INTRODUCCIÓN BLOQUE IV

Este es el cuarto y último volumen de un total de cuatro libros que forman el temario actualizado de Oposiciones de Secundaria de Informática. Su objetivo fundamental es el de facilitar al opositor la preparación de la prueba escrita.

El contenido de este libro está desarrollado basándose en la legislación actual que regula el contenido de estas pruebas.

Este volumen contiene desde el tema 56 hasta el 74. En estos temas se tratan la gestión de proyectos de ingeniería del software, redes y sistemas multimedia ofreciendo un contenido totalmente actualizado.

Cada uno de los temas consta de un índice que presenta el esquema general del tema, la introducción, el desarrollo del tema en cuestión, una conclusión y bibliografía/webgrafía.

En la prueba escrita es recomendable que se introduzca el punto de bibliografía/webgrafía al final del tema. En este libro se presenta la bibliografía agrupada por bloques con el fin de facilitar al opositor la tarea de recordarla. La bibliografía está situada al final de este volumen.

Los temas se presentan de forma acotada para que el opositor sea capaz de desarrollarlo en el tiempo estipulado, asegurando que se tratan todos los puntos de interés con la profundidad adecuada.

Los temas pertenecientes al mismo bloque tienen contenidos en común, lo que permitirá al opositor rentabilizar tiempo de estudio y poder amortizar píldoras de conocimiento aplicables a distintos temas.

Además, este volumen viene acompañado de material adicional en el que el lector puede encontrar trucos sobre cómo afrontar el examen, ejemplos para añadir a los temas, contextualización en los ciclos formativos y otros recursos de interés.

TEMA 56

ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

Tema 56

ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

56.1 INTRODUCCIÓN

En los inicios de la ingeniería del software el paradigma de programación que triunfó fue la programación estructurada. Este paradigma se basa en la definición del comportamiento del programa utilizando recursos como las estructuras condicionales o las iteraciones. Este modelo de programación está basado en el comportamiento, siendo los datos meros instrumentos con los que trabajar.

Más adelante, con el objetivo de aumentar la reutilización de código apareció la programación modular. Este paradigma consiste en agrupar el código en funciones o procedimientos que pueden reutilizarse una y otra vez. De esta manera se conseguía eliminar código duplicado y aumentar la productividad en el desarrollo de software.

El siguiente paradigma de programación que surgió fue la programación orientada a objetos, paradigma surgido en los años 70 debido a la crisis del software que no sabía solucionar los nuevos problemas que se le planteaban. Este paradigma trata de modelizar y representar entidades del mundo real que variaban con su comportamiento a lo largo de su vida. Para ello se desarrolló el concepto de objeto, el cual cumplía las siguientes propiedades:

- Los objetos son entidades que tienen un estado, comportamiento e identidad.
  - El estado está compuesto de datos a los que se habrán asignado valores.
  - El comportamiento está definido por los mensajes a los que responde.
  - La identidad es una propiedad que lo diferencia del resto.

La programación orientada a objetos difiere de la programación estructurada en que en esta última los datos y los procedimientos están separados y sin relación. La programación estructurada se centra en las estructuras de programación, mientras que la orientación a objetos se centra en abstraer la realidad y concretarla en propiedades y métodos que diferencien a las instancias de las entidades. Además, la POO permite mejorar la fase de diseño, acercándola más a la fase de análisis, utilizando conceptos comunes.

Este tema se centra en esta etapa de análisis y diseño, que en la orientación a objetos se convierte en una etapa única desdibujándose la diferencia entre una y otra. En el análisis orientado a objetos el objetivo es definir la realidad basándose en el concepto de clase y objeto. El lenguaje gráfico más utilizado para este análisis es UML, y es el que se va a presentar en este tema.

56.2 ORIENTACIÓN A OBJETOS

Antes de la aparición de la orientación a objetos, ya surgió la necesidad de modelizar estructuras complejas. La primera aproximación que se hizo hacia el concepto de objeto con la programación estructurada fueron los tipos abstractos de datos. Con los tipos abstractos de datos se crearon estructuras como pilas, colas o listas que almacenaban un conjunto de datos y además se asociaron a esas estructuras funciones de manipulación que permitían, por ejemplo, insertar o borrar elementos en ellas.

Poco después surgió la necesidad de almacenar información sobre datos cuyo estado iba cambiando a lo largo de su vida, y con ello surgieron los conceptos de clase y objeto.

56.2.1 Clases

Una clase es una plantilla para la creación de objetos. En ella se definen las características que va a tener ese objeto (atributos), y las funciones para modelar su comportamiento (métodos). Todos los objetos de una clase compartirán esa plantilla.

Ejemplo

Clase: Automóvil
Atributos: matrícula, cilindrada, color, cantidad de combustible
Métodos: Arrancar, reponer combustible, frenar

56.2.2 Objetos

Un objeto es cada una de las instancias de una clase. Es decir, siguiendo la plantilla definida en la clase, un objeto tendrá para cada uno de los atributos un valor y los métodos definirán su comportamiento.

Ejemplo

Un objeto de la clase automóvil se crea dando valor a cada uno de los atributos de la clase:

OBJETO DE LA CLASE AUTOMÓVIL
Matrícula: 5566FVV
Cilindrada: 400cv
Color: blanco
Cantidad de combustible: 20 litros

56.2.3 Atributos

Los atributos definen las características de una clase, son las propiedades de los que interesa almacenar información. Todas las clases deben contener atributos. En el ejemplo anterior los atributos son matrícula, cilindrada, color y cantidad de combustible.

56.2.4 Métodos

Los métodos definen el comportamiento de una clase.

Los métodos pueden clasificarse en distintos tipos:

- **Métodos específicos de la clase**

  Definirán los métodos específicos para poder realizar el comportamiento propio de la clase.

- **Métodos constructores o destructores**

  Los métodos constructores se utilizan para crear objetos de la clase reservando memoria para ellos, mientras que los destructores lo que hacen es liberar la memoria asociada al objeto en cuestión porque ese objeto ya no va a utilizarse. En algunos lenguajes de programación orientada a objetos como Java no es necesario implementar los métodos destructores porque cuenta con su propio recolector de basura que libera la memoria de los objetos que no se utilizan.

- **Métodos getters y setters**

  Un método getter sirve para consultar el valor de un atributo de un objeto determinado, mientras que un método setter sirve para modificar un valor de un atributo determinado del objeto.

Por ejemplo, en el ejemplo del automóvil se tendrían los métodos getters:

- Obtener_matrícula.
- Obtener_cilindrada.
- Obtener_color.
- Obtener_candidad_combustible.

Y como métodos setters:

- Fijar_matrícula(nueva_matrícula).
- Fijar_cilindrada(nueva_cilindrada).
- Obtener_color(nuevo_color).
- Obtener_candidad_combustible(nuevo_combustible).

56.2.5 Propiedades de la OO

El diseño OO, así como la programación OO, tiene una serie de características que le ofrecen un amplio potencial:

- Relación de herencia: hace que la reutilización se amplíe en gran medida pudiendo heredar unas clases a otras y luego refinarlas.
- Polimorfismo: algunos objetos pueden instanciarse en una clase y comportarse como la clase en la que se ha instanciado o en alguna de sus generalizaciones.
- Encapsulación: ofrece la posibilidad de encapsular propiedades y comportamiento dentro de un mismo "contenedor" de manera que desde fuera solo se conoce su interfaz y su comportamiento pero no su implementación.
- Facilita la reutilización: tanto a nivel de diseño como de implementación. Muchas clases generales se repiten en numerosos proyectos y se puede reutilizar esa "caja negra" sin ni siquiera conocer su interior, solo conociendo su funcionamiento.

56.3 UML

Cuando surgió el concepto de análisis orientado a objetos, los principales expertos de la época como Rumbaugh, Booch o Jacobson definieron sus propios métodos para diseñar la realidad basándose en el paradigma de orientación a objetos.

Poco más tarde se unieron y crearon UML (Unified Modelling Language), que se convirtió en el modelo más utilizado de la industria.

En la modelización del análisis estructurado convivían dos visiones que nunca llegaron a encajar: la visión estática donde se partía de un diagrama Entidad/

Relación y se obtenía un diseño de base de datos, y la visión dinámica donde se partía de un modelo DFD y se obtenía la implementación del comportamiento. Eran dos perspectivas que, aun estando relacionadas, no se sabía cómo encajar o como secuenciar.

UML soluciona este problema aunando el diseño de la parte estática y dinámica en un solo proceso.

En UML se puede modelar la información utilizando multitud de diagramas. Es tan amplio el abanico que solo se presentarán los diagramas más representativos para modelar la parte estática y dinámica de un proyecto.

56.3.1 Modelización de la parte estática

Diagrama de clases

Un diagrama de clases se utiliza para representar las clases y las relaciones entre ellas. Este diagrama permite visualizar la estructura del sistema.

Podría decirse que es una evolución del diagrama entidad/relación, aunque en este caso los objetos también incluyen los métodos que implementan su comportamiento.

En el diagrama de clases cada clase se representa con un rectángulo dividido horizontalmente en tres. En la primera posición se encuentra el nombre de la clase, en la segunda los atributos de la misma y en la tercera la interfaz de sus métodos.

Ejemplo

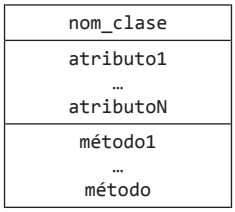


Ilustración 56.1. Representación de una clase en UML

Además de la representación de las clases, en el diagrama de clases se representan también las relaciones entre ellas. Con respecto a las relaciones caben destacar los siguientes conceptos:

Asociaciones

Definen las relaciones estáticas entre clases. Se representan con una línea que puede ir etiquetada con un nombre y pueden establecerse también nombres de rol en caso de considerarse conveniente.

Multiplicidad

En las asociaciones se puede indicar la cardinalidad de la relación, es decir, cuántas instancias de una clase pueden estar relacionada con cada instancia de la otra. En el siguiente ejemplo la multiplicidad indica que un Empleado debe pertenecer a un Departamento y un Departamento puede tener múltiples Empleados.

Ejemplo

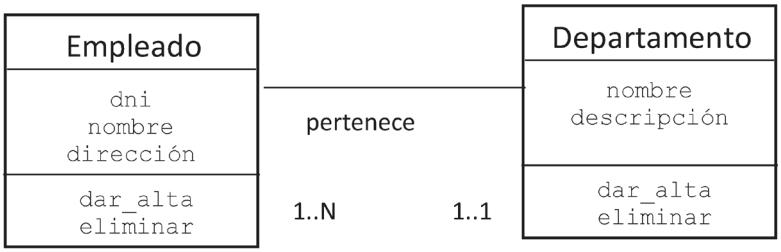


Ilustración 56.2. Ejemplo de asociación en UML

Composición y agregación

La composición es un tipo de relación en la que una clase representa el todo y la otra representa una parte. Además, si la clase que representa el todo desaparece, la clase componente también desaparecerá. Se representa con un pequeño rombo negro en la parte de la clase contenedora.

Ejemplo

Una persona está formada por un conjunto de órganos. Si desaparece la persona, desaparecerían todos sus órganos.

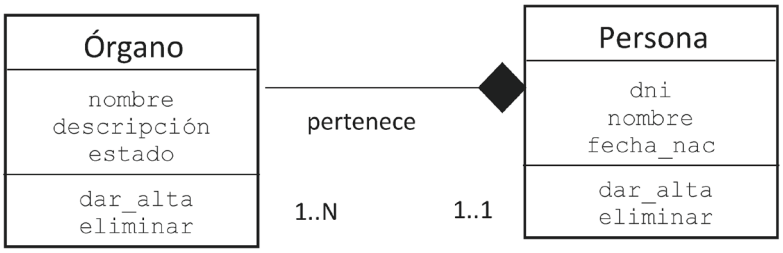


Ilustración 56.3. Ejemplo de composición en UML

La agregación también es una relación entre una clase que representa el todo y otra clase que representa una parte, pero en este caso ambas clases pueden existir sin que exista la otra. Se representa con un pequeño rombo hueco en la parte de la clase contenedora.

Ejemplo

La relación entre empleado y departamento. El departamento está compuesto por empleados, pero si el departamento desaparece los empleados siguen existiendo.

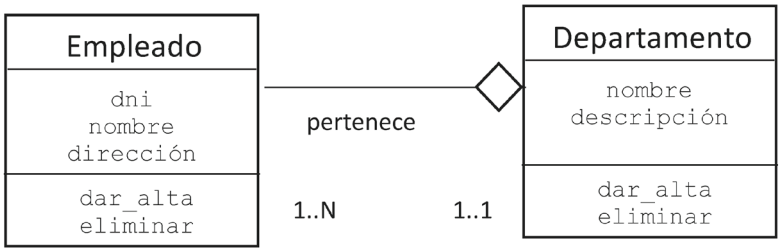


Ilustración 56.4. Ejemplo de agregación en UML

Generalización

La generalización representa la relación de herencia entre clases. Este tipo de relación es extremadamente importante en el diseño OO ya que tiene una traducción directa a la implementación. Los lenguajes OO también implementan relaciones de este tipo.

Se representa mediante un triángulo que apunta a la clase general.

Ejemplo

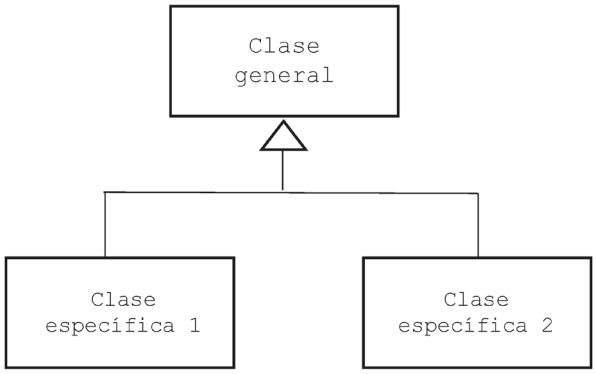


Ilustración 56.5. Ejemplo de generalización en UML

Diagrama de objetos

El diagrama de objetos es la instanciación del diagrama de clases. Se representarán de manera similar a las clases, pero con los valores de los atributos

ya asignados. Además, este diagrama solo muestra las relaciones de determinados objetos en un instante determinado. Es como si mostrara una instantánea de un momento de la ejecución.

56.3.2 Parte dinámica

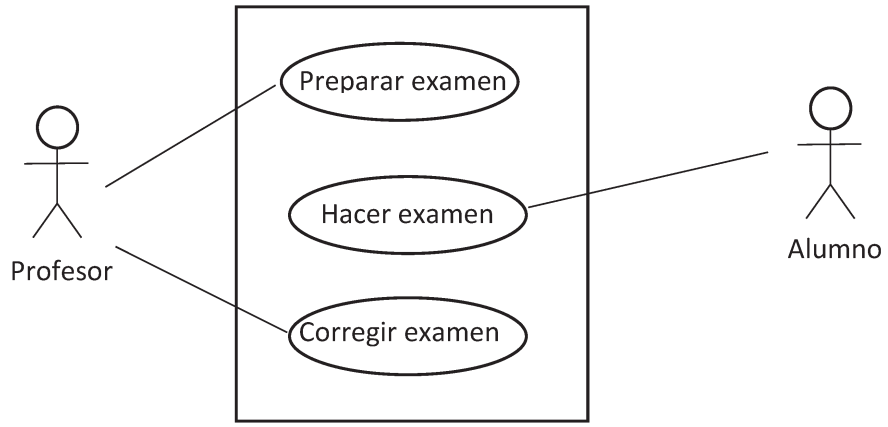
Como ya se ha comentado anteriormente, UML contiene numerosos diagramas. Cada diagrama muestra una perspectiva diferente de los requisitos. Para la parte dinámica destacan por su amplia utilización los siguientes:

Diagrama de casos de uso

Es, probablemente, el diagrama más conocido de UML. Representa la interacción entre los actores (usuarios del sistema) y las distintas acciones que ofrece el sistema.

Muestra gran parte de la funcionalidad del sistema, aunque muestra únicamente qué va a realizar la acción, no cómo. Tampoco se representa la secuenciación de acciones ni su temporalización.

Ejemplo



56.6. Ejemplo de diagrama de casos de uso en UML

Aunque el ejemplo presentado es muy sencillo, en el diseño OO de productos software reales pueden llegar el diagrama de casos de uso puede llegar a tener gran complejidad.

Los casos de uso pueden relacionarse entre sí mediante:

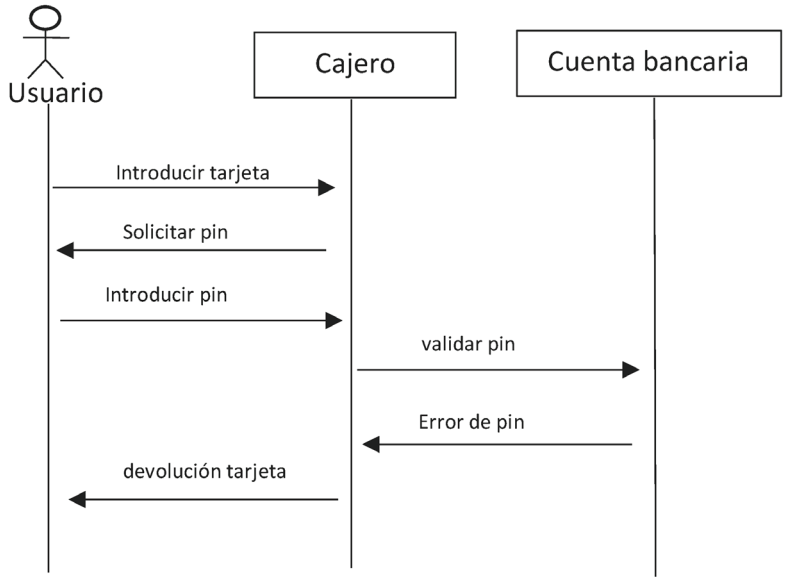
- Include: significa que un caso de uso está dentro de otro.
- Extends: significa que un caso de uso es una especialización de otro.

Diagrama de secuencia

En el diagrama de secuencia se representa una línea de tiempo como una línea vertical que parte de cada objeto que participa en el diagrama. Con flechas se representan las interacciones entre los distintos objetos secuenciados temporalmente. También pueden aparecer actores si estos interactúan directamente con los objetos.

Ejemplo

Suponiendo que se quiere realizar el diagrama de secuencia entre los objetos usuario, cajero y cuenta bancaria y se quiere modelizar la situación en la que un usuario introduce incorrectamente el pin de su tarjeta, un posible diagrama de secuencia sería el siguiente:



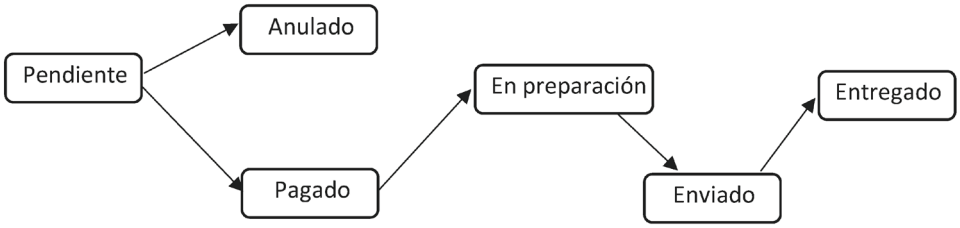
56.7. Ejemplo de diagrama de secuencia en UML

Diagrama de estados

En los diagramas de estado se representa el comportamiento de un objeto. Se muestran los distintos estados por los que puede pasar un objeto a lo largo de su vida y cómo se llega a cada uno de ellos.

Ejemplo

Suponiendo que se realiza un pedido a una tienda online, un posible diagrama de estado sería el siguiente:



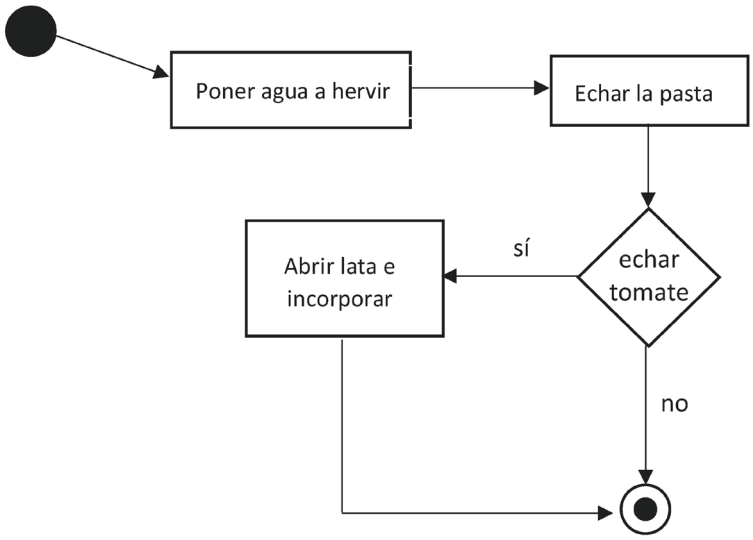
56.8. Ejemplo de diagrama de estados en UML

Diagrama de actividad

El diagrama de actividad representa el flujo de información de un caso de uso. En este caso se detalla el cómo se realiza la acción. Para ello se añaden estructuras como las bifurcaciones. Es muy similar a los diagramas de flujo.

Ejemplo

Si se supone un caso de uso de la preparación de pasta con tomate, un posible diagrama de actividad sería el siguiente:



56.9. Ejemplo de diagrama de actividad en UML

56.3.3 Otros diagramas

Diagrama de despliegue

El diagrama de despliegue indica qué procesos se almacenan en qué dispositivos. Se trata de la modelización de información de bajo nivel.

56.4 RUP

RUP es una especie de ciclo de vida de software que se realiza utilizando UML para la modelización. Se basa en un ciclo de vida iterativo e incremental (forma de espiral), y las fases de cada una de las iteraciones son:

- Planificación.
- Diseño.
- Implementación.
- Prueba.

Los casos de uso son los que integran el trabajo de todas las fases convirtiéndose en el hilo conductor del proceso.

Después de cada iteración se evalúa cómo está el sistema, se realizan los ajustes necesarios y se vuelve al paso de planificación. En algunas de las iteraciones pueden obtenerse resultados parciales que pueden ser bien un prototipo, bien una documentación, etc.

RUP determina 6 buenas prácticas para el desarrollo del software:

1. Gestión de requisitos.
2. Desarrollo de software iterativo.
3. Desarrollo basado en componentes (definir componentes con interfaces claras que más tarde se ensamblarán).
4. Modelado visual utilizando UML
5. Verificación continua de la calidad.
6. Gestión de los cambios.

Las fases por las que pasa cada ciclo que constituye la vida del producto software son:

- Inicio: se realiza el estudio de la viabilidad y el análisis de requisitos.
- Elaboración: se define la arquitectura del proyecto.
- Construcción: se desarrolla el proyecto.
- Transición: se entrega al cliente.

56.5 CONCLUSIÓN

Como se ha podido observar a lo largo del tema, el análisis y diseño orientado a objetos facilita en gran medida la representación del análisis de requisitos con respecto al análisis y diseño estructurado.

Su principal lenguaje de modelización, UML, ofrece numerosos diagramas que ofrecen al diseñador la representación de muchísima información del modelo, tanto de la parte estática como dinámica. En función de la naturaleza del proyecto se utilizarán unos diagramas u otros, aunque en este tema se han presentado los más importantes.

Que un analista conozca y domine los distintos diagramas y cómo utilizarlos va a resultar de gran utilidad a la hora de plasmar las distintas perspectivas del modelo.

Una gran ventaja del análisis y diseño orientado a objetos es que la transformación a la implementación es casi inmediata, facilitando la consistencia de la información y la completitud.

TEMA 57

CALIDAD DEL SOFTWARE. FACTORES Y MÉTRICAS. ESTRATEGIAS DE PRUEBA

Tema 57

CALIDAD DEL SOFTWARE. FACTORES Y MÉTRICAS. ESTRATEGIAS DE PRUEBA

57.1 INTRODUCCIÓN

El software es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación. (IEEE Estándar 729).

Se puede decir que la calidad del software es el grado con el que un sistema, componente o proceso cumple:

- Los requisitos especificados.
- Las necesidades o expectativas del cliente o usuario.

Según Pressman, la calidad de software es la concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados, y con las características implícitas que se espera de todo software desarrollado profesionalmente.

Se deben medir diversos factores como cuánta calidad es necesaria o cómo medirla, ya que, aunque el objetivo utópico sería conseguir el producto software perfecto, eso es imposible. Por ello se debe establecer un punto en el que se considere que el producto está listo.

Al igual que un desarrollador de software puede decir que el código está listo diciendo que "compila", en algún momento los testers necesitan ser capaces de decir "funciona y está prácticamente libre de fallos".

Al igual que un programa está lejos de estar listo cuando compila también es difícil decir cuándo el programa funciona correctamente adaptándose perfectamente a los requisitos iniciales.

Una métrica de software es un conjunto de medidas destinadas a conocer una característica determinada. Generalmente sirven para realizar comparativas o para planificar.

Dada la importancia de la calidad, se han establecido numerosas normas que establecen distintos factores y métricas a evaluar, siendo una de las más importantes la norma ISO/IEC 9126.

Aunque hay distintas clasificaciones en función de la bibliografía consultada, en este tema se va a utilizar la siguiente clasificación de la calidad del software:

- Calidad funcional, que se centra en comprobar que el producto resuelve el problema que se planteó en los requisitos.
- Calidad estructural, que se centra en los aspectos más técnicos del producto, generalmente referente al código.
- Calidad del proceso, que se centra en el propio proceso del desarrollo del software (cumplir plazos de entrega, ajustarse al presupuesto, etc.

Estos 3 aspectos están muy interrelacionados, por ejemplo, mejorar la calidad del proceso con una metodología ágil suele mejorar indirectamente la calidad funcional.

Generalmente, los desarrolladores suelen estar más preocupados de la calidad estructural y la calidad del proceso mientras que lo usuarios suelen centrase más en la calidad funcional y del proceso.

El objetivo es encontrar un equilibro en el que los tres factores de calidad tengan una calidad aceptable, de nada sirve obtener una buena calidad del proceso si se sacrifica la calidad funcional o viceversa.

57.2 CALIDAD FUNCIONAL. FACTORES Y MÉTRICAS

La calidad funcional indica el grado de cumplimiento de un software respecto a los requisitos funcionales o especificaciones. Es el grado de correctitud del software producido.

La mayoría de los tests funcionales y tests de aceptación se centran en alcanzar la calidad funcional. Esto se verá en el apartado final de estrategias de prueba.

Para conseguir calidad funcional, se tendrán en cuenta los siguiente factores y métricas.

57.2.1 Factores

Funcionalidad

La funcionalidad se encarga de realizar software que alcance los requisitos especificados. La calidad funcional tiene que ver principalmente con la cantidad de funciones requeridas por el usuario que son resueltas por el producto software final.

Fiabilidad

La fiabilidad consiste en crear software que tenga pocos fallos. Para que el software tenga calidad funcional no basta con que implemente la funcionalidad requerida, sino que debe hacerlo correctamente, sin fallos. Por ello uno de los factores más importantes es que el software sea fiable.

Rendimiento

El rendimiento indica si, desde el punto de vista del usuario, el producto tiene un desempeño adecuado, es decir, el funcionamiento percibido por parte del usuario del software es el correcto. Por ejemplo, el usuario podría percibir que el producto funciona con demasiada lentitud.

Usabilidad

El software tiene que ser fácil de aprender y fácil de usar para el usuario. La usabilidad tiene que ver con la facilidad de uso del software por parte del usuario para realizar las tareas encomendadas.

57.2.2 Métricas

Las métricas pueden definirse para medir los distintos factores:

Métricas de funcionalidad

- Número de casos de uso implementados

  Una de las métricas más importantes es el número de casos de uso implementados. Puede hacerse a nivel de iteración o a nivel de todo el proceso.

- Porcentaje de casos de uso implementados

La métrica más sencilla es comprobar que las funciones o los casos de usos definidos en el análisis de requisitos se cumplen. Estos requisitos deben ser mapeados a casos de tests individuales. Si un ciclo de prueba consiste en miles de tests, entonces cada uno de estos tests debe definir si ha pasado o fallado el test. Es posible que se le asigne un peso a cada caso de uso en base a su importancia y se pueda calcular el porcentaje del proyecto implementado en base al porcentaje de casos de uso implementados.

- Número de casos de uso bloqueados

  Si un caso de uso es difícil de implementar y hay riesgo de que no se pueda llevar a cabo se dice que está en estado bloqueado. Otra métrica importante es el número de casos de uso en estado bloqueado del proyecto.

- Tests de aceptación

  Los tests de aceptación son una de las formas de medir automáticamente si un caso de uso está implementado. Por ello el número de tests de aceptación pasados o fallidos, o su porcentaje en relación con el total, puede dar una buena estimación de los casos de uso que se llevan realizados y del estado del proyecto desde un punto de vista funcional.

Esto es así siempre y cuando se aplique una metodología en la que cada caso de uso esté siempre emparejado con un caso de prueba.

Métricas de fiabilidad

- Los fallos pueden clasificarse según su importancia, en fallos leves, importantes o críticos. Una de las métricas de software que se pueden utilizar para medir la calidad funcional es el número de fallos de cada una de las categorías.
- Densidad de fallos por módulo: mide la cantidad de fallos en función del número de módulos.
- Tiempo medio de fallo: otra medida es el tiempo medio de fallo, es decir, cuánto tiempo pasa de media hasta que se descubre un fallo en el software.
- Densidad de defectos: mide los fallos en relación con el tamaño del software expresado en líneas de código o puntos de función.
- MTBF (Mean Time Between Failures): esto ayuda a medir la frecuencia de los problemas que encuentran los usuarios en el producto.
- ACR (Application Crash Rate): definen la ratio de fallos en la aplicación dividido por el número de ejecuciones.

Métricas de rendimiento

Este factor depende de la opinión del usuario, por lo que puede medirse, por ejemplo, mediante cuestionarios de satisfacción.

Métricas de usabilidad

Una métrica es la efectividad de uso, el número de tareas realizadas dividido por el número de tareas totales a realizar. Puede estar o no basada en un tiempo dado.

57.3 CALIDAD ESTRUCTURAL

La calidad estructural se refiere a los requisitos no funcionales que soportan la entrega de los requisitos que sí son funcionales.

Principalmente se refiere al cumplimiento de requisitos no funcionales, como pueden ser la robustez o la mantenibilidad. Se refiere a si el software funciona como se desea a nivel técnico, sin problemas.

Este apartado se va a centrar en la definición de factores y métricas para la calidad estructural de la programación orientada a objetos, ya que actualmente es la más utilizada.

57.3.1 Factores

El código tiene capacidad para ser probado. Mediante las pruebas se pueden establecer criterios que debe cumplir un sistema o componente y determinar si se cumplen dichos criterios.

Algunos de los factores determinantes de la calidad estructural son:

Testeabilidad

Es la propiedad que nos indica si el código está desacoplado y es fácil de testear, o lo que es lo mismo, si se organiza el código de forma que es fácil realizar pruebas automáticas. Si el código es testeable significa que tiene bajo acoplamiento y alta cohesión.

Mantenibilidad del código

Un software que es mantenible significa que puede ser modificado de una forma eficaz, o lo que es lo mismo se podrá alterar fácilmente sin modificar su eficiencia.

Modularidad

Cualquier cambio en un módulo del código no hará que el resto de los componentes se vean afectados.

Reusabilidad

Propiedad mediante la cual se podrán reutilizar diferentes componentes o trozos de código en otros módulos.

Legibilidad del código

Facilidad con la que se puede evaluar el impacto de un cambio sobre el resto del software. Si el código es complejo será difícilmente mantenible.

Eficiencia del código

El código fuente y la arquitectura de software son elementos que aseguran un alto rendimiento una vez la aplicación se está ejecutando. La eficiencia es especialmente importante en entornos donde la velocidad de ejecución es importante, como en el proceso algorítmico o transaccional, donde el rendimiento y la escalabilidad son muy importantes.

Un análisis de la eficiencia del código fuente y su escalabilidad advierte de riesgos de negocio latentes y el daño que pueden causar a la satisfacción del cliente debido a la degradación del tiempo de respuesta.

Seguridad del código

Es una medida de la probabilidad de brechas potenciales de seguridad debido a malas prácticas de arquitectura y codificación. Cuantifica el riesgo de encontrar vulnerabilidades críticas que puedan dañar el negocio.

Confidencialidad

Capacidad de protección contra el acceso a datos y/o información no autorizados, ya sea accidental o deliberadamente.

Integridad

Capacidad del sistema o componente para prevenir modificaciones no autorizadas a datos o programas de ordenador que puedan llevar al sistema a un estado incorrecto.

Autenticidad

Capacidad de demostrar la identidad de un sujeto o un recurso.

57.3.2 Métricas de la calidad estructural

La aplicación de prácticas de arquitecturas de software como los principios SOLID en la programación orientada a objetos mejoran la testeabilidad. Una razón es que las clases y los métodos que tienen bajo acoplamiento y alta cohesión tienen mayor facilidad para ser testeados.

Por lo tanto, una buena métrica para ver si se está realizando código testeable es utilizar métricas que midan el acoplamiento y la cohesión.

A continuación se exponen algunas métricas que en general sirven para medir la calidad estructural.

Es interesante utilizar herramientas automáticas como SonarQube e integrarlas en el ciclo de integración continua para verificar la calidad en tiempo real y verificar las métricas.

Métricas de testeabilidad / mantenibilidad

Estas métricas miden el acoplamiento y la cohesión:

- LCOM4 (Lack of Cohesion of Methods)

  Esta métrica mide la falta de cohesión de una determinada clase. La cohesión mide la especialización de una determinada clase o, lo que es lo mismo, cómo de relacionados están los distintos elementos (atributos y métodos) que la componen. LCOM4 calcula un valor para una determinada clase. Si ese valor es mayor que 1, a la clase le falta cohesión. A mayor valor, menor cohesión.

  Para el cálculo del LCOM4 de una clase se mide el número de relaciones diferentes que se establecen entre métodos. Un método se considera relacionado con otro si acceden a un atributo común de la clase o si uno llama al otro. Para el cálculo no se tienen en cuenta los constructores o los getters y setters.

- Índice de interdependencia entre paquetes (acoplamiento):

  Esta métrica calcula los ciclos que hay entre paquetes en el sistema, quiere decir que esos paquetes dependen unos de otros y no se pueden dividir, reduciendo la modularidad del sistema y haciendo que el sistema sea más difícil de probar y menos mantenible. Es deseable minimizar cuanto se pueda la interdependencia entre paquetes.

- Evidencias y Cumplimiento de reglas (code smells)

  Los “code smells” son posibles evidencias de que el código no cumple con buenas prácticas de desarrollo como los principios SOLID. Existe

software que automatiza la detección de code smells y que incluso se integra en ciclos de integración continua, como un test más.

Ejemplos de software que realizan este análisis estático de detección de "code smells" son PMD, CheckStyle y FindBugs.

Algunos ejemplos de code smells:

- **Longitud demasiado larga de clases o métodos:** indicio de que se puede estar violando el primer principio de SOLID (Single Responsibility Principle), que una clase tenga un único propósito general.
- **Uso excesivo de tipos primitivos en vez de objetos:** utilizar tipos básicos como float, int, en vez de encapsular en tipos con referencias (objetos).
- **Comentarios:** dentro de los comentarios y documentación del proyecto, cabe destacar el tanto por cien de APIs documentadas. Esta métrica indica el porcentaje de clases públicas, interfaces, constructores y métodos públicos que tienen comentarios Existen ciertas excepciones como constructores vacíos o getters/setters.
- **Complejidad ciclomática:** tiene que ver con el número de posibles caminos que puede tomar una función. Por ejemplo, si hay un condicional, bucles for, anidamientos, etc. Cuantas más estructuras de este tipo se encuentren, mayor complejidad ciclomática.
- **Bloques duplicados:** el típico copiar y pegar no aporta nada bueno a la calidad estructural.

Métricas de eficiencia

Las métricas de eficiencia se centran en el análisis de costes de los algoritmos, tanto el coste temporal como el coste espacial.

Estas métricas indican cómo medir esos costes de forma práctica o analizando el algoritmo, computan el tiempo de ejecución de algoritmos y lo comparan con otros algoritmos.

Métricas de seguridad

Estas métricas miden criterios como el número de vulnerabilidades conocidas (CVE) y su nivel.

Se conoce como CVE, Common Vulnerability Exposure, a cada una de las vulnerabilidades detectadas. Para cada CVE existe un POC para comprobarlo (Prove Of Concept), que es replicar el escenario en el cual se da la vulnerabilidad para poder encontrar alguna técnica que solucione ese escenario.

57.4 CALIDAD DEL PROCESO

El proceso se refiere al propio procedimiento de creación del producto. Se establecen unas pautas que deben intentar cumplirse en su totalidad. Aunque la calidad funcional y estructural son muy importantes, también es importante la calidad del proceso.

Factores a tener en cuenta en la calidad del proceso son:

- Cumplir con las fechas de entrega, no solo el producto final, si no los distintos prototipos y versiones.
- Ajustarse al presupuesto planificado.

Se necesita un proceso que no estrese al equipo de desarrollo, y que permita cumplir con los objetivos anteriores. Se deben establecer unos parámetros adecuados que permitan alcanzar la calidad funcional y estructural pero sin que afecte a la calidad el proceso.

Las metodologías de desarrollo ágiles hacen énfasis en conseguir procesos que puedan repetirse y mantenerse, ya que estas metodologías:

- Dan prioridad a los individuos e interacciones en lugar de a los procesos y herramientas.
- Dan prioridad al software funcionando con respecto a documentación extensiva.
- Dan prioridad a la colaboración con el cliente con respecto a la negociación contractual.
- Dan prioridad a la respuesta ante el cambio con respecto a seguir un plan.

57.4.1 Factores

Fechas de entrega

Se deben cumplir con las fechas de entrega estipuladas en las negociaciones, no solo para la entrega del producto final, sino de los distintos prototipos y versiones.

Presupuesto

El producto final debe ajustarse al presupuesto estipulado.

57.4.2 Métricas

Dependen de la metodología de desarrollo. A continuación, se comentan algunas usadas en metodologías ágiles.

- **Velocidad del equipo**

  Influye en el factor de cumplimiento de fechas. Esta métrica mide cuanta funcionalidad es capaz de entregar el equipo en un sprint (iteración del ciclo de vida).

- **Burndown chart** (fechas y tareas pendientes):

  Es una representación gráfica del trabajo que queda por hacer en un proyecto representada en el tiempo. Usualmente el trabajo remanente (o backlog) se muestra en el eje vertical y el tiempo en el eje horizontal.

Ambas afectan al cumplimiento de fechas, pero por tanto, también al factor de ajustarse al presupuesto, ya que ambos factores están íntimamente relacionados. Hay que tener en cuenta que si el desarrollo de un producto se dilata en el tiempo aumenta el coste de personal, entre otras cosas.

57.5 ESTRATEGIAS DE PRUEBA

La fase de pruebas es una de las fases más importantes del ciclo de desarrollo del software. Se trata de una fase que valida las anteriores, y por tanto, da información de si el producto está consiguiendo los resultados deseados. Se trata de la fase en la que se detectan los errores cometidos en las otras fases anteriores.

Las pruebas de software consisten en definir un conjunto de actividades, normalmente planificadas previamente, que deben realizarse para testear el funcionamiento. El objetivo de estas actividades es abarcar el número máximo de escenarios posibles en la interacción con la aplicación.

57.5.1 Principios básicos de las pruebas

Los principios básicos de las pruebas son:

- Principio 1: las pruebas deben evaluar el comportamiento de un programa en función de las entradas y resultados esperados.
- Principio 2: las pruebas deben basarse en la especificación de requisitos del programa para garantizar que se cumple el objetivo que se buscaba.
- Principio 3: se deben documentar los casos de prueba realizados.

- Principio 4: El proceso de pruebas es un proceso continuo a lo largo del desarrollo del programa.
- Principio 5: Se deben utilizar distintas estrategias de prueba para evaluar los distintos aspectos del programa.
- Principio 6: Se deben realizar pruebas de forma automatizada para realizar pruebas de forma rápida y eficiente.
- Principio 7: Se deben realizar pruebas desde la implementación del programa hasta el mantenimiento.
- Principio 8: El objetivo final de las pruebas es mejorar la calidad del código y determinar si el programa cumple o no los requisitos establecidos.

57.5.2 Casos de prueba

Una prueba en un programa es el proceso de ejecución de un programa para encontrar errores de forma general, sin embargo, un caso de prueba es un conjunto de pasos y condiciones que se deben de seguir para evaluar el comportamiento de un programa.

Los pasos a seguir en un caso de prueba son:

1. Identificar el objetivo de la prueba

   El primer paso es ver cuál es el objetivo de la prueba y con qué propósito se va a llevar a cabo.

2. Diseñar el caso de prueba

   Una vez se conoce el propósito de la prueba, se ha de diseñar un caso de prueba. Se deben definir qué datos de entrada se van a utilizar, así como qué resultados se esperan en cada momento.

3. Ejecutar el caso de prueba

   Una vez diseñado el caso de prueba hay que iniciar la ejecución.

4. Documentar el resultado de la prueba

   Se debe registrar cada caso de prueba para disponer de un registro de datos que sea útil en caso de errores. De este modo, se podrá garantizar una mejor calidad del programa.

57.5.3 Tipos de pruebas

Una vez se ha desarrollado y compilado un programa es necesario verificar si este cumple con los requerimientos demandados en la primera fase de desarrollo.

Existen varios tipos de pruebas de programas, cada una de ellas se realiza en diferentes momentos del ciclo de vida del desarrollo de cada programa y se pueden realizar de forma individual o utilizando herramientas automatizadas.

A continuación, se presentan las pruebas unitarias, de integración, de sistema y de aceptación.

- **Pruebas unitarias**

  Las pruebas unitarias se utilizan para verificar que cada módulo del código del programa funciona de forma aislada. Verifican si el módulo produce los resultados esperados dadas distintas situaciones.

  Se realizan en fases tempranas del desarrollo del software y suelen realizarlas los programadores.

  JUnit es una herramienta de código abierto que permite realizar pruebas unitarias en Java.

- **Pruebas de integración**

  Son pruebas que se utilizan para verificar si los distintos módulos del programa funcionan correctamente cuando se combinan.

- **Pruebas de sistema**

  Las pruebas de sistema se utilizan para verificar que un programa funciona correctamente en su entorno de ejecución completo. Se realizan después de las pruebas de integración, suelen ser realizadas por usuarios finales.

- **Pruebas de aceptación**

  Las pruebas de aceptación son utilizadas para verificar si un programa cumple con los requisitos que se establecieron al principio.

Existe además un tipo de pruebas especial que se conoce como **pruebas de regresión**. Estas pruebas se realizan cuando hay algún cambio en el sistema y se realizan con el fin de comprobar que el cambio no ha producido errores en el programa.

57.5.4 Técnicas de pruebas

Existen distintas técnicas de prueba de programas, se determinará cuál utilizar en función del objetivo que se quiera obtener. Hay dos enfoques fundamentales desde el punto de vista de la ingeniería del software: pruebas de caja blanca y pruebas de caja negra.

Pruebas de caja blanca

En esta estrategia de prueba se testea la estructura interna del código. Se deben elegir las entradas necesarias para ejercitar los diferentes caminos y comprobar las salidas. Es necesario conocer cómo funciona el código.

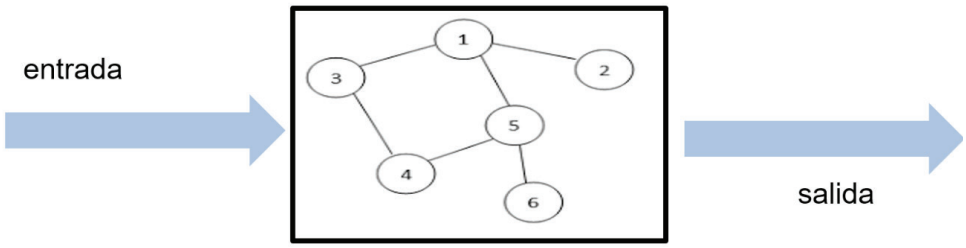


Ilustración 57.1. Prueba de caja blanca

Las principales técnicas de las pruebas de caja blanca son:

- Prueba de interfaz: es la primera que se debe realizar. Se basa en analizar los datos que entran y salen del módulo mediante la interfaz interna y externa.
- Prueba de estructura de datos locales: se debe asegurar la integridad de los datos durante toda la ejecución del programa.
- Prueba del camino básico: se basa en analizar las distintas rutas de ejecución del código.
- Prueba de bucles: se centran en la identificación de los posibles casos de entrada y salida que puedan afectar a los bucles del programa. El primer paso es detectar todos los bucles del programa y el segundo consiste en analizar la entrada y salida de datos en cada uno de ellos.

Para que sea más fácil de entender, es como si alguien va a comprar un coche, pero antes de comprarlo quiere saber si tiene alguna avería, por lo tanto lo lleva al mecánico para que miren si funciona bien.

Pruebas de caja negra

Las pruebas de caja negra son aquellas en las que se realizan las pruebas en los programas teniendo en cuenta únicamente su interfaz externa, no importa la implementación interna.

Las estrategias de prueba de caja negra, también llamados tests de comportamiento, son un método de tests de software en el que la estructura del elemento a testear no es conocido.

Ilustración 57.2. Prueba de caja negra

Por ejemplo, un tester, sin conocimiento de la estructura interna de una página web, testea las páginas web usando un navegador, haciendo clic y verificando la salida en relación con la salida esperada.

Las principales técnicas de pruebas de caja negra son:

- Particiones de equivalencia: también es conocida como partición de clases de equivalencia. En esta técnica los valores de entrada al programa se dividen en distintos grupos según el resultado que se desea obtener. Por ejemplo, un programa que pide la edad del usuario y debe estar entre los 18 y los 60 años. Se podrían hacer tres grupos de prueba: el de menores de 18, el de edades comprendidas entre 18 y 60 y el de mayores de 60 años.
- Análisis de valores límite: esta técnica se centró en los valores límite, en el momento en el que el programa está al límite de cambiar su comportamiento. Por ejemplo, si el programa pide valores entre 1 y 100 puede probar los valores que están al límite: el 0, el 1, el 100 y el 101.
- Error al adivinar: el desarrollador del programa se centra en los errores comunes que suele realizar programando para probarlos en el nuevo programa. Por ejemplo: divisiones entre cero, cargar archivos sin adjuntos, etc.
- Pruebas de comparación: se compara el programa con versiones anteriores o similares.

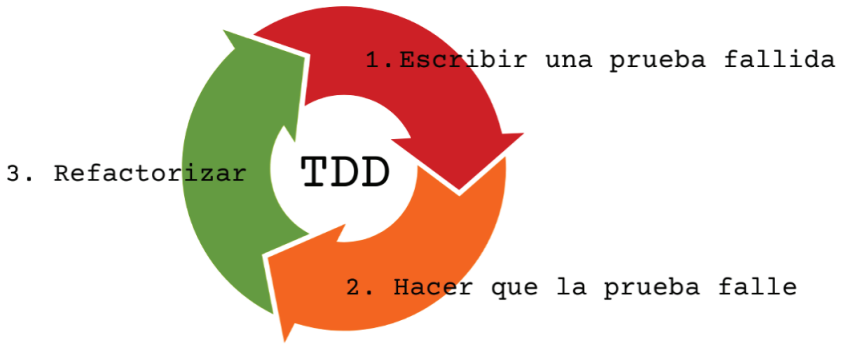
En la analogía de la compra del coche, en el caso de la caja negra el comprador del coche probaría el coche dando una vuelta en él para probar si funciona correctamente.

57.5.5 TDD

El Desarrollo Dirigido por Pruebas (TDD) es una práctica de programación, creada por Kent Beck. Se trata de una nueva metodología en la que primero se escriben las pruebas y después se implementa el código. Una vez se ha llegado a la fase en la que el código está escrito, se pasan todas las pruebas y si es necesario se procede a los cambios de mejora.

Los pasos que se siguen en el TDD son:

- Primero se escribe una prueba que recoja los requisitos.
- Se ejecuta la prueba y la prueba tiene que fallar, en caso contrario es que no se está desarrollando bien, por lo tanto no es válida.
- Se arregla el código para que pase la prueba.
- Se vuelve a ejecutar la prueba y ahora el código sí debe funcionar.



Como el código que se habrá generado está por pulir, se refactoriza y se modifica para que sea más eficiente y se vuelven a pasar los tests una vez se ha modificado el código para ver si sigue pasando correctamente las pruebas.

57.5.6 Herramientas de pruebas

Cada vez son más las herramientas de pruebas que están disponibles en el mercado. A continuación se muestra una tabla con algunas de las herramientas utilizadas y más adelante una breve explicación de algunas de ellas.

TIPO DE PRUEBAS	HERRAMIENTAS	LENGUAJE
UNITARIA	Junit Jtiger	Java
	PHPUnit Simple Test	PHP
FUNCIONAL	Selenium	Java PHP Python Ruby
	HTTP Unit	Java
INTEGRACIÓN	Hudson/Jenkins Conituum	Java
ACEPTACIÓN	Concordion	Java Phython Ruby .net
	Nessus	XML HTML

- JUnit

  Es un framework que se utiliza para hacer pruebas unitarias en aplicaciones desarrolladas en Java. Permite ejecutar programas de forma controlada para evaluar el funcionamiento de cada uno de los métodos implementados.

- Selenium

  Herramienta de código abierto que permite pruebas funcionales de varios módulos de aplicaciones web y navegadores. Admite, entre otros, los siguientes lenguajes: C#, Java, Python, Ruby, etc.

- Hudson (actualmente Jenkins)

  Es una herramienta que se utiliza para automatizar y mantener aplicaciones. Se suele usar en desarrollo de software pero se puede utilizar en otras áreas, por ejemplo en construcción de edificios o fabricación de productos.

- JMeter

  Es una herramienta de Apache y permite probar el rendimiento de sitios y aplicaciones web dinámicas. Es muy sencillo de utilizar.

57.6 CONCLUSIÓN

A lo largo de la historia del software se han producido diversas crisis por la utilización de ciclos de vida poco adecuados que generaban productos finales con un código poco eficiente, llenos de "parches" para solucionar los problemas que iban surgiendo, por lo que el producto final tenía poca calidad.

En la actualidad, sin embargo, se utilizan ciclos de vida mucho más eficientes, y se han desarrollado métricas para cada una de las etapas, con el único objetivo de conseguir un producto software de máxima calidad.

Es tal la importancia que se le ha dado a la medición de la calidad, que la cantidad de métricas desarrolladas puede ser abrumadora, pudiendo caer en el error de que un programador invierta más tiempo aplicando las diferentes métricas para medir su software que realmente produciéndolo, aunque la experiencia y el trabajo en equipo ayudan a detectar este tipo de problemática.