1. Factores de calidad del software

- **Oportunidad:** Capacidad de un sistema de ser lanzado cuando el usuario lo necesita, o antes.
- **Economía:** Los costes del producto. A veces lo más importante de todo.

1.1 Factores externos e internos

- **Eficiencia:** Realizar correctamente aquello para lo que ha sido creado de la mejor forma posible. Algunos factores son: el espacio en memoria utilizado por el programa y el tiempo de ejecución, espacio en disco, etc.
- **Portabilidad:** capacidad o facilidad del producto de ejecutarse en otro hardware diferente o en otro sistema operativo diferente. Es importante que el programa no haga uso de características de bajo nivel del hardware o que asile la parte dependiente del hardware para solo modificar dicha parte al portarlo

1.2 Factores externos

- **Fiabilidad:** El producto proporciona unos resultados con la precisión requerida o con la total satisfacción por parte del usuario, de forma que el usuario pueda confiar totalmente en la bondad de los resultados que especifico.
- **Robustez:** Capacidad del producto de manejar correctamente situaciones imprevistas, de error o fuera de lo normal.
- **Corrección:** Capacidad del producto para realizar de forma adecuada aquello para lo que fue creado.
- **Compatibilidad:** Facilidad que tienen los programas de combinarse entre sí, los datos de salida de un programa como entrada de otro programa.
- **Seguridad:** Capacidad del producto de proteger sus componentes de usos no autorizados y de situaciones excepcionales de perdida de información.
- **Integridad:** El producto no debe corromperse por el simple hecho de su utilización masiva o por una gran acumulación de datos, o por operaciones complejas posibles, pero no previstas al cien por cien.
- **Facilidad de uso:** Facilidad al introducir datos, interpretar datos, etc.
- **Accesibilidad general:** Tener acceso, paso o entrada a un lugar o actividad sin limitación alguna por razón de deficiencia, discapacidad o minusvalía.
- **Accesibilidad informática**: Acceso a la información sin limitación alguna por razones de deficiencia, incapacidad o minusvalía. Un programa accesible es aquel que permita el acceso a la información sin limitación alguna por razón de deficiencia, incapacidad o minusvalía.

1.3 Factores internos:

- **Reutilidad/Reusabilidad**: Capacidad del producto de ser reutilizado en su totalidad o en parte por otros productos, con el objetivo de ahorrar tiempo en soluciones redundantes ya hechas con anterioridad. Un programa debe agrupar en módulos aislados los aspectos dependientes de la aplicación particular.
- **Extensibilidad:** Adaptar el producto a cambios en la especificación de requisitos.

2. Descomposición:

La descomposición busca agrupar las estructuras de datos con los procedimientos que los manipulan en módulos, o bien agrupar en módulos un conjunto de procedimientos relacionados.

2.1 5 Criterios para evaluar la descomposición

- Descomposición modular: Un método de construcción de software debe ayudar en la tarea de descomponer el problema de software en un pequeño número de subproblemas menos complejos, interconectados mediante una estructura sencilla y suficientemente independientes para permitir que el trabajo futuro pueda proseguir por separado en cada uno de ellos.

- Composición modular: La producción de elementos de software se pueden combinar libremente unos con otros para producir nuevos sistemas.

- Comprensibilidad modular: Un lector humano puede entender cada módulo sin tener que conocer los otros.

- Continuidad modular: Si hay un pequeño cambio en la especificación o en los requisitos del problema, provoca sólo cambios en un solo módulo o en muy pocos módulos.

- Protección modular: Si existe un problema dentro de un módulo, no se propaga a otros módulos vecinos.

2.2 5 Reglas

- Correspondencia directa: La estructura modular obtenida debe ser compatible con la estructura modular del dominio del problema.

- Pocas interfaces: Un módulo debe comunicarse con el menor número posible de módulos.

- Pequeñas interfaces: Los módulos deben intercambiar la menor información posible.

- Interfaces explícitas: Las interfaces deben ser obvias a partir de su simple lectura.

- Ocultación de la información: El diseñador de cada módulo debe seleccionar un subconjunto de propiedades del módulo como información oficial sobre el módulo para ponerla a disposición de otros autores de módulos clientes.

2.3 5 Principios

- Unidades modulares lingüísticas: Los módulos deben corresponderse con las unidades sintácticas del lenguaje de programación utilizado.

- Auto-documentación: El diseñador de un módulo debiera esforzarse por lograr que toda la información relativa al módulo forme parte del propio módulo.

- Acceso uniforme: Todos los servicios ofrecidos por un módulo deben estar disponibles a través de la notación uniforma sin importar si están implementados a través del almacenamiento o de un cálculo.- Principio abierto-cerrado: Se debe facilitar la posterior modificación, pero un módulo debe cerrarse para ser utilizado.

- Elección única: Si hay varias alternativas, sólo un módulo del sistema conocerá la lista completa.

3. ¿Qué es la clase raíz de un sistema software orientado a objetos y cuando se dice que un sistema software orientado a objetos está cerrado (cierra de un sistema software)?

- Clase raíz: Clase de la cual derivan todas las demás clases del sistema software.
- Un sistema es cerrado si contiene todas las clases que necesita la clase raíz.

4. Personas importantes y sus aportes:

- Dennis Ritchie: C (1972), Unix (1970).
- Ken Thomson: Unix (1970).
- Bjarne Stroustrup: C++ (1980).
- Richard Stallman: GNU (1983).
- Steve Jobs: Apple (1976).
- Linus Torvalds: Linux (1991).

5. ¿Qué patrones de diseño conoces? Enuméralos todos y comenta en más profundidad uno de ellos a tu elección.

Iterador, observador, objeto compuesto, strategia, triada modelo-vista-controlador.

ITERADOR

Nombre: iterator

Estructura:

- Procesa elementos de una colección
- Sin la exposición de detalles internos
- C++ STL iterators. Ruby: each, collect.

Multitud de aplicaciones

Consecuencias:

- Sencillez de uso
- Mismo uso para todas las colecciones
- Independencia de la representación interna

OBSERVADOR

Nombre: observer. También conocido como publish-subscribe, dependents

Estructura:

- Es un ejemplo de clases cooperantes que se da mucho en diseño de software
- Sujeto: los datos. En su interfaz tiene un método para comunicar cambios a los observadores asignados
- Observador: puede hacer varios asignados al mismo sujeto. En su interfaz tiene un método para actualizarse

Consecuencias:

- Ambos elementos son independientes.
- Se puede reutilizar por separado
- Se puede añadir observadores nuevos

Aplicaciones:

- Infinidad de aplicaciones tienen clases cooperantes de este tipo
- Relación entre modelo y vista en el patrón MVC

STRATEGIA

Nombre: strategy

Estructura:

- Familia de algoritmos intercambiables
- Se prepara una descripción genérica del algoritmo para que posteriormente se instancie con el que convenga

Aplicaciones:

- Cuando preveamos distintos comportamientos en un futuro, podemos habilitarlo
- Estructura de datos complejas que podrán implementarse en un futuro de otras formas

Consecuencia:

- Posibilidad de mejorar eficiencias y rendimientos en el futuro
- Permitir otras estrategias de solución distintas a la propuesta inicialmente
- Facilitar aplicaciones

6. Define qué es patrón de diseño y cuando debe utilizarse. Describe los elementos esenciales (nombre, estructura, aplicación y consecuencias) de los tres patrones de diseño principales en los que se basa la tríada MVC

- **Definición:** Descripciones de clases, objetos y relaciones entre sí, que están especializadas en resolver un problema de diseño general en un determinado contexto.

Triada MVC

Nombre: MVC

Estructura:

- Modelo: objeto de la aplicación
- Vista: su presentación o representación
- Controlador: define el modo en que se reacciona ante la entrada, por ejemplo, del usuario

Usa las propiedades de varios patrones de diseño que cooperan: observer, composite, strategy.

Tiene su origen en el lenguaje Smalltalk-80

Aplicaciones: modernos

Todos los entornos de desarrollo de aplicaciones

Cualquier aplicación

Consecuencias:

- Simplificación del desarrollo
- Separar desarrollos
- Varias presentaciones para un mismo modelo

7. Encapsulado y ocultación de la información

- Un TAD tiene una vista exterior, en la que se ocultan detalles, estructuras de datos y su comportamiento interno. También tiene una vista interior, en la que se puede ver cómo está hecha por dentro y cómo se comporta internamente cada operación, las estructuras de datos utilizadas, el código, etc.

- La separación de estas dos vistas es muy importante, ya que en el diseño de un TAD no debe intervenir la vista interna. Para el cliente del TAD tampoco es necesario conocer el interior de un TAD para poder usarlo. Esto facilita el diseño y lo hace de más calidad, también hace la labor del cliente más sencilla y potente.

- No solo es cuestión de facilitar su uso, también evita que el cliente tenga que modificar la parte interna por cualquier razón y provoque operaciones y usos no permitidos, errores, etc. Es mejor que el cliente se limite a conocer la vista externa y a usar la interfaz del TAD.

- En C++, la vista exterior (interfaz) son todos los métodos y datos que se encuentran dentro de la sección public. La sección private, por el contrario, se usa para la vista interna.

8. La Programación Orientada a Objetos (POO)

- Es una forma más natural y cercana a la realidad de programar.

- La base de la programación estructurada es la función, que es más difícil de comprender.

- La POO usa como base la clase, que integra los siguientes conceptos:

o Representa un objeto del mundo real.

o Tiene atributos.

o Tiene comportamiento.

o Tiene entidad por sí misma.

o Tiene sentido por sí misma, se entiende, se comprende fácilmente.

o Se puede reutilizar en distintos problemas.

- La POO es unan forma natural de modelar problemas reales mediante programación.

9. Abstracción

Consiste en destacar los detalles importantes e ignorar los irrelevantes. Los detalles son irrelevantes a cierto nivel, luego serán relevantes a otros niveles.

9.1 Abstracción por parametrización

El uso de parámetros permite abstraer de los detalles de los datos sobre los que se aplica un procedimiento o función.

9.2 Abstracción por especificación

Permite abstraer de los detalles del cómo, considerando únicamente el qué. Informa de los detalles relevantes de datos, funciones… y evita los irrelevantes.

9.3 Abstracción de datos

- Consiste en la especificación de TADs.

- Un TAD es una colección de datos y operaciones sobre estos datos, que se define mediante una especificación que es independiente de cualquier implementación.

10. Operaciones con TADS

- Constructores: llevan al objeto a su estado inicial.

- Observadores: acceso de lectura a una característica del objeto.

- Modificadores: acceso de escritura a una característica del objeto.

- Destructores: llevan al objeto a su estado final descartando posibles efectos laterales imprevistos.

11. Polimorfismo

Una de las formas de polimorfismo es a través de la sobrecarga de funciones: una o más funciones pueden compartir nombre siempre y cuando las declaraciones de sus parámetros sean diferentes.

12. Referencia

- Son alias o nombres alternativos que pueden darse a una variable u objeto. Al definirse siempre se ha de indicar a qué variable hace referencia. Ejemplo: int &p = i;

- Se pueden pasar parámetros por referencia a una función para que se modifiquen dentro. Ejemplo: int funcion (int &variable);

13. Constructores y destructores

- El constructor es una función miembro de la clase que tiene el mismo nombre que la propia clase y que, si existe, se ejecuta automáticamente en el punto del programa donde se declara el objeto.

- El constructor puede recibir parámetros.

- El destructor tiene igual nombre que el constructor, pero anteponiendo el carácter ~. No puede recibir parámetros. Se suele usar para liberar memoria.

14. Funciones Friend

Son aquellas que, no siendo miembro de una clase, pueden acceder a la parte privada de ella. Se tienen que declarar en las dos clases (la que hace uso de la función y la que tiene datos en la parte privada a los que se quiere acceder).

15. Polimorfismo

- Es el proceso mediante el cual se puede acceder a diferentes implementaciones de una función utilizando el mismo nombre. Este proceso atiende al esquema: una interfaz, múltiples métodos.

- Clases de polimorfismo:

o En tiempo de compilación: Se refiere a la sobrecarga de funciones y de operadores.

o En tiempo de ejecución: Uso conjunto de clases derivadas y funciones virtuales.

16. ¿Qué es la STL?

La STL (Standard Template Library) de C++ es una colección genérica de plantillas de clases y algoritmos que permite a los programadores implementar fácilmente estructuras estándar de datos como colas, listas y pilas.