Programación Orientada a Objetos - Teoría

1. Factores de calidad del software

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

1.1. Factores externos e internos

- **Eficiencia**: Se trata de conseguir que el programa además de realizar correctamente aquello para lo que ha sido creado, lo realice 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 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 aisle la parte dependiente del hardware solo para 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, tal y como se definió en los documentos de especificación y requerimentos.
- **Compatibilidad**: Facilidad que tienen los programas para combinarse entre si, los datos de salida de un programa sirven como entrada para otro. La clave es la estandarización y el consenso entre organismos, instituciones y empresas, en primera instancia, y entre programadores en última instancia.
- **Seguridad**: Capacidad del producto de proteger sus componentes de usos no autorizados y de situaciones excepcionales de pérdida de información. Para ello debe prever mecanismos de control de acceso, encriptación, clave de acceso y también, desde el otro punto de vista copias de seguridad, procesos rutinarios de comprobación, etc.

- **Integridad**: El producto no debe corromperse por el simple hecho de su utilización masiva o por gran acumulación de datos, o por operaciones complejas posibles, pero no previstas al 100%.
- **Facilidad de uso**: Facilidad al introducir los datos, interpretar datos, etc.
- **Accesibilidad general**: Tener acceso, paso o entrada a un lugar sin limitación alguna por razón de deficiencia, discapacidad o minusvalia.
- **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 de los atributos anteriormente citados.

1.3. Factores internos:

- **Reusabilidad/Reutilidad**: 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**: Adapatar el producto a cambios en la especificación.

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. Criterios para evaluar la descomposición (5)

- **Descomposición modular**: Un método de construcción de software debe ayudar en la tarea de descomponer el problema de software en un pequeño numero de subproblemas menos comlejos, 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**: Un método satisface la composición modular si favorece la producción de elementos software que se puedan combinar libremente unos con otros para producir sistemas, posiblemente en un entorno bastante diferente de aquel en que fueron desarrollados inicialmente.
  - Comentarios: Los módulos son independientes y fáciles de combinar entre sí para varias aplicaciones. Bibliotecas de programas, comando '`pipe` de la Shell de UNIX, etc. Como contraejemplo los módulos que dependen de la aplicación, los que no se pueden compilar o combinar juntos, etc.

- **Comprensibilidad modular**: Un método favorece la comprensibilidad modular si ayuda a producir software en el cual un lector humano puede entender cada módulo sin tener que conocer los otros, o, en el peor caso, teniendo que examinar solo unos pocos de los restantes módulos.
  - Comentarios: Afecta mucho a la comprensión y al mantenimiento. Se suele asumir que si nos resulta dificil explicar que hace o para que se ha creado un módulo, ese módulo no está bien diseñado (lo mismo ocurre a otros niveles con los procedimientos/funciones).
- **Continuidad modular**: Un método satisface la Continuidad Modular si en el sistema resultante desarrollado con este método, un pequeño cambio en la especificación o en los requisitos provoca solo cambios en un único módulo o en un reducido número de ellos.
  - Comentarios: El mantenimiento es una fase muy importante del ciclo de vida del software. Si no nos preparamos para él, fracasaremos. Si nuestros módulos no facilitan el mantenimiento, no serán unos buenos módulos. Por otro lado, el cambio en cualquier fase del desarrollo y durante el propio mantenimiento es algo bastante frecuente.
- **Protección modular**: Un método satisface la protección modular si produce arquitecturas en las cuales el efecto de una situación anormal que se produzca dentro de un módulo durante la ejecución queda confinado a dicho módulo o en el peor caso se propaga sólo a a unos pocos módulos vecinos.
  - Comentarios: Si un error dentro de un módulo no es detectado por el módulo y siguen ejecutandose otros modelos arrastrando el error, este será dificil de corregir y de impredecibles consecuencias

2.2. Reglas (5)

- **Correspondencia directa**: La estructura modular obtenida debe ser compatible con la estructura modular del dominio del problema.
  - **Comentarios**: Si ya se ha estructurado el problema en módulos, se han identificado, etc. Su correspondencia con módulos software ayudará mucho en el desarrollo del sistema.
- **Pocas interfaces**: Un módulo debe comunicarse con el menor número posible de módulos.
  - **Comentarios**: Debe ser independiente, si se comunica con muchos modulos es que tiene un grado de dependencia alto.
- **Pequeñas interfaces**: Los módulos deben de intercambiar el menor número posible de módulos.
  - **Comentarios**: *El mismo sentido que el anterior*

- **Interfaces explicitas**: Las interfaces deben de ser obvias a partir de su simple lectura.
  - **Comentarios**: Lo mismo ocurre si un módulo debe de llamar a otro, debe entenderse de forma lógica el motivo de esa llamada.
- **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.
  - El resto de información será privada al desarrollador y no debe hacerse pública puesto que no es necesario que se conozca para el uso del módulo, es más, su conocimiento puede resultar contraproducente puesto que será información de bajo nivel de la cual el usuaario no debe depender.

2.3. Principios (5)

- **Unidades modulares lingüisticas**: Los módulos deben correspnderse con las unidades sintacticas del lenguaje de programación utilizado.
  - Ejemplo: Si hacemos un módulo necesitamos un lenguaje que tenga un buen soporte de módulos. Que se puedan compilar por separado, etc. De otra forma habría que hacer traducciones indeseables, adaptaciones complejas y costosas
- **Auto-documentación**: El diseñador de un módulo debiera esforzarse por lograr que toda la información relativa al módulo forme parte del propio módulo. El ideal sería que la simple lectura del código del módulo es su propia documentación
- **Acceso uniforme**: Todos los servicios ofrecidos por un módulo deben estar disponibles a través de la notación uniforma sin importar si están implementados a traves del almacenamiento o de un cálculo. La uniformidad es importante. Ya que el usuario hace un esfuerzo por entender el módulo, no le cambiemos la nomenclatura, las reglas básicas, etc. en el otro módulo.
- **Principio abierto-cerrado**: Puede parecer contradictorio pero son de naturaleza diferente. Los módulos deben ser abiertos y cerrados.
  - Abiertos: Esforzarse en facilitar la posterior modificación, ampliación, etc. Arquitecturas abiertas, estandares, convenciones, etc.
  - Cerrado: El módulo debe estará cerrado si está disponible para ser usado por otros módulos. Debe completarse, quizá no siendo demasiado ambicioso en las versiones iniciales, ir cerrando las cosas y no dejandolas para luego de forma que el módulo sea imposible de usar incluso despues de un largo periodo de trabajo en él.

- **Elección única**: Relacionado con el principio abierto-cerrado y la ocultación de la información. Muchas veces haremos un software que, por ejemplo, usa una estructura de datos susceptible de ser ampliada en un futuro. Pero es necesario hacerlo cerrado, y que pueda usarse, pero a la vez abierto para facilitar la ampliación mencionada. Lo que busca este principio es que la mencionada ampliación solo afecte a un solo módulo.
  - Entonces: Solo un módulo debe conocer la estructura de datos completa con todas sus variantes. Y este módulo será el único que se modifique al ampliar/cambiar la estructura de datos. Este módulo será el único que elija (por elección única) opciones según la instancia de las 7 estructuras de datos utilizadas en cada momento. En el resto de módulos no influirán estas elecciones porque para ellos están ocultas.

3. ¿Que es la clase raíz de un sistema 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 Thompson: Unix(1970)
· Bjarne Stroustru: C++(1980)
· Richard Stallman: GNU(1983)
· Steve Jobs: Apple(1976)
· Linus Torvalds: Linux(1991)

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

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

- **Iterador**:
  - **Estructura**:
    - Procesa simple y uniformemente elementos de cualquier colección de datos.
    - No hay exposición de detalles internos.

    - Ejemplos C++: STL iterators: each, collect. Python: for ... int etc.
  - **Aplicaciones**: Múltiples y en cada trozo de software donde hay una colección (también denominados ***contenedores*** en C++)
  - **Consecuencias**:
    - Sencillez de uso
    - Acceso uniforme para todas las colecciones.
    - Independencia de la representación interna.
  - **Ejemplo**:

```
x=[1,2,True, 4.5,"hola"]

for i in x:
    print i

from itertools import count
counter = count(start=13)   # Iterador
counter.next()              # Contador del iterador

from itertools import cycle
colors = cycle(['red', 'white', 'blue'])    # Iterador
colors.next()                               # Contador del it.

fo=open("fich.txt")     # Iterador del fichero
print fo.next()         # Contador del iterador
```

- **Observador**:
  - **Estructura**:
    - Esquema de **clases cooperantes** el cual **se da mucho en diseño de software** (**clases cooperantes**: varias clases interaccionan entre sí dentro del patrón).
    - Estructura definida por el par: sujeto-observador.
    - Sujeto: los datos
      - Conoce a los observadores asignados
      - En su interfaz tiene un método para comunicar cambios
    - Observador: objetos que se nutren de los datos
      - Puede haber varios asignados (suscritos) al mismo sujeto
      - En su interfaz tiene un método para actualizarse.
  - **Aplicaciones**:
    - Infinidad de aplicaciones tienen clases cooperantes de este tipo.
    - Relación entre modelo y vista en el patrón de diseño MVC.
  - **Consecuencias**:
    - Ambos elementos son independientes.
    - Se pueden reutilizar por separado.

    - Se pueden añadir observadores nuevos.
    - Mezclar datos y observador es un ERROR de diseño.
  - **Ejemplo**: ***datos vs vistas***
    - Datos. Es el sujeto. Puede ser una base de datos
    - Vistas (observadores/suscriptores):
      - Estadística
      - Gráfica
      - Hoja de cálculo
      - Texto, etc.

- **Strategia**:
  - **Estructura**:
    - Familia de algoritmos intercambiables
    - Se prepara una descripción genérica del algoritmo para que posteriormente se instancie con el que convenga.
    - Se prepara la ***llamada*** al algoritmo que será el mas conveniente en cada caso.
  - **Aplicaciones**:
    - Cuando preveamos distintos comportamientos en un futuro, podemos habilitarlo.
    - Estructuras de datos complejas que podrán implementarse en un futuro de otras formas.
  - **Consecuencias**:
    - Posibilidad de mejorar eficiencias y rendimientos en el futuro
    - Permitir otras estrategias de solución (más adecuadas a otros casos) distintas a la propuesta inicialmente.
    - Facilitar ampliaciones.
  - **Ejemplo**: ***el editor***:
    - La forma de presentar el texto dependerá del algoritmo usado: sin formato, con formato, HTML, LaTeX, etc.
    - Será el algoritmo concreto que se instancie para visualizar el que determinará el resultado.
    - Otras estrategias: para ordenar, clasificar, filtrar, etc.
    - Cliente:
      - El cliente usa la interfaz a la estrategia
      - La estrategia concreta es indiferente
      - Cliente -> editor
      - Estrategia -> visor HTML, LaTeX, PDF, etc.

- **Objeto compuesto**:
  - **Estructura**:
    - Se crean jerarquías de manera que se tratan igual a objetos.
    - Clases abstractas abiertas a incorporación de nuevos componentes que se tratarán igual.
  - **Aplicaciones**:

        * Allí donde se quiera simplificar la interfaz sean primitivas, sean grupos complejos de objetos.
    - **Consecuencias**:
        * Simplificación
        * Interfaz sencilla
        * Acceso uniforme
    - **Ejemplo**:
        * Dibujar: linea o rectángulo
        * Igual que Dibujar: dibujo1
        * Siendo dibujo1: linea + rectángulo + dibujo2
        * Los objetos se tratan de manera uniforme sean primitivas o grupos
    - **Otros ejemplos**:
        * Menús
        * Jerarquía de objetos: QObject, ficheros y directorios
        * Trabajadores
        * Contenedores donde cada elemento puede ser un contenedor

- **Factory**
    - **Definición**:
        * Crea el objeto pero ayudando a elegir entre varios de una misma familia.
        * Ayuda en la construcción de objetos de diferente tipo derivados de una misma clase base.
        * Selecciona fácilmente objetos de una misma familia.
        * Factory simplemente devuelve el nuevo objeto seleccionado.
    - **Solución**:
        * Definir una clase que ayuda a la selección del objeto de una familia.
    - **Conclusiones**:
        * Los patrones de diseño son de gran utilidad en el diseño del software.
        * La reutilización tambien en los "*diseños*".
        * Son cada vez más usados y tendencia actual.
    - **Ejemplo**: `factory.cc`
        * La clase `PizzaFactory` es la que ayuda en la construcción de objetos de diferente tipo derivados de la misma clase.
        * `PizzaFactory` hace la selección, en este caso en base a un parámetro.
        * Las pizzas concretas (`HamAndMushroomPizza`,`DeluxePizza` ,`HawaiianPizza`) se crean de la misma forma:

```
Pizza* pizza;
pizza = PizzaFactory::createPizza(PizzaFactory::HamMushroom);
pizza->getPrice()
...
```

```
pizza = PizzaFactory::createPizza(PizzaFactory::Deluxe);
pizza->getPrice()
...
pizza = PizzaFactory::createPizza(PizzaFactory::Hawaiian);
pizza->getPrice()
```

6. Define que 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 triada MVC.

- **Definición**: Descripciones de clases, objetos y relaciones entre si, que están especializadas en resolver un problema de diseño general en un determinado contexto.
- **Nombre**: **MVC** (***origen en el lenguaje Smalltalk-80***)
- **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 varios patrones de diseño que cooperan: observer, composite, strategy, etc.
- **Aplicaciones**:
  - Presente en casi todos los frameworks de desarrollo modernos.
  - Se puede aplicar a cualquer aplicación.
- **Consecuencias**:
  - Simplificación del desarrollo.
  - Separar desarrollos.
  - Varias presentaciones para un mismo modelo.

7. Encapsulado y ocultación de la información

- Un tipo abstracto de dato (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 utilizados, el código, etc.

- La separación de estas dos vistas es muy importante, ya que 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, tambien hace la labor del cliente más sencilla y potente.
- No solo es cuestión de facilitar su uso, tambien 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 utiliza 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 mas difícil de comprender.
- La POO usa como base la clase, que integra los siguiente conceptos:
  - Representa un objeto del mundo real.
  - Tiene atributos.
  - Tiene comportamiento.
  - Tiene entidad por si misma.
  - Tiene sentido por si misma, se entiende, se comprende facilmente.
  - Se puede reutilizar en distintos problemas.
- La POO es una 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 unicamente el qué. Informa de los detalles relevantes de datos, funciones y evita los irrevelantes.

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 la implementación.

10. Operaciones con TADs

- **Constructores**: Llevan el 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 el 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 parametros 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 que variable hace referencia. Ejemplo: `int &p = i;`.
- Se pueden pasar parametros 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 automaticamente en el punto del programa en el que se declara el objeto.
- **El constructor puede recibir parametros**.
- **El destructor** tiene igual nombre que el constructor, pero anteponiendo el caracter `~`. **No puede recibir parámetros, solo se utiliza para liberar memoria**.

14. Funciones Friend

Son **aquellas que, no siendo miembro de una clase pueden acceder a la parte privada de ella**. Se tiene 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 II

- 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:
  · **En tiempo de compilación**: Se refiere a la sobrecarga de funciones y operadores.
  · **En tiempo de ejecución**: Uso conjunto de clases derivadas y funciones virtuales.

16. ¿Que 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 facilmente estructuras estándar de datos como colas, listas y pilas.

Examen Enero 2017

1. Describe el requisito para una buena descomposición modular denominado "ocultación de la información".

- **Definición**: El diseñador de cada módulo debe seleccionar un suconjunto de propiedades del módulo comoo información oficial sobre el módulo para ponerla a disposición de los autores de módulos clientes.

- **Comentarios**: El resto de la información será privada (encapsulada, oculta) y no debe hacerse pública puesto que no es necesario que se conozca para el uso del módulo, es más, su conocimiento puede resultar contraproducente puesto que será información de bajo nivel de la cual el usuario no debe depender.

2. Indica el factor de calidad del software que no se cumple en los siguientes casos:

a) El software funciona con operaciones sencillas pero falla cuando se usa en operaciones permitidas pero mas complejas.

Intergridad (Factor externo): 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.

b) El software no puede ejecutarse nada mas que en un sistema operativo concreto.

Portabilidad (Interno/Externo): Es la capacidad o la facilidad del producto de ejecutarse en otro hardware diferente o en otro sistema operativo diferente.

Aquí es muy importante que el programa no haga uso de características de bajo nivel del hardware o que aisle la parte dependiente del hardware para que al portarlo solo sea necesario modificar dicha parte.

c) El software es atacado por piratas informáticos con facilidad.

Seguridad (Factor interno): Es la capacidad del producto de proteger sus componentes de usos no autorizados y de situaciones excepcionales de pérdida de información. Para ello debe prever mecanismos de control de acceso y también, desde el otro punto de vista copias de seguridad, procesos rutinarios de comprobación, etc.

d) La página web no está adaptada a personas con discapacidad.

Accesibilidad informática(Factor externo): Consiste en el acceso a la información sin limitación alguna por razones de deficiencia, incapacidad o minusvalía. Con el fin de poder elaborar, reproducir, manipular, etc. dicha información, así como el acceso a herramientas y opciones.

e) La interfaz de usuario es excesivamente compleja.

Facilidad de uso (Factor externo): Facilidad al introducir datos, interpretar datos, comprender errores, interpretar resultados, etc. Para usuarios con diferentes formaciones y aptitudes. Instalación, operación y supervisión. Capacidad multilingüe.

3. Para un fichero de código fuente que se llama `persona.h` escribe el código correspondiente a las guardas de inclusión que se escribe al principio del fichero y el código de dichas guardas de inclusión que se escribe al final del fichero. El resto del contenido del fichero dejalo en blanco.

```
#ifndef PERSONA_H
#define PERSONA_H

/*
    Código

*/

#endif
```

4. ¿Que es un patrón de diseño? Escribe el nombre de al menos patrones de diseño. ¿Por que en C++ conviene pasar objetos grandes como referencia constante?

Un patrón de diseño es una descripción de las clases y objetos que lo componen, y de sus relaciones entre sí. Cada patrón está especializado en resolver un problema de diseño general en un determinado contexto.

Se utiliza la referencia para evitar el proceso de copia del objeto y el consecuente gasto en memoria y tiempo de ejecución, y pasar el objeto como constante evita que este pueda ser modificado.

Examen Febrero 2017

1. Describe el requisito para una buena descomposición modular denominado "interfaces explicitas".

- **Definición**: Las interfaces deben ser obvias a partir de su simple lectura.
- **Comentarios**: La claridad del módulo, afecta en gran medida a su calidad y, por tanto, es determinante para que un módulo sea bueno. Lo mimso ocurre si un módulo debe llamar a otro, debe entenderse de forma lógica el motivo de esa llamada.

2. Indica el factor de calidad del software que no se cumple en los siguientes casos:

a) El software no hace aquello para lo que fue creado

Corrección (Factor externo): Es la capacidad del producto para realizar de forma adecuada aquello para lo que fue creado, tal y como se definió en los documentos de especificación y requerimientos.

b) El software no maneja correctamente situaciones imprevistas

Robustez (Factor externo): Capacidad del producto de manejar correctamente situaciones imprevistas, de error, o fuera de lo normal. Por ejemplo que por culpa de un programa, el SO quede bloqueado.

c) El software no proporciona resultados con muchos errores de precisión

Fiabilidad (Factor externo): Es la medida en que 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 según se especificó. Depende tambien de la corrección y la robustez.

d) El software no se combina bien con otros programas con los que sería útil que se comunicara y se combinara

Compatibilidad (Factor externo): Es la facilidad que tienen los programas de combinarse entre sí. Es la posibilidad de utilizar los datos de salida de un programa como entrada de otro programa. La clave es la estandarización y el consenso.

3. Escribe todo lo necesario para definir la clase *Persona* con un constructor que recibe como parámetros opcionales el nombre y la edad, ambos de tipo `string` , con valores por defecto igual a "XX", y los asigna a variables privadas de dicha clase. Escribelo todo en un

único fichero que se llame `persona.h` y escribe todo el código necesario desde el principio del fichero hasta el final: Todas las líneas necesarias para que compile perfectamente sin ningún error y con buena calidad del software.

```
// Fichero: persona.h

#ifndef __PERSONA_H__
#define __PERSONA_H__

class Persona{

    private:

        string nombre_, edad_, direccion_, localidad_,
        provincia_, pais_;

    public:

        Persona(string nombre="", string edad="", string dirección="XX",
        string localidad="XX", string provincia="XX", string pais="XX"){
                nombre_=nombre;
                edad_=edad;
                direccion_=direccion;
                localidad_=localidad;
                provincia_=provincia;
                pais_=pais;

        };

};

#endif
```

4. Escribe una función en C++ de tipo void que se llame "intercambia" y que reciba como parámetro dos variables usando referencias e intercambie sus valores.

```
void intercambia(int &a, int &b){

    int aux = 0;

    aux = i;
    i = a;
    a = aux;
```

```
}
```

5. ¿Por qué en C++ conviene pasar objetos grandes como referencia constante?

Se utiliza la referencia para evitar el proceso de copia del objeto y el consecuente gasto en memoria y tiempo de ejecución, y pasar el objeto como constante evita que este pueda ser modificado.

Preguntas varias y frecuentes de examenes

1. Escribe el código en C++ de una función que se llame ... y que realice relacionadas con:

- Orden de los parámetros que posee una función
- Usar referencias para el intercambio de valores de dos variables

2. Define el concepto de invariante de clase y escribe dos ejemplos, es decir, describe dos clases y el invariante de cada una. No hay que codificar las clases ni el invariante, simplemente describirlos con tus palabras. ¿Cuando es correcta una clase?

Propiedad que hace que un objeto esté bien definido, caracterizado por su utilidad y simpleza. Basicamente es una verificación práctica del estado de un objeto.

Ejemplo:

- class Fecha:
  - fecha correcta
- class Ruleta:
  - banca > 0

Una clase es correcta si y solo si su implementación es consistente con las precondiciones, postcondiciones y los invariantes.

3. Aserciones. Definición y utilidad. Escribe un breve ejemplo en C++.

- Definición: Predicado incluido en un programa el cual siempre se cumple en ese punto del flujo del programa. Una precondición (aserto al comienzo del código) determina que se espera del conjunto de sentencias que siguen sean ejecutadas y una postcondición (colocada al final) describe el estado que se espera alcanzar al final de la ejecución.

Utilidades: Son utiles para especificar programas y para razonar la corrección de los mismos.

Función `assert()`:

- Llama a la función `abort()` si el resultado es false.
- Muestra nombre del archivo fuente y linea.
- `#define NDEBUG` // evita la ejecución de assert().

Ejemplo:

```
assert (p!=0) // Si p es cero -> abort()
```

4. ¿Que características tienen las clases en las que es imprescindible el uso de copia?

- Que poseen funciones que se le pasan objetos como argumentos por valor.
- Las funciones que posee tienen un objeto como valor de retorno.

5. ¿Quienes son Dennis M. Ritchie y Richard Stallman? ¿Cuales son sus aportaciones en el área de la programación de computadoras?

- **Dennis M. Ritchie** fue un científico de la computación estadounidense. Colaboró en el diseño y desarrollo de los sistemas operativos **Multics** y **Unix**, asi como el desarrollo de varios lenguajes de programación como C.
- **Richard Stallman** es un programador estadounidense y fundador del movimiento por el software libre en el mundo. Creó el editor de texto GNU Emacs, el compilador GCC, y el depurador GDB, bajo la trayectoria del proyecto GNU. Sin embargo, es principalmente conocido por promover el software libre, como una alternativa al desarrollo y distribuición del software no libre o privativo. Tambien es inventor del concepto ***copyleft***.

6. Escribe los nombres de: el principal autor del lenguaje C, el autor del lenguaje C++, los principales autores del Sistema Operativo UNIX, del proyecto GNU, y del cofundador y presidente durante muchos años de Apple.

- C: Dennis M. Ritchie
- C++: Bjarne Stroustrup
- Unix: Ken Thompson y Dennis Ritchie
- Proyecto GNU: Richard Stallman
- Cofundador y presidente durante muchos años de Apple: Steve Wozniak