**Encapsulamiento**

El encapsulamiento (encapsulation) es un término formal que se refiere al mecanismo que reúne el estado y el comportamiento de un objeto en una única unidad lógica

**Ocultación de información**

La ocultación de información (information hiding) es una buena práctica de programación en la cual se oculta cómo los servicios se realizan y los datos que se mantienen internamente para soportar los servicios.

**Sobrecarga**

La sobrecarga (overloading) es un mecanismo de los lenguajes que permite que dos o más métodos de la misma clase tengan el mismo nombre si tienen argumentos diferentes.

**Clase (en software)**

Una clase define las características - atributos y métodos - que cada objeto que

pertenece a la clase posee. Puede ser visto como un molde.

**Objeto (en software)**

Un objeto es un artefacto software que representa una abstracción de un objeto del

mundo real por medio de su estado (atributos) y comportamiento (métodos).

**Componentes**

Ejecutable

Fuentes

Manual de usuario

Manual técnico

**Calidad**

Corrección

Extensibilidad

Facilidad de Uso

Eficiencia

Portabilidad

**Etapas**

Requisitos

Análisis

Diseño

Construcción

Pruebas

**SOLID**  
Solid es un acrónimo que describe los 5 principios de la programación orientada a objetos. Este acrónimo se refiere a:

S – Single responsability Principle(SRP)

O – Open close principle(OCP)

L – Liskov substitution principle(LSP)

I -  Interface segretation principle.(ISP)

D – Dependency inversión principle.(DIP)

**SRP:**Cada clase o objeto debe de tener una responsabilidad única , sencilla, concreta.

**OCP:**Los objetos o componentes deben estar abiertos para desarrollar en base a ellos pero cerrados a su  modificación.

**LSP:**Una subclase puede ser substituida por una super clase por ejemplo puedo substituir la subclase apartamento por la superclase vivienda y este no debe de dar error.

**ISP :**Cada clase debe de implementar las interfaces que va a usar.

**DIP:**Las implementaciones de alto nivel no deben de depender de las de bajo nivel ni viceversa.

**Code the Unit Test First**

Cuando cree sus pruebas primero, antes del código, le resultará mucho más fácil y rápido crear su código. El tiempo combinado que lleva crear una prueba unitaria y crear algún código para que pase es casi lo mismo que simplemente codificarlo de inmediato. Pero, si ya tiene las pruebas unitarias, no necesita crearlas después del código, ahorrándole algo de tiempo ahora y mucho más tarde.  
La creación de una prueba unitaria ayuda a un desarrollador a considerar realmente lo que se debe hacer. Los requisitos se establecen firmemente mediante pruebas. No puede haber malentendidos en una especificación escrita en forma de código ejecutable.  
También tienes retroalimentación inmediata mientras trabajas. A menudo no está claro cuándo un desarrollador ha terminado toda la funcionalidad necesaria. El desplazamiento del alcance puede ocurrir a medida que se consideran las extensiones y las condiciones de error. Si creamos nuestras pruebas unitarias primero, sabremos cuándo hemos terminado; la unidad prueba todo ejecutado.

**BDD**

Es una estrategia de desarrollo dirigido por comportamiento.  
Las ventajas de usar la estrategia de desarrollo BDD en una empresa son:  
-Todo el trabajo de diseño y desarrollo está directamente relacionado con los objetivos de negocio.  
-La aplicación se desarrolla desde el punto de vista del usuario, lo que resulta en una mejor experiencia del usuario.  
-El enfoque centrado en el negocio prioriza primero la entrega de características críticas para el negocio.  
-Todas las partes interesadas comparten la misma información y evalúan la solicitud sobre un terreno común.  
-Un lenguaje y vocabulario compartidos para la descripción de las funcionalidades permiten una comunicación eficiente entre los equipos.  
-La excelente calidad del código se mantiene desde el primer día, ya que no hay confusión sobre el resultado final y el flujo de usuario.

**MDD**  
El MDD (Model Driven Development) es un paradigma de construcción de software que asigna a los modelos un rol central y se derivan modelos que van desde los más abstractos a los más concretos. Este paradigma, además, proporciona un marco de trabajo que permite a los interesados compartir sus puntos de vista y manipular las representaciones de las entidades del dominio. Las principales motivaciones de este paradigma son la independencia de los productores de software a través de estandarizaciones y la portabilidad de los sistemas de software.  
  
El objetivo de MDD es separar el diseño del sistema de la arquitectura de las tecnologías, para que puedan ser modificados independientemente. Para lograr esto, se asigna a los modelos un rol central y activo bajo el cual se derivan

**1.** ¿Cuándo utilizaría herencia y cuando una interfaz y por qué?

Se utiliza herencia cuando se quieren generalizar comportamientos diciendo el cómo, en cambio el uso de la interfaz no. Esto porque en la herencia puede tener métodos que no son abstractos y por la tanto se puede decir el cómo, y en la interfaz todos los métodos son abstractos, además usuaria una interfaz cuando se necesite hacer herencia múltiple.

**2.** Explique cuales son los 3 momentos de una excepción

Los tres momentos de una excepción son: lanzar, propagar y atender.

**1)** ¿Qué se puede hacer con las excepciones? ¿Cómo se diseñan estas acciones? ¿Cómo se codifican en java?

Las excepciones sirven para enviar mensajes de que algo fuera de lo normal está pasando.

Se diseñan en tres momentos: lanzar, propagar y atender.

Loas anteriores se codifican de la siguiente manera:

Lanzar:

**throw new <clase excepción>(<clase excepción>.<atributo excepción>)**

Propagar:

public void method() **throws <clase excepción>{}**

Atender

**Try {**

**} Catch (<clase exepción> e) {**

**}**

**2)** ¿Qué significa las primeras dos letras de SOLID? Escoja dos puntos del diseño de SwFactory para ilustrar: ¿cómo se cumple la S? ¿cómo se cumple la O?

S: Single responsability Cada clase o objeto debe de tener una responsabilidad única , sencilla, concreta.

O: Open-Closed Los objetos o componentes deben estar abiertos para desarrollar en base a ellos pero cerrados a su  modificación.

**1**. ¿Cuáles son las características de una interfaz?

* No se puede instancia
* Sus métodos son abstractos
* Generaliza comportamientos sin decir el cómo.
* Permite hacer herencia múltiple.

**2.** ¿Cuándo es conveniente utilizar un método abstracto y por qué?

Cuando el comportamiento es generalizado pero la implementación es especifica de cada subclase.

**1.** ¿Qué tipos de métodos tiene una clase abstracta? Explique cuándo usar cada uno de ellos.

Una clase abstracta tiene métodos abstractos y concretos. Se usan los abstractos cuando la implementación de estos es diferente en todas las subclases, en cambio los concretos la mayoría lo implementan de la misma forma.

**2.** ¿Qué tipos de métodos puede tener una interfaz? Explique cuándo usar cada uno de ellos

Las interfaces tienen métodos que solo son abstractos que se usan cuando todas las clases que lo implementan lo deben definir obligatoriamente, también tiene los default que ya cada clase decide si implementarlos o no.

**1.** ¿Qué es una clase abstracta? ¿Cuál podría ser una diferencia con una interfaz?

Una clase abstracta es aquella que generaliza comportamientos y característica, pero no se pueden instanciar objetos de ellas.

Una diferencia con la interfaz es que esta no dice el cómo, en cambio la abstracta pude decir el cómo. Otra diferencia es que una clase puede implementar varias interfaces, en cambio solo puede hacer herencia a una clase abstracta.

2. ¿Cuál es la diferencia entre error y excepción? Puede usar un ejemplo para relacionar la diferencia.

Los errores no pueden ser controlados por lo programadores en cambio las excepciones sí.

Por ejemplo, un error puede ser de memoria lo cual el programador no puede controlar y una excepción puede ser.