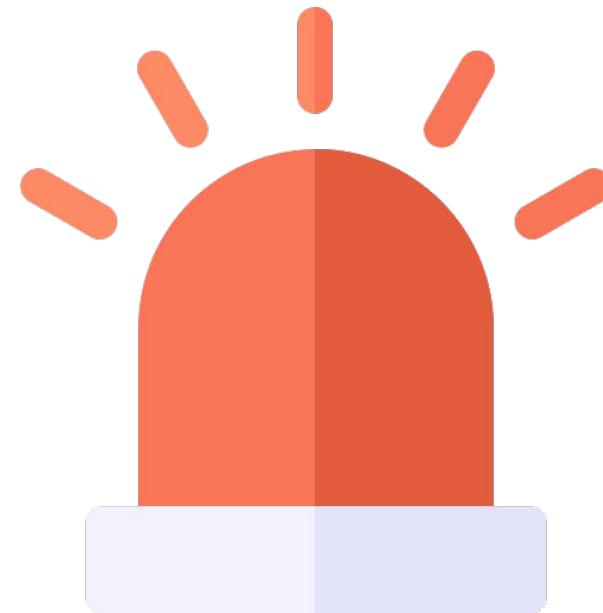
Es así que Lehman observó que se configuraba una nueva problemática en el desarrollo de sistemas, es decir, la crisis se expande ahora en el sentido que no sólo se requiere reflejar lo más fielmente posible las necesidades del usuario, sino que ahora los ambientes en que el sistema está inserto están sujetos a cambios y estos cambios inciden en la efectividad del software desarrollado.



¿Por qué diseñó leyes?

A su vez, ocurre un fenómeno: la **crisis del software**. Esto se refiere a los problemas que, desde sus inicios, ha ido experimentando el software, muchas veces problemas de gran magnitud, debido, principalmente, a la mínima eficacia que presentan una gran cantidad de empresas al momento de realizar un software; sería algo así como el conjunto de dificultades o errores ocurridos en la planificación, estimación de los costos, productividad y calidad de un software.

Sin embargo, la crisis del software no causaba solamente problemas económicos...



Ej. Crisis del software

1986: En abril de 1986 un avión de combate se estrelló por culpa de un giro descontrolado atribuido a una expresión “if then”, para la cual no había una expresión “else”, debido a que los desarrolladores del software lo consideraron innecesario.

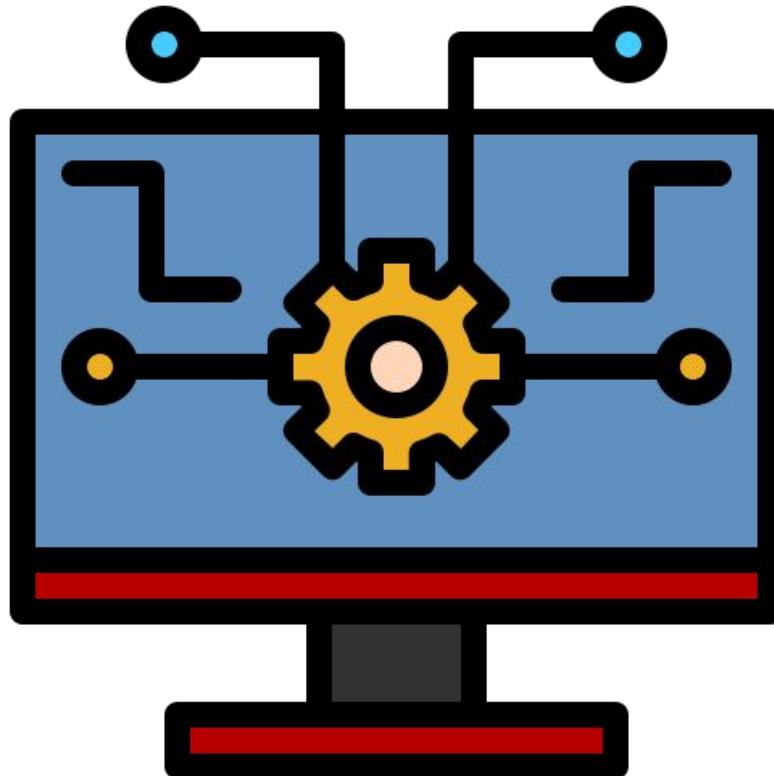


1985-1987: El Therac-25 fue una máquina de radioterapia que causó la muerte de varios pacientes en diversos hospitales de Estados Unidos y Canadá, debido a las radiaciones de alto poder aplicadas sin control, las cuales fueron atribuidas a la falta de control de calidad del software médico.

Leyes de Lehman

Lehman propone que la evolución de un sistema de software está sujeta a varias leyes. Ha determinado estas leyes a partir de observaciones experimentales de varios sistemas, como los grandes sistemas operativos.

Lehman señala, a partir de esto, 3 tipos de programas, y varias leyes.

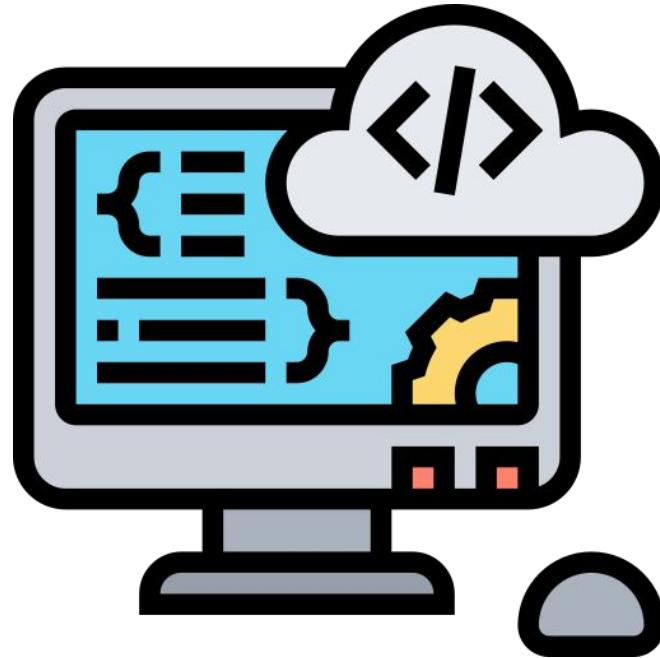


‘Programas’ de Lehman

Un ‘S-Program’ es escrito de acuerdo a una especificación exacta de qué puede hacer el programa. Por ejemplo, un programa que consista en un algoritmo determinado, estático y sin evolución; el problema de las 8 damas.

Un ‘P-Program’ es creado para implementar ciertos procedimientos que determinan completamente qué puede hacer el programa de acuerdo al contexto; un motor de ajedrez.

Un ‘E-Program’ se crea para hacer actividades de la vida real; cómo debe comportarse está fuertemente vinculado al entorno en el que corre, y semejante programa necesita adaptarse a requerimientos variados y circunstancias en ese entorno.



Leyes de Lehman

- **Cambio continuo:** Un programa que se utiliza en un ambiente del mundo real debe cambiar o será cada vez menos útil en ese ambiente.
- **Complejidad creciente:** A medida que un programa en evolución cambia, su estructura se hace más compleja, a menos que se lleven a cabo esfuerzos activos para evitar este fenómeno.
- **Evolución del programa:** La evolución del programa es un proceso autorregulado, y una medición de atributos del sistema, como el tamaño, el tiempo entre versiones, el número de errores advertidos, etc., revela las tendencias estadísticas significativas y las características invariantes.



Leyes de Lehman

- **Conservación de la estabilidad organizativa:** Durante el tiempo de vida de un programa, su rapidez de desarrollo es casi constante e independiente de los recursos dedicados al desarrollo del sistema.
- **Conservación de la familiaridad:** A medida que evoluciona el sistema, todo lo asociado con ello, desarrolladores, personal de ventas y usuarios, por ejemplo, deben mantener maestría de su contenido y comportamiento para lograr la evolución satisfactoria y constante. El crecimiento excesivo disminuye esta maestría. Entonces cada incremento de los incrementos debe mantenerse promedio e invariante.



Leyes de Lehman

- **Crecimiento continuo:** El contenido funcional de un sistema debe ser incrementado continuamente para mantener la satisfacción de los usuarios y clientes a lo largo de su ciclo de vida.
- **Calidad declinante:** La calidad de un sistema va a ser aparentemente declinante salvo que se mantenga y se adapte rigurosamente para acatar los cambios operacionales ambientales.
- **Sistema de retroalimentación:** El proceso de evolución de un sistema constituye sistemas de retroalimentación multi-nivel, multi-iterativos, y multi-agentes, y deben ser tratados para denotar una mejoría significativa.



Interpretaciones

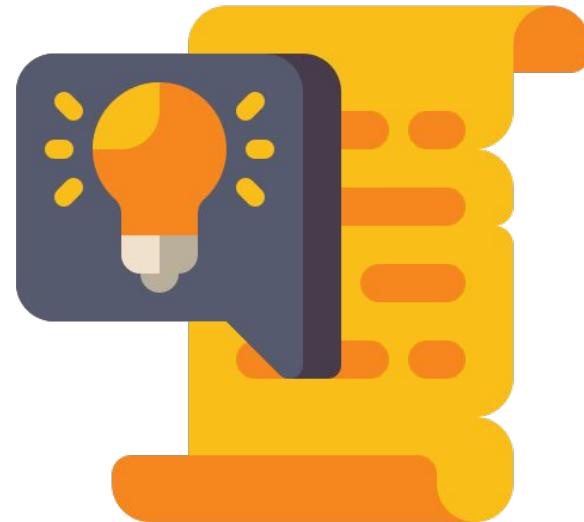
Somerville: Considera las "leyes" de Lehman como elementos positivos en la configuración del ámbito en que se desarrollan los sistemas, por ejemplo, dice que las dos primeras hipótesis de Lehman son casi con certeza válidas.

Interpretando la primera como: "...El razonamiento subyacente en la primera ley... es que cuando un sistema de software (sobre todo uno grande) se construye para modelar algún ambiente y después se introduce en él, modificándose así el ambiente original con la presencia del sistema de software... No puede ser más interesante para los propósitos de esta revisión la característica planteada por esta hipótesis, es decir, si se desarrolla software no se podría pensar desde la perspectiva de estabilidades, todo lo contrario, necesariamente es el cambio el motor y el configurador de los sistemas de software."



Sommerville

La segunda ley en el análisis de Sommerville corresponde al hecho de que la estructura del programa original se estableció para aplicarlo a un conjunto de necesidades iniciales. A medida que se produce el cambio evolutivo de esas necesidades, la estructura original se degrada y con ello se va haciendo más compleja ya que al ir incrementándose la distancia conceptual entre el software y el medio que lo contiene, éste va estando cada vez más presente en el hacer cotidiano, formulando quiebres constantes sobre el sistema de trabajo lo que imprime un sello de complejidad a la utilización del software.



Sommerville

Las siguientes leyes tienen relación con las características de las organizaciones y de los individuos que participan en el proceso de desarrollo de software. Lehman afirma que las organizaciones se esfuerzan por lograr la estabilidad e intentan evitar cambios drásticos o repentinos.

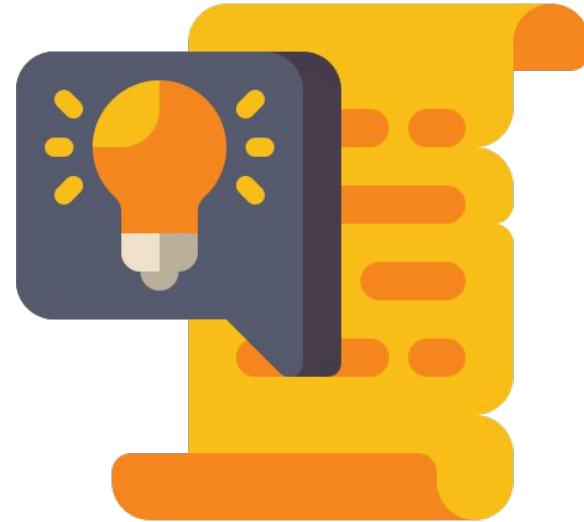


Por lo tanto, a medida que se añaden más recursos a un proyecto de software, el efecto evolutivo de la adición de nuevos recursos se va reduciendo, hasta que la adición de nuevos recursos no produce ningún efecto. Si bien, estas últimas leyes no resultan tan obvias como las primeras y podrían ser cuestionadas, es posible que la última, que tiene que ver con la conservación de la familiaridad sea la más útil, como también la de reducción de personal, en el sentido que cuanta más gente trabaje, menos productivo será cada miembro del proyecto.

Richard Fairley

El desarrollo de productos mediante el método de versiones sucesivas es una extensión del método de prototipos en el que se refina un esqueleto inicial del producto obteniendo así, cada vez más capacidades. En dicho método, cada versión es un sistema funcional y capaz de realizar trabajo útil.

En realidad, el ciclo de desarrollo de un producto de programación es una combinación de los distintos modelos presentados. Las organizaciones y proyectos especiales pueden adoptar alguno de estos modelos en particular; sin embargo, ciertos elementos de ellos se encuentran en todo proyecto de programación. Por ejemplo, para proyectos de desarrollo de programación no es extraño adoptar el modelo de fases como marco de referencia básico, e incluir prototipos y versiones sucesivas en el desarrollo.

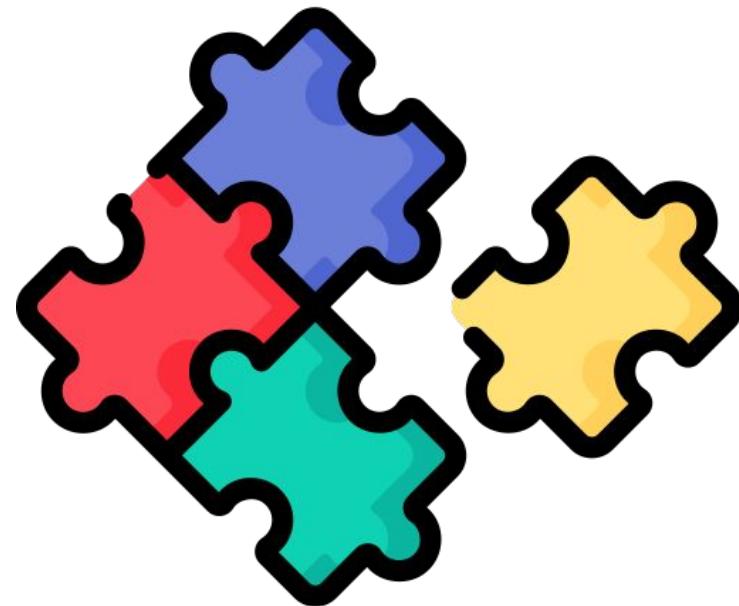


Reutilización de Componentes

Fausto Panello 2025

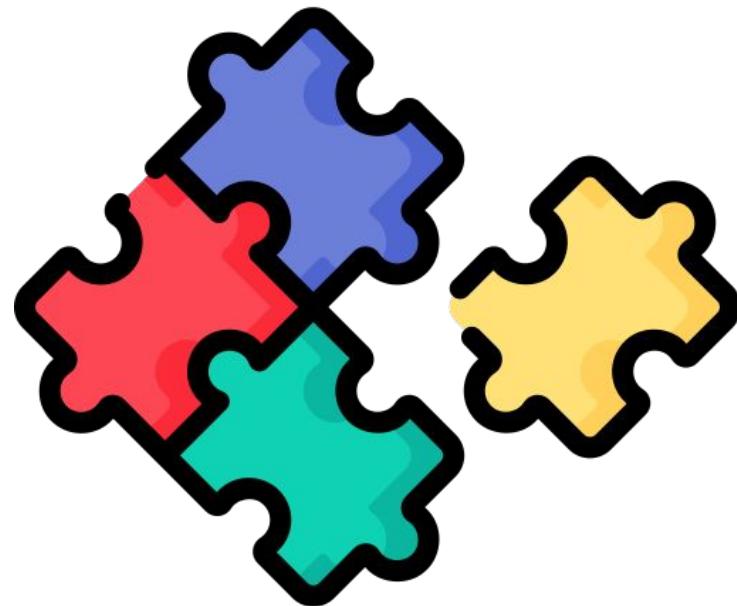
Modelo de Reutilización de Componentes

Es el proceso de creación de sistemas de software **a partir de un software ya existente**, para de ese modo evitar un diseño de sistema desde el principio. En los años 60, se construyeron bibliotecas de subrutinas científicas reutilizables con un dominio de aplicación limitado, en la actualidad se crean componentes comunes a varios procesos con el fin de poder utilizarlos en la construcción de software.



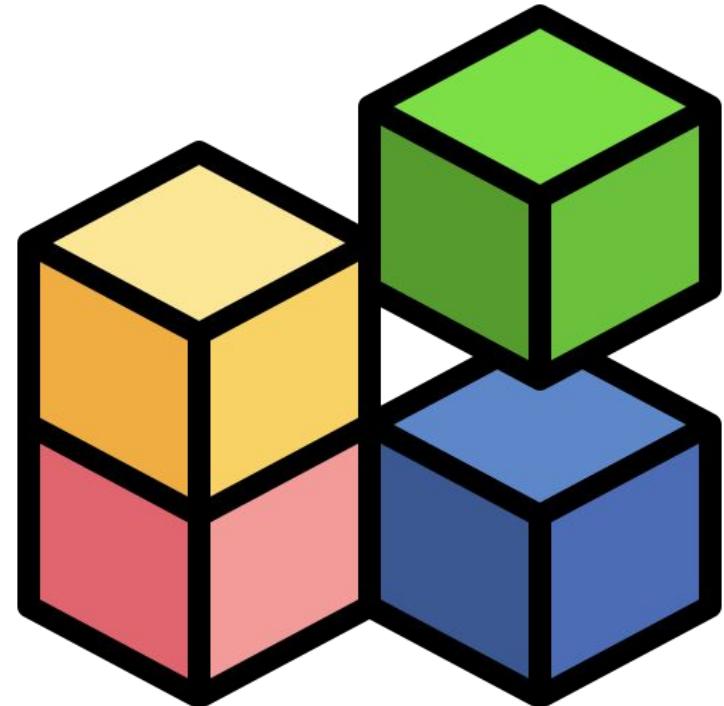
Modelo de Reutilización de Componentes

Podemos definirla como el empleo de elementos de software u otros de nivel superior, creados en desarrollos anteriores, para de este modo reducir los tiempos y simplificar el desarrollo del software, mejorando la calidad y reduciendo su costo. Un componente de software individual es un paquete de software, un servicio web API por ejemplo, o un módulo que encapsula un conjunto de funciones relacionadas (o de datos).



Modelo de Reutilización de Componentes

Todos los procesos del sistema son colocados en componentes separados de tal manera que todos los datos y funciones dentro de cada componente están semánticamente relacionados. Debido a este principio, con frecuencia se dice que los componentes son **modulares** y **cohesivos**.



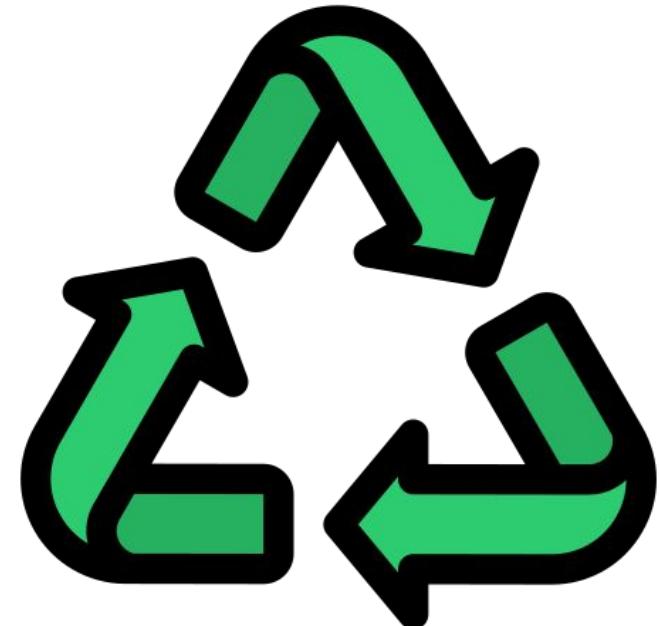
Modelo de Reutilización de Componentes

Los componentes se comunican uno con el otro por medio de interfaces (de entrada y salida).

Cuando un componente ofrece servicios al resto del sistema, este adopta una **interfaz** proporcionada que especifica los servicios que otros componentes pueden utilizar, y cómo pueden hacerlo. Esta interfaz puede ser vista como una **firma** del componente. El cliente no necesita saber sobre los funcionamientos internos del componente (su implementación) para hacer uso de ella. Este principio resulta en componentes referidos como **encapsulados**.

Elementos que intervienen en la reutilización

- Especificaciones de requerimientos previamente concebidas
- Diseños previamente definidos (Estructuras de datos, algoritmos, etc.)
- Código probado y depurado con anterioridad
- Planes y casos de prueba previamente utilizados
- Personal calificado (aprovechamiento de la experiencia de los ingenieros de un proyecto a otro)
- Paquetes de software de propósito general.



Conceptos de reutilización de software

- La reutilización de software aparece como una alternativa para desarrollar aplicaciones y sistemas de una manera más eficiente, productiva y rápida.
- La idea es reutilizar elementos y componentes en lugar de tener que desarrollarlos desde un principio.
- Surge formalmente en 1968.
- La idea principal era producir componentes de software como si de componentes eléctricos se tratara.
- El objetivo es reutilizar lo existente sin tener que volver a rediseñarlo desde el principio.



Dificultades al aplicar reutilización

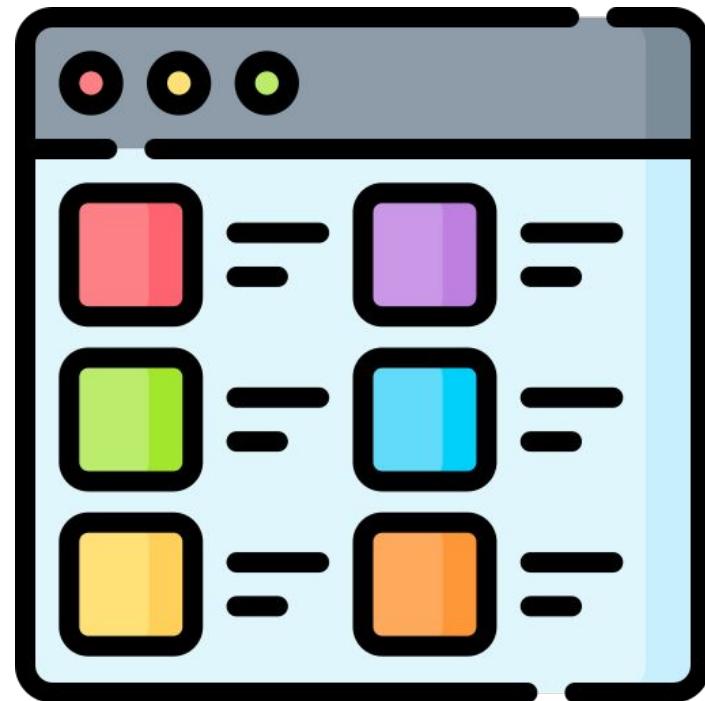
- En muchas empresas no existe plan de reutilización (No se considera prioritario).
- Escasa formación (y personal técnico disponible) en dicho campo.
- Pobre soporte metodológico (los componentes son paquetes habitualmente cerrados y el soporte metodológico puede ser complejo o incompleto).
- Uso de métodos que no promueven la reutilización (Estructurados).



Categorías de recurso de software

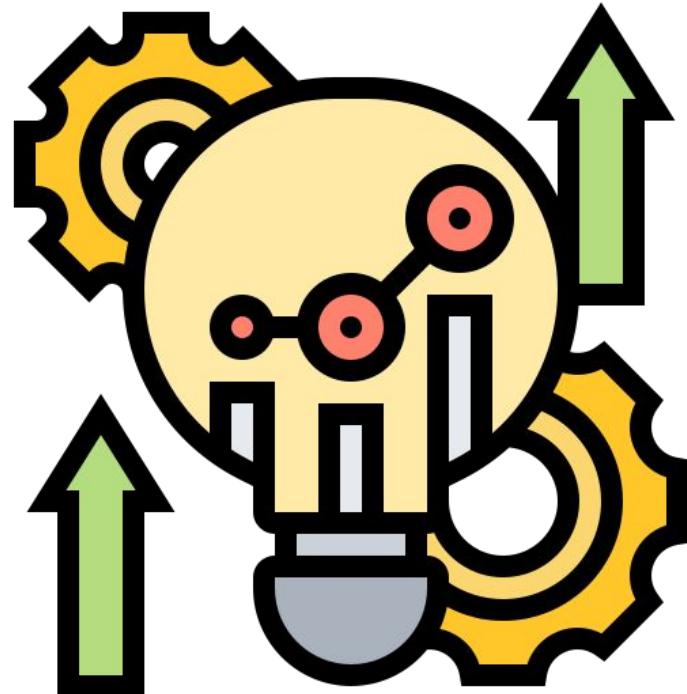
Hay diferentes categorías de recurso de software a reutilizar:

- Componentes ya desarrollados
- Componentes ya experimentados
- Componentes con experiencia parcial
- Componentes nuevos



Componentes ya desarrollados

El software existente se puede adquirir de una tercera parte o provenir de uno desarrollado internamente para un proyecto anterior. Llamados componentes CCYD (componentes comercialmente ya desarrollados), estos componentes están listos para utilizarse en el proyecto actual y se han validado totalmente.



Ejemplos de componentes ya desarrollados

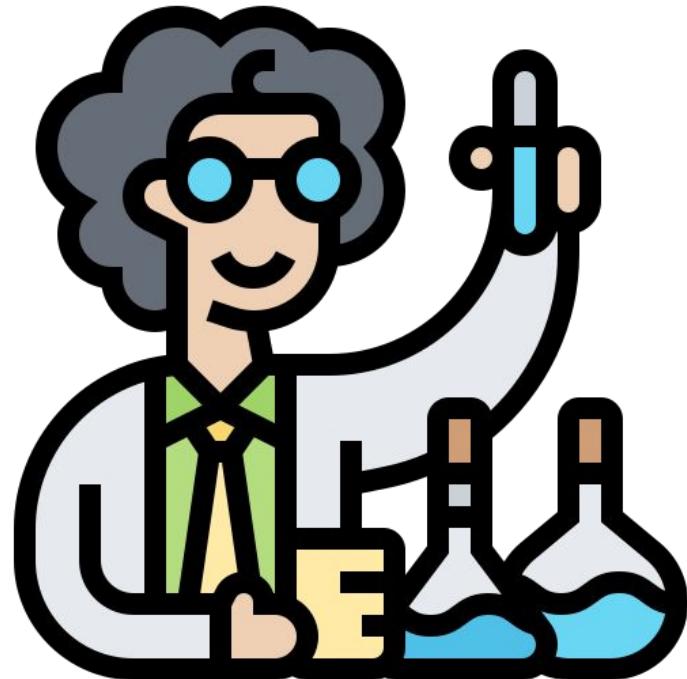
Biblioteca de gráficos 3D: Un proyecto de desarrollo de un videojuego puede reutilizar una biblioteca de gráficos 3D existente, como Unity o Unreal Engine, para crear los efectos visuales y la renderización del juego.

Sistema de autenticación: Un proyecto de desarrollo web puede utilizar un componente de software de autenticación ya desarrollado, como OAuth o Firebase Authentication, para gestionar la autenticación de usuarios en la aplicación.



Componentes ya experimentados

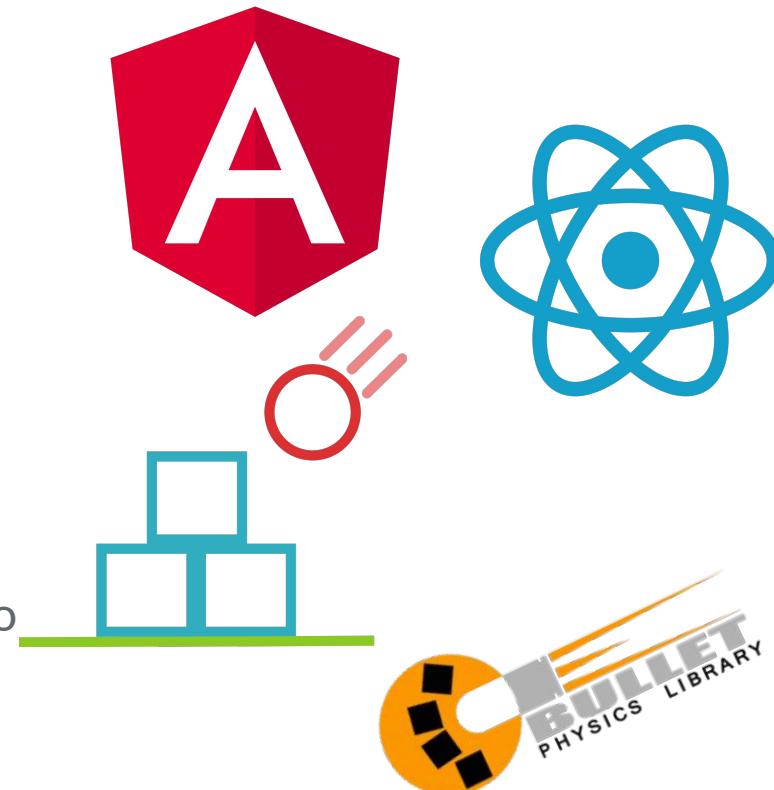
Especificaciones, diseños, código o datos de prueba existentes desarrollados para proyectos anteriores que son similares al software que se va a construir para el proyecto actual. Los miembros del equipo de software actual ya han tenido la experiencia completa en el área de la aplicación representada para estos componentes. Las modificaciones, por tanto, requeridas para componentes de total experiencia, tendrán un riesgo relativamente bajo.



Ejemplos de componentes ya experimentados

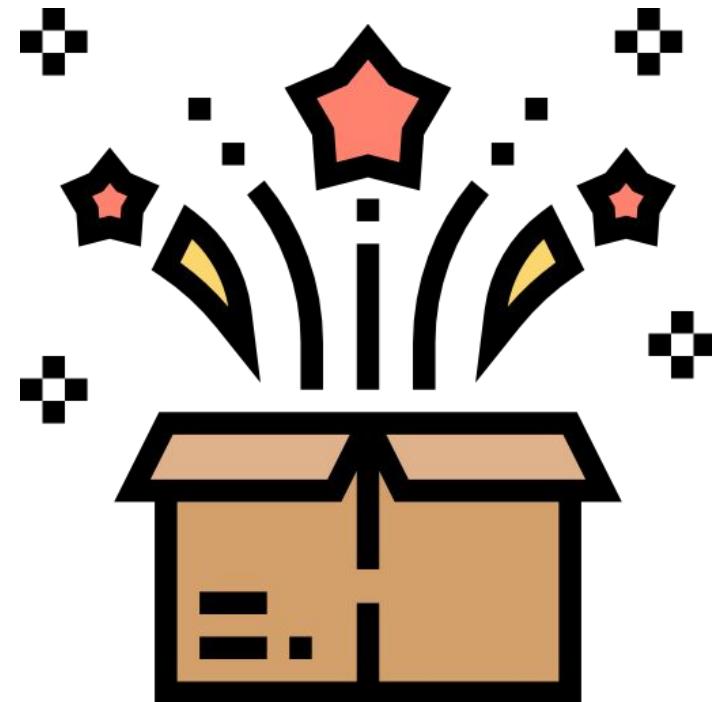
Framework de desarrollo web: Un equipo de desarrollo web que ha utilizado anteriormente un framework como Angular o React puede reutilizar el código, las especificaciones y los diseños para desarrollar un nuevo proyecto web con características similares.

Motor de física: Un proyecto de simulación de física puede reutilizar un motor de física experimentado, como Box2D o Bullet Physics, que ya ha sido utilizado por el equipo para proyectos anteriores con éxito.



Componentes con experiencia parcial

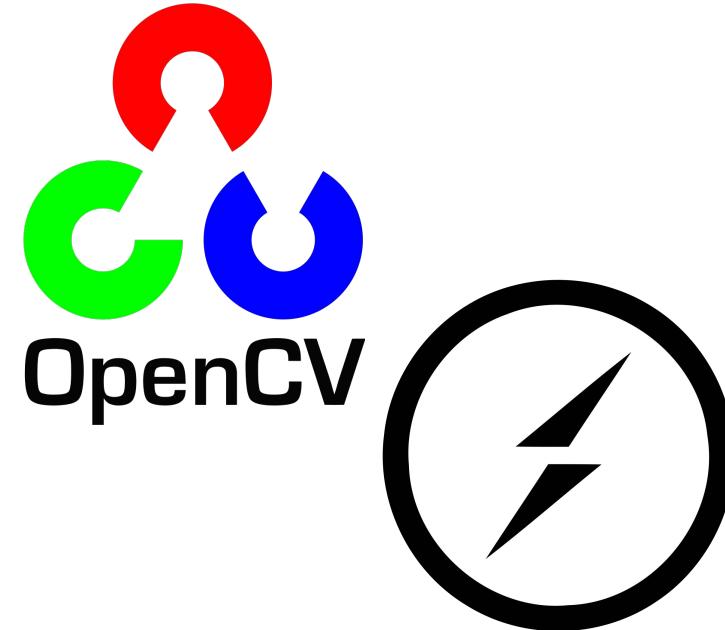
Especificaciones, diseños, código o datos de prueba existentes desarrollados para proyectos anteriores que se relacionan con el software que se va a construir para el proyecto actual, pero que requerirán una **modificación sustancial**. Los miembros del equipo de software actual han limitado su experiencia sólo al área de aplicación representada por estos componentes. Las modificaciones, por tanto, requeridas para componentes de experiencia parcial tendrán bastante grado de riesgo.



Ejemplos de componentes con experiencia parcial

Biblioteca de procesamiento de imágenes: Un equipo de desarrollo que ha trabajado previamente en el procesamiento de imágenes puede reutilizar una biblioteca existente, como OpenCV, para realizar ciertas operaciones de procesamiento de imágenes, pero necesitará modificarla para adaptarla a los requisitos específicos del proyecto actual.

Componente de comunicación en tiempo real: Un proyecto de desarrollo de aplicaciones de chat puede reutilizar un componente de comunicación en tiempo real existente, como Socket.IO, pero deberá realizar modificaciones sustanciales para adaptarlo a las necesidades específicas del proyecto.



Componentes nuevos

Los componentes de software que el equipo de software **debe construir específicamente para las necesidades del proyecto actual**. Ejemplos:

Motor de recomendaciones personalizadas: Un proyecto de desarrollo de una plataforma de comercio electrónico puede requerir la construcción de un motor de recomendaciones personalizadas que tenga en cuenta el historial de compras y preferencias del usuario para ofrecer recomendaciones relevantes.

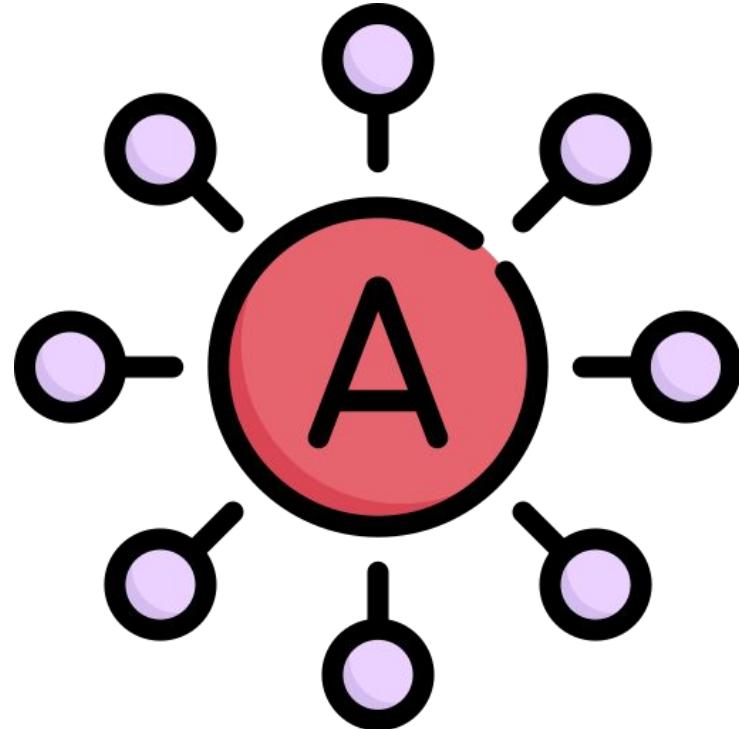
Algoritmo de procesamiento de señales: Un proyecto de desarrollo de software de procesamiento de señales de audio puede requerir la creación de un algoritmo personalizado para el análisis y la manipulación de las señales de audio de entrada.

Sistema de análisis de sentimientos en redes sociales: Para un proyecto de análisis de datos en redes sociales, se puede construir un componente nuevo que utilice técnicas de procesamiento de lenguaje natural y aprendizaje automático para analizar los sentimientos expresados en los mensajes de los usuarios. Este componente extraería información valiosa sobre las opiniones y emociones de los usuarios en relación con ciertos temas y proporcionaría análisis detallados. Requeriría el diseño e implementación de algoritmos de análisis de sentimientos, procesamiento de texto y visualización de datos.



Tipos de reutilización

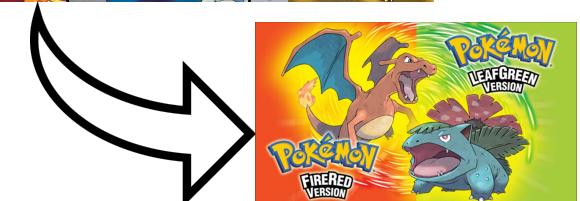
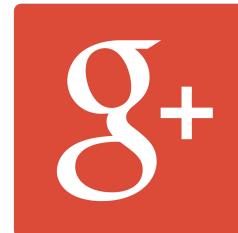
- **Oportunista:** El ingeniero de software reutiliza piezas que él sabe que se ajustan al problema.
- **Sistemática:** Esfuerzo a nivel organizacional y planificado de antemano. Todo componente reutilizado ha de ser ideado, a priori, para ser reutilizado. Implica inversiones iniciales para recoger frutos en el futuro. Diseñar componentes genéricos para que sean reutilizados con facilidad.
- **Bottom-Up:** Se desarrollan pequeños componentes para una determinada aplicación. Se incorpora a un repositorio.
- **Top-Down:** Se determinan las piezas necesarias que encajan unas con otras. Se van desarrollando poco a poco. Requiere alta inversión al comienzo. Se recogerán beneficios en el futuro. Análisis de escenarios para la reutilización.



¿Cuándo voy a requerir elementos de reutilización?

Existen al menos 4 escenarios en los que un proyecto de software requerirá elementos de reutilización:

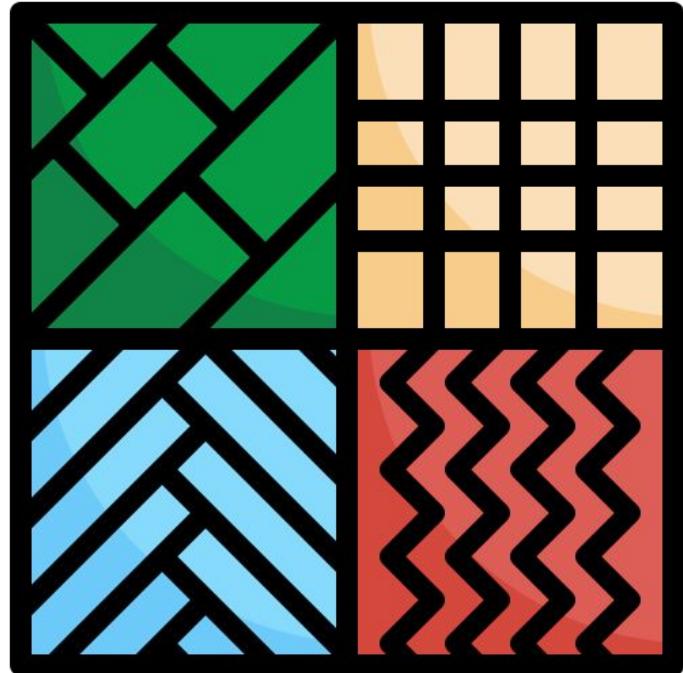
1. El proyecto es similar a uno anterior (reutilización de un proyecto existente).
2. Mismo proyecto con configuración diferente (reutilizan productos actuales).
3. Características de uso basado en productos existentes.
4. Nueva arquitectura con capacidades o elementos existentes.



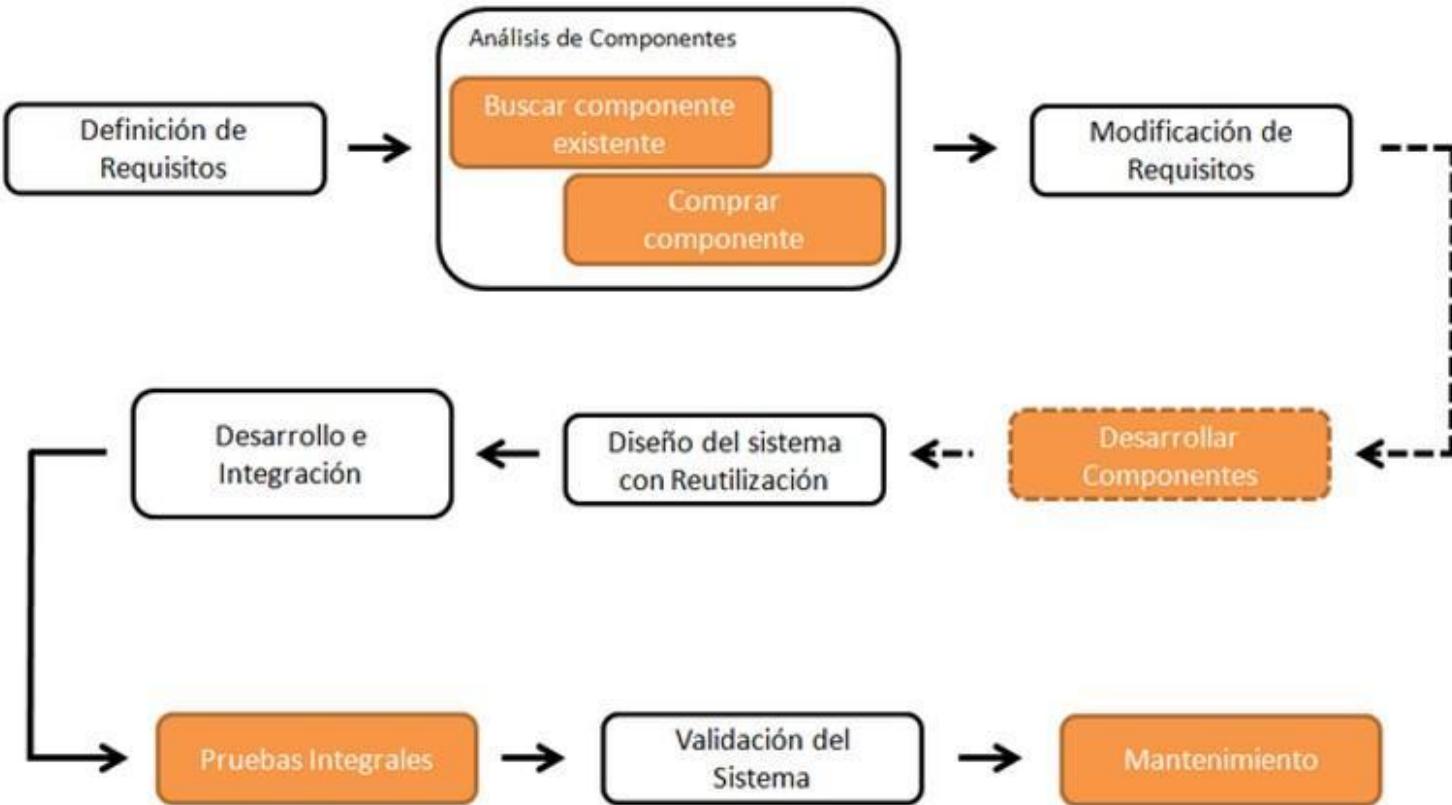
Patrones de diseño

Los patrones y los lenguajes de patrones son formas de describir las mejores prácticas, buenos diseños y encapsulan la experiencia de tal forma que es posible para otros reutilizar dicha experiencia. Algunos elementos esenciales del patrón de diseño pueden ser:

- Nombre que es una referencia significativa del patrón.
- Una descripción del área del problema que explica cuándo puede aplicarse el patrón
- Descripción de las partes de la solución del diseño, sus relaciones y sus responsabilidades.
- Una declaración de las consecuencias de aplicar el patrón.



Representación gráfica del ciclo de vida



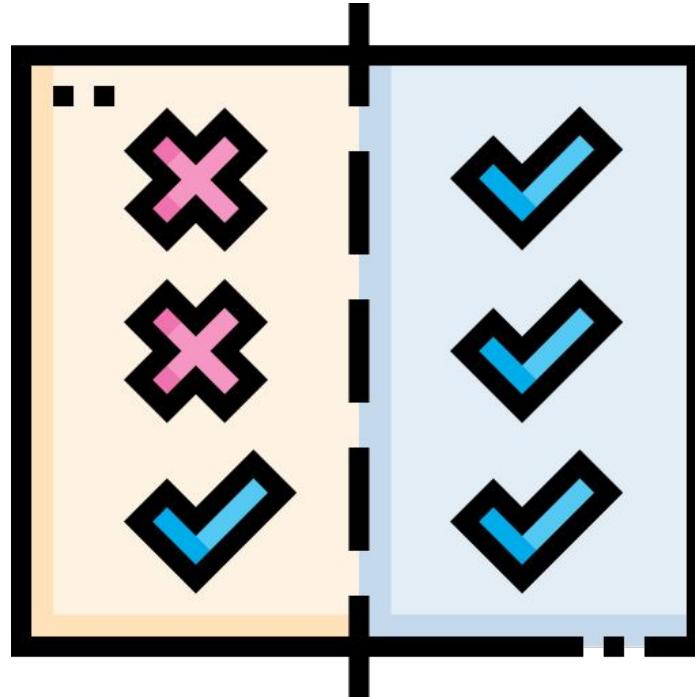
Ventajas y desventajas

Ventajas

- Reduce el tiempo de desarrollo.
- Reduce los costos.
- No es necesario reinventar las soluciones.

Desventajas

- Necesidad de invertir antes de obtener resultados (se puede hacer entregas luego de la puesta en marcha completa del sistema).
- Carencia de métodos adecuados (dependen y se amoldan a los módulos).
- Necesidad de formar al personal por la complejidad de implementar el sistema sobre módulos preexistentes.



XP Programming

ADS 2025

XP (eXtreme Programming)

Es una metodología de desarrollo de la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, Extreme Programming Explained: Embrace Change (1999).

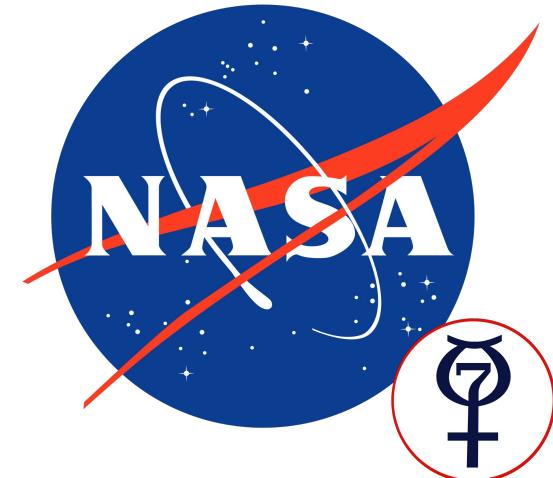
La programación extrema se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad. (Metodología ágil).



Historia y origen

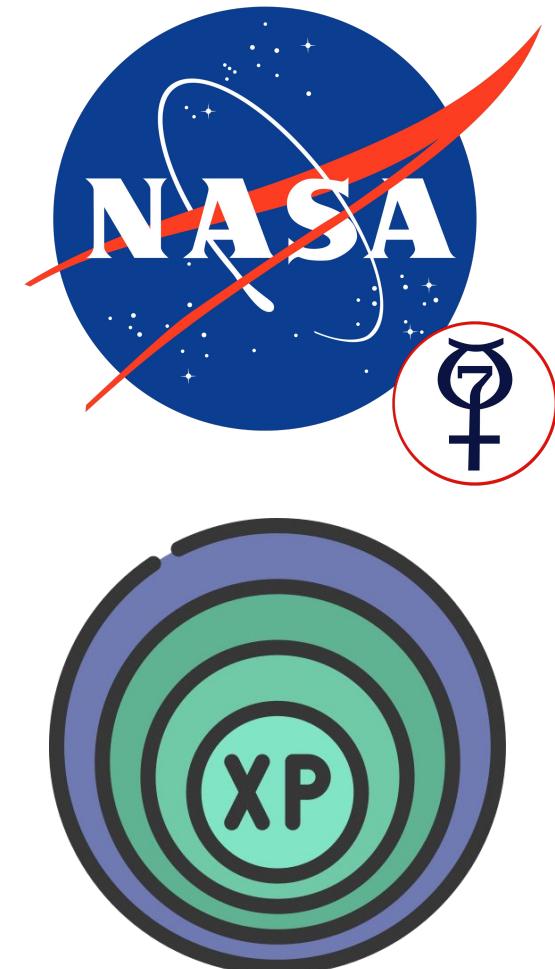
Kent Beck desarrolló una “programación extrema” durante su trabajo en el proyecto de nóminas para el Sistema de Compensación Integral de Chrysler. A medida que su trabajo iba avanzando, comenzó a refinar la metodología de desarrollo utilizada en el proyecto y escribió un libro sobre la metodología (1999).

Muchas de las prácticas de la programación extrema existen desde hace algún tiempo; la metodología lleva las “mejores prácticas” a niveles “extremos”. Por ejemplo, la “práctica del desarrollo de la prueba primero, la planificación y la escritura de pruebas antes de cada microincremento” se utilizó ya en el Proyecto Mercury de la NASA (primer programa espacial de EEUU), a principios de la década de 1960...



Historia y origen

... Para acortar el tiempo total de desarrollo, se desarrollaban algunos documentos de prueba formales (como para las pruebas de aceptación) en paralelo con (o poco antes) que el software estuviera listo para la prueba. Un grupo de prueba independiente de la NASA podía escribir los procedimientos de prueba, basados en requisitos formales y límites lógicos, antes de que los programadores escriban el software y lo integren con el hardware. XP lleva este concepto al nivel extremo, escribiendo pruebas automatizadas (a veces dentro de módulos de software) que validan el funcionamiento de incluso pequeñas secciones de codificación de software, en lugar de solo probar las funciones más grandes.



Influencias

Dos influencias importantes dieron forma al desarrollo de software en la década de 1990:

Internamente, la programación orientada a objetos reemplazó a la programación procedural (secuencial) como el paradigma de programación preferido por algunos desarrolladores.

Externamente, el auge de Internet y el auge de las puntocom enfatizaron la velocidad de comercialización y el crecimiento de la empresa como factores comerciales competitivos.

Los requisitos rápidamente cambiantes exigían ciclos de vida más cortos del producto y, a menudo, chocaban con los métodos tradicionales de desarrollo de software.



Concepción de los métodos

Chrysler contrata a Kent Beck para su Sistema de Compensación Integral. Dentro del proceso de desarrollo, Beck nota varios problemas, y aprovechó esta oportunidad para proponer e implementar algunos cambios en las prácticas de desarrollo. Beck describe la concepción temprana de los métodos así:

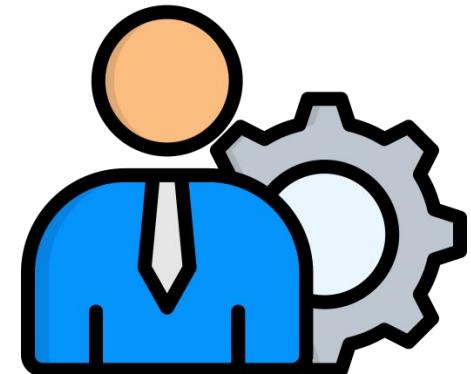
"La primera vez que me pidieron que liderara un equipo, les pedí que hicieran algunas de las cosas que pensaba que eran sensatas, como pruebas y revisiones. La segunda vez había mucho más en juego. Pensé: "Al diablo con los torpedos, al menos esto será un buen artículo", y le pedí al equipo que subiera todas las perillas a 10 en las cosas que pensaba que eran esenciales y omitiera todo lo demás."



Valores, Características, y roles.

Al crear la serie de libros sobre XP, Beck remarca tres elementos, con subelementos por cada uno, que se van a ir interrelacionando para crear el mejor desarrollo XP posible, y alcanzar los objetivos. Estos son:

- **Valores:**
 - Simplicidad
 - Comunicación
 - Retroalimentación (feedback)
 - Coraje o valentía
 - Respeto
- **Características:**
 - ... ya las vamos a ir viendo.
- **Roles:**
 - ... también los vamos a ver ahora.



Valores: Simplicidad

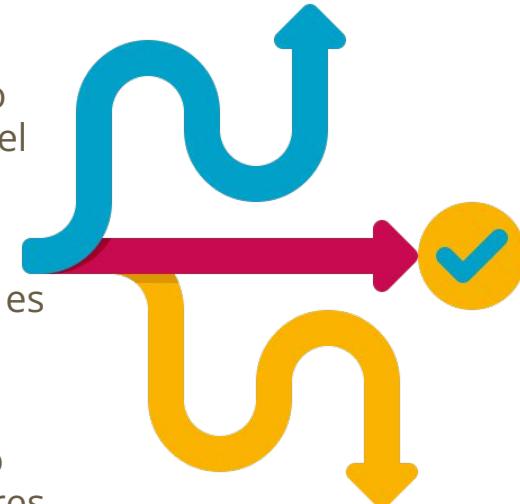
XP propone el principio de hacer la cosa más simple que pueda funcionar, en relación al proceso y la codificación. Es mejor hacer hoy algo simple, que hacerlo complicado y probablemente nunca usarlo mañana.

La simplicidad es la base de la programación extrema. Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento. Un diseño complejo del código junto a sucesivas modificaciones por parte de diferentes desarrolladores hacen que la complejidad aumente exponencialmente.

Para mantener la simplicidad es necesaria la refactorización del código, esta es la manera de mantener el código simple a medida que crece.

También se aplica la simplicidad en la documentación, de esta manera el código debe comentarse en su justa medida, intentando eso sí que el código esté autodocumentado. Para ello se deben elegir adecuadamente los nombres de las variables, métodos y clases. Los nombres largos no decrementan la eficiencia del código ni el tiempo de desarrollo gracias a las herramientas de autocompletado y refactorización que existen actualmente.

Aplicando la simplicidad junto con la autoría colectiva del código y la programación por parejas se asegura que cuanto más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.



Valores: Comunicación

Algunos problemas en los proyectos tienen origen en que alguien no dijo algo importante en algún momento. XP hace casi imposible la falta de comunicación.

La comunicación se realiza de diferentes formas. Para los programadores el código comunica mejor cuanto más simple sea. Si el código es complejo hay que esforzarse para hacerlo legible. El código autodocumentado es más fiable que los comentarios ya que estos últimos pronto quedan desfasados con el código a medida que es modificado. Debe comentarse solo aquello que no va a variar, por ejemplo el objetivo de una clase o la funcionalidad de un método.

Las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de como utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la programación por parejas. La comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo. El cliente decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.



Valores: Retroalimentación o feedback

Retroalimentación concreta y frecuente del cliente, del equipo y de los usuarios finales da una mayor oportunidad de dirigir el esfuerzo eficientemente.

Al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real.

Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en lo que es más importante.

Considérense los problemas que derivan de tener ciclos muy largos. Meses de trabajo pueden tirarse por la borda debido a cambios en los criterios del cliente o malentendidos por parte del equipo de desarrollo. El código también es una fuente de retroalimentación gracias a las herramientas de desarrollo. Por ejemplo, las pruebas unitarias informan sobre el estado de salud del código. Ejecutar las pruebas unitarias frecuentemente permite descubrir fallos debidos a cambios recientes en el código.



Valores: Coraje o valentía

El coraje (valor) existe en el contexto de los otros 3 valores. (si funciona...mejoralo).

Muchas de las prácticas implican valentía. Una de ellas es siempre diseñar y programar para hoy y no para mañana. Esto es un esfuerzo para evitar empantanarse en el diseño y requerir demasiado tiempo y trabajo para implementar el resto del proyecto. La valentía le permite a los desarrolladores que se sientan cómodos con reconstruir su código cuando sea necesario. Esto significa revisar el sistema existente y modificarlo si con ello los cambios futuros se implementarán más fácilmente. Otro ejemplo de valentía es saber cuándo desechar un código: valentía para quitar código fuente obsoleto, sin importar cuanto esfuerzo y tiempo se invirtió en crear ese código. Además, valentía significa persistencia: un programador puede permanecer sin avanzar en un problema complejo por un día entero, y luego lo resolverá rápidamente al día siguiente, solo si es persistente.



Valores: Respeto

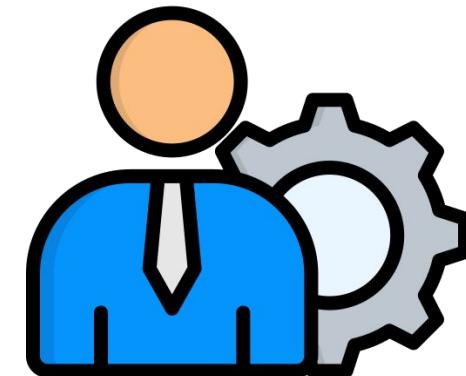
El respeto se manifiesta de varias formas. Los miembros del equipo se respetan los unos a los otros, porque los programadores no pueden realizar cambios que hacen que las pruebas existentes fallen o que demore el trabajo de sus compañeros. Los miembros respetan su trabajo porque siempre están luchando por la alta calidad en el producto y buscando el diseño óptimo o más eficiente para la solución a través de la refactorización del código. Los miembros del equipo respetan el trabajo del resto no haciendo menos a otros, una mejor autoestima en el equipo eleva su ritmo de producción.



Características

Al crear la serie de libros sobre XP, Beck remarca tres elementos, con subelementos por cada uno, que se van a ir interrelacionando para crear el mejor desarrollo XP posible, y alcanzar los objetivos. Estos son:

- Valores:
 - ... ya los vimos 
- **Características:**
 - Equipo completo
 - Pruebas unitarias continuas
 - Programación en parejas
 - Refactorización del código
 - Propiedad del código compartida
 - Versiones pequeñas
 - Integración continua
 - Metáforas
 - Ritmo sostenible
 - Simplicidad en el código (como la del valor)
- Roles:
 - ... ya los vamos a ver ahora.



Características: Equipo completo

Forman parte del equipo todas las personas que tienen algo que ver con el proyecto, incluido el cliente y el responsable del proyecto.

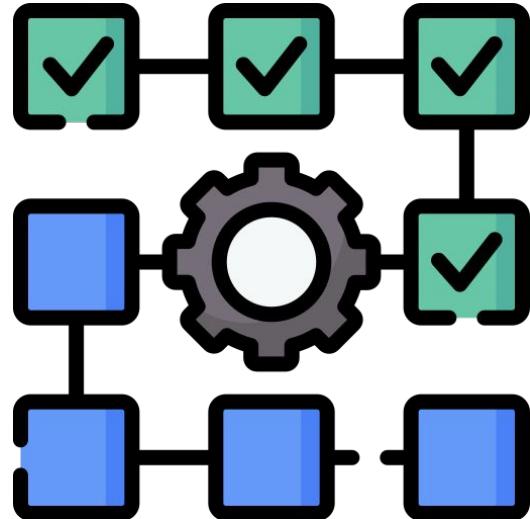
Todos los miembros del equipo de desarrollo tienen habilidades y conocimientos suficientes para realizar cualquier tarea necesaria para el proyecto. Esto significa que los desarrolladores no están limitados a roles específicos y están dispuestos a asumir diferentes responsabilidades según sea necesario.

El concepto de equipo completo se basa en la premisa de que la colaboración y la comunicación efectiva son fundamentales para el éxito del proyecto. Al tener a todos los miembros del equipo capacitados y comprometidos para asumir diferentes tareas, se fomenta un ambiente de trabajo colaborativo y se promueve la flexibilidad en la asignación de trabajo.



Características: Pruebas unitarias continuas

Una característica primordial es la de las pruebas unitarias continuas, frecuentemente repetidas y automatizadas, incluyendo pruebas de regresión. Se aconseja escribir el código de la prueba antes de la codificación. Véase, por ejemplo, las herramientas de prueba JUnit orientada a Java, DUnit orientada a Delphi, NUnit para la plataforma .NET o PHPUnit para PHP. Estas tres últimas inspiradas en JUnit, la cual, a su vez, se inspiró en SUnit, el primer framework orientado a realizar tests, realizado para el lenguaje de programación Smalltalk.



Características: Programación en parejas

Se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto. La mayor calidad del código escrito de esta manera -el código es revisado y discutido mientras se escribe- es más importante que la posible pérdida de productividad inmediata.

A su vez, en la programación en parejas, uno de los programadores actúa como el "conductor" mientras que el otro es el "navegante". El conductor es responsable de escribir el código, mientras que el navegante revisa y ofrece sugerencias. Estos roles se pueden intercambiar periódicamente para mantener una colaboración equilibrada y asegurarse de que ambos programadores estén involucrados activamente en el proceso.



Características: Refactorización del código

Reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo.

La refactorización se enfoca en abordar los llamados "**malos olores**" en el código, que son señales de diseño o implementación deficiente. Estos pueden incluir duplicación de código, código largo y complicado, métodos o clases demasiado acoplados, entre otros. Identificar y eliminar estos malos olores mejora la mantenibilidad, la extensibilidad y la comprensión del código.

La refactorización puede implicar una variedad de técnicas, como la extracción de métodos o clases para eliminar duplicación de código, la simplificación de estructuras de control complejas, la mejora de nombres de variables y métodos para una mayor legibilidad, y la reorganización de la estructura del código para una mejor modularidad. XP enfatiza la importancia de utilizar técnicas de refactorización seguras y respaldadas por pruebas para garantizar que los cambios no introduzcan errores en el código.



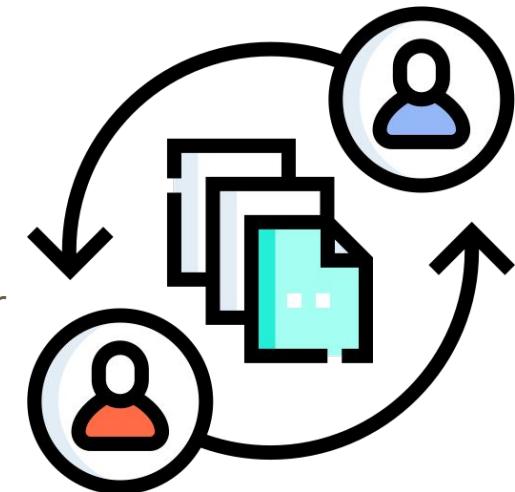
Características: Propiedad del código compartida

Cualquiera puede y debe tocar y conocer cualquier parte del código.

En vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados.

A su vez, la propiedad del código compartida fomenta una colaboración activa entre los miembros del equipo. Se alienta a todos a revisar, analizar y brindar retroalimentación sobre el código desarrollado por otros. Esto promueve una mayor comunicación y comprensión del código en todo el equipo, lo que a su vez mejora la calidad general del software.

Al compartir la propiedad del código, se fomenta la transferencia de conocimiento entre los miembros del equipo. Cada miembro tiene la oportunidad de comprender diferentes partes del sistema y aprender de las habilidades y experiencias de otros. Esto no solo mejora la capacidad del equipo para mantener y mejorar el código, sino que también crea un ambiente de aprendizaje continuo.



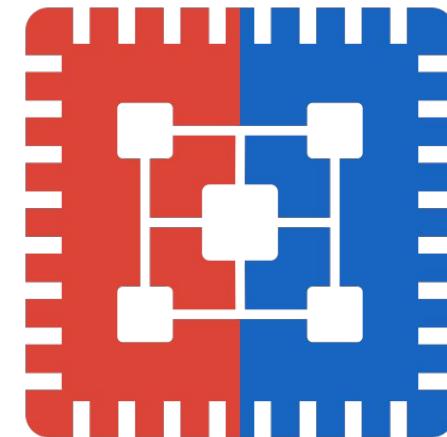
Características: Versiones pequeñas o microincrementos

Las mini-versiones deben ser lo suficientemente pequeñas como para poder hacer una cada pocas semanas. Deben ser versiones que ofrezcan algo útil al usuario final y no trozos de código que no pueda ver funcionando.

Para lograr las versiones pequeñas, los equipos de XP trabajan en ciclos de desarrollo cortos llamados iteraciones o sprints. Estas iteraciones típicamente duran de una a tres semanas y se enfocan en entregar un conjunto de características o mejoras definidas y probadas.

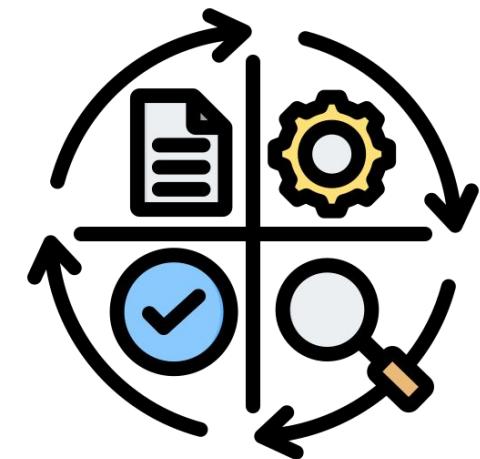
La práctica de versiones pequeñas prioriza la entrega de funcionalidad clave y de alto valor para el cliente. Se busca identificar las características más importantes y desarrollarlas en incrementos pequeños y manejables. Esto permite una mayor adaptabilidad a medida que los requisitos pueden cambiar o surgir nuevas prioridades.

Al entregar versiones pequeñas, el equipo de desarrollo puede obtener retroalimentación temprana del cliente o usuario final. Esto facilita la identificación de posibles problemas, errores o cambios necesarios. El equipo puede ajustar rápidamente su enfoque y hacer mejoras en función de la retroalimentación recibida, lo que resulta en un producto final más alineado con las necesidades y expectativas del cliente.



Características: Integración continua

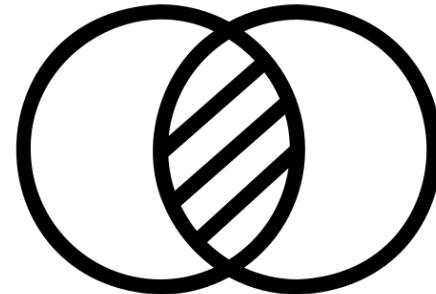
Debe tenerse siempre un ejecutable del proyecto que funcione y en cuanto se tenga una nueva pequeña funcionalidad, debe recompilarse y probarse. Es un error mantener una versión congelada dos meses mientras se hacen mejoras y luego integrarlas todas de golpe. Cuando falle algo, no se sabe qué es lo que falla de todo lo que hemos metido.



Características: Metáforas

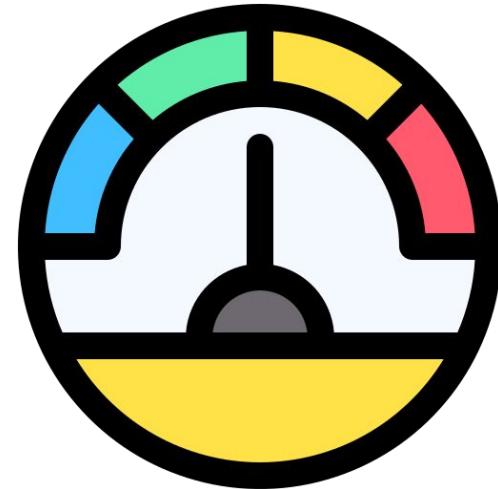
Hay que buscar unas frases o nombres que definen cómo funcionan las distintas partes del programa, de forma que sólo con los nombres se pueda uno hacer una idea de qué es lo que hace cada parte del programa. Un ejemplo claro es el "recolector de basura" de java. Ayuda a que todos los programadores (y el cliente) sepan de qué estamos hablando y que no haya malentendidos.

La metáfora puede ayudar a simplificar el diseño y la arquitectura del sistema. Al relacionar el sistema con algo familiar y comprensible, se pueden tomar decisiones de diseño más intuitivas y coherentes. Además, la metáfora puede proporcionar una estructura conceptual que oriente el desarrollo y facilite la comprensión del sistema en su conjunto.



Características: Ritmo sostenible

Se debe trabajar a un ritmo que se pueda mantener indefinidamente. Esto quiere decir que no debe haber días muertos en que no se sabe qué hacer y que no se deben hacer un exceso de horas otros días. Al tener claro semana a semana lo que debe hacerse, hay que trabajar duro en ello para conseguir el objetivo cercano de terminar una historia de usuario o mini-versiones.

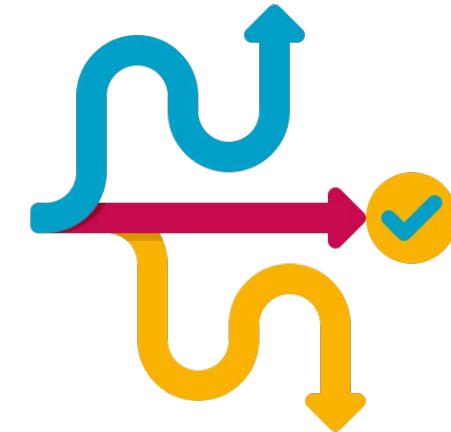


Características: Simplicidad en el código

Es la mejor manera de que las cosas funcionen. Cuando todo funcione se podrá añadir funcionalidad si es necesario. La programación extrema apuesta que es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo.

La simplicidad en el código promueve la legibilidad y la comprensión. Se busca escribir un código claro y conciso que pueda ser fácilmente entendido por otros miembros del equipo. Se prefieren nombres de variables y métodos significativos y se evita la complejidad innecesaria en la estructura y el flujo del código. Esto facilita la colaboración y el mantenimiento a largo plazo.

Se evita la sobrecarga y la complejidad innecesaria. Se enfoca en encontrar la solución más simple y directa para un problema en lugar de implementaciones complicadas o excesivamente abstractas. Esto facilita la comprensión, el razonamiento y la corrección de errores en el código.

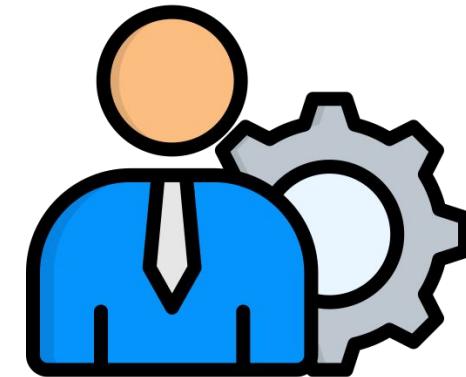
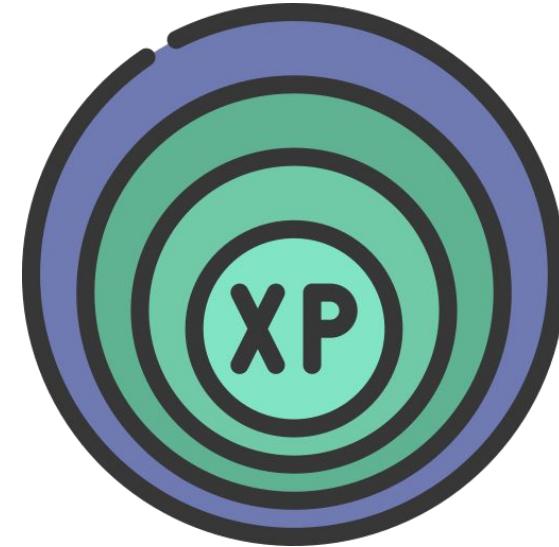


(y todo lo de la diapositiva 8)

Roles

Al crear la serie de libros sobre XP, Beck remarca tres elementos, con subelementos por cada uno, que se van a ir interrelacionando para crear el mejor desarrollo XP posible, y alcanzar los objetivos. Estos son:

- Valores:
 - ... ya los vimos 
- Características:
 - ...tambien las vimos 
- **Roles (pag 105):**
 - Programador
 - Test developer
 - Cliente
 - Tester
 - Tracker
 - Entrenador (coach)
 - Consultor
 - Gestor (Big Boss)



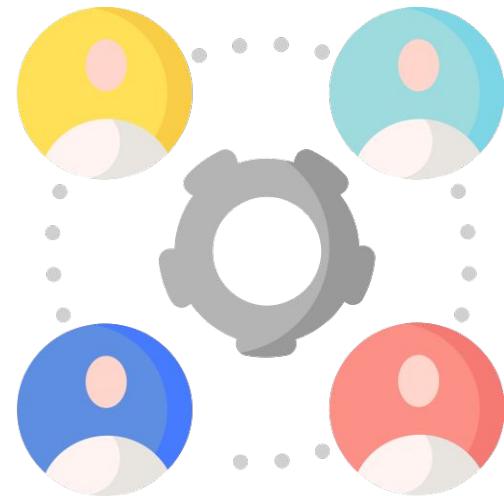
Roles

Programador: Produce el código del sistema. Es la esencia del equipo.

Test developer: Produce el código de los test unitarios del sistema. Es uno de los roles más importantes.

Cliente: Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar el mayor valor de negocio.

Tester: Interpreta el pedido del cliente y ayuda al equipo de desarrollo a escribir las pruebas funcionales. Ejecuta pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.



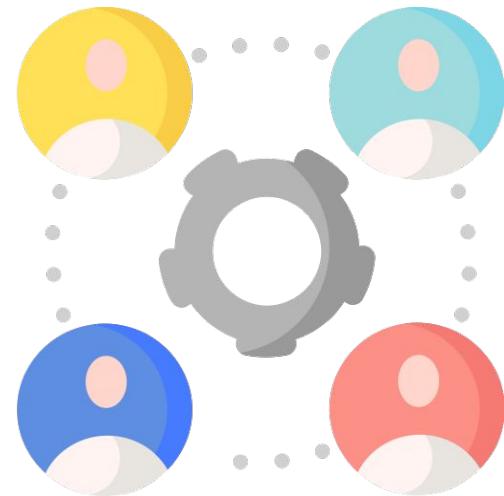
Roles

Tracker: Es el encargado de seguimiento. Proporciona retroalimentación al equipo. Debe verificar el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, comunicando los resultados para mejorar futuras estimaciones.

Entrenador (coach): Responsable del proceso global. Guía a los miembros del equipo para seguir el proceso correctamente.

Consultor: Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto. Ayuda al equipo a resolver un problema específico. Además este tiene que investigar según los requerimientos.

Gestor (Big Boss): Es el dueño de la tienda y el vínculo entre clientes y programadores. Su labor esencial es la coordinación.



Características

En conclusión, XP es una metodología ágil centrada en **potenciar las relaciones interpersonales** como clave para el éxito en desarrollo de software, promoviendo **el trabajo en equipo**, preocupándose por el **aprendizaje de los desarrolladores**, y propiciando un **buen clima de trabajo**. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, **comunicación fluida** entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.



Ventajas y desventajas

Ventajas:

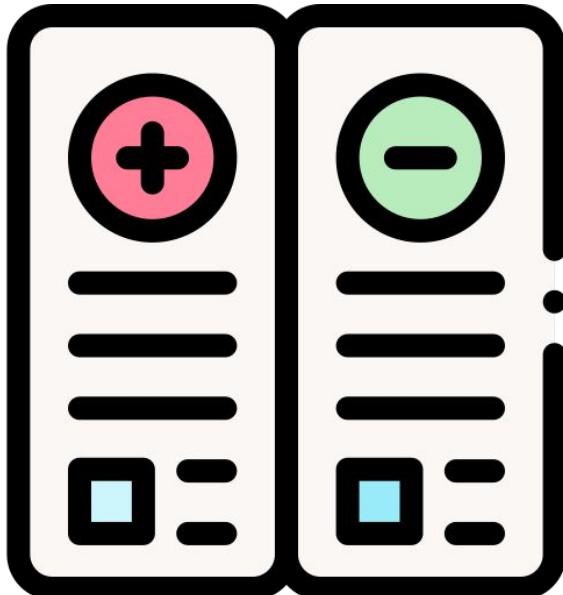
- Programación organizada.
- Menor tasa de errores.
- Satisfacción del programador.

Desventajas:

- Es recomendable emplearlo solo en proyectos a corto plazo.

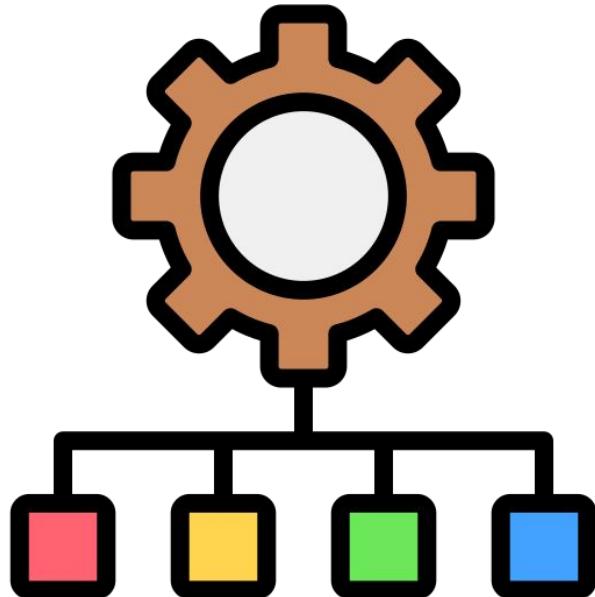
Adecuado para proyectos:

- De corto plazo.



Programación organizada

La programación en parejas, escritura de pruebas antes de implementar código, iteraciones cortas y retroalimentación continua permiten a los equipos abordar rápidamente cualquier problema o error que surja durante el desarrollo. A su vez, darle prioridad a la calidad y la corrección del código, hacen que XP ayude a minimizar los errores y aumentar la estabilidad y confiabilidad del producto final.



Satisfacción del programador

XP promueve un ambiente de trabajo colaborativo y de confianza, donde los programadores tienen una mayor participación en el proceso de toma de decisiones y en la planificación de las tareas. Esto ayuda a fomentar un sentido de propiedad y empoderamiento en el equipo de desarrollo, lo que a su vez aumenta la satisfacción laboral. Además, la programación en parejas y la revisión continua del código brindan oportunidades de aprendizaje y crecimiento profesional, ya que los programadores pueden compartir conocimientos y habilidades entre ellos. La colaboración estrecha también puede reducir el estrés y la presión individual, ya que el equipo se apoya mutuamente en el desarrollo del software. En general, XP busca crear un entorno de trabajo satisfactorio y enriquecedor para los programadores, lo que puede conducir a una mayor motivación, productividad y retención del talento.



Solamente adecuado para proyectos de corto plazo

Escalabilidad limitada: XP se centra en la colaboración y la comunicación constante entre los miembros del equipo. Si el equipo de desarrollo es grande, puede ser difícil mantener este nivel de interacción y coordinación eficientemente. A medida que el número de personas involucradas aumenta, la complejidad de la comunicación y la coordinación también crece, lo que dificulta la implementación efectiva de las prácticas de XP. Por lo tanto, a largo plazo o en proyectos de mayor envergadura, puede resultar más complicado aplicar las técnicas y principios de XP de manera efectiva.

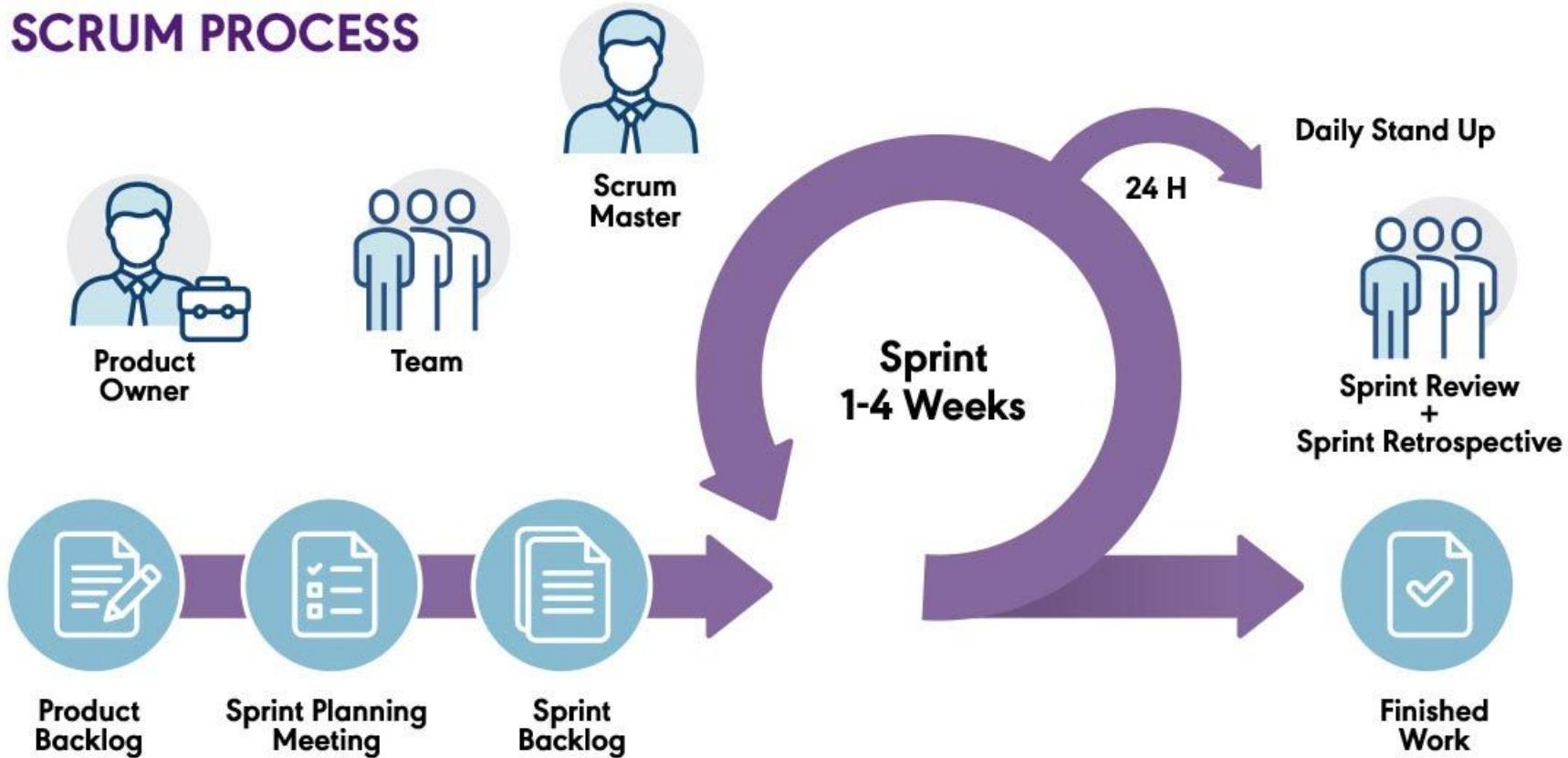


Sólo adecuado para proyectos de corto plazo

Necesidad de compromiso y experiencia: XP requiere un alto nivel de compromiso por parte de todos los miembros del equipo, así como una sólida experiencia en desarrollo ágil y programación en parejas. No todos los equipos o desarrolladores están preparados para adoptar XP y aplicar sus prácticas de manera efectiva. Requiere un cambio de mentalidad y una comprensión profunda de los principios y técnicas de XP. Si un equipo no está adecuadamente capacitado o no tiene la experiencia necesaria en XP, puede resultar difícil implementar correctamente esta metodología y obtener los beneficios esperados.



SCRUM PROCESS



SCRUM

Scrum es un marco de trabajo para desarrollo ágil de software que se ha expandido a otras industrias.

Es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo y obtener el mejor resultado posible de proyectos.

Scrum es el marco de trabajo **más utilizado hoy en día** por empresas de desarrollo de software, gracias a su gran capacidad para desarrollar incrementalmente, y su organización inmaculada.



Origen de Scrum

Este modelo fue identificado y definido por Ikujiro Nonaka y Takeuchi a principios de los 80, al analizar cómo desarrollaban los nuevos productos las principales empresas de manufactura tecnológica (Canon, Honda, Fuji, Epson, Hewlett-Packard). En su estudio, Nonaka y Takeuchi compararon la nueva forma de trabajo en equipo, con el avance en formación de **scrum** de los jugadores de **Rugby**, a raíz de lo cual quedó acuñado el término “scrum” para referirse a ella.



Aunque esta forma de trabajo surgió en empresas de productos tecnológicos, es apropiada para cualquier tipo de proyecto con requisitos inestables y para los que requieren rapidez y flexibilidad, situaciones frecuentes en el desarrollo de determinados sistemas de software.

Origen de Scrum

Nonaka y Takeuchi describieron una nueva manera de desarrollar productos comerciales, que incrementarían ampliamente la velocidad y flexibilidad, basado en casos de estudio de las compañías mencionadas previamente. Estos casos de estudio fueron llevados a cabo principalmente por Schwaber y Ogunnaike, en la Universidad de Delaware.

En estos estudios, Ogunnaike advirtió que los intentos de desarrollar productos complejos, como software, que no estuvieran basados en el empirismo, estaban condenados a mayores riesgos y altas tasas de fracaso a medida que cambiaban las condiciones iniciales y los supuestos. El empirismo, utilizando inspección y adaptación frecuentes, con flexibilidad y transparencia, es el enfoque más adecuado.

Canon



FUJI
innovative spirit

EPSON

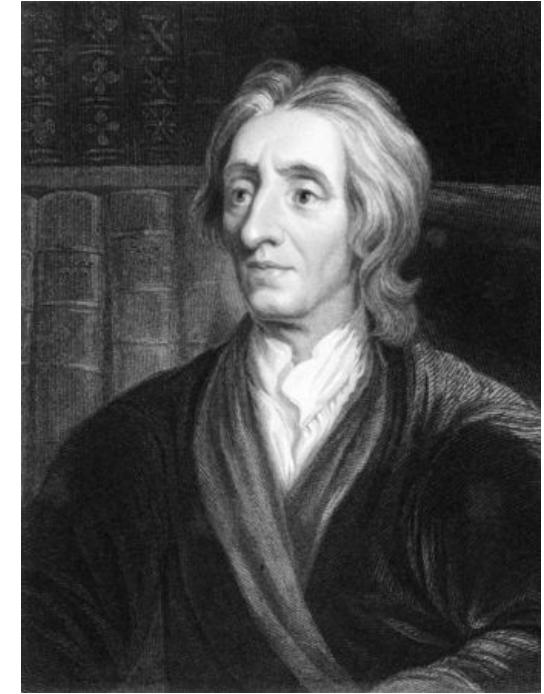


Empirismo? 🤔

Empirismo (Locke), dice que la experiencia y la evidencia es el punto de partida y fundamento de todo conocimiento posible. Es así que el empirismo sostiene que todo nuestro conocimiento del mundo proviene de los sentidos, a su vez que la experiencia misma posee un papel epistémico insustituible en la justificación de la creencia.

Es así que la idea de Scrum es tomar decisiones basados en la información concreta obtenida de la observación que muestra el progreso del desarrollo de producto, los cambios en el mercado y los comentarios de los clientes.

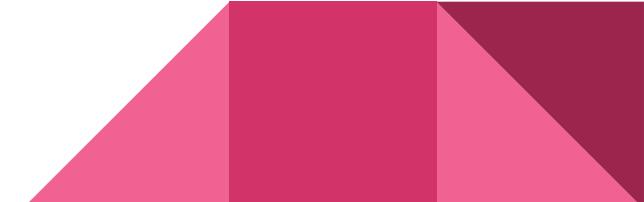
Lo contrario al empirismo (o scrum), es usar planificación previa, procesos definidos, planes predictivos, hechos no concretos.



Scrum

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad, y la productividad, son fundamentales.



Sprint

Sprint es el nombre que va a recibir cada uno de los ciclos o iteraciones que vamos a tener dentro de un proyecto Scrum.

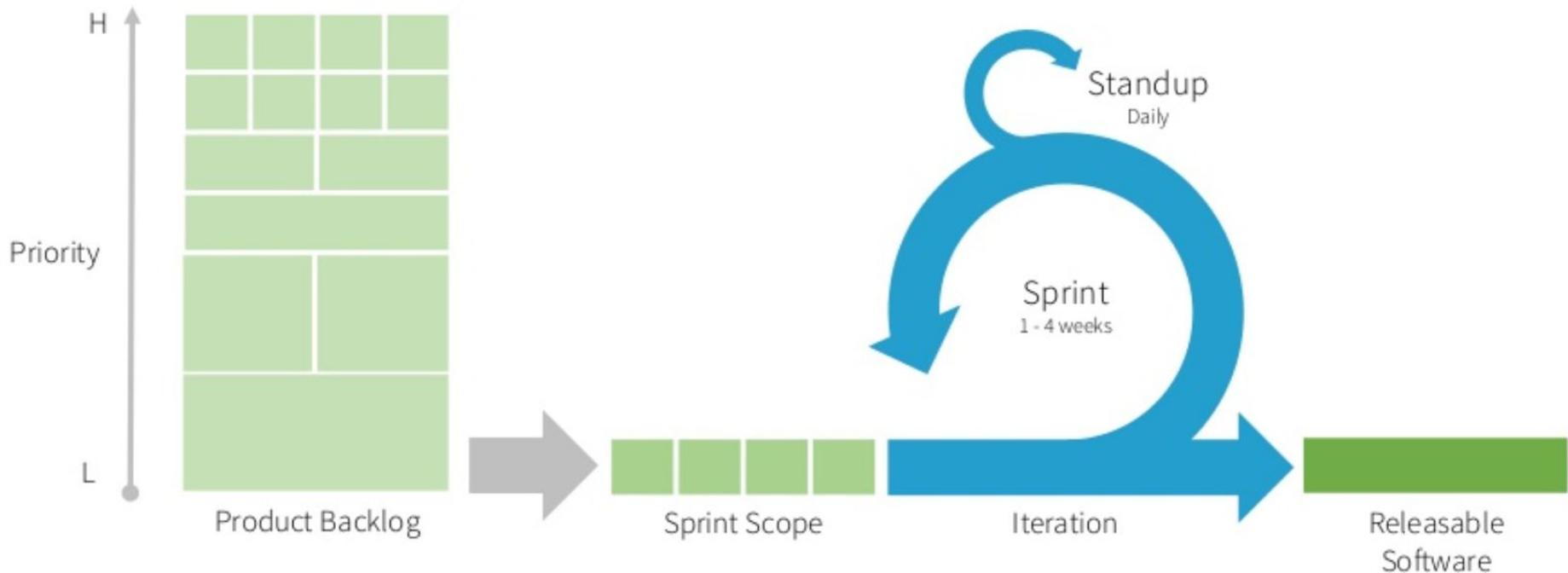
Nos van a permitir tener un ritmo de trabajo con un tiempo prefijado, siendo la duración habitual de un Sprint unas dos semanas, aunque lo que la metodología dice, es que debería estar entre una semana y un máximo de un mes.

En cada Sprint o cada ciclo de trabajo lo que vamos a conseguir es lo que se denomina un entregable o incremento del producto, que aporte valor al cliente.

La idea es que cuando tenemos un proyecto bastante largo, por ejemplo un proyecto de 12 meses, vamos a poder dividir ese proyecto en doce Sprints de un mes cada uno. En cada uno de esos Sprints vamos a ir consiguiendo un producto, que siempre, y esto es muy importante, sea un producto que esté funcionando.



Traditional Scrum

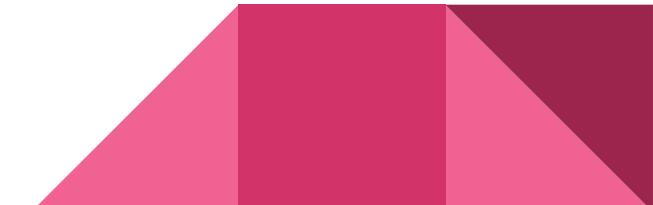


Actores o roles principales

El Dueño del Producto (**Product Owner**), representa a los inversionistas o las personas que requieren el software.

El Director Scrum (**Scrum Master**), es el facilitador del equipo, supervisa al equipo y verifica que se lleven a cabo las reuniones y se haga uso de los artefactos. Ayuda a que el proyecto tenga éxito. Elimina los problemas e impedimentos que se pudieran presentar. Ayuda a los miembros del equipo a tomar decisiones responsables y los asesora en todas las maneras posibles para que alcancen sus objetivos.

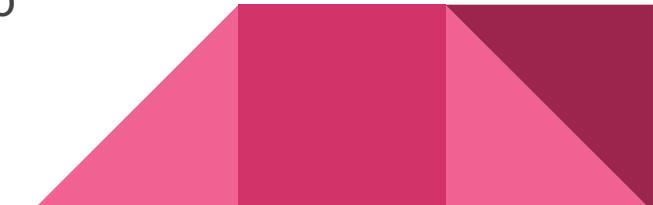
Los miembros del equipo (**Team Members**), son los que desarrollan el software, poseen las capacidades técnicas para fabricar el producto.



Product Owner

El **Product Owner** es el responsable de maximizar el valor del producto resultante del trabajo del equipo Scrum. La forma en que se hace esto puede variar ampliamente entre organizaciones, equipos Scrum e individuos.

Como miembro del Equipo Scrum, el **Product Owner** proporciona claridad al equipo sobre la visión y el objetivo de un producto. Todo el trabajo se deriva y se prioriza, en función del objetivo del producto, para ofrecer valor a todas las partes interesadas, incluidas las de su organización y todos los usuarios, tanto internos como externos. Los Product Owners identifican, miden y maximizan el valor a lo largo de todo el ciclo de vida del producto.



Scrum Master

El Scrum Master es responsable de promover y apoyar el modelo Scrum. Los Scrum Masters hacen esto al ayudar a todos a comprender la teoría, las prácticas, las reglas y los valores de Scrum. Tienen dos funciones principales:

- Gestionar el proceso Scrum:** el Scrum Master se encarga de gestionar y asegurar que el proceso Scrum se lleva a cabo correctamente, así como de facilitar la ejecución del proceso y sus mecánicas. Siempre atendiendo a los tres pilares del control empírico de procesos y haciendo que la metodología sea una fuente de generación de valor.
- Eliminar impedimentos:** esta función del Scrum Master indica la necesidad de ayudar a eliminar progresiva y constantemente impedimentos que van surgiendo en la organización y que afectan a su capacidad para entregar valor, así como a la integridad de esta metodología. El Scrum Master debe ser el responsable de velar porque Scrum se lleve adelante, transmitiendo sus beneficios a la organización facilitando su implementación.



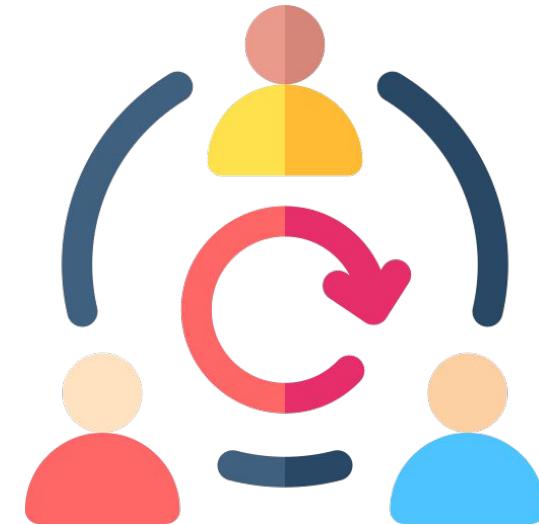
Team Members o equipo de desarrollo

El equipo de desarrollo suele estar formado por entre 3 a 9 profesionales que se encargan de **desarrollar el producto**, auto-organizándose y auto-gestionándose para conseguir entregar un incremento de software al final del ciclo de desarrollo.

El equipo de desarrollo se encargará de crear un incremento terminado a partir de los elementos del Product Backlog seleccionados (Sprint Backlog) durante el Sprint Planning. (ahí vemos qué significa esto).

Es importante que en la metodología Scrum todos los miembros del equipo de desarrollo conozcan su rol, siendo solo uno común para todos, independientemente del número de miembros que tenga el equipo y cuales sean sus roles internos. Cómo el equipo de desarrollo decida gestionarse internamente es su propia responsabilidad y tendrá que rendir cuentas por ello como uno solo; hay que evitar intervenir en sus dinámicas.

Habitualmente son equipos 'cross-funcional', capaces de generar un incremento terminado de principio a fin, sin otras dependencias externas.

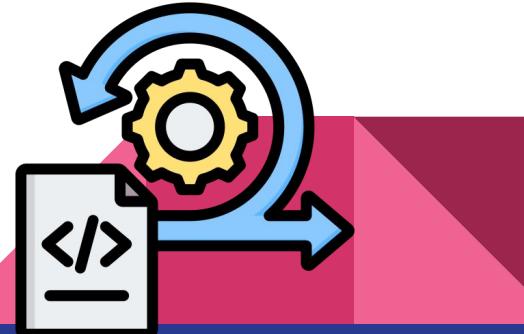


Tres artefactos

La Pila de Producto (**Product BackLog**), que es una lista de todas las cosas “Por Realizar” del proyecto. Esta lista es confeccionada por el Dueño del Producto de acuerdo a los requerimientos y esta ordenada por la prioridad que tiene cada elemento en la pila. Es decir, de mayor a menor importancia.

La Pila del Sprint (**Sprint BackLog**), son las actividades que se van a realizar dentro de un sprint.

El Grafico de Trabajo Pendiente (**Burndown Chart**). Representa visualmente el trabajo que está por hacer versus el tiempo restante del proyecto. El trabajo pendiente se representa en el eje vertical y el tiempo en el eje horizontal. Es útil para predecir en que tiempo se terminaría todo el trabajo, incluso se puede establecer el ritmo de avance del proyecto del equipo.

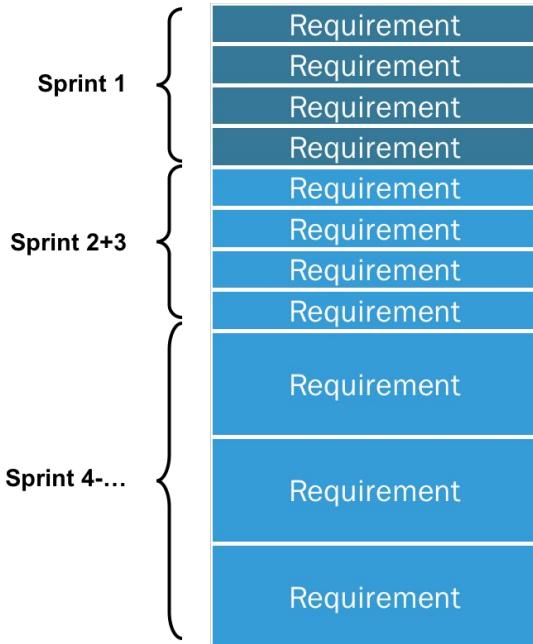


Product BackLog

El Product Backlog (o pila de producto) es un listado de todas las tareas que se pretenden hacer durante el desarrollo de un proyecto.

Algunos product backlog pueden asociarse con proyectos de varios años, incluso.

Todas las tareas deben listarse en el product backlog, para que estén visibles ante todo el equipo y se pueda tener una visión panorámica de todo lo que se espera realizar. Si el proyecto o producto amerita una lista larga, podemos tener cientos o miles de actividades en el product backlog. Se asignan prioridades sobre el product backlog, en base a las necesidades del cliente y la complejidad de cada actividad. Así mismo, de forma sucesiva, cada sprint produce detalles adicionales, en el diseño, en la funcionalidad, y la verificación de actividades.



Más detalle

Historia de usuario (4-8 días/programador)

RELEASE ACTUAL
PRÓXIMO SPRINT

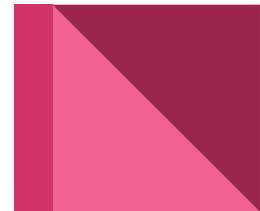
Epic (2-4 semanas/equipo)

RELEASE SIGUIENTE

FUTURAS RELEASES

Temas

Menos detalle



Sprint BackLog

El sprint backlog es básicamente una lista de tareas identificadas por el scrum team; ésta deberá ser completada durante cada sprint.

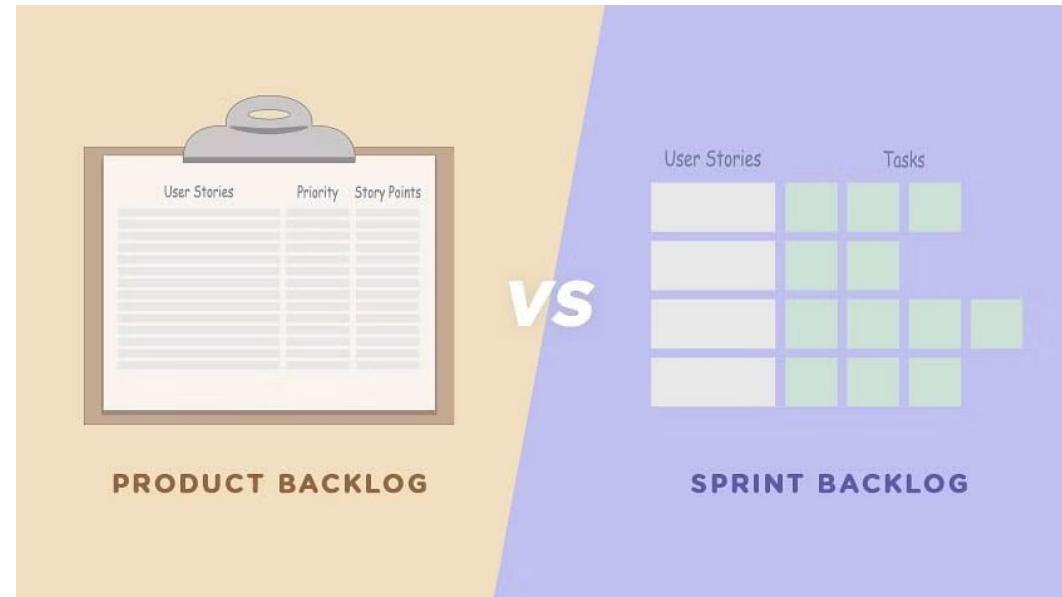
El sprint backlog es representado a través de un tablero de tareas; hace visible todo el trabajo necesario para alcanzar el compromiso que se hizo con el product owner para el sprint. Permite ver las tareas donde el equipo está teniendo problemas y no avanza, para tomar decisiones al respecto.

Para cada uno de los objetivos/requisitos del proyecto se muestran las tareas a cubrir; el esfuerzo pendiente para finalizarlas y la auto asignación que han hecho los miembros del equipo, también son visibles.



Sprint BackLog vs Product BackLog

La diferencia principal entre Sprint y Product BackLog, es que el product BackLog es una parte generalizada, mientras que el sprint BackLog es específico y más orientado a detalle. Ambos backlogs requieren una gestión estratégica, y diferente entre ellos.

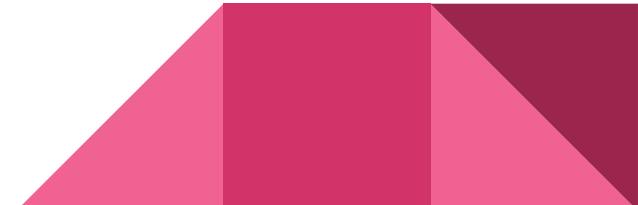


Objetivo del Sprint	Pendiente	En progreso	Completo
<i>Habilitar todas las partes esenciales de la tienda online para permitir que los usuarios puedan experimentar un proceso de compra completo. Esto hará que otras características de la página web sean más significativas.</i>			Item #1 t.1.6 t.1.1 t.1.3 t.1.5 t.1.2
	Item #2 t.2.7	t.2.6 t.2.5	t.2.1 t.2.2 t.2.3 t.2.4
	Item #3 t.3.4 t.3.5 t.3.3 t.3.2	t.3.1	
	Item #4 t.4.4 t.4.2 t.4.5		
	Item #5		

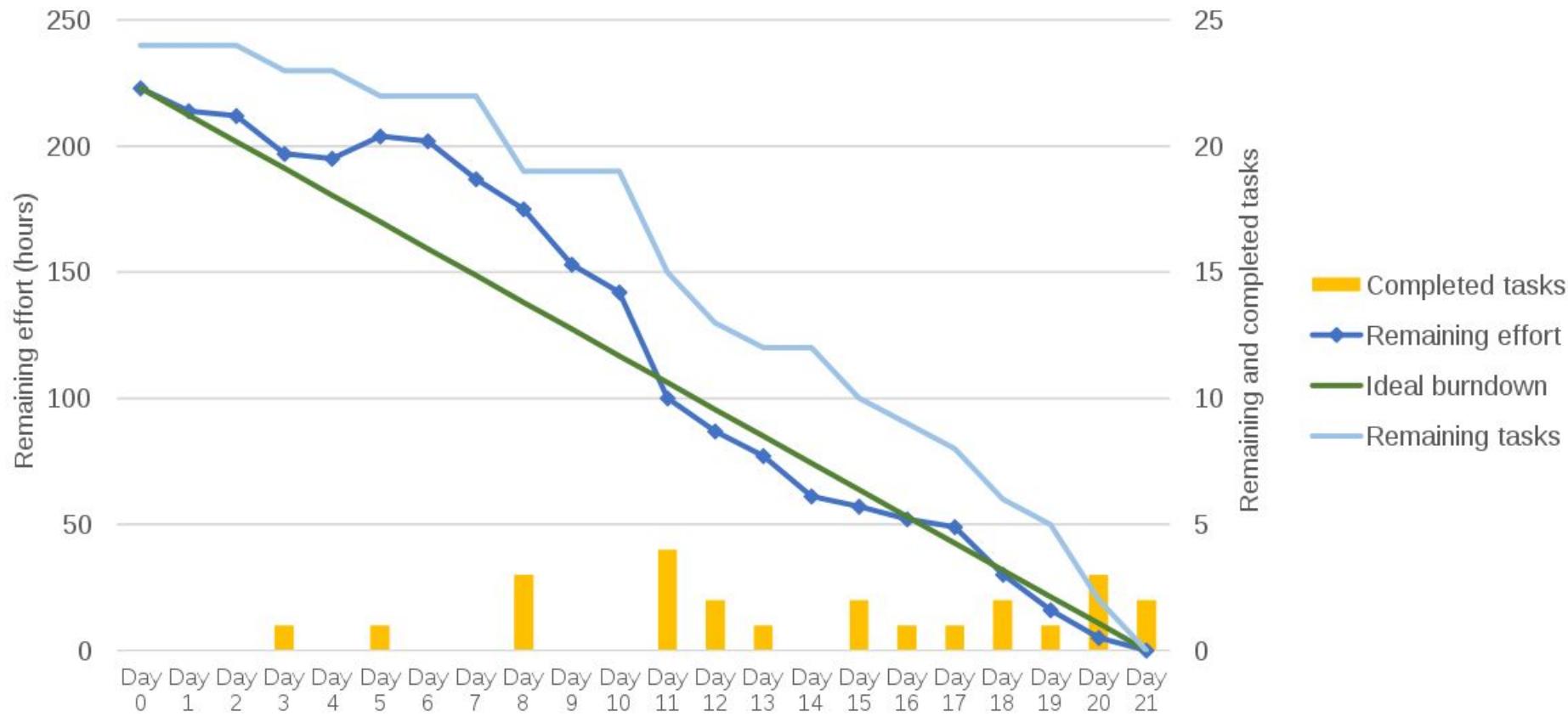
Burndown Chart

Un Burndown Chart, diagrama de quemado o gráfica de trabajo pendiente es una gráfica en la que podemos ver el estado del progreso de un Sprint.

Un burndown chart puede ofrecernos mucha más información que el avance de un Sprint si sabemos interpretarla correctamente y podremos ayudar mucho más a los equipos gracias a esto.



Sample Burndown Chart



Construcción de un Burndown Chart

1. Eje X: Tiempo

En el eje X de nuestro burndown representamos el tiempo. Pero un tiempo finito. En el caso de un Sprint, cuando $x=0$ debe ser el inicio del Sprint y el último valor de nuestro eje X, (en el ejemplo de la diapositiva pasada, 21 días), debe ser el final del Sprint.

2. Eje Y: Cantidad de trabajo

En el eje Y de la gráfica reflejamos la cantidad de trabajo. Es el trabajo que hay comprometido para realizar en el tiempo estipulado. En el caso de Scrum, sería el Sprint Backlog. Para poder crearla gráfica es necesario que las tareas estén estimadas (da igual que sean puntos de historia, jornadas, horas, etc). De esta forma podremos pintar nuestra línea guía.

Construcción de un Burndown Chart

3. Línea guía: Referencia

Llamamos línea guía a la diagonal que une el último valor de nuestro eje Y (el máximo de trabajo comprometido) con el último valor del eje X (la que dice “Ideal Burndown”). Esta será nuestra referencia para saber si vamos bien, vamos mejor que bien, o si está todo mal.

4. Línea de progreso

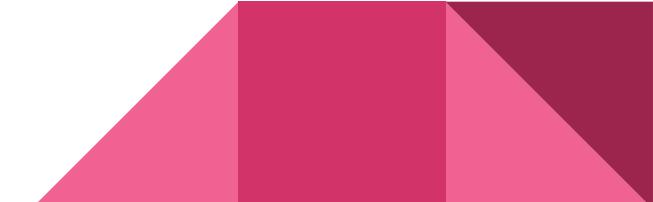
Arrancaremos una nueva línea que irá actualizándose con cada tarea (que, recordemos, debe estar estimada) que se termine.

Por cada tarea terminada, bajaremos la cantidad proporcional a la estimación de la tarea terminada, y, en una situación ideal, tendremos un burndown chart en el que las tareas pendientes y el tiempo estimado “bajan” de manera ideal.

El proceso

En Scrum un proyecto se ejecuta en bloques temporales cortos y fijos (iteraciones de un mes natural y hasta de dos semanas, si así se necesita). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto final que sea susceptible de ser entregado con el mínimo esfuerzo al cliente cuando lo solicite.

El proceso parte de la lista de objetivos/requisitos priorizada del producto, que actúa como plan del proyecto. En esta lista el cliente prioriza los objetivos balanceando el valor que le aportan respecto a su coste y quedan repartidos en iteraciones y entregas.



Planificación de la iteración (Sprint Planning)

El **primer día** de la iteración se realiza la reunión de planificación de la iteración. Tiene dos partes:

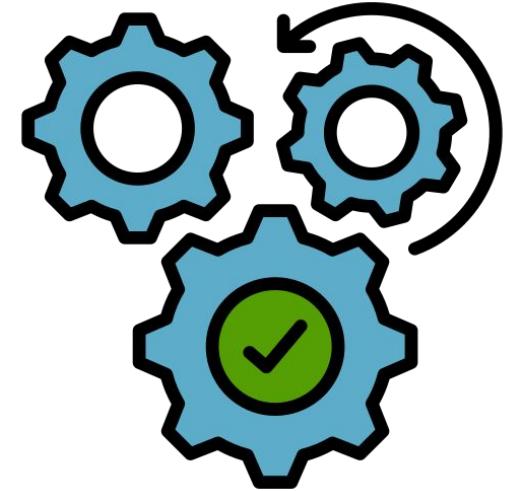
Selección de requisitos (4 horas máximo). El cliente presenta al equipo la lista de requisitos priorizada del producto o proyecto. El equipo pregunta al cliente las dudas que surgen y selecciona los requisitos más prioritarios que se compromete a completar en la iteración, de manera que puedan ser entregados si el cliente lo solicita.



Planificación de la iteración (4 horas máximo). El equipo elabora la lista de tareas de la iteración necesarias para desarrollar los requisitos a que se ha comprometido. La estimación de esfuerzo se hace de manera conjunta y los miembros del equipo se autoasignan las tareas.

Ejecución de la iteración (Scrum/Sprint Planning)

Planeación del Sprint (**Scrum Planning**), con la ayuda del Product Owner y el Scrum Master y el equipo, se compromete con las actividades que se deben realizar de la Pila del Producto. Es decir se seleccionan actividades prioritarias y relacionadas y con la restricción del tiempo existente, para que estas formen parte de la siguiente iteración (Sprint).



Ejecución de la iteración (Daily Scrum)

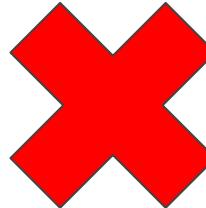
Reunión Diaria de Scrum (**Daily Scrum**), esta actividad se lleva a cabo todos los días y debe durar quince minutos o menos (si bien suele durar alrededor de 40 minutos en la vida real). Cada miembro del equipo debe comunicar al resto del equipo y al Scrum Master, tres ítems indicados en la Tabla de Tareas: lo que se realizó, lo que se va realizar el día de hoy, y si existe algún impedimento para llevar a cabo el trabajo. Cada tarea debe evolucionar desde Por hacer (**To Do**), En Progreso (**In Progress**), Realizadas (**Done**). Al finalizar el sprint todas las tareas deben estar realizadas. Esta reunión es muy importante porque acá los miembros comparten información, para determinar la mejor manera de cumplir sus tareas y poder terminar con las metas del Sprint. Se manifiestan los riesgos encontrados y es útil porque permite que los miembros del equipo sean más eficaces.



Ejecución de la iteración (Sync meeting)

Cada día el equipo realiza una reunión de sincronización (15 minutos máximos). Cada miembro del equipo inspecciona el trabajo que el resto está realizando (dependencias entre tareas, progreso hacia el objetivo de la iteración, obstáculos que pueden impedir este objetivo) para poder hacer las adaptaciones necesarias que permitan cumplir con el compromiso adquirido. En la reunión cada miembro del equipo responde a tres preguntas:

¿Qué hice desde la última reunión de sincronización?



¿Qué voy a hacer a partir de este momento?

¿Qué impedimentos tengo o voy a tener?



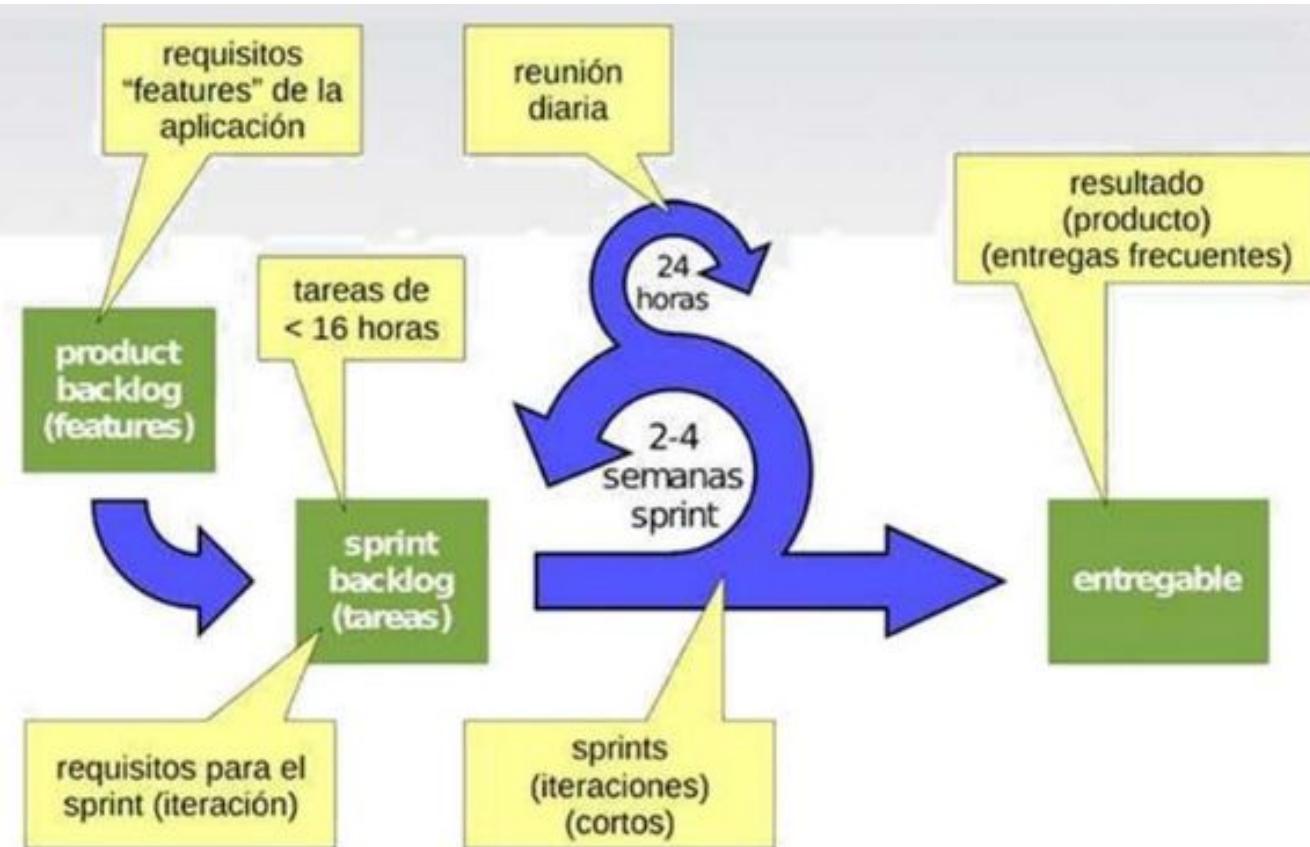
Ejecución de la iteración (Sync meeting)

La sync meeting siempre ocurre el **último día de la iteración**, y tiene dos partes:

Demostración (4 horas máximo). El equipo presenta al cliente los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo. En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, el cliente realiza las adaptaciones necesarias de manera objetiva, ya desde la primera iteración, replanificando el proyecto.

Retrospectiva (4 horas máximo). El equipo analiza cómo ha sido su manera de trabajar y cuáles son los problemas que podrían impedirle progresar adecuadamente, mejorando de manera continua su productividad. El Facilitador se encargará de ir eliminando los obstáculos identificados.

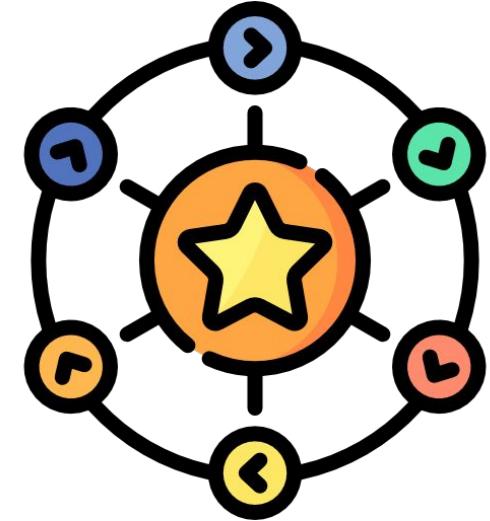




Ventajas de scrum

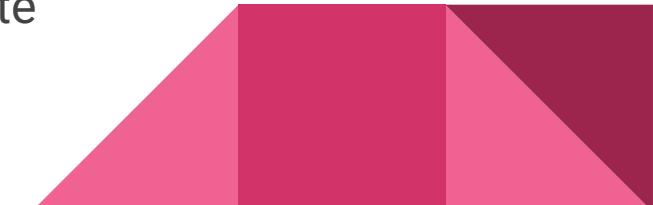
Ventajas:

- El cliente puede comenzar a utilizar el producto rápidamente.
- El cliente puede decidir los nuevos objetivos a realizar.
- Se agiliza el proceso, porque se divide el problema en pequeñas tareas.
- Menos probabilidad de que se den sorpresas o desarrollos inesperados porque el cliente va viendo poco a poco lo que se está desarrollando.



El cliente puede comenzar a utilizar el producto rápidamente

La ventaja de que el cliente pueda comenzar a utilizar el producto rápidamente en el modelo Scrum se debe a la entrega incremental y frecuente de funcionalidades durante cada sprint. En lugar de esperar hasta que todo el producto esté completo, el equipo de desarrollo trabaja en iteraciones cortas para entregar partes funcionales del producto en intervalos regulares. Aunque el producto aún no esté completamente desarrollado, el cliente puede recibir y comenzar a utilizar las funcionalidades entregadas al final de cada sprint. Este enfoque de entrega incremental permite que el cliente experimente y evalúe el producto en etapas tempranas del desarrollo.



El cliente puede decidir los nuevos objetivos a realizar

Durante las reuniones de Scrum planning, el cliente tiene la oportunidad de priorizar los elementos del backlog del producto y definir los objetivos específicos que le gustaría que el equipo de desarrollo logre en el próximo sprint. Esta participación activa del cliente en la definición de los objetivos permite una mayor flexibilidad en el proceso de desarrollo. A medida que se avanza en el proyecto y se obtiene una mejor comprensión de los requisitos y necesidades del cliente, es posible que surjan nuevos objetivos o cambios en los existentes. En lugar de seguir un plan rígido y predefinido, el modelo Scrum permite que los objetivos sean reevaluados y ajustados en cada sprint.

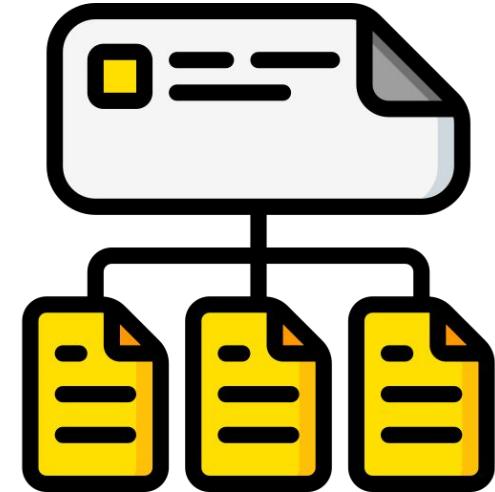
A su vez, al permitir que el cliente defina los nuevos objetivos, se fomenta una mayor satisfacción del cliente y se maximiza el valor entregado por el producto final.



Se agiliza el proceso, porque se divide el problema en pequeñas tareas.

Recordemos que el equipo de desarrollo divide el problema o el proyecto en pequeñas tareas o funcionalidades que pueden ser abordadas de manera individual. Estas tareas se registran en el backlog del producto y se priorizan según su importancia. Durante cada sprint, el equipo selecciona un conjunto de tareas del backlog para trabajar en ellas.

Al dividir el problema en tareas más pequeñas, se logran varios beneficios. En primer lugar, las tareas individuales son más fáciles de comprender y abordar por parte del equipo de desarrollo. Esto permite una mayor eficiencia y reduce el riesgo de errores o malentendidos. Cada tarea se puede asignar a un miembro del equipo, lo que permite una distribución equitativa de la carga de trabajo y una mayor especialización en la resolución de problemas específicos.



Menos probabilidad de que se den sorpresas o desarrollos inesperados porque el cliente va viendo poco a poco lo que se está desarrollando.

Al mostrar al cliente el producto en desarrollo de manera incremental, se minimiza la posibilidad de sorpresas o desarrollos inesperados en etapas posteriores del proyecto. El cliente tiene la oportunidad de revisar y proporcionar retroalimentación sobre el producto en cada entrega y asegurarse de que se esté alineando con sus expectativas y requisitos. También existe una visibilidad y transparencia por parte del cliente, a la hora de llevar a cabo un modelo scrum. Esto permite que el cliente tenga una comprensión clara y actualizada del estado del proyecto en todo momento. Esto reduce la incertidumbre y la probabilidad de que se den sorpresas desagradables más adelante, ya que el cliente tiene la oportunidad de corregir el rumbo o realizar ajustes a medida que avanza el desarrollo.

Desventajas de scrum

Desventajas:

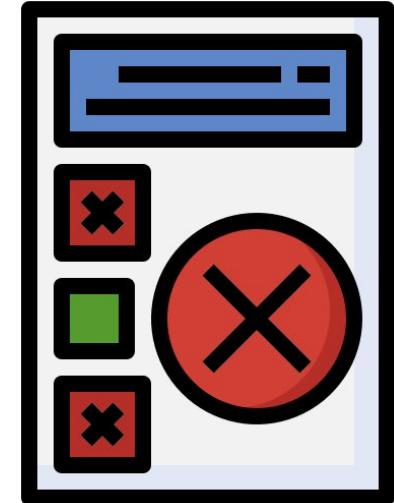
- Existe la tendencia de que se deje una tarea sin terminar y que por las exigencias del Product Owner se deban realizar otras nuevas. Estas tareas no terminadas pueden obstaculizar la planeación de nuevos Sprint y se deba volver al problema original.
- Alto nivel de estrés de los miembros del equipo. El desgaste puede ser excesivo y estresante, lo que puede disminuir el rendimiento.
- La necesidad de contar con equipos multidisciplinarios puede ser un problema, porque cada integrante del equipo debe estar en capacidad de resolver cualquier tarea y no siempre se cuenta con este perfil en la empresa.
- El equipo puede estar tentado de tomar el camino más corto para cumplir con un sprint, que no necesariamente puede ser el de mejor calidad en el desarrollo del producto.



Tareas sin terminar

Existe la tendencia de que se deje una tarea sin terminar y que por las exigencias del Product Owner se deban realizar otras nuevas. Estas tareas no terminadas pueden obstaculizar la planeación de nuevos Sprint y se deba volver al problema original.

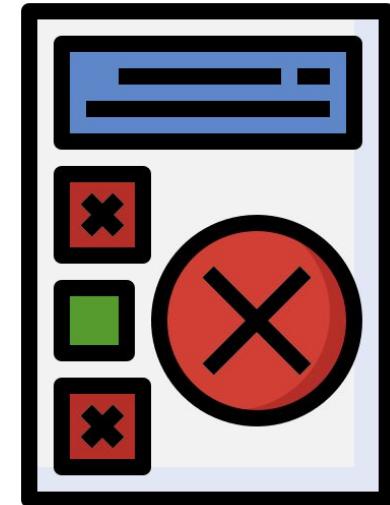
Durante el proceso de desarrollo, es posible que el Product Owner solicite cambios o agregue nuevas tareas que deben abordarse de manera prioritaria. Esta situación puede generar un desafío para el equipo de desarrollo, ya que pueden encontrarse en la posición de tener que dejar una tarea incompleta para abordar las nuevas exigencias. Si esta tendencia se repite en varios sprints, puede haber un acumulamiento de tareas incompletas y un retraso en la finalización del trabajo original.



Tareas sin terminar

Cuando se dejan tareas sin terminar, esto puede tener un impacto negativo en la planificación de los nuevos sprints. El equipo de desarrollo puede enfrentar dificultades para estimar de manera precisa y planificar las tareas futuras si hay trabajo pendiente y no se ha completado. Esto puede llevar a una falta de visibilidad y una planificación deficiente, ya que es necesario considerar tanto las tareas pendientes como las nuevas tareas solicitadas.

Además, si no se abordan adecuadamente las tareas incompletas, puede haber una acumulación de trabajo no terminado que afecte la calidad del producto y la satisfacción del cliente. Las tareas no completadas pueden generar dependencias y afectar la funcionalidad general del producto, lo que requiere que el equipo deba volver al problema original y dedicar tiempo adicional para completar las tareas pendientes.



¿Cómo abordar la problemática de las tareas inconclusas?

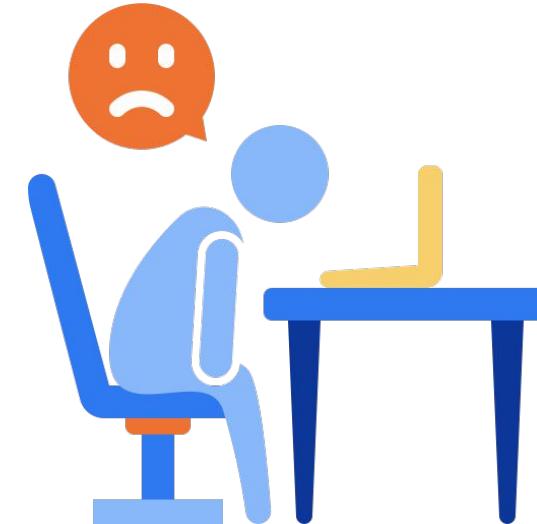
Para abordar esta desventaja, es importante tener una comunicación clara y continua entre el equipo de desarrollo y el Product Owner. Es fundamental establecer expectativas realistas y priorizar adecuadamente las tareas para evitar una sobrecarga de trabajo y una acumulación excesiva de tareas no terminadas. Además, es importante realizar una gestión eficaz del backlog y asegurarse de que las tareas sean adecuadamente refinadas y estimadas antes de comprometerse a trabajar en ellas.



Alto nivel de estrés

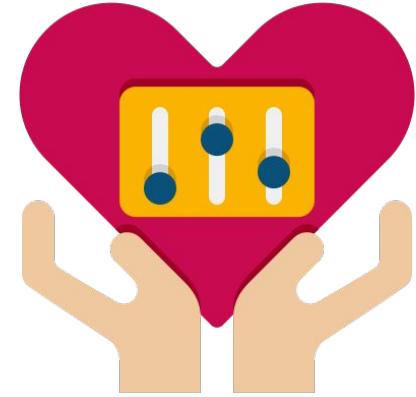
Alto nivel de estrés de los miembros del equipo. El desgaste puede ser excesivo y estresante, lo que puede disminuir el rendimiento. Esto se debe al ritmo de trabajo acelerado y las exigencias constantes del proceso. Esto puede generar un nivel de estrés elevado en los miembros del equipo. Pueden surgir situaciones en las que los plazos sean ajustados, los recursos sean limitados o los obstáculos sean difíciles de superar. Esto puede llevar a una sensación de presión constante y una carga de trabajo abrumadora.

El desgaste excesivo y el estrés prolongado pueden tener un impacto negativo en el rendimiento y la salud de los miembros del equipo. El estrés crónico puede llevar a la disminución de la productividad, la falta de concentración, la toma de decisiones deficientes y la disminución de la calidad del trabajo. Además, el alto nivel de estrés puede afectar negativamente el bienestar emocional y físico de los miembros del equipo, lo que puede llevar a un agotamiento o burnout.



Cómo evitar el estrés en los miembros del equipo

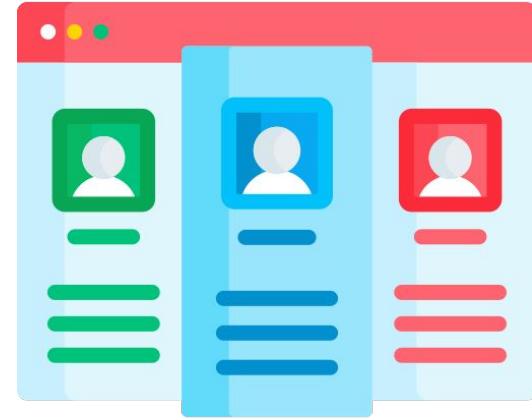
Es importante reconocer y abordar este desafío para mantener un ambiente de trabajo saludable y productivo. Los equipos Scrum pueden implementar prácticas como la gestión efectiva de la carga de trabajo, la planificación realista, el establecimiento de límites y la promoción de un equilibrio adecuado entre el trabajo y la vida personal. Además, la comunicación abierta y el apoyo mutuo dentro del equipo son fundamentales para identificar y abordar los problemas de estrés y encontrar soluciones adecuadas.



Equipos multidisciplinarios

En el modelo Scrum, se espera que los equipos de desarrollo sean multidisciplinarios, es decir, que cuenten con miembros que posean habilidades y conocimientos diversos. Esto se debe a que las tareas o funcionalidades en el backlog del producto pueden requerir diferentes áreas de experiencia, como diseño, desarrollo de software, pruebas, implementación, entre otros.

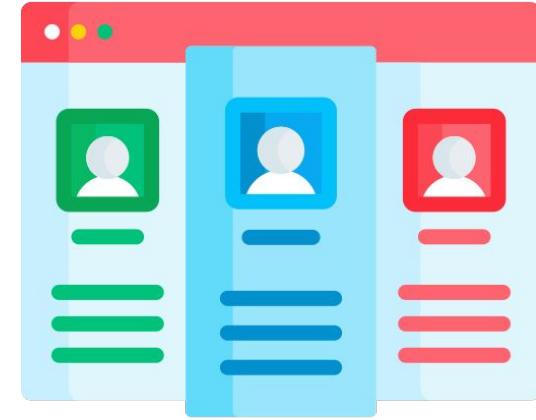
Sin embargo, puede haber situaciones en las que la empresa no cuente con miembros del equipo que posean todas las habilidades necesarias para abordar todas las tareas del proyecto. Esto puede deberse a limitaciones de recursos, falta de personal especializado o falta de diversidad en las habilidades dentro de la organización.



¿Cómo hacer que esto no sea una desventaja?

La falta de un equipo multidisciplinario puede generar desafíos y obstáculos en el desarrollo del proyecto. Si un miembro del equipo no tiene las habilidades necesarias para abordar una tarea específica, puede haber retrasos en su implementación o la calidad del trabajo puede verse comprometida. Además, puede generar una dependencia excesiva en ciertos miembros del equipo que poseen las habilidades requeridas, lo que puede generar desequilibrios y sobrecargas de trabajo.

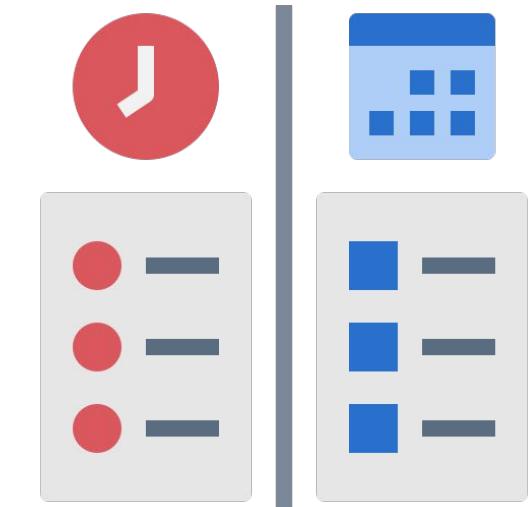
Para abordar esta desventaja, es importante evaluar y planificar adecuadamente los recursos necesarios para el proyecto. Esto puede implicar la contratación de personal adicional con habilidades específicas o la capacitación y desarrollo de los miembros del equipo existentes para adquirir las habilidades requeridas. También es posible buscar apoyo externo, como consultores o especialistas, para cubrir las áreas de experiencia faltantes.



Tomar el camino más corto

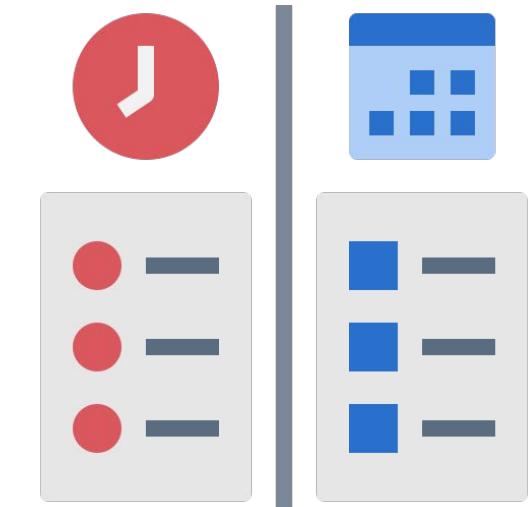
Recordemos el tema del estrés y la presión por llegar al plazo previsto. Teniendo esto en cuenta, esta presión puede llevar al equipo a tomar atajos o tomar el camino más corto para cumplir con las entregas en el tiempo asignado. Esto puede significar omitir ciertos pasos de desarrollo, como pruebas exhaustivas, revisiones de código o refinamiento del diseño. El objetivo principal se convierte en completar la tarea dentro del sprint en lugar de garantizar la calidad óptima del producto.

El problema con tomar el camino más corto es que puede afectar negativamente la calidad del producto y la satisfacción del cliente a largo plazo. Al omitir pasos importantes o realizar un desarrollo apresurado, pueden surgir problemas técnicos, errores o deficiencias en la funcionalidad. Estos problemas pueden requerir más tiempo y esfuerzo para solucionar en futuros sprints, lo que puede ralentizar el progreso general del proyecto.



¿Qué se debe hacer en estos casos?

Para abordar esta desventaja, es importante fomentar un enfoque equilibrado entre la entrega dentro del sprint y la calidad del producto. El equipo de desarrollo debe ser consciente de la importancia de cumplir con los plazos establecidos, pero también de la necesidad de garantizar la calidad en cada entrega. Se deben establecer estándares de calidad claros y promover prácticas de desarrollo sólidas, como pruebas exhaustivas, revisiones de código y refinamiento del diseño.



¿Para qué proyectos es adecuado usar scrum?

Scrum se utiliza para resolver situaciones en que **no se está entregando al cliente lo que necesita**, cuando **las entregas se alargan demasiado, los costes se disparan o la calidad no es aceptable**, cuando se necesita capacidad de reacción ante la competencia, cuando **la moral de los equipos es baja y la rotación alta**, cuando **es necesario identificar y solucionar ineficiencias sistemáticamente** o cuando **se quiere trabajar utilizando un proceso especializado en el desarrollo de producto**.

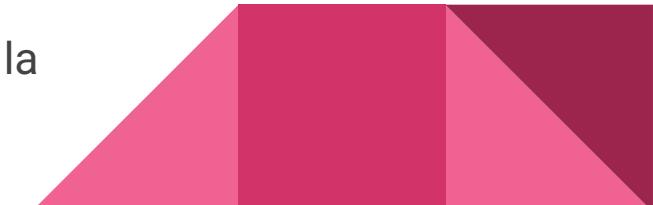
Vamos a analizar cada una de las frases que están resaltadas:

“No se está entregando al cliente lo que necesita”

Scrum se utiliza cuando existe una brecha entre lo que se está entregando al cliente y sus necesidades reales. Al adoptar Scrum, se enfatiza la colaboración con el cliente a través de la participación regular y activa en el proceso de desarrollo. Esto permite una retroalimentación constante y una mayor alineación entre las necesidades del cliente y el producto final.

“Las entregas se alargan demasiado”

Podemos también abordar el tema de las entregas prolongadas al trabajar en sprints de tiempo fijo. Estos sprints tienen una duración predeterminada, generalmente de una a cuatro semanas. Al dividir el trabajo en incrementos manejables y establecer plazos definidos, Scrum fomenta la entrega regular de funcionalidades y productos, evitando así la prolongación innecesaria de los plazos de entrega. Obvio que también causa la desventaja que fue mencionada previamente.

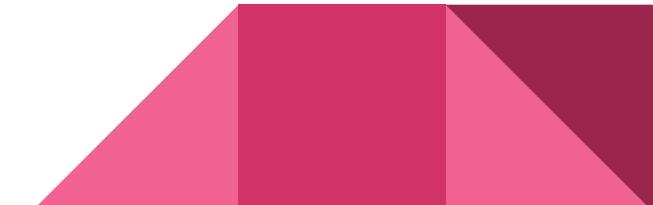


"Los costes se disparan"

Scrum busca controlar los costos mediante la planificación y priorización efectiva del trabajo. Al tener ciclos de desarrollo más cortos y entregar incrementos de valor con mayor frecuencia, se pueden identificar y abordar rápidamente los riesgos y desviaciones que pueden llevar a un aumento de los costos. Además, el enfoque en la colaboración y la transparencia dentro del equipo de desarrollo y con el cliente ayuda a optimizar la eficiencia y evitar gastos innecesarios.

"La calidad no es aceptable"

Se promueve la calidad a través de la participación activa del cliente y la retroalimentación continua. Al entregar incrementos de valor en cada sprint, se fomenta la detección temprana de problemas de calidad y se pueden realizar correcciones rápidas. Además, la planificación y ejecución de pruebas periódicas aseguran que los estándares de calidad se cumplan en cada entrega.

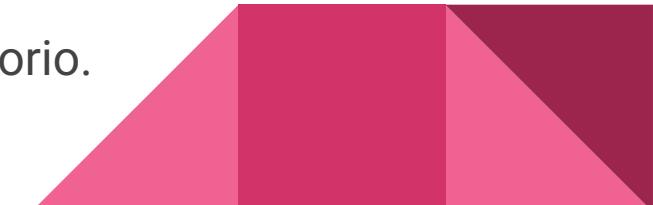


"Se necesita capacidad de reacción ante la competencia"

Podemos obtener una mayor capacidad de respuesta al mercado y a la competencia gracias a scrum. Al trabajar en sprints y realizar entregas regulares, se pueden realizar ajustes y mejoras en función de las necesidades cambiantes del mercado. Esto permite adaptarse rápidamente a la competencia y mantener la relevancia del producto.

"La moral de los equipos es baja y la rotación alta"

Scrum busca mejorar la moral y la retención del equipo al fomentar la autogestión y la colaboración, de una manera muy similar a cómo lo hacía XP (clase pasada). Al proporcionar un marco de trabajo que valora la participación activa, la toma de decisiones conjunta y el empoderamiento del equipo, se promueve un entorno de trabajo más motivador y satisfactorio.



"Es necesario identificar y solucionar ineficiencias sistemáticamente"

Obviamente, al ser un modelo incremental, por naturaleza se fomenta la mejora continua a través de la revisión regular del proceso y la identificación de ineficiencias. A su vez, mediante la celebración de reuniones de retrospectiva al final de cada sprint, se pueden analizar y abordar las áreas de mejora, lo que permite un desarrollo más eficiente y efectivo.

"Se quiere trabajar utilizando un proceso especializado en el desarrollo de producto"

Al final del día, scrum es un marco de trabajo ágil especializado en el desarrollo de productos complejos. Al utilizar Scrum, se aprovechan las mejores prácticas y principios específicos para gestionar eficientemente el desarrollo de productos, lo que garantiza una mayor eficacia y éxito en el resultado final, al igual que ocurre en los modelos ágiles que vimos durante la cursada.

DevOps I - Introducción y aplicación en tecnología de la información

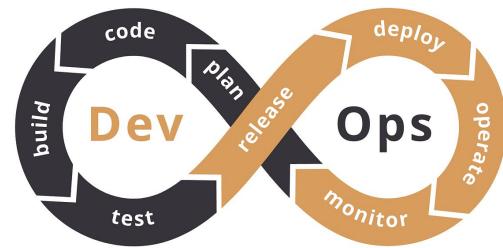




¿Qué es DevOps?

DevOps es una metodología de desarrollo que integra y automatiza el trabajo del **desarrollo de software** (dev) y **operaciones de tecnología de la información** (ops), con el objetivo de mejorar y acortar el ciclo de vida de un sistema.

© DISSTITUTE





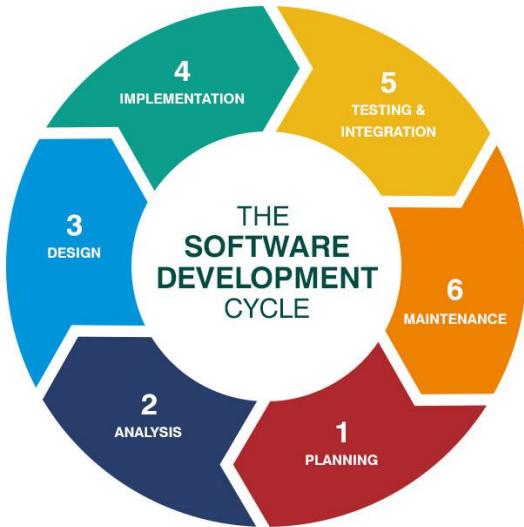
Origen

Ya en los 80s y 90s, existían propuestas para combinar metodologías de desarrollo de software con conceptos de operaciones.

Entre 2007 y 2008, surgieron preocupaciones dentro de las comunidades de desarrollo de software y TI de que la separación entre las dos industrias, donde una escribía y creaba software completamente separada de aquellos que lo implementan y apoyan, estaba generando un nivel fatal de **disfunción** dentro de la industria.

En la conferencia Agile 2008 Toronto, Yhens Wasna y Patrick Debois introdujeron el término en su charla sobre "Infraestructura Ágil". A partir de 2009, el término DevOps se ha promocionado constantemente y se ha incorporado a un uso más general a través de una serie de "devopsdays", que comenzaron en Bélgica y ahora también se han extendido a otros países.

Desarrollo de software



Recordemos que desarrollo de software incluye no solamente **programar** y realizar **mantenimiento** sobre un código dado, como paso central para el proceso, sino que también incluye el **análisis** de requerimientos del sistema, **diseño**, **testeo**, y **lanzamiento**.

Tecnología de la información

Por otra parte, las operaciones de la tecnología de la información tienen más relación con **otras** actividades que son esenciales para el funcionamiento y mantenimiento de sistemas de software en producción, incluyendo:

- **Gestión de infraestructura**, como configuración y mantenimiento de servidores, redes y otros componentes de hardware.
- **Monitoreo y rendimiento**, fijarse bien de que la disponibilidad, rendimiento, y salud general de los sistemas esté asegurado a través de una supervisión constante.
- **Seguridad**, que incluye la protección contra amenazas y vulnerabilidades externas o internas...



Tecnología de la información

- **Despliegue**, es decir la entrega de software (con DevOps se busca automatizarla).
- **Backups**, muchas estrategias para garantizar que los datos se respalden y puedan recuperarse en caso de fallos.
- **Escalabilidad**, muchas veces los sistemas no pueden manejar un aumento en la carga y el tráfico.
- **Configuración y gestión de versiones**, muchos controles relacionadas a las configuraciones y versiones de software para que los entornos de desarrollo y producción mantengan la coherencia; muchas veces quedan desalineados.



Hay muchísimas herramientas de DevOps para cada uno de estos ítems, y los vamos a ir viendo en esta presentación:



Herramientas para gestión de infraestructura

Terraform: definir, aprovisionar y gestionar infraestructuras de manera eficiente utilizando un lenguaje declarativo llamado **HashiCorp Configuration Language (HCL)**, o incluso JSON.



- Se define todo el código en archivos de configuración
- Se describe directamente el estado final que queremos tener en nuestra infraestructura, y Terraform va a automáticamente intentar llegar a ese estado (*declarativo*)
- Compatible con los mejores proveedores de nube y servicios.
- Permite planificación de cambios, gestión del ciclo de vida de la infraestructura y estado de la infraestructura.
- Crea módulos reutilizables, fomentando la **modularización** y **reutilización** del código.
- Gestiona dependencias de forma automática.

¿Cómo funciona Terraform internamente?

Funciona con JSON o un lenguaje **declarativo**, llamado **HashiCorp Configuration Language (HCL)**.

Ejemplo concreto - instancia de una máquina virtual en la nube de Amazon (**EC2 en AWS**):

```
provider "aws" {  
    region = "sa-1"  
}
```

#Acá definimos el proveedor de Amazon Web Service (AWS), y qué región nos conviene a nosotros.

```
resource "aws_instance" "mi_instancia_con_terraform" {
```

ami = "ami-0c55b159cbfafe1f0" #Imagen del Sist. Operativo para la máquina virtual. En este ejemplo, usamos Linux.

instance_type = "t2.micro" #La instancia que vamos a usar, esta es gratis. Le podemos poner un nombre que queramos nosotros a la instancia con la siguiente línea de código:

```
    tags = {  
        Name = "InstanciaTerraformADS2024"  
    }  
}
```



¿Cómo funciona Terraform internamente?

Después tenemos comandos básicos para lo que vayamos necesitando. Por ejemplo:

terraform init #Iniciar un proyecto

terraform plan #Planificar los cambios

terraform apply #Aplicar los cambios

terraform destroy #Destruir recursos innecesarios

Para hacer funcionar el ejemplo del proyecto anterior, nos faltarían los siguientes pasos:



export AWS_ACCESS_KEY_ID = "clave_de_acceso"

export AWS_SECRET_ACCESS_KEY = "clave_secreta" #Estas claves las encontramos en el archivo de configuración de AWS con la clave única para nuestro proyecto.

terraform init #Inicializamos terraform

terraform plan #Esto nos va a mostrar un plan de acción en base a lo existente.

terraform apply #Si nos gustó el plan que hizo terraform, creamos la instancia.

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Listo, tenemos nuestra infraestructura hecha :)

Herramientas para monitoreo y rendimiento

Hay bastante software de monitoreo, ninguno que sea concretamente superior. Algunos ejemplos son:

Grafana: es de código abierto, podemos consultar datos en bases de datos. Se conecta a bases de datos como Oracle, SQL, o la que usemos, recibe los datos, y los transforma en gráficos, tableros e informes. Tiene sistemas de alertas, por si se llega a algún número umbral que no queramos de cualquier información que sea, aparte de muchos plugins e integraciones.



Grafana

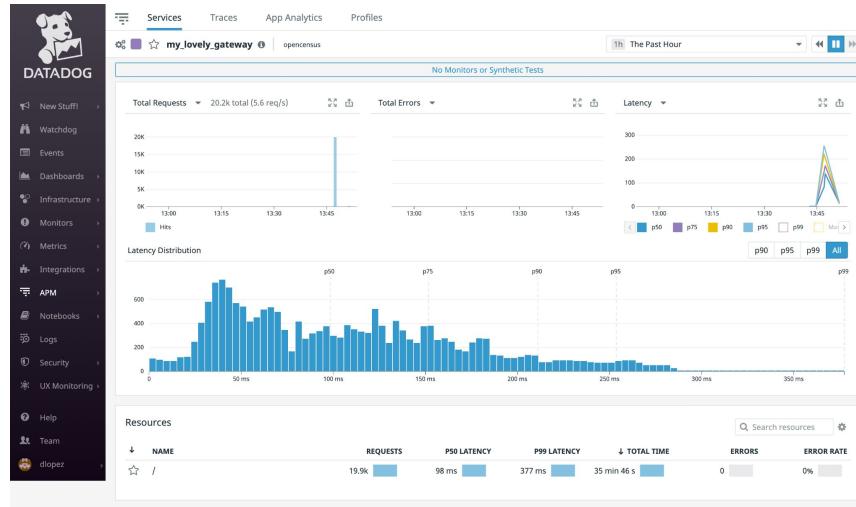


Herramientas para monitoreo y rendimiento

Datadog: muy similar a Grafana, pero mayormente ofrece visibilidad de infraestructura, aplicaciones y logs. Monitorea todo el **stack** tecnológico en servidores, bases de datos, servicios en la nube (como el que creamos con Terraform!!), entre otros. También tiene alertas como Grafana, y se integra con muchas otras herramientas DevOps que están en esta presentación.



DATADOG





Herramientas para seguridad

En términos de seguridad, hay dos vertientes: una que se refiere al **scanning de vulnerabilidades**, es decir, problemas que pueda haber en nuestro código y las dependencias, y otra que se refiere a la **autenticación y autorización seguras**, para que no entren hackers a nuestro sistema, etc..

Algunas herramientas pueden ser...

Herramientas para escaneo de vulnerabilidades

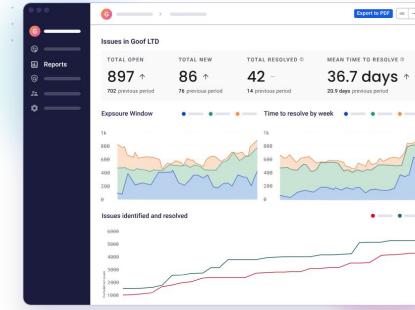


Snyk: Identifica, y corrige vulnerabilidades en dependencias de código, contenedores y configuraciones de infraestructura como código.

- Primero, Snyk se fija que los archivos de dependencias de los proyectos (package.json), (node-modules), etc.. no tengan vulnerabilidades .
- Después, escanea archivos de configuración (tipo **terraform**) para ver que no haya puertos abiertos, permisos muy amplios, o cualquier vulnerabilidad de ese estilo.
- Una vez que las detecta, Snyk sugiere correcciones automáticas o parches para problemas en las dependencias.



snyk



Herramientas para escaneo de vulnerabilidades

SonarQube: analiza código estático, mejorando así la calidad del código y detectando vulnerabilidades de seguridad en el código fuente. Se encarga especialmente de mantener algún patrón de diseño que tenga el código, y de que no se duplique para que esté bien modularizado.

- Detecta bugs y se revisa el código sin ejecutarlo, así que antes de que el software esté en ambiente de explotación, ya podemos ver algún problema que haya.
 - Internamente SonarQube tiene miles de reglas de calidad que hay en los distintos lenguajes de programación (como Java, Python, JavaScript, C#, PHP, etc.), y evalúan la estructura y calidad del código.
 - **No detecta tantas vulnerabilidades como Snyk!!**
 - Genera informes sobre la calidad del código y las vulnerabilidades detectadas.



Helicopter View

ALL PROJECTS

Calificación SQALE

A

Ratio De Deuda Técnica

5.2%

Deuda Técnica Líneas De Código
 39,904d ↘ 12,515K ↗

ALL PROJECTS

Debt

39,904d ↘

Evidencias

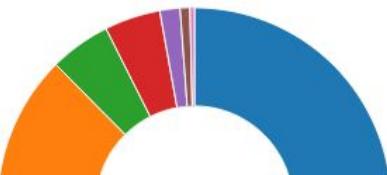
1,825,913 ↗

- ! Bloqueante 18,646 ↗
- + Crítica 56,384 ↗
- Mayor 818,158 ↘
- ✓ Menor 846,298 ↗
- ▼ Info 86,427 ↗

Global Security Issue Tags

error-handling	52379	multi-threading	5856	sans-top25	1049
owasp-a6	939	owasp-a2	807	owasp-a1	802
sans-top25-risky	745	owasp-top10	314	denial-of-service	214
injection	14				

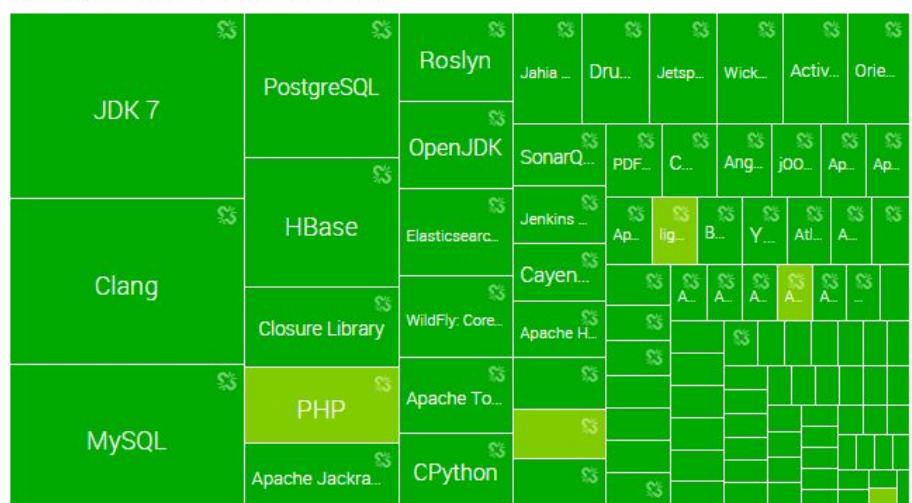
FORGES



- Apache
- Others
- Sourceforge
- Codehaus
- OW2
- OPS4J

ALL PROJECTS

Size: Líneas de código Color: Calificación SQALE

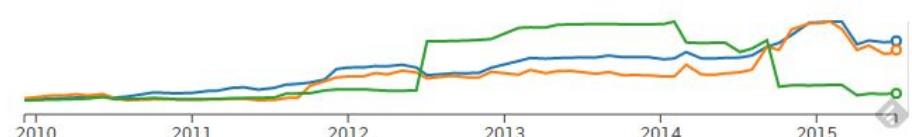


Solo se muestran los primeros 100 componentes

ALL PROJECTS

6 de julio del 2015 ● Líneas de código: 12,515,044 ● Líneas duplicadas: 2,365,719

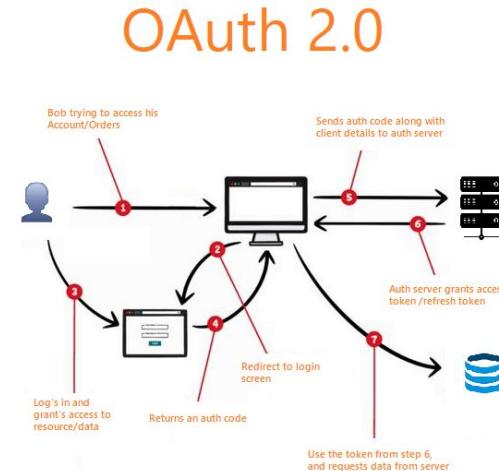
● Tests unitarios: 118,967



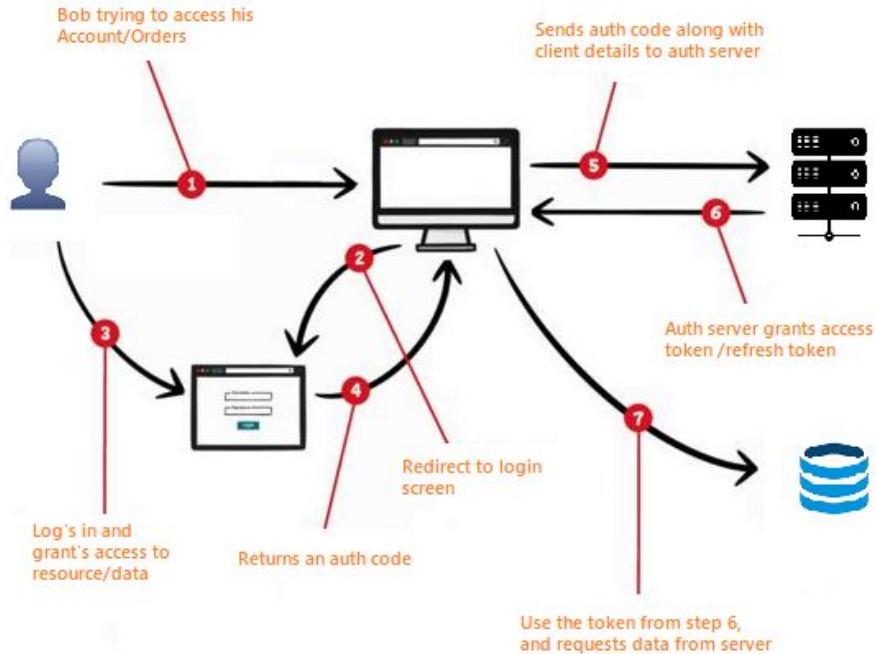
Herramientas para autenticación y autorización seguras

OAuth: protocolo de autorización que permite a un cliente acceder a recursos de un usuario en otro servicio sin compartir las credenciales. Por ejemplo, se usa para cuando uno apreta el botón de "Iniciar sesión con Google" en una aplicación, en ese caso, en lugar de ingresar tus credenciales en la app, OAuth permite que la app acceda a tu perfil en Google de manera segura usando tokens de acceso.

OAuth utiliza internamente algo llamado **OpenID Connect (OIDC)**, una capa de identidad construida sobre OAuth 2.0 que agrega autenticación a las capacidades de autorización de OAuth, lo que nos permite OIDC es verificar la identidad del usuario.



OAuth 2.0



Herramientas para autenticación y autorización seguras

HashiCorp Vault: Permite gestionar secretos y proteger el acceso a datos sensibles en aplicaciones, como contraseñas, tokens de API, entre otros.

- Los 'secretos' se almacenan de manera cifrada.
- Tiene políticas de acceso para controlar quién puede acceder a qué secreto, definiendo qué usuario o aplicación tiene permiso para leer, escribir o generar secretos.

Ejemplo: Una aplicación que necesita acceder a una base de datos obtiene sus credenciales de forma dinámica usando esta herramienta, en lugar de almacenarlas en un archivo de configuración existente en la memoria del usuario o de los servidores (muy vulnerable), mejorando así la seguridad de las credenciales.



Herramientas para despliegue



Jenkins: Automatiza código abierto, permitiendo principalmente integración continua (CI) y despliegue continuo (CD).



Jenkins

- **Integración continua:** Cualquier desarrollador que esté metido en el mismo proyecto Jenkins puede integrar su cambio de código de manera frecuente en un repositorio central, ejecutando automáticamente pruebas unitarias y de integración, asegurando que el código no se rompa.
- **Despliegue continuo:** Despues de que el código previamente integrado pasa las pruebas, se puede automatizar su entrega a entornos de desarrollo, pruebas o producción.

Ejemplo: Cuando un desarrollador hace un commit de código en Git, Jenkins automáticamente detecta el cambio, ejecuta pruebas automáticas, compila el código, y si anduvo todo bien, lo despliega en un servidor de prueba, notificando al equipo si hay problemas.

Herramientas para despliegue

GitLab: extremadamente similar a Jenkins en muchos aspectos. Permite integración continua y entrega continua de la misma manera que Jenkins, pero fomentando más aún la colaboración en proyectos porque ofrece un repositorio de código basado en Git y un entorno unificado para los desarrolladores. Dentro del repositorio, se pueden crear muy fácilmente ramas, hacer pull requests, realizar revisiones de código, y ver versiones viejas y tomar partes de versiones viejas (cherry pick) del proyecto.



Herramientas para despliegue



AWS CodePipeline: Amazon Web Service Code Pipeline permite compilar, probar y desplegar aplicaciones de manera rápida.

- Define flujos de trabajo o **pipeline**: Define el flujo de trabajo completo y despliegue de una aplicación.
- Automatiza el flujo de desarrollo, detectando un cambio y ejecutando el pipeline definido para el proyecto.
- Tiene excelente control de versiones, facilitando hacer rollback (volver a una versión vieja), si hubo algún tipo de problema en producción.



AWS CodePipeline

Herramientas para backups

Las distintas herramientas utilizadas en DevOps también utilizan automatización de backups.

Veeam: Realiza copias de seguridad a nivel de imagen, lo que significa que puede capturar el estado completo de una máquina virtual (VM), servidor físico o una instancia en la nube, permitiendo restaurar todo el sistema operativo, aplicaciones y datos si es necesario. También tiene la capacidad de restaurar archivos individuales, algún correo electrónico e incluso registros dentro de bases de datos.

Además de los backups, Veeam puede replicar máquinas virtuales o servidores a un sitio alternativo (DR - **disaster recovery**). Esto permite que en caso de un fallo o desastre en el centro de datos principal, se pueda hacer un **failover** a la réplica.



Herramientas para backups

AWS Backup: Amazon Web Service backup permite, al igual que Veeam, la automatización de copias de seguridad y recuperación de recursos almacenados en la nube.

- Tiene centralización de backups - realiza copias de seguridad múltiples de otros servicios AWS.
- Distintas políticas de backup para respaldar los recursos.
- Copias cruzadas entre distintas regiones de AWS.
- **Automatización;** se configuran las políticas para que se ejecuten en intervalos específicos de tiempo.

(Entre nosotros... Veeam es mucho más completo y mejor)



Herramientas para escalabilidad

Microsoft Azure: Azure tiene un montón de soluciones para almacenamiento, cómputo, redes, entre otros. En términos de escalabilidad, Azure tiene muchas herramientas:

- **Virtual Machines:** Se pueden escalar máquinas virtuales bajo demanda, permitiendo cambiar el tamaño de las VMs o aumentar el número de instancias.
- Azure contiene una plataforma para desplegar aplicaciones web y APIs (PaaS - Platform as a Service). Esta plataforma, llamada **Azure App Service** tiene escalabilidad automática basada en el tráfico; si hay mucho uso entonces se agregan instancias de aplicaciones de forma automática.
- Con respecto a CPU, memoria o cantidad de solicitudes, Azure también permite configurar reglas de **escalado automático**.
- El Azure **Load Balancer** nos deja distribuir el tráfico entre varios recursos para que la performance sea la mejor posible. Estos load balancer son particularmente importantes en Azure porque permiten gestionar grandísimos volúmenes de tráfico eficientemente, sin perder performance.

Herramientas para configuración y gestión de versiones



Git: Es posiblemente una de las herramientas más utilizadas, sino la más utilizada en contextos de DevOps, y es elemental entenderla al ingresar a cualquier tipo de proyecto que aplique metodologías ágiles. El funcionamiento de Git es simple y bastante directo:

- **Repositorio:** Lugar local o en una nube (GitHub, GitLab) donde se almacenan los archivos y su historial de cambios.
- **Commit:** Registro de los cambios realizados en el código. Lo puede realizar cualquier desarrollador (esto no significa que vaya a aprobarse), describiendo los cambios realizados.
- **Branching:** Ramas para trabajar en diferentes características o correcciones de manera independiente. Permite desarrollar nuevas funcionalidades sin afectar el código principal (muy importante para trabajar con metodologías como SCRUM o XP que tienen **historias de usuario**).



Herramientas para configuración y gestión de versiones

- **Merge:** Si los cambios en una rama son completados y probados, y funcionan correctamente, se aprueba un **commit** para ser combinados (mergeados) en otra rama superior, como la principal.
- **Clonación y pushing:** Es posible clonar un repositorio para hacer cambios locales en el mismo. Si realizamos cambios y los aprobamos internamente, hacemos *push* para enviarlos de vuelta al repositorio remoto.

Ejemplos de comandos básicos en Git:

  **git clone <url_repository>** #Esto permite clonar un repositorio a nuestra computadora

git checkout -b rama_nueva_funcionalidad #Acá creamos una nueva rama que se llama 'rama_nueva_funcionalidad'

git add . #Hace los cambios internamente.

git commit -m "Descripción del commit" #Realiza un commit, confirmando los cambios agregados previamente, con un mensaje que describa qué cambios se realizaron

Herramientas para configuración y gestión de versiones

`git push origin rama_nueva_funcionalidad` #Esto sube los cambios del commit al repositorio remoto, agregándolos a nuestra nueva rama.

`git merge rama_nueva_funcionalidad` #Finalmente, unimos los cambios a la rama principal en el caso en el que el código realizado en la nueva rama haya sido aprobado para que podamos mandarlo al sistema completo.



Herramientas para configuración y gestión de versiones



Ansible: Es una herramienta para **automatización de configuración** y gestión de infraestructuras. Automatiza aprovisionamiento de servidores, instalación de software, entre otras.

Ansible tiene algo llamado **playbooks**, que contienen “*plays*” que describen tareas a realizar; qué hacer y dónde hacerlo. Para saber dónde hacerlo, se utilizan los **inventarios**, especificando en qué servidores deben ejecutarse las tareas.

Aparte, Ansible tiene **módulos**, pequeñas unidades (a veces atómicas) de código que realizan tareas específicas, y **roles**, que permiten estructurar las tareas en módulos reutilizables.

Ansible utiliza archivos de tipo **YAML** (es una onda JSON).

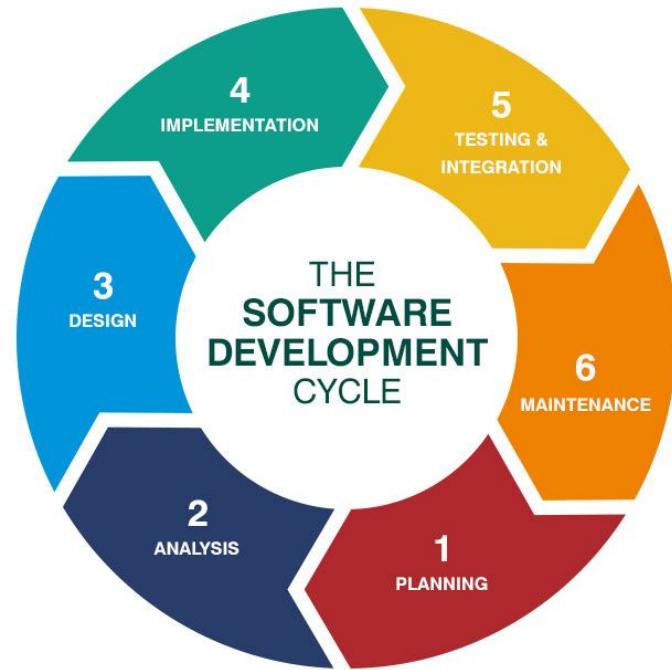


ANSIBLE

DevOps II - Aplicación en desarrollo de software

ADS

DevOps en Desarrollo de software



Con respecto al desarrollo de software, también se aplican muchas herramientas DevOps. Vamos a ir separando los distintos pasos de acuerdo al círculo que está en esta diapositiva:

- Planeamiento general del proyecto
- Análisis de requisitos
- Diseño
- Implementación
- Testeo e integración
- Mantenimiento

Obviamente, se van a repetir muchas herramientas que ya vimos con respecto a las prácticas en tecnología de la información.



Herramientas para planeamiento general del proyecto

Acá entran en juego todas las metodologías de desarrollo; SCRUM, XP, modelo evolutivo, por prototipos, etc..

El planeamiento desde la génesis del proyecto, donde se detallan los pasos a seguir para intentar conseguir que el mismo supla a la empresa con un sistema de calidad, es el que contiene la metodología de desarrollo subyacente.

Más allá de la metodología de desarrollo, hay herramientas para **organización y automatización del seguimiento de tareas** que entran en esta categoría de planeamiento general del proyecto.

Herramientas para planeamiento general del proyecto

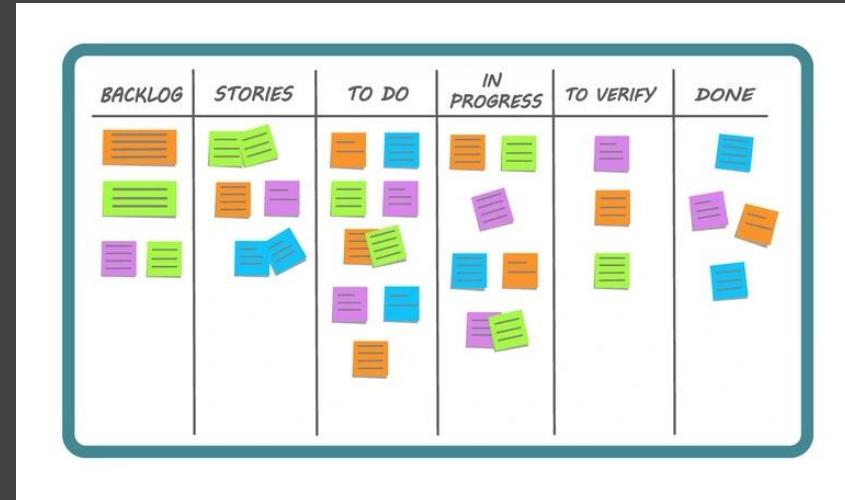
Kanban: si bien Kanban se puede entender como metodología ágil en general, en realidad vamos a ver que en el contexto de DevOps, junto con cualquier otra metodología de desarrollo ágil (como vimos con SCRUM), permite **visualizar y gestionar el trabajo** para lograr lo que todas las metodologías ágiles pretenden: mejorar la eficiencia, reducir el tiempo de entrega y asegurar un flujo continuo de trabajo.



Kanban - Introducción

El principio básico de Kanban (看板) es **visualizar el trabajo** dividido en columnas que representan diferentes etapas o estados del flujo de trabajo. Siempre están estas tres mínimo:

- **To do:** a hacer, pero todavía no comenzamos.
- **In Progress:** en progreso, trabajando actualmente.
- **Done:** tareas completadas.



Kanban - Tarjetas

Cada tarea o elemento de trabajo se representa con una **tarjeta** en el tablero. Las tarjetas pueden tener ciertos detalles:

- Un título
- Una descripción
- Asignaciones
- Fechas límite
- Prioridad



Kanban - WIP Limits

Una característica de Kanban es la de limitar la cantidad de trabajo que hay en la columna de trabajo en progreso, a esto se lo conoce como **WIP Limits**.

La idea es que el equipo no se sobrecargue y termine las tareas ya asignadas antes de comenzar nuevas.

Durante el desarrollo, es fácil decir “Bueno, voy a parar de hacer esto un segundo y arranco con otra cosa”, sin embargo esto es muy ineficiente; toma tiempo y quita concentración de cada tarea, dividiéndola en dos.

Es así que si el **WIP Limit** para ‘In Progress’ es 3, entonces solo podemos tener 3 tareas en esa columna, y si todas las tareas en esa columna están ocupadas, el equipo no puede comenzar una nueva hasta que terminen una de las que ya está en progreso.



Kanban - Mejora Continua

Obviamente, al ser una metodología ágil, y aplicar leyes de Lehman con relación a la autorregulación, Kanban promueve la **mejora continua** del flujo de trabajo.

Por lo general, se recolectan métricas como **lead time** (tiempo que tarda una tarea en moverse de una columna “To Do” a “Done”), y ajustar prácticas para mejorar la eficiencia.

Por ejemplo, si se están acumulando muchas tareas en “In Progress”, hay que modificar ya el WIP Limit porque estamos siendo inefficientes.

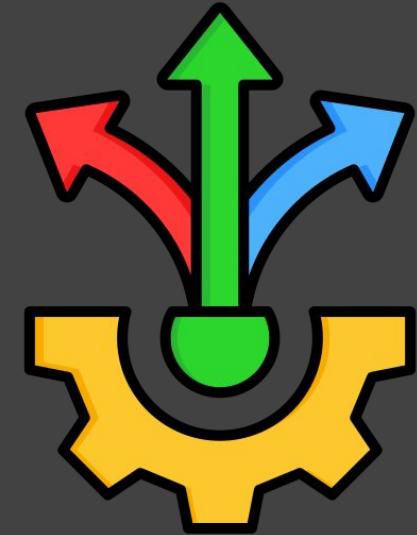


Kanban - Prioridades flexibles

Kanban permite **priorización dinámica**; las prioridades pueden cambiar a medida que surgen nuevas necesidades al principio de cada sprint (o incluso durante los sprints con los **Kanban de emergencia**).

Los Kanban de Emergencia se emiten de modo temporal cuando se requiera hacer frente a partes defectuosas del código, trabajos extraordinarios o esfuerzos especiales, si la situación lo amerita.

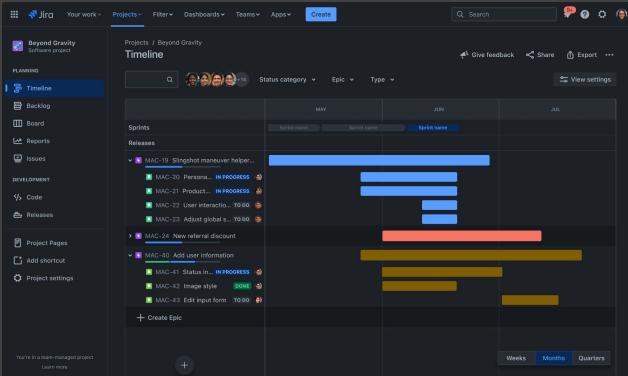
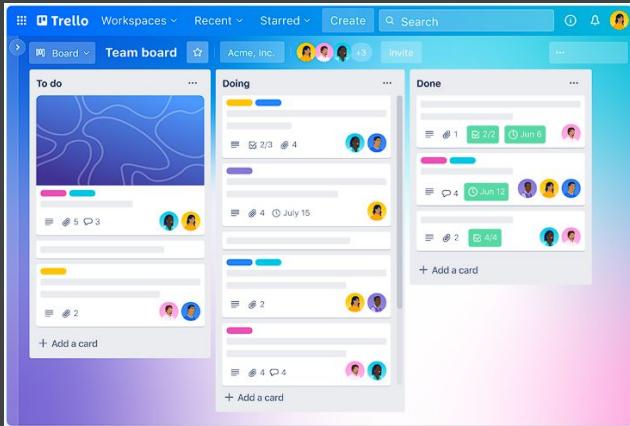
Por ejemplo, si aparece una tarea urgente, se agrega al tablero en la columna “To Do” y se trabaja en eso sin esperar a la finalización de un ciclo completo.



Herramientas para aplicar Kanban

Las herramientas más utilizadas para aplicar Kanban son:

- **Trello**, que dispone de un tablero visual para pequeñas y medianas empresas.
- **Jira**, una herramienta de gestión usada en empresas mucho más grandes.
- **Gitlab** y **Azure** también tienen sus propios tableros Kanban integrados para poder manejar el flujo de trabajo.





Herramientas para el análisis de requisitos

Lo más relevante en las metodologías de desarrollo de hoy en día para realizar un análisis de requisitos satisfactorio es dividir los mismos en historias manejables para facilitar la entrega iterativa y la priorización en ciclos de desarrollo: **historias de usuario**.

Más allá de las historias de usuario y las metodologías de desarrollo que las promueven, existen algunas herramientas que permiten la realización de un análisis de requisitos de manera más cómoda...

Herramientas para el análisis de requisitos



Confluence: permite crear, organizar y compartir documentos dentro de un equipo o empresa; funciona como una *wiki* empresarial para almacenar y gestionar información relevante.

- **Documentación de requisitos:** Creamos páginas para documentar los requisitos del proyecto de manera detallada, ya sean **funcionales** como **no funcionales**.
- **Bien organizado:** Los requisitos se organizan en una jerarquía clara, utilizando títulos, subtítulos y links internos (como Wikipedia!!).
- **Historial de versiones:** Confluence guarda versiones anteriores de los documentos, permitiendo ver el historial de cambios; útil para mantener un registro de cómo fueron evolucionando los requisitos.

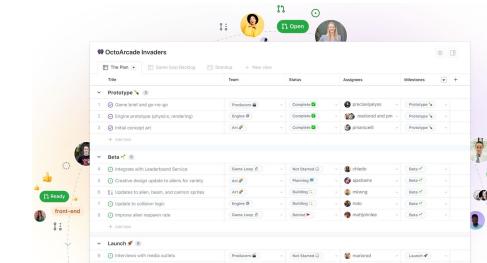


Herramientas para el análisis de requisitos



GitHub Issues: Es mucho más simple que Confluence; permite gestionar principalmente bugs y mejoras. Muy útil para proyectos que se manejan en GitHub.

- **Issues:** ‘incidentes’. Éstos pueden crearse para cada requisito, conteniendo una descripción detallada del mismo, links, discusiones, entre otros.
- **Asignación:** Los issues pueden ser asignados a desarrolladores o equipos para hacerse responsables del análisis o implementación del mismo, permitiendo una buena organización.
- **Commits y pull requests:** Al estar integrado directamente con el repositorio de código en GitHub, podemos referenciar commits o pull requests directamente desde los issues.





Open



OctoArcade Invaders



The Plan

Game loop Backlog

Standup

+ New view

Title	Team	Status	Assignees	Milestones
Prototype 🌱 3				
1 Game brief and go-no-go	Producers 🎬	Complete ✓	preciselyalyss	Prototype 🌱
2 Engine prototype (physics, rendering)	Engine ⚙️	Complete ✓	marirod and pm	Prototype 🌱
3 Initial concept art	Art 🌈	Complete ✓	pmarsceill	Prototype 🌱
+ Add item				
Beta 🌱 5				
4 Integrate with Leaderboard Service	Game Loop 📈	Not Started 🕒	chiedo	Beta 🌱
5 Creative design update to aliens for variety	Art 🌈	Planning 📈	ajashams	Beta 🌱
6 Updates to alien, beam, and cannon sprites	Art 🌈	Building 🏗️	mkwng	Beta 🌱
7 Update to collision logic	Engine ⚙️	Building 🏗️	mdo	Beta 🌱
8 Improve alien respawn rate	Game Loop 📈	Behind ➡️	mattjohnlee	Beta 🌱
+ Add item				
Launch 🚀 6				
9 Interviews with media outlets	Producers 🎬	Not Started 🕒	marirod	Launch 🚀
10 Save score across levels	Game Loop 📈	Not Started 🕒	pmarsceill	Launch 🚀



Open



Ready



front-end



Herramientas para el diseño

Una herramienta que ayuda mucho al diseño, y vimos la clase pasada es **Terraform**, ya que permite estructurar el proyecto de manera automática.

Otro puede ser **AWS CloudFormation**, que permite modelar, aprovisionar y gestionar recursos en la nube de AWS usando JSON o YAML.



CloudFormation usa un archivo de **plantilla** que describe los distintos **recursos** y su **configuración**. Cuando se manda el archivo a la nube, CloudFormation crea y gestiona los recursos en el orden adecuado, garantizando que se manejen las dependencias entre ellos.

Al conjunto completo de recursos que definió la plantilla, lo llamamos **pila** o **stack**.

Herramientas para el diseño

AWS CloudFormation contiene:

- **IaC (Infraestructura como Código):** Permite que toda la infraestructura de un proyecto se defina como código en un archivo de plantilla, convirtiéndo a la infraestructura y al código fuente en repetible y visionable.
- **Despliegue de recursos:** Los recursos definidos se van a crear y configurar de manera consistente, ya que la plantilla está automatizada, minimizando los riesgos y errores.
- **Gestión de dependencias:** Se hace automáticamente; si hay dependencias en el proyecto entre sí, las mismas van a gestionarse de manera automática para que los recursos se creen en el orden correcto.

Herramientas para la implementación



Se puede hacer una relación entre seguridad (TI) e implementación; aplicaciones como **SonarQube** o **Snyk**, que realizan revisiones automáticas de calidad y seguridad del código aplican tanto para tecnología de la información como para desarrollo de software.

Git, por otra parte, al tener un funcionamiento muy amplio como software de control de versiones, también puede aplicar para control de implementación, permitiendo el acceso al código a cualquier desarrollador del proyecto.

Con relación a integración continua y despliegue continuo (CI/CD), también nos vamos a referir a la implementación del código: herramientas que vimos la clase pasada como **Jenkins** pueden ser de gran ayuda para desarrollo de software también.

Herramientas para la implementación



GitHub Actions es otra herramienta que puede ser esencial para la implementación si estamos trabajando con repositorios en GitHub. Este nos permite automatizar, customizar y ejecutar los flujos de trabajo del desarrollo de software en nuestro repositorio.

La página web de GitHub actions lo explica perfectamente:

<https://docs.github.com/es/actions/about-github-actions/understanding-github-actions>



GitHub Actions



Herramientas para testeo e integración

Hay tres vertientes por las cuales se puede analizar el testeo e integración de una aplicación:

- A través de **pruebas automatizadas**, creando así pruebas unitarias, de integración, de regresión y funcionales para validar automáticamente el comportamiento del código en cada etapa.
- A través de **pruebas en entornos de staging**, permitiendo configurar pruebas en entornos que simulan producción (staging).
- A través de **monitorización de logs**, capturando y analizando logs en tiempo real durante el testing y después de la integración.

Herramientas para pruebas



JUnit es un framework de pruebas unitarias para Java. Permite validar que cada unidad de código (métodos, clases, funciones) funcionen de manera correcta en aislamiento.

JUnit tiene la capacidad de realizar **pruebas unitarias y pruebas de regresión**; esta última nos permite detectar si los cambios en el código realizados para arreglar pruebas unitarias causan errores en otras partes que anteriormente funcionaban.



JUnit dispone de:

- **Anotaciones:** definen el comportamiento de los métodos de prueba.
Algunas anotaciones pueden ser:
 - `@Test`: marca un método como una prueba.
 - `@Before` y `@After`: definen acciones antes y después de cada prueba.
 - `@BeforeClass` y `@AfterClass`: ejecutan configuraciones y limpiezas a nivel de clase.

JUnit Herramientas para pruebas

JUnit también dispone de **Assertions**. Estas permiten verificar condiciones esperadas. Si una condición falla, la prueba se marca entonces como fallida:

- assertEquals()
- assertTrue()
- assertNotNull()

Ejemplo:

```
public class Calculadora {  
    public int Suma(int a, int b) {  
        return a + b;  
    }  
}  
  
//en base a este método, hago un test...
```

```
public class TestCalculadora {  
    @Test  
    public void testSuma() {  
        Calculadora c = new Calculadora();  
        int resultado = c.Suma(2, 3);  
        assertEquals(5, resultado, "La  
        suma da 5");  
    }  
}  
  
//acá creamos un test que utiliza el método  
Suma de la clase Calculadora, y verifica si  
el resultado es correcto.
```

Herramientas para pruebas



Selenium se utiliza para pruebas de interfaz de usuario (UI) en el front-end de páginas de internet. Permite automatizar la interacción del usuario con el browser.

Selenium simula el comportamiento del usuario en aplicaciones web, permitiendo ser ejecutado en **diferentes navegadores** (google chrome, mozilla firefox, microsoft edge, etc.), y se integra con otros frameworks de pruebas como JUnit, TestNG o cualquiera que usemos.



Esta herramienta usa una API llamada **WebDriver** para poder interactuar con el browser. La misma nos permite localizar elementos de la página mediante selectores (*id*, *name*, *class*, *CSS selector*, etc.), para simular un clic, envío de texto, o verificación de estado.

Herramientas para pruebas

Ejemplo:

```
public class LoginTest {  
  
    @Test  
  
    public void testLogin() {  
  
        WebDriver driver = new  
ChromeDriver();  
  
        driver.get("http://app.com/login");  
  
        WebElement usuario =  
driver.findElement(By.id("usuario"));  
  
        WebElement contrasenia =  
driver.findElement(By.id("contrasenia"));  
  
        WebElement botonLogin =  
driver.findElement(By.id("botonLogin"));
```

```
        usuario.sendKeys("admin");  
  
        contrasenia.sendKeys("111111");  
  
        botonLogin.click();  
  
        assertTrue(driver.getCurrentUrl().contains("/paginaPrincipal"));  
  
        driver.quit();  
    }  
}
```

//Acá lo que hicimos fue simular el ingreso de un usuario llamado “admin” con contraseña “111111”. En el caso de que el login haya funcionado correctamente, debería habernos redirigido a la paginaPrincipal, entonces utilizamos la ruta del mismo como assertTrue para verificar.

Herramientas para pruebas



Postman sirve para hacer pruebas de APIs. Nos deja enviar solicitudes HTTP y verificar las respuestas sin necesidad de programar.

Las respuestas son verificadas de dos maneras:

- Podemos ver si contienen los datos esperados
- Verificamos el estado HTTP.
- Podemos crear colecciones de pruebas y ejecutarlas automáticamente.

Supongamos que queremos realizar una petición GET con una API de Pokemon o con la de países y banderas...



POSTMAN

Herramientas para pruebas

Vamos a usar este URL: <https://pokeapi.co/api/v2/pokemon/pikachu>

... o este para el de los países: <https://restcountries.com/v3.1/name/argentina>

Vamos a utilizar **GET**.

Podemos usar la versión online de Postman también!!

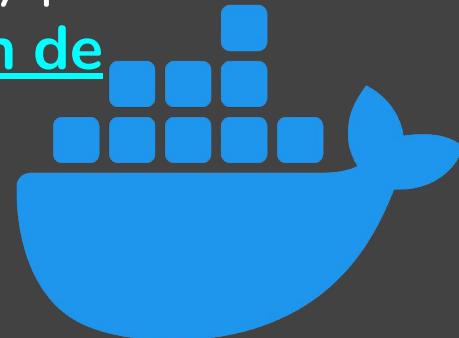
Después esto se prueba de manera automática:

```
pm.test("Status code is 200", function () {  
    pm.response.to.have.status(200);  
});  
  
pm.test("Name is Pikachu", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData.name).to.eql("pikachu");  
});  
  
pm.test("Type is Electric", function () {  
    var jsonData = pm.response.json();  
    var type = jsonData.types[0].type.name;  
    pm.expect(type).to.eql("electric");  
});  
  
//verificamos estado HTTP, nombre del  
pokemon y qué tipo es
```

Docker - Introducción

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de **contenedores de software**, en lugar de virtualizar hardware (como lo hacen las máquinas virtuales).

En el contexto de DevOps y desarrollo de software, Docker se convierte en una herramienta importantísima para simplificar la integración y entrega continua y para estar seguros de que las aplicaciones se ejecuten de manera consistente.

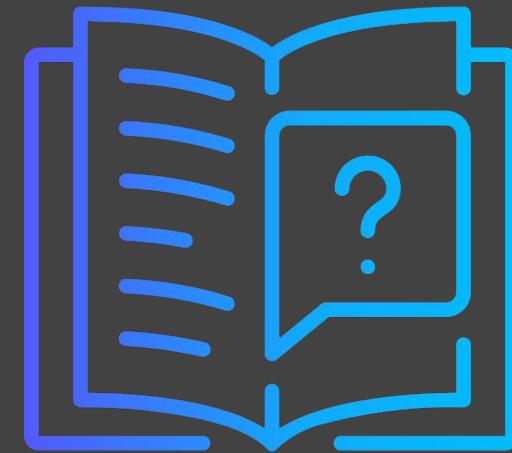


Docker - Origen

En la era moderna del desarrollo de software, los entornos de trabajo estandarizados y la capacidad de empaquetar aplicaciones junto con todas sus dependencias son requisitos fundamentales. A medida que los proyectos de software crecen y se vuelven más complejos, la necesidad de un **entorno uniforme** se hace cada vez más evidente, y Docker es una tecnología que responde a esta necesidad.

Imaginemos la situación común en la que una aplicación funciona perfectamente en la computadora de un desarrollador, pero al desplegarse en otro entorno (como en el servidor de producción o en la máquina de otro desarrollador), aparecen errores o se empieza a comportar de manera inesperada.

Ahí es donde tenemos que intentar manejar entornos complejos y dependencias entre sistemas; Docker nos provee un entorno para que repliquemos el sistema en cualquier lugar sin importar la arquitectura, utilizando algo llamado **contenedores**.

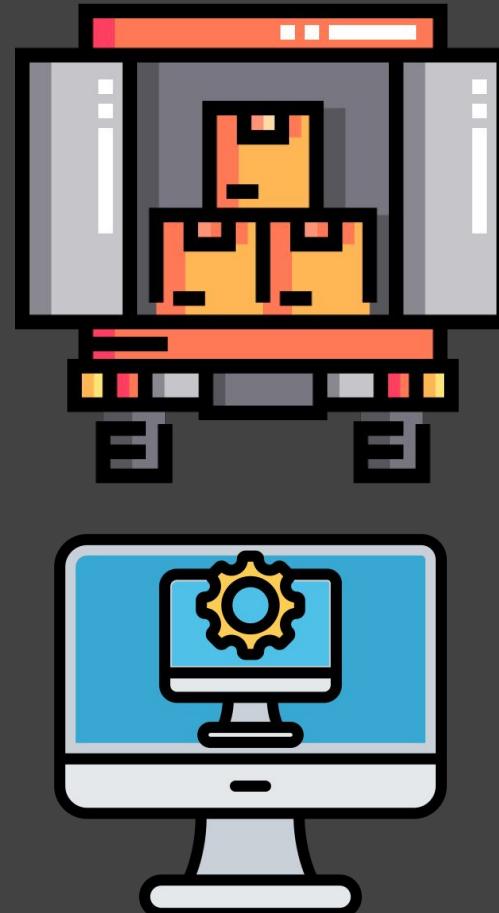


Docker - Contenedores vs. Máquinas Virtuales

Para entender Docker, primero hay que entender en qué se diferencian los contenedores de las máquinas virtuales:

Recordemos que una **máquina virtual** emula todo un sistema operativo completamente de arriba a abajo con su propio kernel y recursos (CPU, RAM, espacio en disco), lo que las hace muy pesadas en términos de recursos porque requieren su propio kernel y todos los drivers del sistema operativo para funcionar.

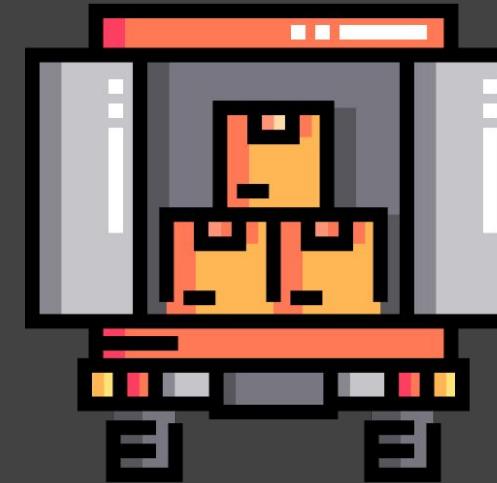
Por otra parte, un **contenedor** comparte el kernel del sistema operativo anfitrión, pero crea un entorno aislado que contiene solo la aplicación y sus dependencias. Al no emular el sistema operativo, los contenedores son **significativamente más ligeros y pueden ejecutarse a gran velocidad** con menos consumo de recursos.



Docker - Qué ventajas nos dan los contenedores

Gracias a los contenedores, podemos trabajar en un entorno controlado, predecible y replicable. Sus características ventajosas incluyen:

- **Consistencia entre entornos:** podemos asegurar que la aplicación funcione de la misma manera en cualquier entorno.
- **Escalabilidad:** las aplicaciones escalan en función de la demanda, ya que cada componente es muy fácil de desplegar y replicar, lo que lo convierte en crucial en aplicaciones de alta disponibilidad que necesiten adaptarse a fluctuaciones de tráfico (como vimos en DevOps I - herramientas para escalabilidad).
- **Automatización y CI/CD:** se integran los contenedores en pipelines de integración y entrega continua. Esto permite desplegar nuevas versiones rápidamente y con menos riesgo de errores.
- **Aislamiento de aplicaciones:** es posible ejecutar múltiples instancias de aplicaciones en el mismo sistema sin preocuparse de conflictos de dependencias ya que cada contenedor funciona de manera aislada.



Docker - Funcionamiento interno

Docker combina varias tecnologías:

- **Namespaces**: aíslan los recursos del sistema para que los contenedores parezcan entornos independientes.
- **Control Groups**: Limitan y distribuyen los recursos del sistema (CPU, RAM, etc.), a cada contenedor.
- **Union File Systems**: Nos proveen con un sistema de archivos en capas que hace que las imágenes de Docker sean eficientes en almacenamiento.
- **Networking**: Configura redes virtuales para que los contenedores puedan comunicarse entre sí y con el exterior sin interferir con el sistema anfitrión.



Kernel de Linux - Namespaces

Los namespaces son una funcionalidad del kernel de Linux que Docker usa para aislar diferentes aspectos de los contenedores. En Docker, los namespaces permiten que cada contenedor se ejecute en su propio “universo” sin interferir con otros contenedores o con el sistema anfitrión. Los namespaces más usados son:

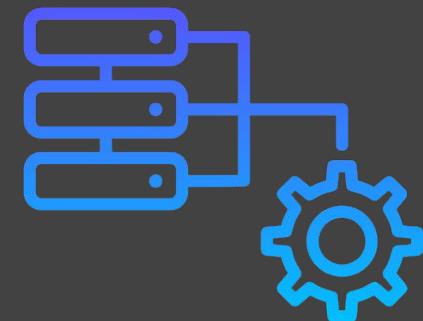
- **PID (Process ID)**: Aísla el espacio de los procesos para que cada contenedor tenga su propia numeración de PID. Esto significa que un proceso dentro de un contenedor no puede ver ni interferir con los procesos de otros contenedores ni del anfitrión.
- **Mount**: Controla el sistema de archivos que cada contenedor puede ver. Esto permite que un contenedor vea únicamente su sistema de archivos en lugar de todo el sistema de archivos del anfitrión.
- **Network**: Aísla las interfaces de red. Cada contenedor puede tener su propio conjunto de interfaces de red y su propio espacio de red, incluyendo puertos y direcciones IP.
- **IPC**: Aísla la comunicación entre procesos, de modo que los procesos dentro de un contenedor solo puedan comunicarse con otros procesos dentro del mismo contenedor.
- **User**: Permite que los contenedores tengan su propio espacio de usuarios, creando un entorno seguro en el que los usuarios dentro del contenedor pueden no tener privilegios en el sistema anfitrión, independientemente de su privilegio dentro del contenedor.



Kernel de Linux - CGroups

Los cgroups o “grupos de control” son otra funcionalidad del kernel de Linux que Docker utiliza para asignar y limitar los recursos que cada contenedor puede utilizar. Mediante los cgroups, Docker puede imponer límites estrictos en la cantidad de CPU, memoria RAM, disco y otras métricas de recursos que cada contenedor puede consumir.

Por ejemplo, si un contenedor ejecuta un proceso que consume mucha CPU, los cgroups pueden limitar el uso de CPU de ese contenedor, asegurando que otros contenedores o el sistema anfitrión no se vean afectados por este consumo excesivo.



Docker - UnionFS

Docker implementa UnionFS, que es un sistema que permite que cada imagen de Docker se construya sobre capas que representan cambios individuales en el sistema de archivos, en lugar de copiar archivos completos. Cada capa es inmutable, lo cual significa que no se modifica una vez creada. En lugar de eso, cada cambio crea una nueva capa.

Cuando Docker crea un contenedor a partir de una imagen, apila las capas y crea una capa adicional, llamada la capa de contenedor o capa de escritura. Solo esta última capa es modificable y contiene los cambios específicos del contenedor.

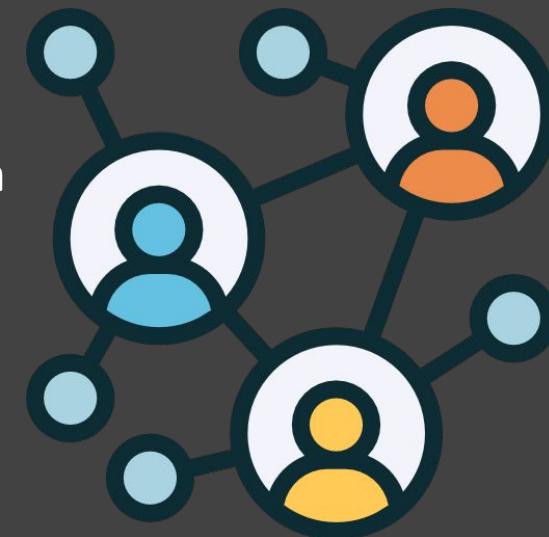


Docker - Redes

Docker tiene una infraestructura de red propia que es muy flexible. Los modos de red principales son:

- **Bridge**: Crea una red virtual privada donde los contenedores pueden comunicarse entre sí.
- **Host**: El contenedor comparte la red del anfitrión en lugar de tener su propia red aislada, sin ofrecer aislamiento de red (al contrario de **bridge**).
- **Overlay**: Permite crear redes que abarcan múltiples hosts, ideal para clústeres Docker en producción.
- **None**: Desactiva la red del contenedor.

Obviamente, Docker no prescinde de funcionalidades de Linux en términos de las redes, utilizando de hecho iptables (reglas de Firewall de Linux).



Docker - ¿Cómo lo hacemos andar?

Dockerfile: Primero, creamos un archivo llamado Dockerfile que construye la imagen del contenedor, definiendo el sistema operativo base, las dependencias de la aplicación, configuraciones específicas, variables de entorno y comandos a ejecutar.

Build: Ejecutamos ‘docker build’, procesando el Dockerfile y construyendo la imagen a partir del mismo. Cada instrucción en el Dockerfile crea una nueva capa en la imagen, usando UnionFS para apilar las capas de manera eficiente.

Run: Ejecutamos ‘docker run’, lanzando un contenedor a partir de una imagen tomada. Docker crea una capa de escritura temporal para almacenar los cambios realizados durante la ejecución y usa namespaces y cgroups para aislar el contenedor del sistema anfitrión.



Docker - Cliente

Docker usa arquitectura cliente-servidor. El **Docker Daemon** (*dockerd*) es un proceso en segundo plano que se ejecuta en el sistema anfitrión y maneja todas las operaciones de construcción, ejecución y supervisión de contenedores. El cliente Docker (*docker*) es una interfaz de línea de comandos que los usuarios utilizan para interactuar con el Daemon.

Cuando se ejecuta un comando como *docker run*, el cliente Docker se comunica con el Docker Daemon mediante una API para ejecutar la tarea solicitada.



Docker

Concluimos en que usar Docker para crear y gestionar contenedores puede simplificar la creación de sistemas altamente distribuidos, permitiendo que múltiples aplicaciones, las tareas de los trabajadores y otros procesos funcionen de forma autónoma en una única máquina física o en varias máquinas virtuales.

Docker permite una plataforma como servicio o *PaaS* de estilo de despliegue, además que simplifica la creación y el funcionamiento de las tareas de carga de trabajo.

El enfoque de Docker basado en contenedores impulsa la adopción general de filosofía DevOps, facilitando la colaboración entre desarrollo y operaciones y agilizando los ciclos de vida de CI/CD.



Herramientas para entornos de producción o staging



Kubernetes es una herramienta muy completa que se utiliza en muchas etapas de DevOps. Sin embargo, ayuda principalmente a administrar aplicaciones en contenedores en entornos de **producción o staging**.

- **Orquestación:** Kubernetes administra el ciclo de vida de los contenedores (inicio, paro, reinicio) en un clúster. Coordina el despliegue de las aplicaciones y asegura que la cantidad deseada de réplicas de los contenedores esté siempre en ejecución.
- **Rolling deploy:** Podemos realizar una actualización de una aplicación sin tiempo de inactividad mediante despliegues *rolling*, donde los contenedores se actualizan gradualmente.



Herramientas para entornos de producción o staging



Supongamos que tenemos una aplicación web distribuida y nos falta desplegarla en staging para pruebas y después en producción:

Primero **configuramos** un *cluster* en un proveedor de nube y lo organizamos en namespaces: *production* y *staging*.



Hacemos un **pipeline** para desplegar los cambios automáticamente en el namespace que hicimos antes, *staging*, cada vez que haya un merge a la rama de desarrollo. Después de eso, lo desplegamos en *production*.

En este momento, podemos ir probando nuevas versiones en el namespace *staging* y después mandamos un **Canary release** (para una cantidad limitada de gente) en *production*, así vemos qué piensan los usuarios y si es estable o no.

Configuramos alguna de las herramientas que vimos la clase pasada relacionada a monitoreo para ir verificando la respuesta de usuarios y la estabilidad antes de redirigir todo el tráfico.

Herramientas para monitoreo de logs



ELK Stack permite recolectar, almacenar y visualizar datos de logs. El mismo tiene tres componentes (E, L y K - Elasticsearch, Logstash y Kibana)

Elasticsearch es el núcleo del stack, una base de datos de búsqueda y análisis que permite almacenar y buscar grandes volúmenes de datos en tiempo real. Indexa los logs que vienen de múltiples fuentes y permite búsquedas rápidas en los datos históricos y actuales, ayudando a identificar patrones de errores y métricas de rendimiento.

Logstash es un motor de procesamiento de datos que ingiere, transforma y envía los logs a Elasticsearch. Se conecta a diversas fuentes de datos (servidores, aplicaciones, bases de datos), transforma los logs al formato deseado y los envía a Elasticsearch, permitiendo almacenar datos complejos antes de almacenarlos.

Kibana es la herramienta de visualización y dashboard de ELK, que facilita el análisis de datos almacenados en Elasticsearch mediante gráficos y paneles.

Herramientas para monitoreo de logs



Grafana Loki trabaja en conjunto con Grafana. Está optimizado para entornos en contenedores y se integra muy bien con clusters de Kubernetes.

Loki es el servicio de almacenamiento y gestión de logs. A diferencia de ELK, Loki almacena logs en texto plano y sólo indexa metadatos, lo que hace que consuma menos recursos. Es ideal para entornos de microservicios, especialmente Kubernetes, ya que puede recolectar logs directamente de los contenedores y organiza los logs en base a los labels de Kubernetes, como el nombre del namespace.

Herramientas para mantenimiento

Con respecto al mantenimiento, podemos destacar 4 campos distintos que son muy relevantes:



Despliegue continuo: Se pueden usar pipelines para garantizar el despliegue continuo, esto consiste principalmente en actualizaciones frecuentes y controladas en entornos de producción sin inactividad.

Monitoreo y alertas: Herramientas más relacionadas a TI como las que vimos la clase pasada, para monitorear el rendimiento de la aplicación en producción y recibir alertas por si pasó algo:

- **Prometheus**
- **Grafana**
- **Datadog**

Gestión de parches y actualizaciones: Automatización de instalación de parches de seguridad y actualizaciones de software, como por ejemplo:

- **Ansible** (apareció la clase pasada)
- **Chef**

Automatización de backups: Herramientas como las que ya vimos que realizan backups del sistema, intentando minimizar el riesgo por si ocurre algún desastre, como:

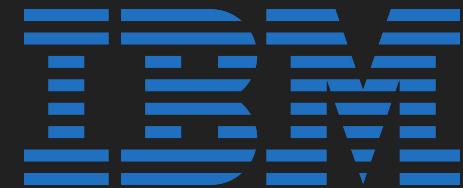
- **Veeam**
- **AWS Backup**

RUP - Rational Unified Process

Fausto Panello

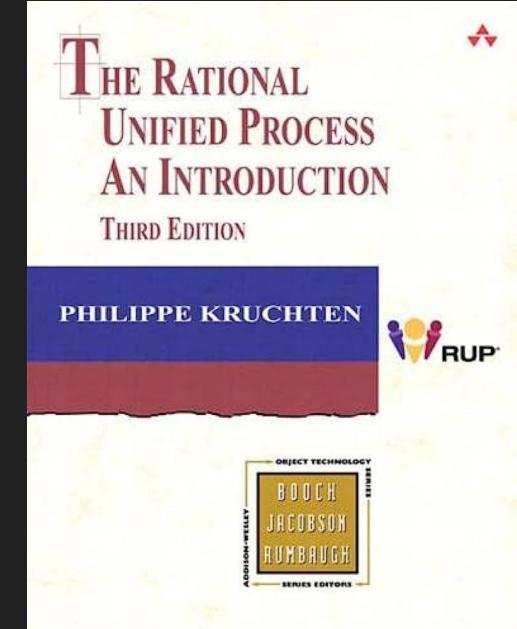
¿Qué es RUP?

El Proceso Racional Unificado o RUP (por sus siglas en inglés de Rational Unified Process) es un proceso de desarrollo de software desarrollado por la empresa Rational Software, actualmente propiedad de IBM. Junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, diseño, implementación y documentación de sistemas orientados a objetos.



Origen de RUP

Los orígenes de RUP se remontan al modelo espiral original de Barry Boehm. Ken Hartman, uno de los contribuidores claves de RUP colaboró con Boehm en la investigación. En 1995, Rational Software compró una compañía sueca llamada Objectory AB, fundada por Ivar Jacobson, famoso por haber incorporado los casos de uso a los métodos de desarrollo orientados a objetos. El Rational Unified Process fue el resultado de una convergencia de Rational Approach y Objectory (el proceso de la empresa Objectory AB). El primer resultado de esta fusión fue el Rational Objectory Process, la primera versión de RUP, fue puesta en el mercado en 1998, siendo el arquitecto en jefe Philippe Kruchten, quien escribió un libro “The Rational Unified Process - An Introduction”, en 1999.



RUP - Definición

RUP no es un sistema con pasos firmemente establecidos, sino un conjunto de metodologías adaptables, que se apoya en diversos artefactos, incluido en el software Rational Method Composer (RMC) como herramienta de aplicación para este método. Trabaja en conjunto con **UML**.

Originalmente se diseñó un proceso genérico y de dominio público, el Proceso Unificado, y una especificación más detallada, el Rational Unified Process, que se vendiera como producto independiente.



RUP - Casos de Uso

En el proceso unificado los casos de uso se utilizan para capturar los requisitos funcionales y para definir los contenidos de las iteraciones.

La idea es que en cada iteración, tome un conjunto de **casos de uso** y desarrolle todo el camino por las disciplinas (análisis, diseño, pruebas, implementación, etc.). Acá va un pequeño ejemplo:

Título: Procesar una venta en el punto de venta

Actores principales: Vendedor, Cajero, Comprador

Descripción: Este caso de uso describe cómo se procesa una venta en el punto de venta, desde la interacción del vendedor con el comprador hasta el pago y la finalización de la transacción por parte del cajero.

El vendedor registra los productos seleccionados por el comprador en el sistema de punto de venta.

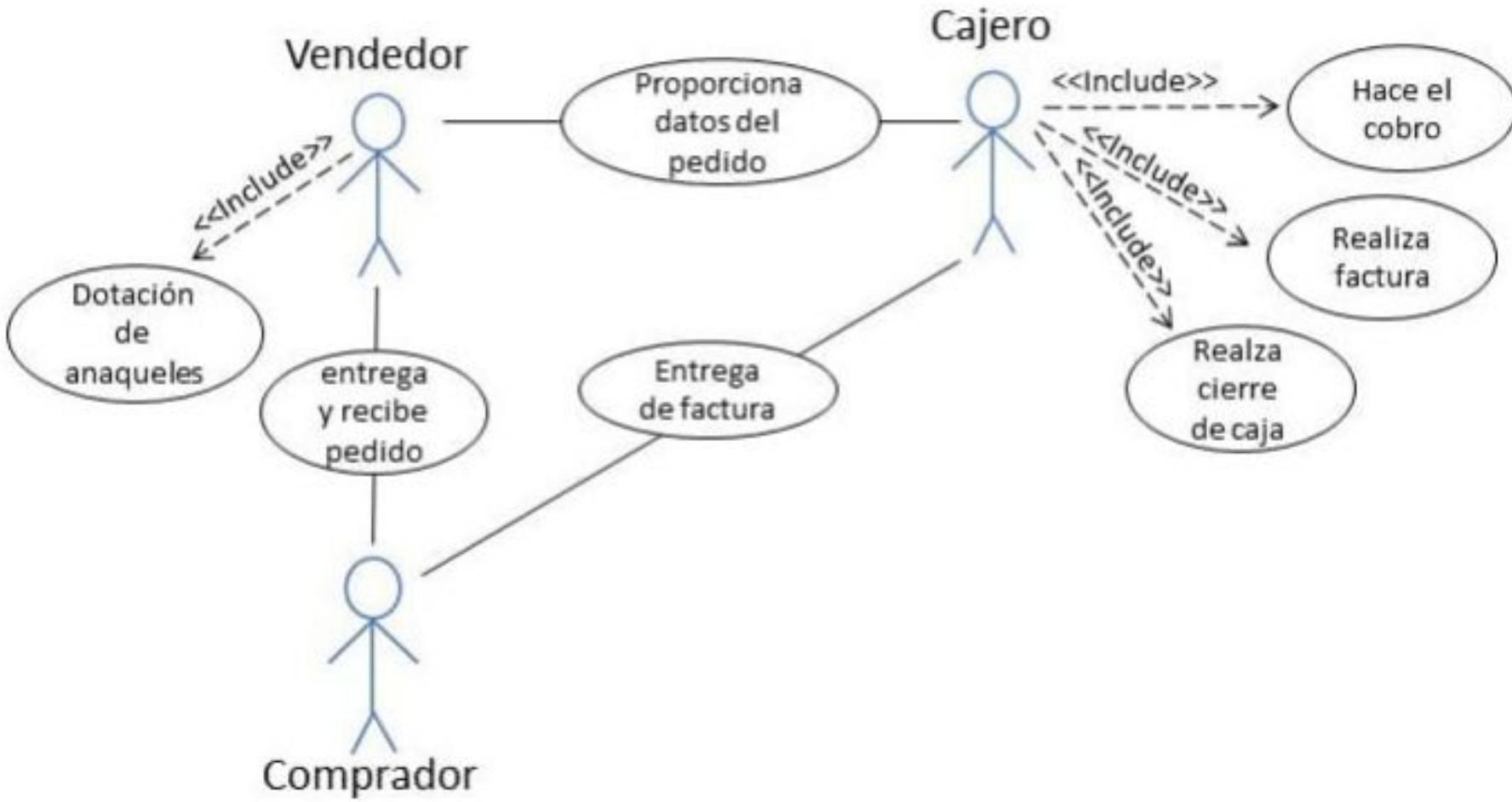
El sistema muestra una interfaz de registro de productos.

El vendedor escanea o ingresa manualmente los códigos de barras de los productos.

El sistema agrega los productos al carrito de compra y muestra el total acumulado.

El vendedor finaliza la selección de productos.

... seguiría el caso de uso.

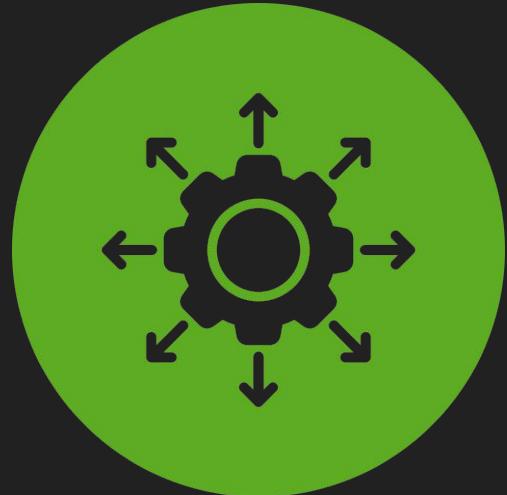


RUP - Principios clave

La Filosofía del RUP está basado en 6 principios clave que son los siguientes:

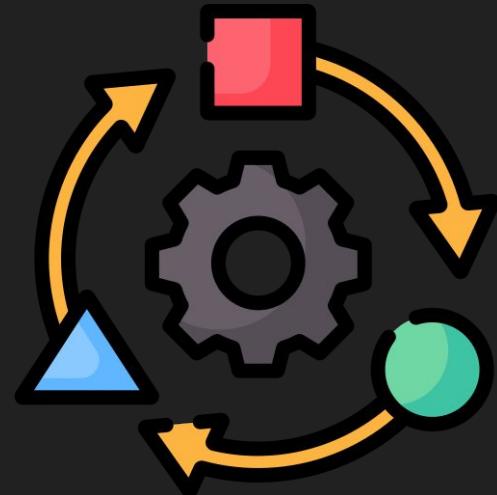
- Adaptar el proceso
- Equilibrar prioridades
- Demostrar valor iterativamente
- Colaboración entre equipos
- Elevar el nivel de abstracción
- Enfocarse en la calidad

Vamos a ver cada uno de los principios.



RUP - Principios clave - Adaptar el proceso

El proceso deberá adaptarse a las necesidades del cliente ya que es muy importante interactuar con él. Las características propias del proyecto u organización. El tamaño del mismo, así como su tipo o las regulaciones que lo condicionen, influirán en su diseño específico. También se deberá tener en cuenta el alcance del proyecto en un área sub formal.



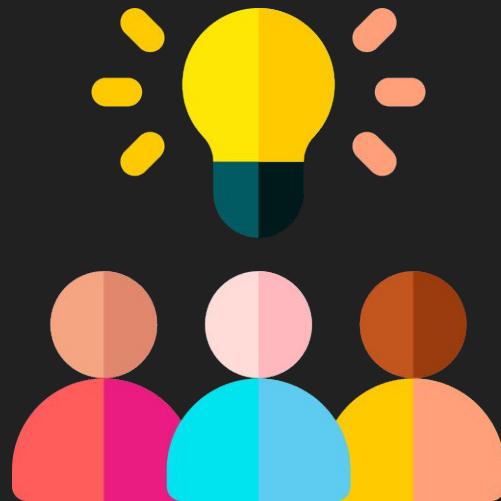
RUP - Principios clave - Equilibrar prioridades

Los requisitos de los diversos participantes pueden ser diferentes, contradictorios o disputarse recursos limitados. Debe encontrarse un equilibrio que satisfaga los deseos de todos. Gracias a este equilibrio se podrán corregir desacuerdos que surjan en el futuro.



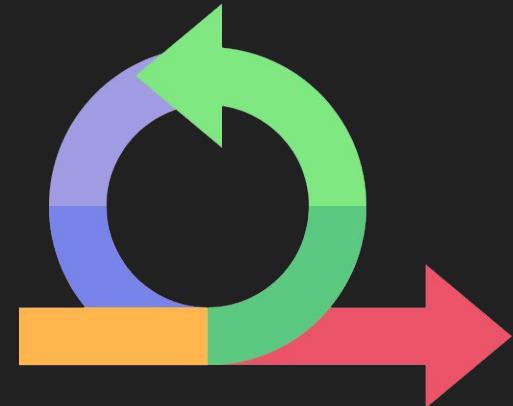
RUP - Principios clave - Colaboración entre equipos

El desarrollo de software no lo hace una única persona sino múltiples equipos. Debe haber una comunicación fluida para coordinar requisitos, desarrollo, evaluaciones, planes, resultados, etc.



RUP - Principios clave - Demostrar valor iterativamente

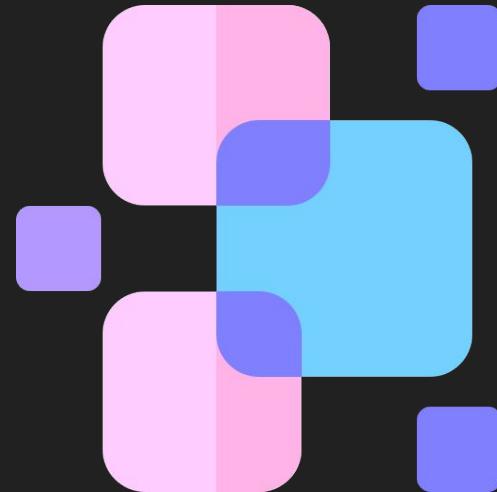
Los proyectos se entregan, aunque sea de un modo interno, en etapas iteradas. En cada iteración se analiza la opinión de los inversores, la estabilidad y calidad del producto, y se refina la dirección del proyecto, así como también los riesgos involucrados



RUP - Principios clave - Elevar el nivel de abstracción

Este principio dominante motiva el uso de conceptos reutilizables tales como patrón del software, lenguajes 4GL o marcos de referencia (frameworks) por nombrar algunos. Esto evita que los ingenieros de software vayan directamente de los requisitos a la codificación de software a la medida del cliente, sin saber con certeza qué codificar para satisfacer de la mejor manera los requisitos y sin comenzar desde un principio pensando en la reutilización del código.

Un alto nivel de abstracción también permite discusiones sobre diversos niveles y soluciones arquitectónicas. Éstas se pueden acompañar por las representaciones visuales de la arquitectura, por ejemplo, con el lenguaje UML.



RUP - Principios clave - Enfocarse en la calidad

El control de calidad no debe realizarse al final de cada iteración, sino en todos los aspectos de la producción. El aseguramiento de la calidad forma parte del proceso de desarrollo y no de un grupo independiente.



RUP - Ciclo de vida

El ciclo de vida RUP es una implementación del desarrollo en espiral. Fue creado ensamblando los elementos en secuencias semi-ordenadas. El ciclo de vida organiza las tareas en fases e iteraciones.

RUP divide el proceso en cuatro fases, dentro de las cuales se realizan pocas pero grandes y formales iteraciones en número variable según el proyecto.

Las fases son:

- Inicio
- Elaboración
- Construcción
- Transición



RUP - Fase de inicio

Esta fase tiene como propósito definir y acordar el alcance del proyecto con los patrocinadores, identificar los riesgos asociados al proyecto, proponer una visión muy general de la arquitectura de software y producir el plan de las fases y el de iteraciones posteriores.



RUP - Elaboración

En esta fase se seleccionan los casos de uso que permiten definir la arquitectura base del sistema y se desarrollan en esta fase, se realiza la especificación de los casos de uso seleccionados y el primer análisis del dominio del problema, se diseña la solución preliminar.



RUP - Construcción

El propósito de esta fase es completar la funcionalidad del sistema, para ello se deben clarificar los requisitos pendientes, administrar los cambios de acuerdo a las evaluaciones realizados por los usuarios y se realizan las mejoras para el proyecto.



RUP - Transición

El propósito de esta fase es asegurar que el software esté disponible para los usuarios finales, ajustar los errores y defectos encontrados en las pruebas de aceptación, capacitar a los usuarios y proveer el soporte técnico necesario. Se debe verificar que el producto cumpla con las especificaciones entregadas por las personas involucradas en el proyecto.



Fases del Método RUP



Ventajas y desventajas

- Es una forma disciplinada de asignar tareas y responsabilidades en una empresa de desarrollo (quién hace qué, cuándo y cómo)..

- Método pesado
- Por el grado de complejidad puede ser no muy adecuado para proyectos de corto alcance.
- En proyectos pequeños, es posible que no se puedan cubrir los costos de dedicación del equipo de profesionales necesarios.



Adecuado para proyectos:

- Al ser las iteraciones un proceso interno, aplica para sistemas de gran tamaño, donde las entregas no necesariamente sean un sistema completo preliminar (ejemplo: sistema operativo).

Introducción a UML

Fausto Panello

¿Qué es UML?

UML es un standard de OMG (Object Management Group) grupo dedicado a gestionar estándares de tecnología orientada a objetos.

Es un lenguaje de modelado. Un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema es capturar las partes esenciales del sistema.

El modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar.

UML es ante todo un **lenguaje**. Un lenguaje proporciona un **vocabulario** y una serie de **reglas** para permitir una comunicación.

En este caso, este lenguaje se centra en la representación gráfica de un sistema.



UML - Origen

El lenguaje UML comenzó a gestarse en octubre de 1994, cuando Rumbaugh se unió a la compañía Rational fundada por Booch (dos reputados investigadores en el área de metodología del software). El objetivo de ambos era unificar dos métodos que habían desarrollado: el método Booch y el OMT (Object Modelling Tool). El primer borrador apareció en octubre de 1995.

En esa misma época otro reputado investigador, Jacobson, se unió a Rational y se incluyeron ideas suyas. Estas tres personas son conocidas como los “tres amigos”. Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML.



UML - Origen

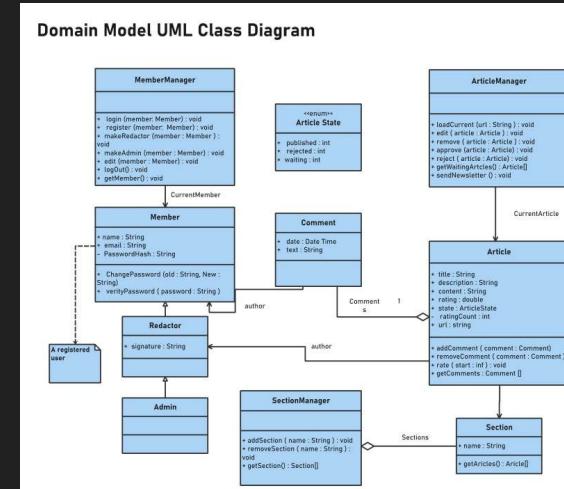
Esta primera versión mencionada la diapositiva anterior se ofreció a un grupo de trabajo para convertirlo en 1997 en un estándar del OMG (Object Management Group <http://www.omg.org>). Este grupo, que gestiona estándares relacionados con la tecnología orientada a objetos (metodologías, bases de datos objetuales, CORBA, etc.), propuso una serie de modificaciones y una nueva versión de UML (la 1.1), que fue adoptada por el OMG como estándar en noviembre de 1997. Desde aquella versión ha habido varias revisiones que gestiona la OMG Revision Task Force.



UML - Modelado visual

Tal como indica su nombre, UML es un lenguaje de modelado. Un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema es capturar las partes esenciales del sistema. Para facilitar este modelado, se realiza una abstracción y se plasma en una notación gráfica. Esto se conoce como modelado visual.

El modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar. De la misma forma que para construir una choza no hace falta un modelo, cuando se intenta construir un sistema complejo como un rascacielos, es necesario abstraer la complejidad en modelos que el ser humano pueda entender.

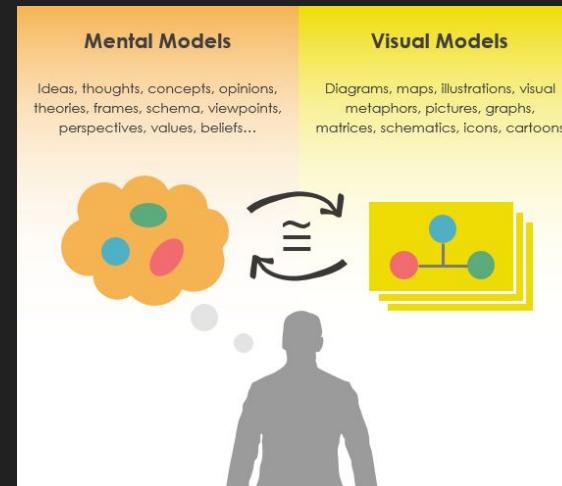


UML - Modelado visual

UML sirve para el modelado completo de sistemas complejos, tanto en el diseño de los sistemas software como para la arquitectura hardware donde se ejecuten. Otro objetivo de este modelado visual es que sea independiente del lenguaje de implementación, de tal forma que los diseños realizados usando UML se puedan implementar en cualquier lenguaje que soporte las posibilidades de UML (principalmente lenguajes orientados a objetos).

UML es además un método formal de modelado. Esto aporta las siguientes ventajas:

- Mayor rigor en la especificación.
- Permite realizar una verificación y validación del modelo realizado.
- Se pueden automatizar determinados procesos y permite generar código a partir de los modelos y a la inversa (a partir del código fuente generar los modelos). Esto permite que el modelo y el código estén actualizados, con lo que siempre se puede mantener la visión en el diseño, de más alto nivel, de la estructura de un proyecto.



UML - Objetivos

Recordemos que en la primera clase vimos “modelos”. Las metodologías de desarrollo siempre tienen como objetivo interpretar y leer los modelos, pero no cómo crearlos. Siempre hay que tener eso en cuenta antes de marcar los objetivos reales de una metodología de desarrollo. Dicho esto, las funciones, propias y naturales de UML incluyen:

- **Visualizar:** UML permite expresar de una forma gráfica un sistema de forma que otro lo puede entender.
- **Especificar:** UML permite especificar cuáles son las características de un sistema antes de su construcción.
- **Construir:** A partir de los modelos especificados se pueden construir los sistemas diseñados.
- **Diagramar estados**
- **Documentar:** Los propios elementos gráficos sirven como documentación del sistema desarrollado que pueden servir para su futura revisión.



UML - Objetivos

Aunque UML está pensado para modelar sistemas complejos con gran cantidad de software, el lenguaje es lo suficientemente expresivo como para modelar sistemas que no son informáticos, como flujos de trabajo (workflow) en una empresa, diseño de la estructura de una organización y por supuesto, en el diseño de hardware.

Un modelo UML está compuesto por **tres clases de bloques de construcción:**

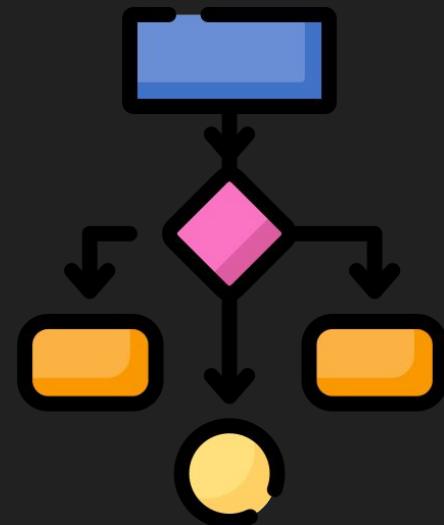
- **Elementos:** Los elementos son abstracciones de cosas reales o ficticias (objetos, acciones, etc.)
- **Relaciones:** relacionan los elementos entre sí.
- **Diagramas:** Son colecciones de elementos con sus relaciones.



UML - Diagramas

Un diagrama es la representación gráfica de un conjunto de elementos con sus relaciones. En concreto, un diagrama ofrece una vista del sistema a modelar. Para poder representar correctamente un sistema, UML ofrece una amplia variedad de diagramas para visualizar el sistema desde varias perspectivas. UML incluye los siguientes diagramas:

- Diagrama de casos de uso.
- Diagrama de clases.
- Diagrama de objetos.
- Diagrama de secuencia.
- Diagrama de colaboración.
- Diagrama de estados.
- Diagrama de actividades.
- Diagrama de componentes.
- Diagrama de despliegue.



UML - Diagramas

Si bien hay 9 diagramas en UML, nos vamos a centrar en:

- Diagrama de casos de uso
- Diagrama de clases
- Diagrama de secuencia

El diagrama de estados también se usa mucho, pero no lo vamos a ver en principio.

UML - Diagrama de caso de uso

El diagrama de uso que cada interactor representa lo que tiene

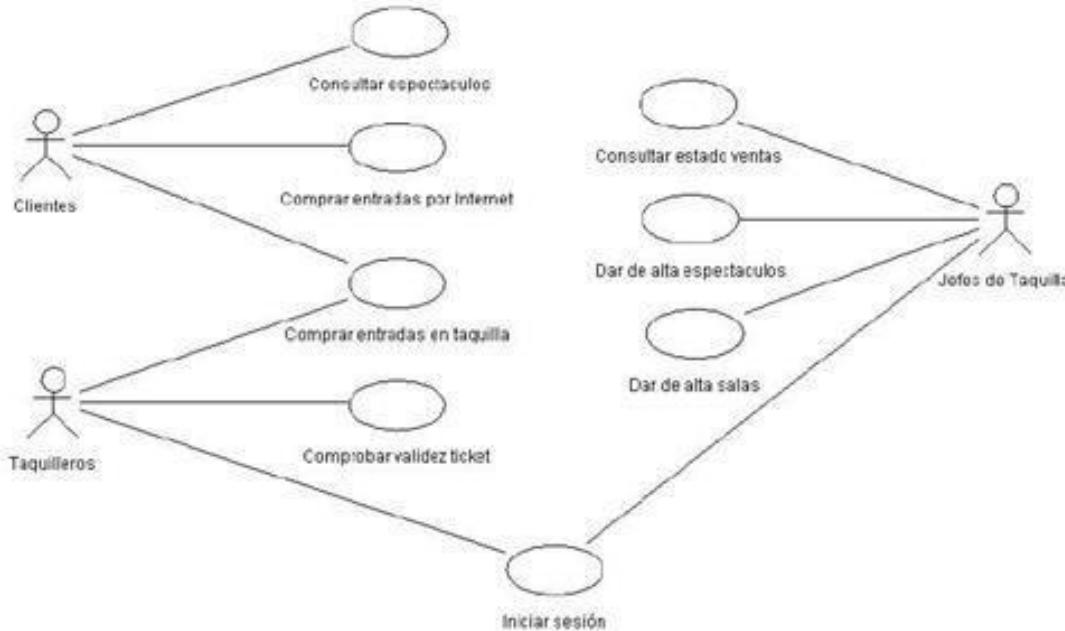


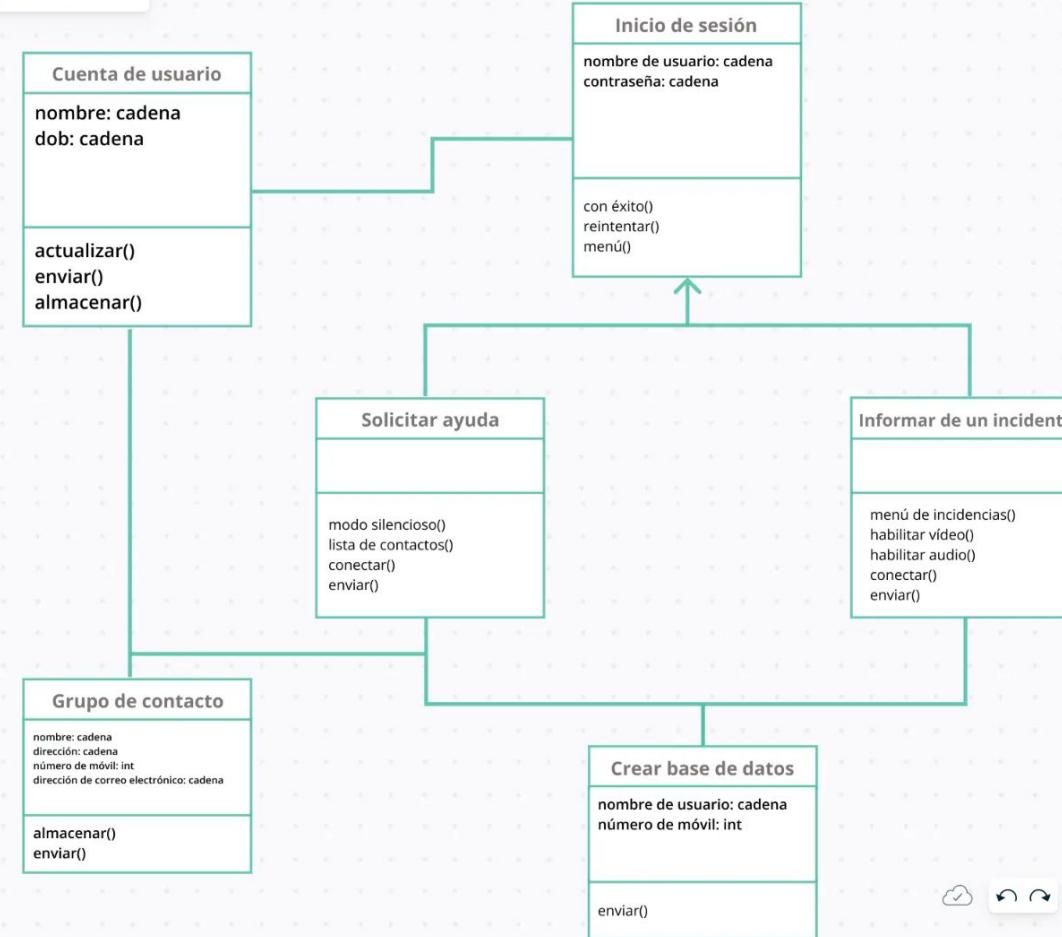
Figura 3: Diagrama de casos de uso



FORMAS SIMPLES

Núcleo Proceso +

Buscar formas



UML - Diagrama de secuencia

En esta figura, el diagrama de secuencia muestra la interacción de los objetos que componen un sistema de forma temporal.

Se muestra la interacción de crear una nueva sala para un espectáculo, por ejemplo.

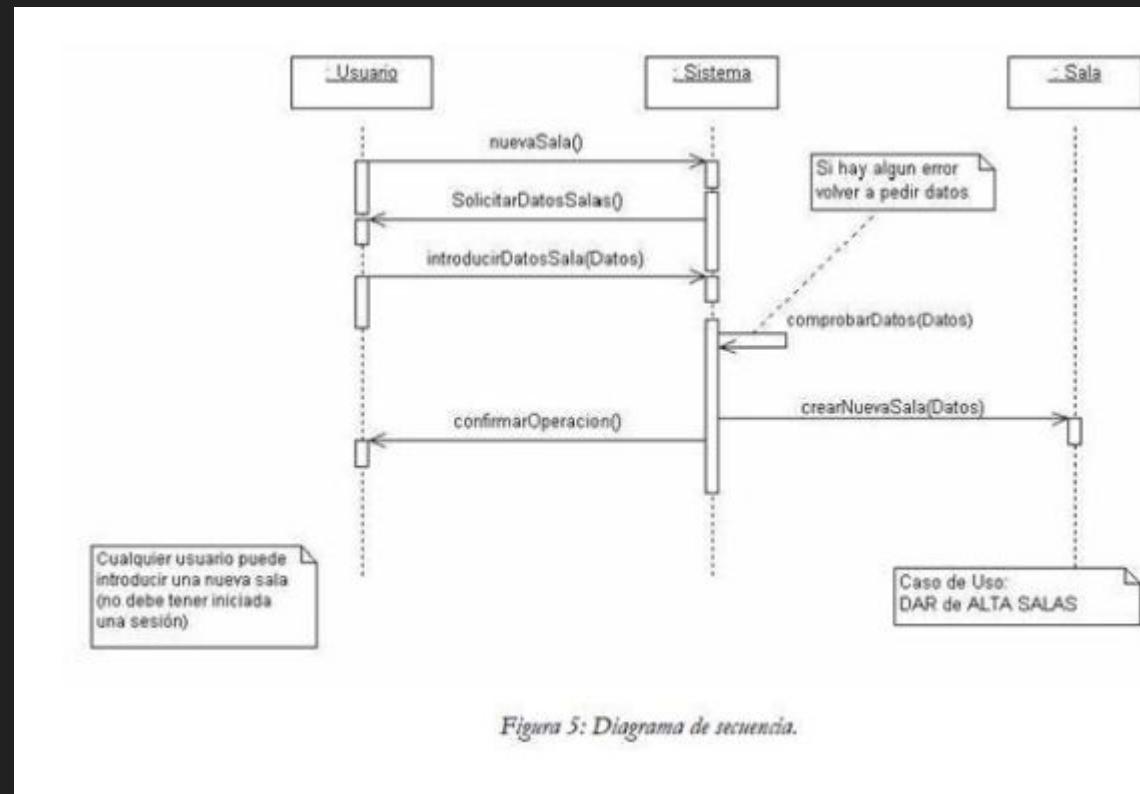


Figura 5: Diagrama de secuencia.

UML - Proceso de desarrollo

Aunque UML es bastante independiente del proceso de desarrollo que se siga, los mismos creadores de UML han propuesto su propia metodología de desarrollo, denominada el Proceso Unificado de Desarrollo.

El Proceso Unificado está basado en componentes, lo cual quiere decir que el sistema software en construcción está formado por componentes software interconectados a través de interfaces bien definidos. Además, el Proceso Unificado utiliza el UML para expresar gráficamente todos los esquemas de un sistema software. Pero realmente los aspectos que definen este Proceso Unificado son tres:

es iterativo e incremental,
dirigido por casos de uso,
centrado en la arquitectura.



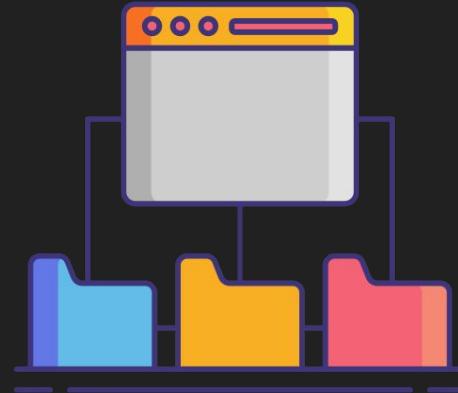
UML - Proceso de desarrollo - Iterativo e incremental

Todo sistema informático complejo supone un gran esfuerzo que puede durar desde varios meses hasta años. Por lo tanto, lo más práctico es dividir un proyecto en varias fases. Actualmente se suele hablar de ciclos de vida en los que se realizan varios recorridos por todas las fases. Cada recorrido por las fases se denomina iteración en el proyecto en la que se realizan varios tipos de trabajo (denominados flujos). Además, cada iteración parte de la anterior incrementando o revisando la funcionalidad implementada. Se suele denominar como “proceso”.



UML - Proceso de desarrollo - Dirigido por casos de uso

Basándose en los casos de uso, los desarrolladores crean una serie de modelos de diseño e implementación que los llevan a cabo. Además, estos modelos se validan para que sean conformes a los casos de uso. Finalmente, los casos de uso también sirven para realizar las pruebas sobre los componentes desarrollados.



UML - Proceso de desarrollo - Centrado en la arquitectura

En la arquitectura de la construcción, antes de construir un edificio éste se contempla desde varios puntos de vista: estructura, conducciones eléctricas, fontanería, etc. Cada uno de estos aspectos está representado por un gráfico con su notación correspondiente. Siguiendo este ejemplo, el concepto de arquitectura software incluye los aspectos estáticos y dinámicos más significativos del sistema.



UML - Resumen

Resumiendo, el Proceso Unificado es un modelo complejo con mucha terminología propia, pensado principalmente para el **desarrollo de grandes proyectos**. Es un proceso que puede adaptarse y extenderse en función de las necesidades de cada empresa.

UML II

Diagramas

Diagrama de casos de uso.

Diagrama de clases.

Diagrama de objetos.

Diagrama de secuencia.

Diagrama de colaboración.

Diagrama de estados.

Diagrama de actividades.

Diagrama de componentes.

Diagrama de despliegue.

Diagrama de casos de uso

El uso de los diagramas de casos de uso será, por lo general, parte de un documento de diseño que el cliente y el equipo de diseño tomarán como referencia de las acciones inmediatas que puede realizar el usuario final sobre el sistema o aplicación desarrollado.

De esta manera se pueden explicar rápidamente conceptos que ayudaran a un analista a comprender como un sistema deberá comportarse. Para ello es necesario aprender a visualizar los conceptos involucrados dentro de los casos de uso para así poder colocar todos los actores en el escenario correspondiente, incluyendo las actividades que pueden producir en dicha situación.

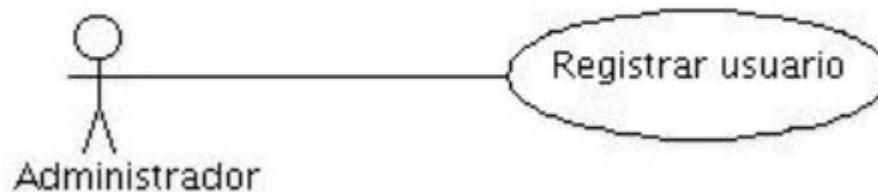
Como se menciona anteriormente los diagramas de casos de uso se emplean para modelar la vista de casos de uso estática de un sistema. Esta vista cubre principalmente el comportamiento del sistema (los servicios visibles externamente que proporciona el sistema en el contexto de su entorno).

Actor



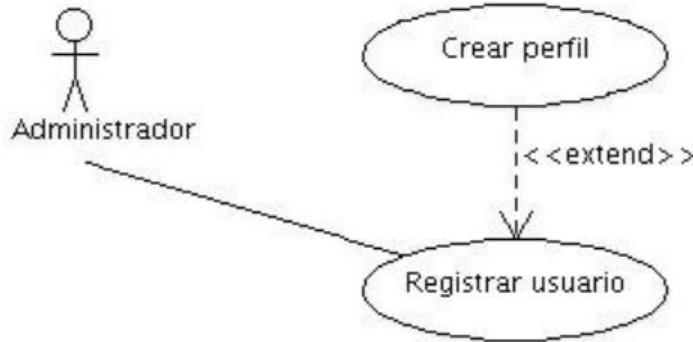
Un actor es una entidad externa al sistema que necesita intercambiar información con el sistema. La notación utilizada en UML para representar un actor es una figura humana con el nombre del actor.

Caso de uso



A diferencia de las técnicas de análisis estructurado, el diagrama de casos de uso no persigue modelar un flujo de datos. Un caso de uso puede definirse como un flujo completo de eventos en el que se especifica la interacción entre el actor y el sistema o como un negocio trabaja actualmente. La ejecución de un caso de uso termina cuando el actor genere un evento que requiera un caso de uso nuevo.

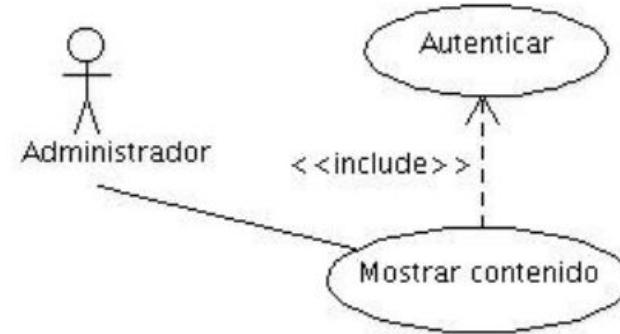
Extensión



Una extensión indica que la realización del caso de uso que extiende no es obligatoria durante la realización del caso de uso que está siendo extendido.

En el ejemplo que se muestra, el caso de uso Crear perfil puede o no ser realizado al ejecutarse el caso de uso Registrar usuario.

Inclusión



Una inclusión indica que un caso de uso se realizará incondicionalmente durante la realización del caso de uso que lo referencia.

En el ejemplo que se muestra, el caso de uso Autenticar debe ser realizado para que el caso de uso Mostrar contenido sea realizado correctamente.

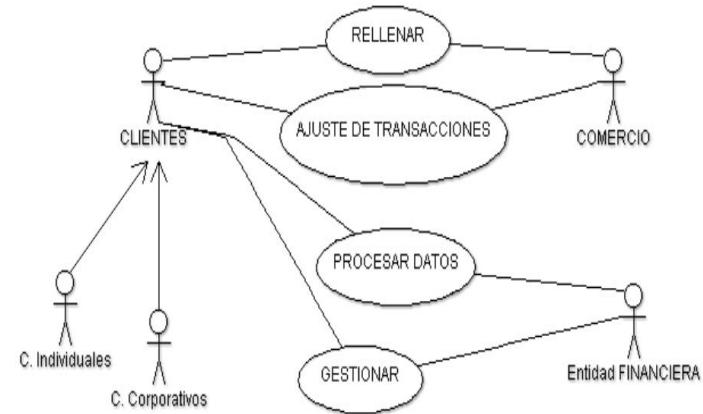
Funcionamiento

Cuando se modela la vista de casos de uso estática de un sistema, normalmente se emplearán los diagramas de casos de uso de una de las dos formas siguientes:

- 1) Para modelar el contexto de un sistema: Modelar el contexto de un sistema implica dibujar una línea alrededor de todo el sistema y asegurar qué actores quedan fuera del sistema e interactúan con él. Aquí, se emplearán los diagramas de casos de uso para especificar los actores y significado de sus roles.
- 2) Para modelar los requisitos de un sistema: El modelado de los requisitos de un sistema implica especificar qué debería hacer el sistema (desde un punto de vista externo), independientemente de cómo se haga. Aquí se emplearán los diagramas de casos de uso, para especificar el comportamiento deseado del sistema. De esta forma, un diagrama de casos de uso permite ver el sistema entero como una caja negra; se puede ver qué hay fuera del sistema y cómo reacciona a los elementos externos, pero no se puede ver cómo funciona por dentro.

Ejemplo del contexto de un sistema

La siguiente figura muestra el contexto de un sistema de validación de tarjetas de crédito, destacando los actores en torno al sistema. Se puede ver que existen Clientes, de los cuales hay dos categorías (Cliente individual y Cliente corporativo). Estos actores representan los roles que juegan las personas que interactúan con el sistema. En este contexto, también hay actores que representan a otras instituciones tales como Comercio (que es donde los Clientes realizan una transacción con tarjeta para comprar un artículo o servicio) y Entidad Financiera (que presta servicio como sucursal bancaria para la cuenta de la tarjeta de crédito). En el mundo real estos dos actores probablemente sean sistemas con gran cantidad de software.



Ejemplo de los requisitos de un sistema

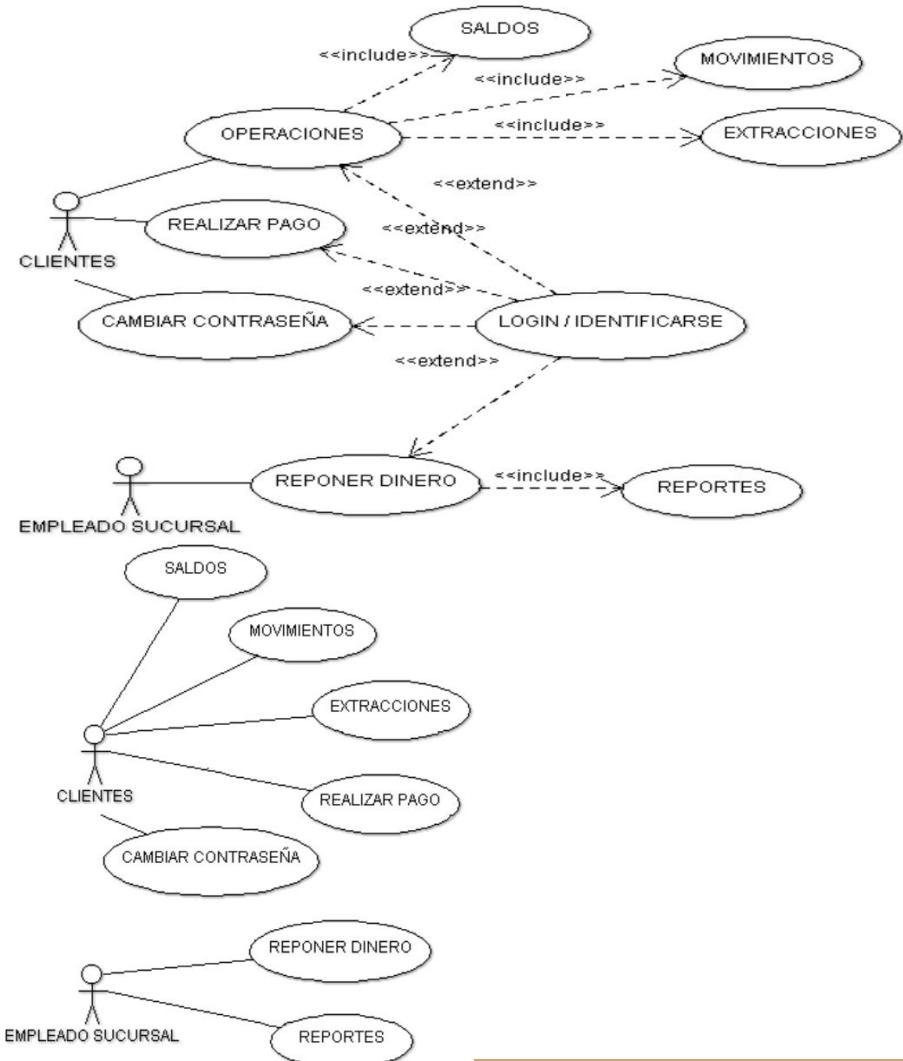
Continuando con el ejemplo anterior, en este caso, aunque omite las relaciones entre los actores y los casos de uso, añade casos de uso adicionales que son invisibles para el cliente normal, aunque son comportamientos fundamentales del sistema. Este diagrama es valioso porque ofrece un punto de vista común para los usuarios finales, los expertos del dominio y los desarrolladores para visualizar, especificar, construir y documentar sus decisiones sobre los requisitos funcionales del sistema.

Por ejemplo, Detectar fraude de tarjeta es un comportamiento importante tanto para el Comercio como para la Entidad Financiera. Análogamente, Informe estado de Cuentas, es otro comportamiento requerido del sistema por varias entidades.



Más ejemplos

Las figura de arriba muestra el caso de uso desplegado y como lo vería un desarrollador y la otra muestra el primer escenario de caso de uso que suele ser utilizado para instruir al usuario regular, cuando se realiza una exhibición de facultades del producto, como es el funcionamiento y las posibilidades que brinda la aplicación.



Ejemplo de modelado de requisitos (más allá del software)

El diagrama de casos de uso se puede utilizar no solo para representar esquemas informáticos, sino que también sirven para demostrar el modelado de requisitos de un sistema.

Ejemplo:

Te encargan realizar una aplicación para la compra-venta de cuadros. En cuanto a la compra de cuadros, una vez que el agente introduce unos datos básicos sobre el cuadro, el sistema debe proporcionar el precio recomendado que el agente de la galería debería pagar. Si el vendedor del cuadro acepta la oferta, entonces el agente de la galería introduce más detalles (sobre el vendedor del cuadro y la venta).

Los datos básicos incluyen el nombre y apellidos del artista, el título y fecha de la obra, sus dimensiones, la técnica (óleo, acuarela u otras técnicas), el tema (retrato, naturaleza muerta, paisaje, otro) y la clasificación (obra maestra, obra representativa, otro tipo). Si es obra maestra, el precio recomendado se calcula comparando el cuadro introducido con los que hay en el registro de cuadros, tomando el más parecido y aplicando un algoritmo que tiene en cuenta la coincidencia de tema, la técnica y las dimensiones del cuadro. El sistema debe utilizar información de subasta de todo el mundo que ahora la galería recibe en un CD de manera mensual. Para una obra representativa, el precio recomendado se calcula como si fuera una obra maestra y luego se aplica una corrección. Para una obra de otro tipo, se calcula utilizando el área del cuadro y un coeficiente de moda para el artista. Si no hay coeficiente de moda para un artista, el agente tiene por norma no comprar el cuadro. El coeficiente de moda varía de mes a mes...

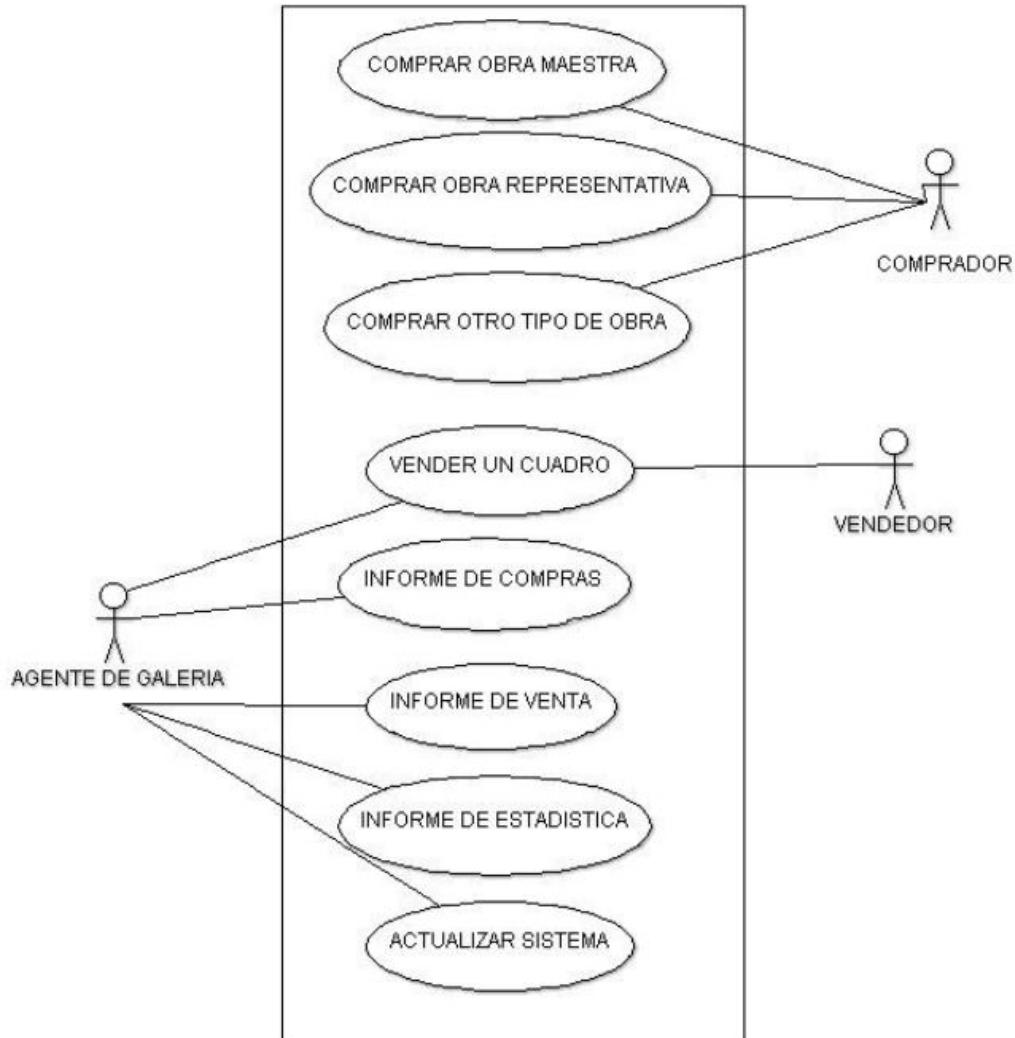
... sigue el ejemplo

Si el cuadro finalmente se compra, se introducen datos adicionales.

En cuanto a la venta de cuadros por parte de la galería, el sistema simplemente registra la fecha de venta, el nombre y dirección del comprador y el precio de venta real.

El sistema también deberá detectar nuevas tendencias en el mercado de arte tan pronto como sea posible. La idea es detectar secuencias de compras por valores mayores que los esperados por la obra de un artista determinado, de tal manera que tu cliente pueda comprar cuadros de ese artista antes de que otros detecten la tendencia. Con el objetivo de detectar cuándo el precio de venta es mayor que el precio esperado cuando tu cliente compró el cuadro, se debe mantener un registro de todas las compras y todas las ventas.

Se quieren generar tres informes: compras y ventas realizadas durante un año, y artistas de moda....



Se puede visualizar en el Diagrama de caso de uso de requisitos sobre las diferentes actividades en las cuales se ven relacionados los diferentes actores.

En tanto el agente galería se encuentra afectado por todos los casos de uso, su relación con el comprador solamente es la de "vender un cuadro", mientras que con el vendedor existen 3 tipos de "compras" posibles, según los detalles diagnosticados en el ejercicio.

Por lo detallado en el esquema se puede visualizar rápidamente dos aspectos fundamentales - 1) Los actores están en lados opuestos que por lo general significa cómo actúan sobre el sistema, uno realiza modificaciones internas (agente) mientras que los otros dos tienen actuaciones externas (vendedor, comprador). 2) Los tipos de compras que puede realizar el comprador están especificadas y no unificadas, esto se debe a que en cada una de las transacciones hay en específico una opción esencial, como por ejemplo si se llegase a vender una obra maestra, por los costos y valor intrínseco de la misma se deberán llenar unos formularios especiales que requieran de donde provienen los fondos de dicha compra, quien es la persona o razón social afectada en esta transacción, etc.

Anteriormente habíamos declarado que en el caso en particular de un Primer escenario debemos ya que no utilizamos los extend o include debemos identificar los datos y acciones relacionadas en el diagrama por tal motivo se desarrolla el siguiente modelo de especificaciones.

Actores:

Agente Galería, vendedor, comprador.

Interesados y Objetivos:

- Agente: quiere obtener una recomendación lo más acertada posible del precio máximo recomendado de manera rápida.
- Vendedor: quiere vender el cuadro a un precio razonable de manera rápida.
- Comprador: quiere comprar los cuadros al mejor precio posible.

Precondiciones:

El agente ha entrado en la aplicación para una negociación, pero eso significa que los precios de los cuadros se pueden modificar.

Garantía de éxito (postcondiciones):

Se registra la venta. Se requiere la identificación de las partes ya sea compra(agente) venta(vendedor) u venta(agente) compra(comprador).

Escenario Principal de Éxito:

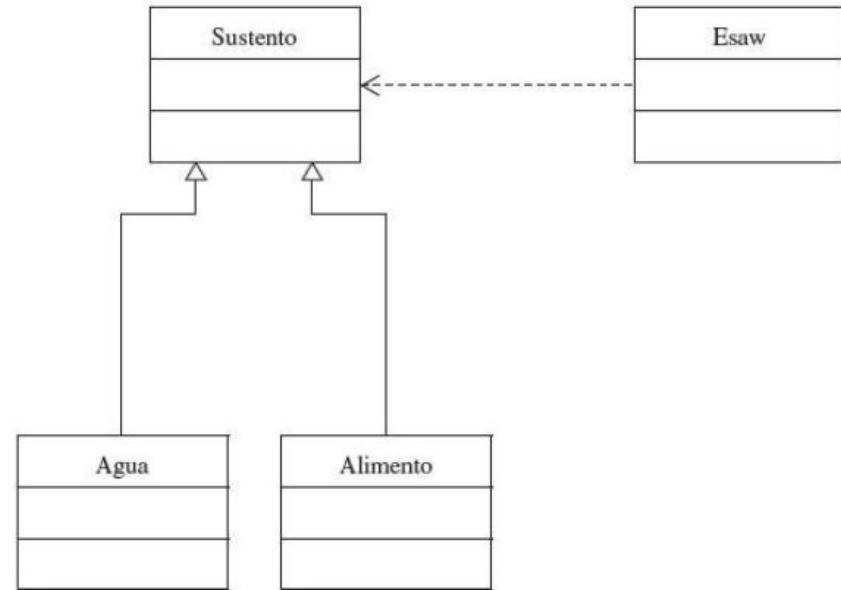
1. El agente introduce la descripción del cuadro para lo que se requiere identificación.
2. El sistema busca el cuadro más parecido del mismo autor.
3. El sistema presenta el precio recomendado, pero el agente es quien dispone del precio de la obra de arte.
4. El agente hace una propuesta por debajo del precio recomendado, y el vendedor acepta la oferta, ambos identificados.
5. El agente introduce información de la venta para lo cual deberá estar identificado.
6. El comprador, compra el cuadro teniendo en cuenta el precio del mismo puede funcionar en modo invitado, pero para comprar se debe identificar.

Diagrama de clase

Los diagramas de clases se usan para mostrar las clases de un sistema y las relaciones entre ellas.

Una sola clase puede mostrarse en más de un diagrama de clases y no es necesario mostrar todas las clases en un solo diagrama monolítico de clases. El mayor valor es mostrar las clases y sus relaciones desde varias perspectivas, de una manera que ayudará a transmitir la comprensión más útil.

Los diagramas de clases muestran una vista estática del sistema; no describen los comportamientos o cómo interactúan los ejemplos de las clases.



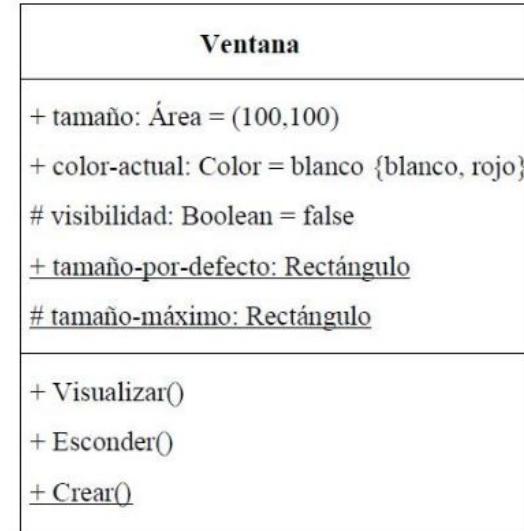
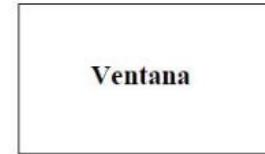
Un diagrama sencillo de clases, quizás uno de muchos, que transmite una faceta del sistema que se está diseñando.

Clases

Las clases describen un conjunto de objetos con características y comportamiento idénticos, es decir, objetos que comparten los mismos atributos, operaciones y relaciones.

Las clases se representan gráficamente por medio de un rectángulo con tres divisiones internas. Los tres compartimentos alojan el nombre de la clase, sus atributos y sus operaciones, respectivamente.

En muchos diagramas se omiten los dos compartimentos inferiores. Incluso cuando están presentes, no muestran todos los atributos y todas las operaciones. Por ejemplo, en la figura de la derecha viene representada la clase Ventana de tres formas diferentes: sin detalles, detalles en el ámbito de análisis y detalles en el ámbito de implementación.



CLASE: Ventana con y sin detalles.

Compartimento del nombre

Cada clase debe tener un nombre que la distinga de las otras clases. Dicho nombre puede ser un nombre simple (nombre de la clase) o un nombre compuesto (nombre del paquete que contiene a la clase: nombre de la clase).

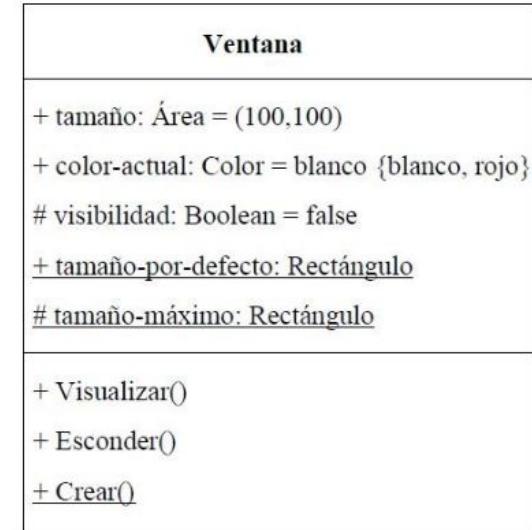
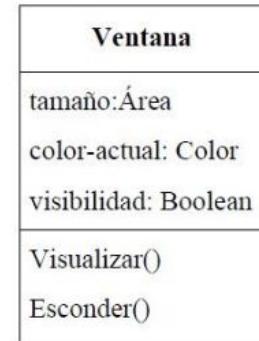
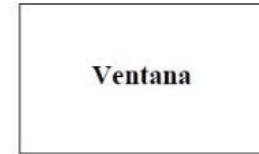
En este ejemplo, Ventana es un nombre simple de clase; pero si estuviese contenida en el paquete Gráficos, entonces el nombre compuesto sería Gráficos: Ventana. Otra forma de representar dicho nombre compuesto es escribir dentro de este compartimento primero el nombre del paquete (en este caso, «Gráficos») y debajo el nombre de la clase contenida en él.

Compartimento de la lista de atributos

Los atributos describen las características propias de los objetos de una clase.

La sintaxis completa de un atributo es:

[visibilidad] nombre [multiplicidad] [: tipo] [= valor-inicial]
[{lista-propiedades}]



Componentes de los atributos (definición):

visibilidad puede ser:

•+ (pública): que hace el atributo visible a todos los clientes de la clase.

•- (privada): que hace el atributo visible sólo para la clase.

(protégida): que hace el atributo visible a las subclases de la clase.

*Si un atributo no tiene asignada ninguna visibilidad, quiere decir que la visibilidad no está definida (no hay visibilidad por defecto).

Atributos

Nombre: es una cadena de texto que representa el nombre del atributo.

Tipo: es un tipo de atributo típico: string, boolean, integer, real, double, point, área y enumeration. Se llaman tipos primitivos. También pueden ser específicos de un cierto lenguaje de programación, aunque se puede usar cualquier tipo.

Multiplicidad: es un indicador de la multiplicidad del atributo, que va encerrado entre corchetes. La ausencia de multiplicidad significa que tiene exactamente valor 1, mientras que una multiplicidad de 0..1 proporciona la posibilidad de valores nulos (la ausencia de un valor).

Valor inicial: es una expresión con el valor inicial que va a contener el atributo cuando se crea de nuevo el objeto.

Lista propiedades: es una lista que contiene los valores permitidos en un atributo. Se utiliza para especificar tipos enumerados tales como color, estado, impresión, etc.

Nota: Los atributos tamaño-por-defecto y tamaño-máximo tienen por tipo la clase Rectángulo, mientras que el resto corresponde a tipos primitivos: Área, Color y Boolean. Asimismo, estos dos atributos, que aparecen subrayados, son atributos cuyo ámbito es la clase.

Operaciones

Ejemplos de operaciones: En el ejemplo de la Figura anterior la clase Ventana tiene las siguientes operaciones:

- + Visualizar()
- + Esconder()
- + Crear()

En estas operaciones no se ha especificado ninguno de los argumentos mencionados antes, excepto el de la visibilidad. La operación Crear es una operación cuyo ámbito es la clase, pues aparece subrayada.

Relaciones de los diagramas de clase

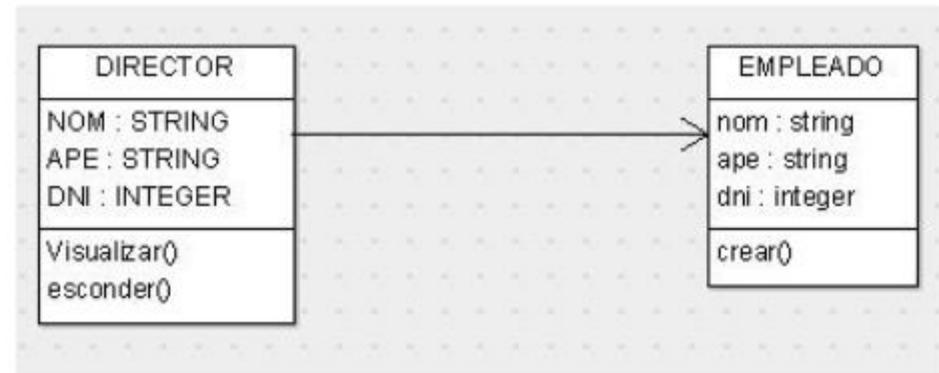
ASOCIACION:

Es una relación de estructura entre clases, es decir, una entidad se construye a partir de otra u otras. Aunque este tipo de relación es más fuerte que la Dependencia es más débil que la Agregación, ya que el tiempo de vida de un objeto no depende de otro.

Representación UML:

Se representa con una flecha continua que parte desde una clase y apunta a otra. El sentido de la flecha nos indica la clase que se compone (base de la flecha) y sus componentes (punta de la flecha).

El nombre de la asociación es opcional y se muestra como un texto que está próximo a la línea. Se puede añadir un pequeño triángulo negro sólido que indique la dirección en la cual leer el nombre de la asociación. En el ejemplo de la Figura se puede leer la asociación como "director manda sobre Empleado".



Relaciones de los diagramas de clase

MULTIPLICIDAD:

La multiplicidad es una restricción que se pone a una asociación, que limita el número de instancias de una clase que pueden tener esa asociación con una instancia de la otra clase. Indica cuántas instancias de una clase pueden surgir fruto de la relación, y se indican mediante:

1 --> sólo uno.

0..1 --> cero o uno.

n --> mediante un número entero se indica cuántas relaciones pueden haber.

n..m --> varios a varios.

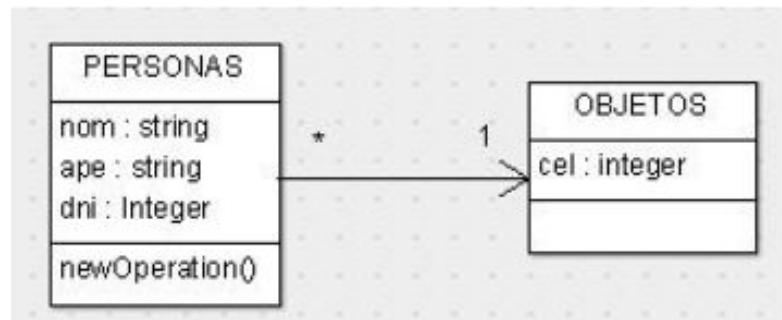
* --> cero o más.

0..* --> cero o más (lo mismo que el anterior).

1..* --> uno o más.

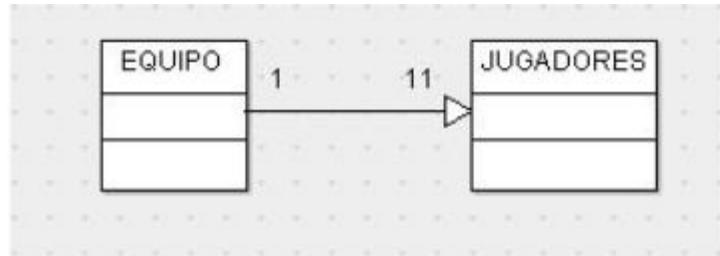


En este ejemplo se ve que el "DIRECTOR" maneja o tiene a su responsabilidad 0 o más empleados, en este caso por el lado de "EMPLEADO" vemos que hay 1 sola dependencia que es "DIRECTOR".



En este ejemplo se define que una instancia de "PERSONAS" puede estar relacionada sólo con 1 "OBJETOS", que a su vez puede estar relacionada con varias instancias de "PERSONAS".

En este tercer caso en particular, la instancia de "CLUB" puede estar relacionada con varias de "Personas", y cada una de ellas puede estar relacionada con varias de "CLUB".

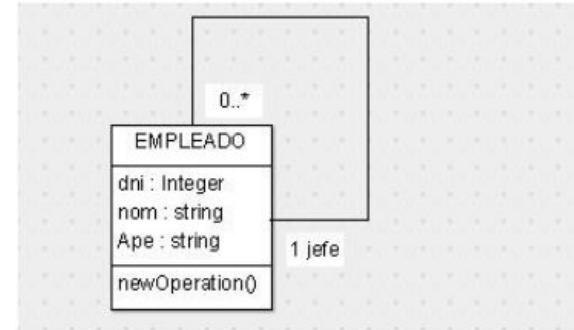


En el último caso, una instancia de "EQUIPO" puede estar relacionada con 11 instancias de "JUGADORES", y cada una de ellas a su vez con sólo una de "EQUIPO".

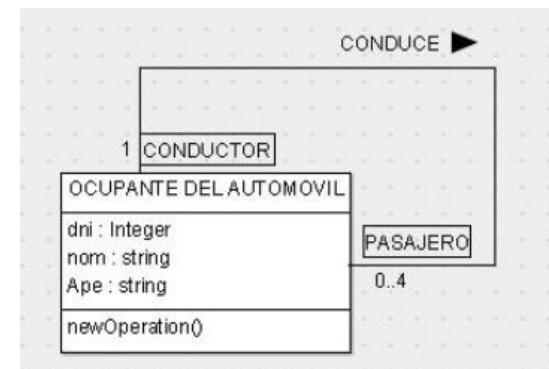
Relaciones de los diagramas de clase

ASOCIACION REFLEXIVA:

Por otro lado, si la asociación es entre objetos de la misma clase nos encontramos ante una asociación reflexiva. En la imagen se muestra que un empleado jefe está relacionado con ninguno o varios empleados, y que un empleado tiene sólo un jefe:



Imaginemos entonces por un momento un Automóvil que puede tener varios ocupantes, uno puede ser el "CONDUCTOR" y otro "PASAJERO". En el papel del conductor, el OCUPANTE_DE_AUTOMOVIL puede llevar ninguno o más clases del mismo objeto, por lo tanto representaremos con una asociación reflexiva como se asocian dichos roles u objetos en el mismo cuadrante.



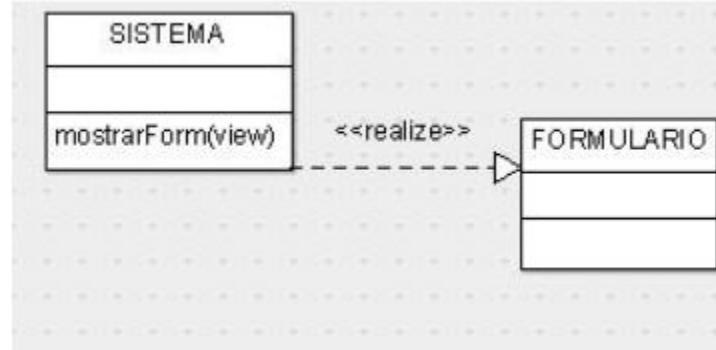
Relaciones de los diagramas de clase

DEPENDENCIA:

Es otro tipo de relación, una clase utiliza o requiere de otra, el uso más común de una dependencia es mostrar que la firma de la operación de una clase utiliza a otra clase.

Supongamos que un sistema tiene 1 o varios formularios corporativos para ser visualizados por pantalla y ser completados. En este tipo de diseños, tendremos una clase "SISTEMA" y una clase "FORMULARIO". Entre algunas de sus muchas operaciones, la clase "SISTEMA" va a permitir mostrarform. El "FORMULARIO" que el sistema será capaz de desplegar, dependerá directamente del tipo que seleccione el usuario.

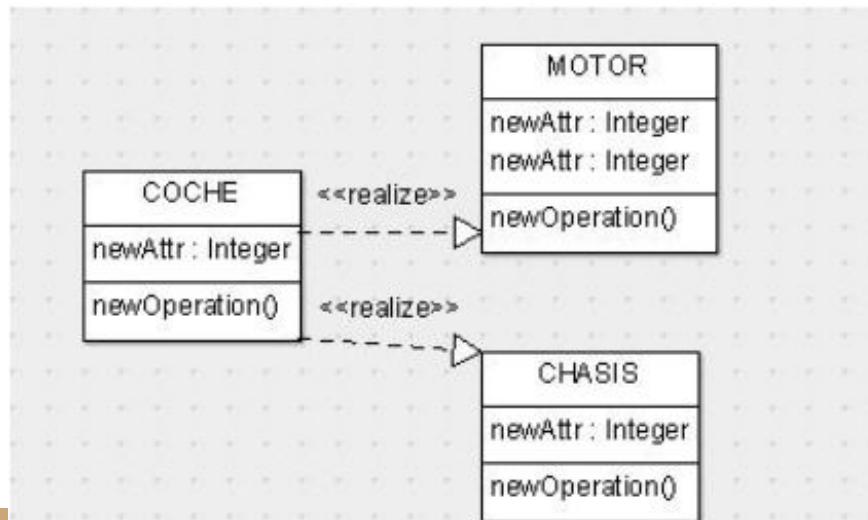
La notación que utiliza UML para este tipo de elementos es una línea discontinua con una punta de flecha en forma de triángulo sin relleno que apunta a la clase de la que depende, como se muestra en la figura de acá a la derecha.



En la Dependencia entre dos clases uno de los elementos es dependiente ("COCHE" según la imagen de ejemplo) del otro, viéndose afectado por los cambios que se produzcan en él y no pudiendo funcionar correctamente si le falta.

Ello no implica que el elemento dependiente deba obligatoriamente contener al otro como uno de sus atributos (puede que lo necesite como parámetro o que lo utilice en un método, por ejemplo).

En lugar de la Dependencia es recomendable definir la relación como Asociación, a menos que sea importante representarla como tal debido a que un cambio en un elemento del que depende(n) otro(s) pueda afectar al funcionamiento del elemento dependiente.



Relaciones de los diagramas de clase

GENERALIZACIÓN O HERENCIA:

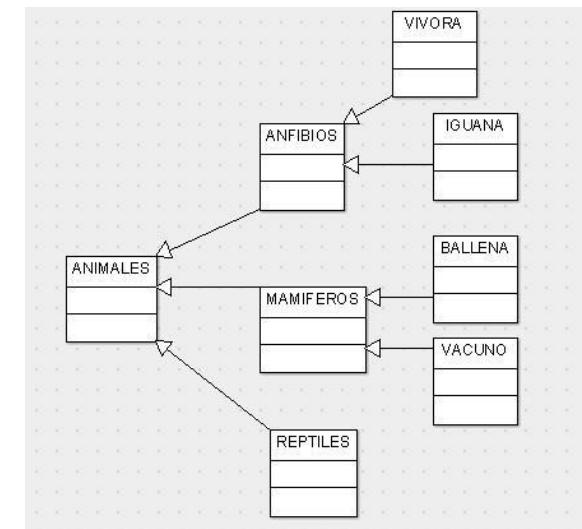
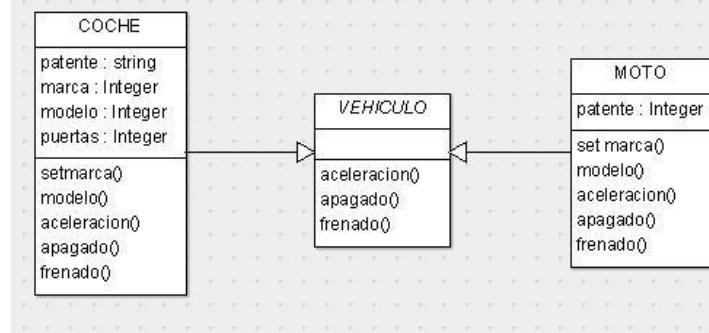
En cuanto a la Generalización, indica una relación de herencia entre dos clases u objetos (el elemento hijo comparte la misma estructura que el elemento madre).

Como se muestra en esta imagen "VEHICULO" hereda sus propiedades aceleración, apagado, frenado a sus dos clases hijos "COCHE" y "MOTO" ambos son vehículos con sus propiedades pero que están ampliamente vinculadas con las de la clase madre.

En el sentido amplio de la programación tener el conocimiento sobre un objeto cuya categoría de cosas, automáticamente sabrá qué elementos podrán ser transferidos a otra clase.

La jerarquía de la herencia no tiene que finalizar en dos niveles, es decir que una clase secundaria puede nacer de otra clase secundaria.

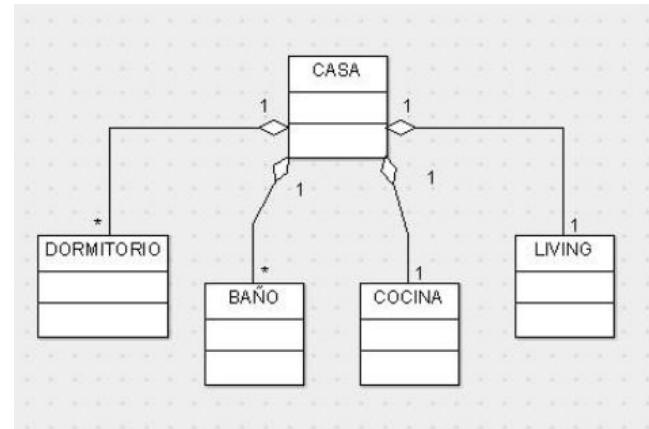
Imaginemos el caso de la clase "ANIMALES" donde sus clases secundarias serían "ANFIBIOS" "MAMIFEROS" "REPTILES" luego de esta clase secundaria se genera una nueva clase dependiente de "REPTILES" llamada "VIVORA""IGUANA" y de la clase "MAMIFEROS" se desprenden "BALLENA" "VACUNO" etc.



Relaciones de los diagramas de clase

AGREGACIÓN:

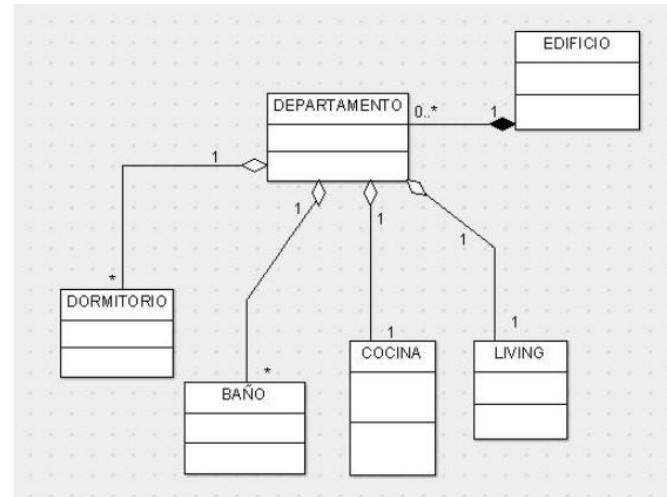
En ocasiones una clase consta de otras clases. Este es un tipo especial de relación conocida como agregación. Los componentes y la clase que constituyen son una asociación que conforma un bloque completo. En la Agregación el conjunto de elementos relacionados forma un "TODO", aunque los elementos relacionados podrían existir y "funcionar" independientemente de él. Se representan mediante una línea con un rombo sin relleno en un extremo.



Relaciones de los diagramas de clase

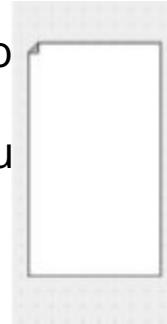
COMPOSICIÓN:

Una composición es un tipo muy representativo de una agregación. Cada componente dentro de una composición puede pertenecer tan solo a un todo. Por ejemplo, pensemos en una mesa de café, tanto la superficie de la tabla como las patas establecen una composición, no existe una mesa sin esos componentes juntos conformando en unidad un TODO, los elementos relacionados no podrían existir y "funcionar" independientemente si no formasen parte de dicho "TODO". El símbolo de una composición es el mismo que el de una agregación, excepto que el rombo está relleno es decir es de color negro.



Notas

Muchas veces vamos a requerir agregar algún texto con información a nuestros diagramas de clases y objetos, insertaremos el mismo en un rectángulo con una de sus esquinas doblada:



Las notas pueden estar en cualquier parte del diagrama sin mantener relación con ningún elemento en particular, aunque se utilizará una línea discontinua para especificar a qué elemento hacen referencia:

