**1. Proceso de desarrollo de software utilizando metodologías tradicionales y ágiles**

**Metodologías Tradicionales**

* **Cascada (Waterfall):**
* El desarrollo se realiza en etapas secuenciales como análisis, diseño, implementación, pruebas, y mantenimiento.
* Cada etapa debe completarse antes de avanzar a la siguiente.
* Los requisitos se definen al inicio y son estáticos.

Beneficios: Claridad en los hitos del proyecto y tiempos predecibles

Desventajas: Falta de flexibilidad ante cambios durante el proyecto.

**Metodologías Ágiles**

* **Agile:**
* El desarrollo es iterativo e incremental, dividiéndose en ciclos cortos llamados "sprints".
* Los equipos colaboran estrechamente y los requisitos evolucionan a lo largo del proyecto.
* Se promueve la entrega continua de software funcional.

Beneficios: Flexibilidad para adaptarse a cambios, retroalimentación continua, y mejora continua.

Desventajas: Puede ser difícil gestionar en proyectos grandes y equipos distribuidos sin buena coordinación.

**2. Rol del analista de calidad de software en un entorno ágil**

El analista de calidad (QA) en un entorno ágil desempeña un rol activo en todas las etapas del desarrollo:

* **Colaboración desde el inicio:** Participa en la planificación de las historias de usuario y la definición de criterios de aceptación.
* **Pruebas continuas:** Ejecuta pruebas manuales y automáticas durante cada sprint, enfocándose en la entrega continua de valor.
* **Prevención y detección temprana de errores:** Trabaja estrechamente con los desarrolladores para identificar y prevenir defectos a lo largo del desarrollo.
* **Automatización de pruebas:** Aumenta la eficiencia implementando pruebas automatizadas para reducir el tiempo de ejecución manual y garantizar la repetibilidad de las pruebas.

**3. Automatización y Programación Orientada a Objetos (POO)**

**A. Pilares de la Programación Orientada a Objetos**

1. **Encapsulamiento:** Agrupar datos y métodos que operan sobre esos datos en una sola entidad (clase).
2. **Abstracción:** Ocultar detalles complejos y mostrar solo las funcionalidades esenciales.
3. **Herencia:** Permitir que las clases hereden propiedades y métodos de otras clases, reutilizando código.
4. **Polimorfismo:** La capacidad de un objeto de tomar diferentes formas, generalmente mediante métodos sobrescritos o sobrecargados.

**B. Consideraciones al automatizar pruebas**

* **Estabilidad del sistema:** Asegurarse de que el sistema bajo prueba esté suficientemente maduro.
* **Mantenimiento del script:** Los scripts de prueba deben ser fáciles de mantener y adaptarse a cambios en la aplicación.
* **Tiempo de ejecución:** Las pruebas automatizadas deben ejecutarse en tiempos razonables.
* **Selección de herramientas:** Usar herramientas adecuadas según el tipo de prueba (UI, API, base de datos, etc.).

**C. Principios SOLID en la automatización de pruebas**

* **Single Responsibility Principle (SRP):** Cada clase de prueba debe tener una única responsabilidad para facilitar su mantenimiento.
* **Open/Closed Principle (OCP):** Las pruebas deben ser abiertas para extensión, pero cerradas para modificación. Nuevas pruebas deben añadirse sin cambiar las existentes.
* **Liskov Substitution Principle (LSP):** Los objetos derivados en las pruebas deben ser reemplazables por objetos de su clase base sin afectar la funcionalidad.
* **Interface Segregation Principle (ISP):** Evitar interfaces grandes y dividirlas en interfaces más específicas que los clientes usen.
* **Dependency Inversion Principle (DIP):** Las pruebas deben depender de abstracciones, no de implementaciones concretas, lo que facilita el mockeo de dependencias.

**D. Pirámide de automatización de Cohn**

* **Base (Pruebas Unitarias):** Representa la mayor cantidad de pruebas, son rápidas, aisladas, y cubren el código a nivel de método o función.
* **Nivel Intermedio (Pruebas de Servicio/API):** Pruebas de integración entre servicios o componentes. Son más costosas pero validan la comunicación entre sistemas.
* **Punta (Pruebas End-to-End):** Pruebas que simulan escenarios de usuario completo. Son más lentas y costosas, por lo que deben minimizarse.

**E. Uso de SonarQube para mejorar la calidad del código**

SonarQube es una herramienta que analiza el código y ofrece métricas clave:

* **Cobertura de pruebas:** Mide qué porcentaje del código está cubierto por pruebas unitarias. Mejora la robustez y reduce el riesgo de errores no detectados.
* **Duplicación de código:** Identifica fragmentos de código repetidos. La duplicación puede generar mantenimiento costoso, por lo que reducirla mejora la mantenibilidad.
* **Code Smells:** Señala malas prácticas de programación. Reducir los "code smells" mejora la legibilidad y facilita la evolución del software.

**4. Pruebas No Funcionales**

**A. Pruebas de Seguridad y Performance**

* **Pruebas de Seguridad:** Evalúan si el sistema es vulnerable a ataques y si los datos están protegidos adecuadamente. Implican pruebas como la validación de controles de acceso, gestión de sesiones, y prevención de inyección SQL.
* **Pruebas de Performance:** Evalúan cómo responde el sistema bajo diferentes niveles de carga, estrés y volumen. Esto incluye medir el tiempo de respuesta, la capacidad de manejar múltiples usuarios, y la estabilidad bajo condiciones extremas.

**B. Herramientas utilizadas para ejecutar estas pruebas**

* **Seguridad:**
* **OWASP ZAP:** Un proxy para encontrar vulnerabilidades en aplicaciones web.
* **Burp Suite:** Herramienta de análisis de seguridad web.
* **Performance:**
* **JMeter:** Simula cargas y mide el rendimiento de aplicaciones web.
* **Gatling:** Herramienta para pruebas de carga y rendimiento de aplicaciones.

**C. Función de las pruebas no funcionales en la calidad del software**

Las pruebas no funcionales aseguran que el sistema no solo funcione correctamente desde el punto de vista funcional, sino que también cumpla con requisitos de rendimiento, seguridad, y usabilidad. Aportan confianza en la estabilidad, capacidad de respuesta y resistencia del sistema bajo diferentes condiciones.