El desarrollo de software y la gestión de proyectos están íntimamente relacionados y se complementan mutuamente. El desarrollo de software implica la creación del producto, mientras que la gestión de proyectos implica la planificación, ejecución, control y entrega exitosa del proyecto. Ambos se coordinan para garantizar que el software se desarrolle de manera ordenada y eficiente, se mantenga dentro de los límites de tiempo, costo y calidad, y se adapte a los cambios en los requisitos y prioridades. La comunicación efectiva y la colaboración entre los equipos de desarrollo y gestión son fundamentales para el éxito del proyecto y la entrega de productos de software que cumplan con las expectativas del cliente y del mercado.

La gestión de proyecto se apoya en Prácticas ágiles (Agile Software Development Manifiesto – 2001)

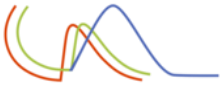
Mientras que las metodologías para el desarrollo de software son enfoques estructurados y sistemáticos que guían el proceso de creación, desarrollo y mantenimiento de software. Estas metodologías proporcionan un conjunto de prácticas, principios y procedimientos que ayudan a los equipos de desarrollo a organizar su trabajo, gestionar el tiempo y los recursos, y asegurar la calidad del software final.

Entre ellas se encuentran:

1. **DDD (Domine Driven Design)**: Se centra en modelar el dominio del negocio y asegurarse de que el software refleja correctamente los conceptos y procesos del negocio.
2. **BDD (Behavior Driven Development)**: Se centra en especificar y verificar el comportamiento del software desde la perspectiva del usuario, utilizando ejemplos concretos y pruebas automatizadas.
3. **TDD** (**Test Driven Development)** es una práctica de desarrollo de software donde escribir pruebas automáticas antes de escribir el código de producción. Luego, se escribe el código de producción para que pase las pruebas.
4. **XP (Extreme Programming)** es una metodología ágil de desarrollo de software que enfatiza la entrega temprana y continua de software funcional, la comunicación constante entre el equipo y el cliente, y la adaptación a los cambios en los requisitos.

Los principios generales de programación son un conjunto de reglas y pautas que guían la escritura de código de alta calidad y mantenible. Estos principios están diseñados para mejorar la legibilidad, la eficiencia y la escalabilidad del código, así como para reducir errores y facilitar su mantenimiento a lo largo del tiempo. Algunos de los principios más comunes incluyen:

1. Principios SOLID
    a. Single Responsability
    b. Open/Closed abiertos para su extensión cerrados para su modificación
    c. Liskov substitution (reemplazables por instancias de sus subtipos)
    d. Interface segregation
    e. Dependency inversión (dependen de abstracciones, no de implementaciones).
2. Don´t Repeat Yourself (DRY)



3. Inversion of Control (IoC)
4. Your Aren´t Gona Need It (YAGNI)
5. Keep It Simple, Stupid (KISS)
6. Ley de Demeter (LoD)

Un patrón de diseño es una solución a un problema de diseño cuya efectividad ha sido comprobada por haber sido empleada para resolver problemas similares en ocasiones anteriores.

Los patrones de diseño, según The Gang of Four (GOF), describen soluciones simples y elegantes a problemas específicos en el diseño de software orientado a objetos.

Según el GOF, los patrones de diseño se agrupan en:

- Patrones de Creación
- Patrones Estructurales
- Patrones de Comportamiento

Patrones de Creación

Patrones de CREACIÓN (o creacionales): definen cómo puede crearse un objeto aislando los detalles de la creación del objeto, de forma que su código no dependa de los tipos de objeto que hay y, por lo tanto, no deba ser modificado al añadir un nuevo tipo de objeto.

Patrones:

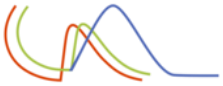
1. Abstract Factory (Factoría abstracta).
2. Builder (Constructor).
3. Factory Method (Factoría).
4. Prototype (Prototipo).
5. Singleton (Instancia única).
6. Model View Controller (Modelo Vista Controlador).

Patrones Estructurales

Patrones ESTRUCTURALES: tratan la manera en que los objetos se conectan con otros objetos, los combinan y forman estructuras mayores, asegurando estabilidad en las conexiones ya que los cambios del sistema no requieren cambiar esas conexiones.

Patrones:

1. Adapter o Wrapper (Adaptador o Envoltorio).
2. Bridge (Puente).
3. Composite (Objeto compuesto).
4. Decorator (Decorador).
5. Facade (Fachada).
6. Flyweight (Peso ligero).
7. Proxy.



Patrones de Comportamiento

Patrones de COMPORTAMIENTO: tratan a los objetos que manejan tipos particulares de acciones y llamadas entre los diferentes objetos dentro de un programa. Éstos encapsulan procesos debe ejecutarse dentro de la funcionalidad de la aplicación, como interpretar un lenguaje, completar una petición, moverse a través de una secuencia o implementar un algoritmo. Patrones:

1. Chain of Responsability (Cadena de responsabilidades).
2. Command (Comando).
3. Interpreter (Intérprete).
4. Iterator (Iterador).
5. Mediator (Mediador).
6. Memento (Recuerdo).
7. Observer (Observador).
8. State (Estado).
9. Strategy (Estrategia).
10. Template Method (Método plantilla).
11. Visitor (Visitante).

Un framework se puede explicar como una estructura de software predefinida y reutilizable que proporciona un conjunto de herramientas, bibliotecas y patrones de diseño para facilitar el desarrollo de aplicaciones. Los frameworks están diseñados para ayudar a los desarrolladores a evitar tener que escribir código repetitivo o reinventar la rueda cada vez que crean una nueva aplicación.

Las empresas dedicadas al desarrollo de software utilizan los frameworks ya que utilizan una serie de beneficios que hacen que valga la pena utilizarlos.

Beneficios:

- **Productividad mejorada**: Los frameworks proporcionan funcionalidades listas para usar que permiten a los desarrolladores construir aplicaciones de manera más rápida y eficiente. Al utilizar componentes y patrones de diseño probados, los desarrolladores pueden concentrarse en resolver problemas específicos de la aplicación en lugar de reinventar soluciones genéricas.
- **Consistencia y estandarización**: Los frameworks establecen convenciones y mejores prácticas que ayudan a mantener la consistencia en el código y facilitan la colaboración entre desarrolladores. Esto es especialmente importante en equipos de desarrollo grandes donde múltiples personas trabajan en el mismo proyecto.
- **Abstracción de complejidad**: Los frameworks ocultan la complejidad técnica detrás de una interfaz simple y coherente. Esto permite a los desarrolladores centrarse en la lógica de la aplicación en lugar de preocuparse por detalles de implementación complejos.
- **Mantenimiento simplificado**: Los frameworks suelen ofrecer actualizaciones y correcciones de errores periódicas, lo que facilita el mantenimiento a largo plazo de las aplicaciones. Además, al seguir las convenciones del framework, las aplicaciones son más fáciles de entender y mantener por otros desarrolladores en el futuro.
- **Comunidad y soporte**: Los frameworks suelen tener una comunidad activa de desarrolladores que comparten conocimientos, resuelven problemas y contribuyen con mejoras al framework. Esto proporciona a los desarrolladores acceso a una amplia gama de recursos, como documentación, tutoriales y foros de discusión, que pueden ayudar en el aprendizaje y la resolución de problemas.

En resumen, los frameworks son herramientas poderosas que pueden acelerar el proceso de desarrollo de software, mejorar la calidad del código y facilitar el mantenimiento a largo plazo de las aplicaciones. Estudiar los frameworks es fundamental para cualquier desarrollador, ya que les permite aprovechar al máximo estas herramientas y construir aplicaciones robustas y escalables de manera más eficiente.

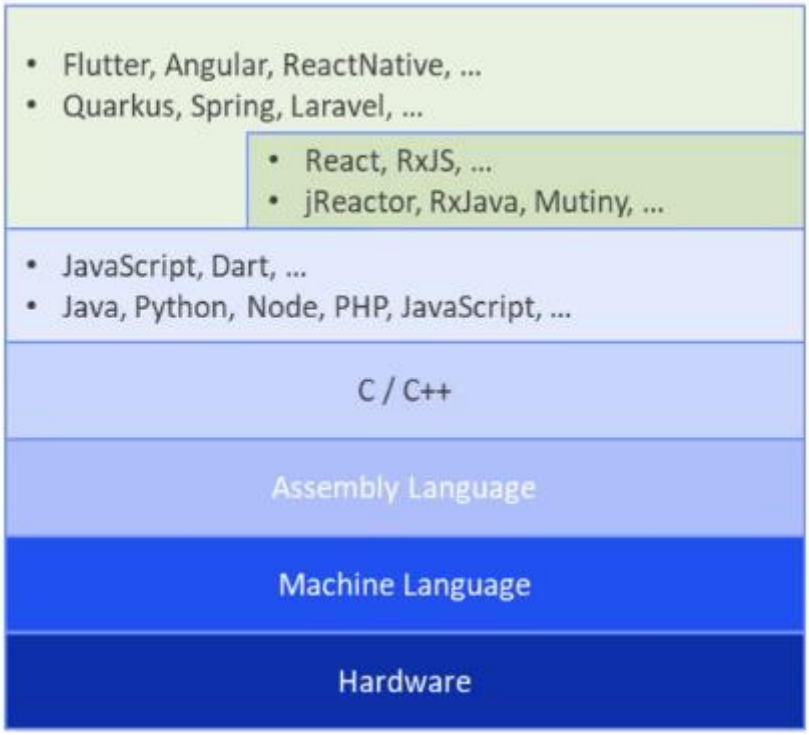


Ilustración 1- Capas de Software en un Framework

Antes de las metodologías ágiles, el ciclo de desarrollo de software seguía generalmente un enfoque secuencial conocido como Modelo en Cascada. Este modelo consta de varias fases bien definidas:

1. **Requisitos**: Recopilación y documentación exhaustiva de los requisitos del cliente.
2. **Diseño**: Creación de la arquitectura del sistema y el diseño detallado del software.
3. **Implementación**: Codificación y desarrollo del software basado en el diseño.
4. **Pruebas**: Verificación y validación del software para asegurar que cumple con los requisitos.
5. **Despliegue**: Instalación y entrega del software al cliente o al entorno de producción.
6. **Mantenimiento**: Resolución de problemas, corrección de errores y actualización del software según sea necesario.

Las características de este modelo son un enfoque lineal y secuencial en el que cada fase debe completarse antes de pasar a la siguiente. Por lo tanto, los cambios son difíciles de manejar una vez que el proyecto está en fases avanzadas, y se requiere una documentación extensa en cada etapa.

A partir del año 2011, con las metodologías ágiles, el ciclo de desarrollo de software cambió de ser iterativo a incremental, adaptándose a los cambios y mejorando continuamente a lo largo del proceso. Algunas de las metodologías ágiles más conocidas son Scrum, Kanban. Las fases en estas metodologías son las siguientes:

Iteraciones (Sprints): El trabajo se divide en ciclos cortos, llamados iteraciones o sprints (generalmente de 2-4 semanas).

Requisitos y Planificación: Los requisitos se priorizan y se planifican en la lista de trabajo (backlog). Cada iteración comienza con una reunión de planificación para determinar qué se puede lograr en ese sprint.

Desarrollo: Los desarrolladores trabajan en pequeñas partes del proyecto, entregando incrementos funcionales de software al final de cada iteración.

Revisión y Retroalimentación: Al final de cada iteración, se revisa el trabajo completado con el equipo y los interesados. Se recolecta retroalimentación para ajustar el trabajo futuro.

Pruebas Continuas: Las pruebas se realizan de manera continua a lo largo de cada iteración para asegurar que cada incremento funcione correctamente.

Entrega Continua: El software se entrega de manera frecuente y regular, lo que permite recibir retroalimentación constante y realizar ajustes necesarios.

Sprint Planning (Reunión de Planificación de Sprint):

Frecuencia: Al inicio de cada Sprint.

Propósito: El equipo Scrum y el Product Owner colaboran para seleccionar elementos del Backlog del Producto y planificar el trabajo que se llevará a cabo durante el Sprint.

Daily Scrum (Scrum Diario):

Frecuencia: Diariamente, durante el Sprint.

Propósito: El equipo se reúne brevemente para compartir actualizaciones rápidas sobre el progreso del trabajo, identificar posibles obstáculos y ajustar el plan para el día siguiente.

Sprint Review (Revisión de Sprint):

Frecuencia: Al final de cada Sprint.

Propósito: Se revisa el trabajo completado durante el Sprint y se recopila la retroalimentación del cliente. Se adapta el Backlog del Producto según sea necesario.

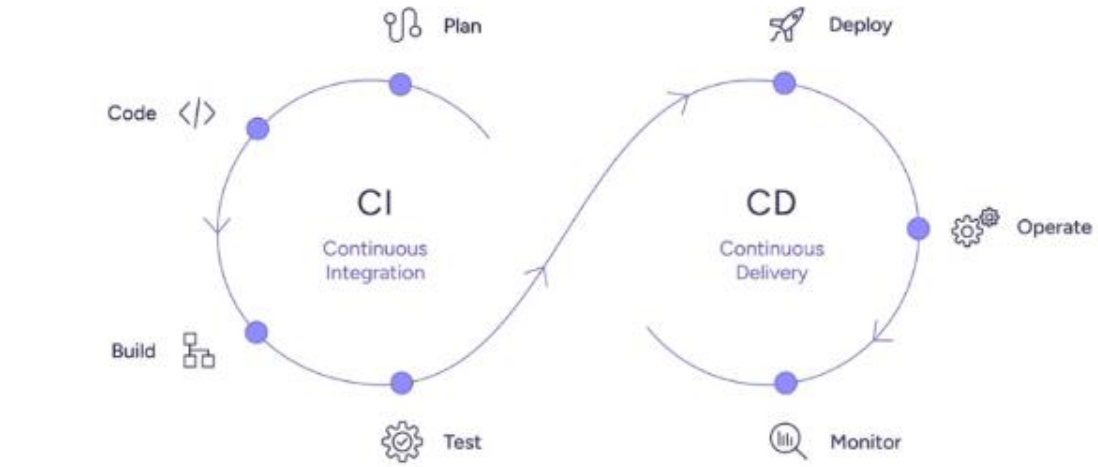
Sprint Retrospective (Retrospectiva de Sprint):

Frecuencia: Al final de cada Sprint, después de la Sprint Review.

Propósito: El equipo reflexiona sobre el Sprint y busca maneras de mejorar el proceso de trabajo para el próximo Sprint.

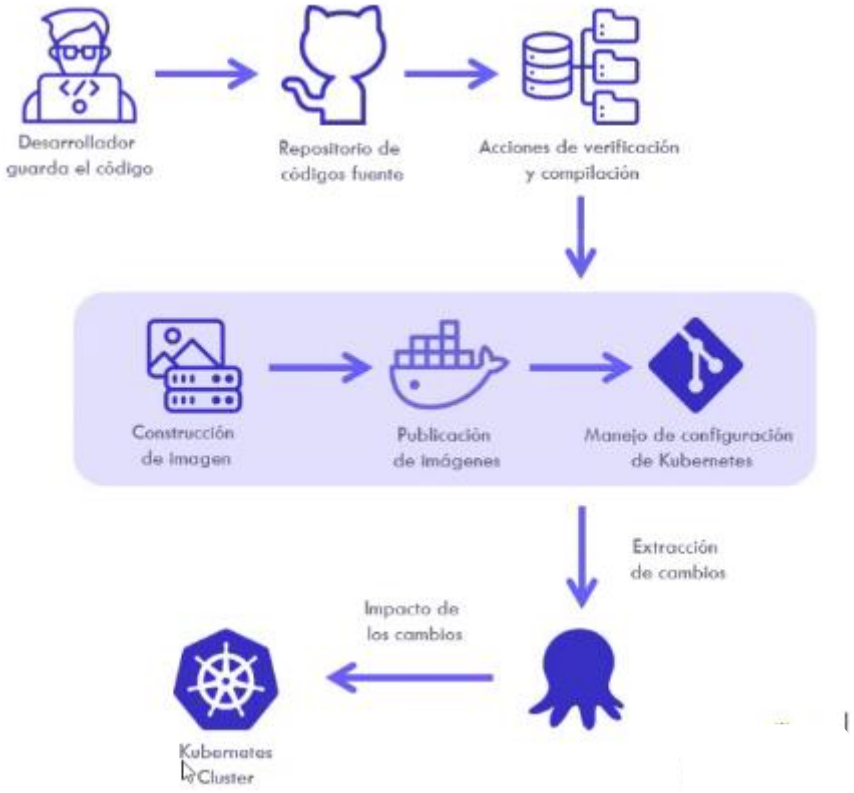
El enfoque ágil ofrece varias ventajas significativas para el desarrollo de software. En primer lugar, proporciona flexibilidad y capacidad de adaptación rápida a los cambios en los requisitos del proyecto. Además, se centra en la entrega continua de valor al cliente, lo que permite obtener retroalimentación temprana y ajustar el producto en consecuencia. A través de la retroalimentación constante, se promueve una mejora continua en el proceso y en el producto final. Además, fomenta una mayor colaboración entre el equipo de desarrollo y los interesados, lo que contribuye a una comprensión más clara de las necesidades y expectativas del cliente.

En resumen, las metodologías ágiles han revolucionado el ciclo de desarrollo de software al hacer el proceso más flexible, colaborativo y centrado en el cliente.



El surgimiento de DevOps se presenta como una evolución natural en el panorama del desarrollo de software ágil. DevOps surge como respuesta a la necesidad de integrar aún más las operaciones con el ciclo de desarrollo, creando un enfoque holístico que abarca desde la planificación y codificación hasta la entrega y operación del software. Esta filosofía se basa en la colaboración estrecha entre los equipos de desarrollo y operaciones, promoviendo la automatización, la comunicación y la

responsabilidad compartida en todo el ciclo de vida del software. La implementación exitosa de DevOps permite una entrega más rápida y confiable de software, una mayor calidad del producto y una mejor capacidad de respuesta a las demandas del mercado y del cliente. En resumen, DevOps se integra de manera natural en el entorno ágil, proporcionando una extensión crucial para maximizar la eficiencia y la efectividad en el desarrollo y operación de software.



Durante la fase de diseño, los arquitectos de software y los diseñadores elaboran una estructura y planificación general para el sistema, definiendo cómo se organizarán los componentes del software, cómo interactuarán entre sí y cómo se cumplirán los requisitos funcionales y no funcionales. La arquitectura del proyecto es esencial para proporcionar una base sólida para el desarrollo posterior, asegurando que el software sea robusto, escalable, mantenible y cumpla con las necesidades del cliente.

Sistemas monolíticos y cliente-servidor

Arquitectura orientada a servicios

Arquitectura de microservicios

Para interconectar los componentes se hace uso de los patrones de arquitectura y algunos ejemplos de ello son:

1. **Command Query Responsibility Segregation** (CQRS): Este patrón propone separar la lógica de lectura (queries) de la lógica de escritura (commands) en una aplicación. Esto permite optimizar cada aspecto por separado y simplificar el modelo de dominio.
2. **Transactional Outbox**: Es un patrón que se utiliza para garantizar la consistencia entre operaciones en una arquitectura distribuida. Consiste en escribir los eventos de dominio en un registro externo (outbox) de la transacción de base de datos, lo que garantiza que los eventos se envíen incluso si la transacción principal falla.
3. **Circuit** Breaker: Inspirado en el concepto eléctrico, este patrón se utiliza para mejorar la resiliencia de un sistema frente a fallas. Monitorea el estado de un servicio y, si detecta una falla continua, interrumpe temporalmente las solicitudes al servicio afectado, evitando así que se propaguen los fallos.
4. **Strangler Fig**: Este patrón se utiliza para migrar gradualmente un sistema monolítico a una arquitectura más modular y distribuida. Consiste en añadir nuevas funcionalidades y componentes como "enredaderas" alrededor del sistema existente, gradualmente estrangulando el monolito original.
5. **Sidecar**: Es un patrón de diseño de microservicios donde un contenedor adicional (sidecar) se adjunta a un servicio principal para proporcionar funcionalidades adicionales, como la gestión de la configuración, la monitorización o la seguridad, sin modificar el servicio principal.
6. **Saga**: Es un patrón de diseño utilizado en sistemas distribuidos para mantener la consistencia entre varias transacciones. Consiste en descomponer una operación que implica múltiples pasos en una serie de pasos más pequeños y atómicos, donde cada paso se puede deshacer o revertir si algo sale mal.

El artículo "The Practical Test Pyramid" de Martin Fowler discute la importancia de la Pirámide de Pruebas como una guía para estructurar las pruebas de software de manera eficiente y eficaz. La Pirámide de Pruebas es un modelo que sugiere tener más pruebas automatizadas en la base (pruebas unitarias) y menos pruebas en la cima (pruebas de interfaz de usuario).

1. **Pruebas Unitarias** (Base de la Pirámide): Estas pruebas son rápidas y verifican pequeños fragmentos de código de forma aislada. Son esenciales para detectar errores básicos y mantener la calidad del código.

2. **Pruebas de Servicio o Integración** (Centro de la Pirámide): Estas pruebas se centran en la interacción entre diferentes partes del sistema, asegurando que los módulos funcionan bien juntos. Aunque son más lentas que las pruebas unitarias, son cruciales para identificar problemas de integración.

3. **Pruebas de Interfaz de Usuario** (Cima de la Pirámide): Estas pruebas son las más costosas y lentas. Verifican que la aplicación funcione correctamente desde la perspectiva del usuario final. Dado su costo y complejidad, se recomienda tener menos de estas pruebas en comparación con las otras capas.

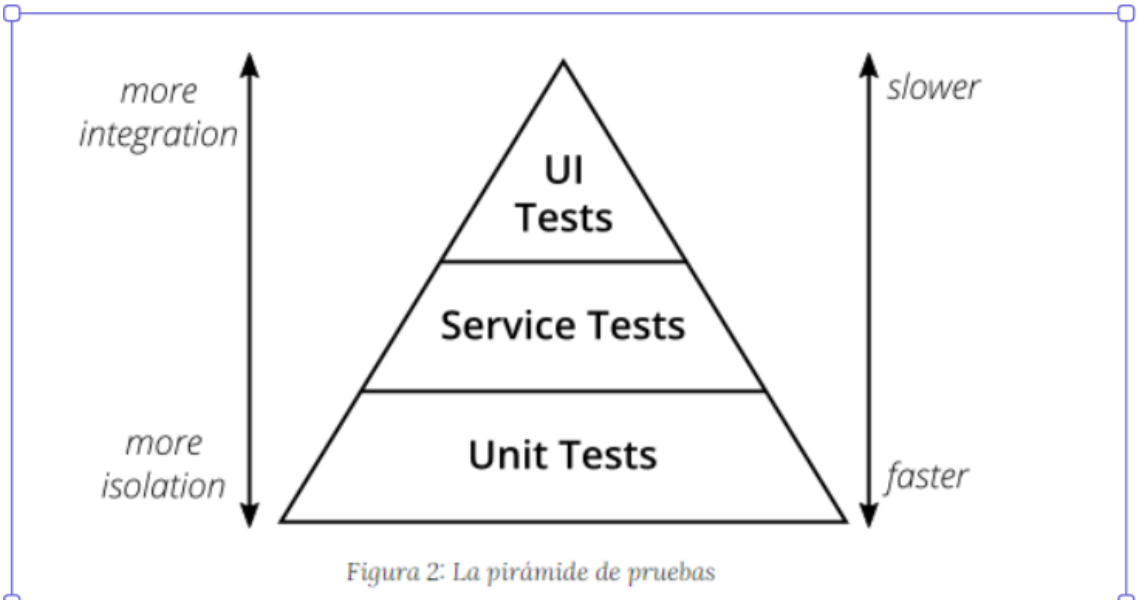
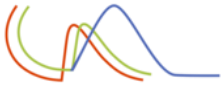


Figura 2: La pirámide de pruebas



Las métricas DORA, desarrolladas por el programa DevOps Research and Assessment, son una métrica para evaluar y mejorar el rendimiento en entornos DevOps. Estas métricas incluyen:

1. **Tiempo de entrega de cambios:** Tiempo promedio desde el commit hasta la implementación en producción.
2. **Frecuencia de despliegue**: Frecuencia con la que se realizan despliegues en producción.
3. **Tiempo de recuperación** de incidentes: Tiempo promedio para restaurar el servicio después de una falla en producción.
4. **Tasa de fallos en cambios**: Porcentaje de despliegues que causan una interrupción o requieren una corrección posterior.