Resumen I2 IIC2113
2021-2

Por Ian Basly y Daniela Poblete

- Buenas prácticas (desde code smells)
  - Code Smells
    - Definición
    - Bloaters
    - Object-Orientation Abusers
    - Change Preventers
    - Dispensables
    - Couplers
  - Refactoring
    - ¿Qué es refactoring?
    - ¿Por qué hacer refactor?
    - ¿Cuándo hacer refactor?
  - Testing
    - ¿Qué es testing?
- Ingeniería Inversa
  - Ingeniería Inversa
    - ¿Qué es Ingeniería Inversa o retroingeniería?
  - Paradigmas de programación
    - ¿Qué son los paradigmas de programación?
- Patrones de diseño
  - ¿Qué son?
  - Tipos de patrones
  - Patrones creacionales
    - Factory method 🏭
    - Abstract Factory 🏭🏭
    - Builder 🏗️
    - Prototype 🦾
    - Singleton ❄️
  - Patrones estructurales
    - Adapter 🔌
    - Bridge 🌉
    - Composite 🌿
    - Decorator 🎀
    - Facade 📱
    - Flyweight 🗄️
    - Proxy 🔗
  - Patrones de comportamiento
    - Interpreter 💬

        - Template Method 📑
        - Chain of Responsibility ⏩
        - Command 📦
        - Iterator ♻️
        - Mediator ⚖️
        - Memento 📌
        - Observer 🔎
        - State 🎛️
        - Strategy 🗂️
        - Visitor 🧳

- Patrones de arquitectura
    - Arquitectura de software
    - ¿Qué son los patrones de arquitectura?
    - Principios
    - Patrones
        - Modelo - Vista - Controlador (MVC)
        - Blackboard
        - Cliente - Servidor
        - Microservicios y monolito
        - Big ball of mud

Buenas prácticas (desde code smells)

1. Code Smells

- Definición

  "*A code smell is a surface indication that usually corresponds to a deeper problem in the system*" Martin Fowler.

- Existen 5 tipos:

  - *Bloaters*: Representan código, métodos y clases que han crecido a tal punto que es difícil trabajar con ellos.

  - *Object-Orientation Abusers*: Implementación incompleta o incorrecta de los principios de programación orientados a objetos.

- *Change Preventers*: Reflejan un posible problema si un cambio en una parte del código implica varios cambios en otras secciones, aumentando el costo de desarrollar.

- *Dispensables*: Fragmentos de código que son innecesarios e inútiles, además su ausencia haría que el código fuese más claro, eficiente y fácil de entender.

- *Couplers*: Exceso de acoplamiento en el código.

- *Bloaters*

Nombre	Síntomas	Razones	Soluciones
Long Method	Método con muchas líneas (10 o más).	Quién escribe el código lo mantiene, "es más fácil" agregar líneas a un método existente que crear nuevos.	Descomponer en más funciones, reducir variables y parámetros o mover el método a su propio objeto.
Large Class	Clase con muchas líneas, atributos y/o métodos.	Las clases crecen en conjunto con la aplicación con baja cohesión. "Es más fácil" agregar líneas a un método existente que crear nuevos.	Descomponer en múltiples clases con asociación o composición o usando herencia.
Primitive Osession	Uso de variables primitivas en vez de objetos pequeños o de constantes para guardar información.	Se agregan atributos a una clase a medida que se necesitan, sin considerar que se pueden agruparse en un objeto. Mala elección de tipos primitivos para representar una estructura de datos.	Convertir conjuntos de variables en objetos pequeños.
Long Parameter List	Más de 3 o 4 parámetros por método. Es difícil entender listas extensas de parámetros.	Se agrupa funcionalidad en un sólo método. Se pasan parámetros a un método directamente, y no a través de objetos que contienen la información o llamar a otros métodos para obtenerlos.	Agrupar conjuntos de variables en objetos pequeños. Conviene ignorarlo si la separación implicaría mucha interdependencia entre los nuevos objetos creados.
Data Clumps	Diferentes partes de una aplicación contienen una definición recurrente de variables.	Una pobre estructura de la aplicación, o copy-paste programming. No se agrupan valores inseparables en un objeto.	Agrupar conjuntos de variables en objetos pequeños. Conviene ignorarlo si se generan dependencias no deseadas entre clases.

- Object-Orientation Abusers

Nombre	Síntomas	Razones	Soluciones
Switch Statements	Se tienen múltiples bloques switch complejos, o grandes secuencias de if/else.	Mal uso de polimorfismo.	Re-organizar código (if/else o switches en otros métodos o clases). Usar polimorfismo si hace sentido.
Temporary Field	Se tienen atributos de una clase que solo tienen valor bajo ciertas circunstancias, estando el resto del tiempo sin definir o nulos.	Se usan atributos de clase en vez de pasar los datos como parámetros.	Quitar los atributos temporales en una clase relacionada (que si los use).
Refused Bequest	Una subclase utiliza algunas propiedades y métodos de sus padres.	Se quiso reutilizar código entre una superclase y una clase, pero son completamente distintas.	Reemplazar la herencia por delegación o asociación. Extraer los métodos comunes del padre y el hijo, para pasarlo a otra clase de la cual ambas clases hereden.
Alternative Classes with Different Interfaces	Dos clases realizan funcionalidades idénticas, pero con nombres de métodos distintos.	Desconocimiento por parte del programador sobre la existencia de la otra clase.	Cambiar los nombres de los métodos, ajustar los parámetros y variables para poder reusar la misma clase y borrar el clon. Si solo una parte está duplicada, crear una superclase para hacer herencia.

- Change Preventers

Nombre	Síntomas	Razones	Soluciones
Divergent Change	Cambios en la clase desencadenan cambios en métodos que no tienen relación entre sí.	Estructura poco pensada. Copypasta programming.	Extraer el comportamiento de las clases si es algo estándar. Combinar clases a través de herencia. Muchas veces se solucionan de forma simple re-ordenando el código dentro de la misma clase.
Shotgun Surgery	Cambios en la clase desencadenan cambios en otras clases.	La responsabilidad fue dividida entre varias clases.	Agrupar la responsabilidad en una sola clase: una ya existente o crear una nueva. Quitar las clases redundantes.
Parallel Inheritance Hierarchies	Hay exceso de dependencia en el comportamiento: Añadir una subclase A a una clase B, implica añadir una subclase C a una clase D.	Cuando las jerarquías son pequeñas, las modificaciones necesarias se tienen bajo control. Pero a medida que el árbol de jerarquías crece, se hace cada vez más difícil realizar cambios.	Combinar las clases en la jerarquía de ser posible.

- Dispensables

Nombre	Síntomas	Razones	Soluciones
Comments	Un método contiene excesivos comentarios.	La intención de los comentarios es buena. Sin embargo más que un comentario explicando la estructura, la verdadera solución es que la estructura sea buena y entendible por sí misma.	El mejor comentario es un buen nombre de clase o método. Re-estructurar el código para que sea entendible por sí mismo.
Código duplicado	Fragmentos idénticos de código.	Puede ser explícita: mismo código repetido en distintas partes. O implícita: dos funciones distintas que cumplen el mismo propósito.	Si es código repetido, usar un módulo común para importar esa clase, método, etc. Si son dos funciones distintas, elegir solo una forma y unificar ese método. En general hay varias formas de re-estructurar el código duplicado.
Lazy Class	Una clase no cumple un rol que realmente justifique su existencia.	Cambios en la estructura del código han generado que la clase deje de ser importante. Mala planificación: posibles funcionalidades que nunca se implementaron. Uso de generadores.	Agrupar el código de acuerdo a las necesidades del problema. Si no se programará en ese momento, mejor no considerarlo.
Data Class	Una clase contiene solamente atributos, sin aplicar comportamiento.	Cambios en la estructura del código que han implicado quitarle métodos. Mala planificación.	Evaluar si efectivamente no tiene comportamiento: ¿Todos los atributos son públicos o privados? ¿Deberían haber getters y setters?.
Dead Code	Una variable, parámetro, campo, método o clase ya no se utiliza.	Los requerimientos cambian, y no se limpia el código. Lógica de condicionales inaccesible.	Limpiar el código borrando el código que no se necesite.
Speculative Generality	Una variable, parámetro, campo, método o clase que no se utiliza, pero se planea usar.	Mala planificación por posibles funcionalidades. Generalizar lógica de código que no es necesaria.	Evitar usar clases abstractas si es que no se tiene claro que más de alguna clase la usara.

- Couplers

Nombre	Síntomas	Razones	Soluciones

Feature Envy	Un método utiliza más información de un objeto que recibe, que del mismo objeto en el que está definido.	Mala modelación. Se crean clases para almacenar datos, pero los métodos que los modifican se crean en otras partes del código.	Mover los métodos a las clases que modifican.
Inappropriate Intimacy	Una clase utiliza métodos y atributos internos de otra clase.	Clases muy ligadas entre sí.	Mover el código donde se usa realmente. Hacer oficial la relación entre las clases (ya sea con composición o herencia). Hacer que la asociación sea solo en una dirección y no cíclica.
Message Chains	Excesivas llamadas a métodos en cadena: a->b()->c()->d().	Dependencia de quien hace la primera llamada sobre el flujo o la estructura.	Mover los métodos y re-definir las responsabilidades. Mover el código importante al inicio de la cadena.
Middle Man	La única funcionalidad de la clase es delegar trabajo sin agregar funcionalidad.	Al reorganizar código puede que haya quedado una clase sin responsabilidades.	Quitar el Middle Man. No quitar si se añadió el Middle Man para evitar dependencias entre clases o si es parte de algún patrón de diseño.
Incomplete Library Class	Una librería presenta problemas y no responde a las necesidades del problema.	Una librería ya no provee las funcionalidades necesarias o tiene un problema de compatibilidad, y no se actualiza en mucho tiempo.	Utilizar soluciones intermedias para hacer compatible la librería. Extraer las funcionalidades importantes de la librería.

2. Refactoring

- ¿Qué es refactoring?

  Es el proceso de cambiar un sistema de software de tal forma que no se altera el comportamiento externo, pero que **mejora la estructura interna del código.**
- Testing: La forma más eficiente de hacer refactoring es teniendo buenos tests acompañando al código.

- ¿Por qué hacer refactor?

  a. Mejora el diseño del software.
  b. Hace el código más fácil de entender.
  c. Ayuda a encontrar bugs.
  d. Ayuda a ganar experiencia y al equipo a programar más rápido.

- ¿Cuándo hacer refactor?

    1. Rule of three: Three strikes and you refactor
    2. Cuando añadas algo nuevo → considerando el ROI.
    3. Cuando corriges bugs.
    4. Cuando tu código vaya a pasar por code review.

- Metáfora de los “Dos sombreros”

  Un desarrollador debería dividir su tiempo en 2 actividades: añadir funcionalidades y hacer refactoring:
    - ○ Dentro de las funcionalidades, solo se añaden nuevas funciones y tests para medir el avance.
    - ○ Cuando se hace refactor, no se añaden más tests, sólo se re-estructura el código.

  Durante el desarrollo, el desarrollador estará cambiando de sombreros de forma constante. Sin embargo es importante para el mismo percatarse cual sombrero tiene puesto.

- Problemas

    - Managers poco técnicos no le ven el valor.
    - No hacer bien el refactoring puede causar cambios en el comportamiento (importancia de los tests).
    - Hacer refactoring sobre código “no estable” trae más problemas que beneficios.
    - Hacer refactoring encima de un deadline puede traer complicaciones.

3. Testing

- ¿Qué es testing?

  Son pruebas que permiten validar y verificar el funcionamiento del software.

  Validación: ¿Estamos construyendo el producto correcto?
  Verificación: ¿Estamos construyendo el producto correctamente?

Artefacto	Tipo de test
Código	Unit testing
Diseño de interfaz	Integration testing
Requerimientos	Validations testing
Sistema	System testing

En la tabla se muestra el orden en que debería hacerse el testing.

Ingeniería Inversa

1. Ingeniería Inversa

- ¿Qué es Ingeniería Inversa o retroingeniería?

  Proceso de analizar la estructura, funcionamiento, características y, en general, los fundamentos técnicos de un sistema o dispositivo ya sea mecánico o electrónico, e incluso un programa computacional. El término se puede aplicar tanto a productos de software, hardware, y hoy en día a todo tipo de productos. Este proceso se realiza porque no se tiene acceso a las especificaciones ni documentación del producto.

- Nivel de extracción:
  Se refiere a la sofisticación del diseño extraído de un software. A mayor nivel, más información se puede extraer. De mayor a menor nivel:
    - Diseño de procesos
    - Diseño estructural
    - Modelo del sistema
    - Relación entre componentes

- Completitud:
  Dado un nivel de extracción, se refiere a la cantidad de detalle que se puede obtener. Generalmente es inversamente proporcional al nivel de extracción.

- Ventajas:
    - Reducir la complejidad del sistema
    - Recuperar o actualizar información
    - Identificar el alcance de un producto
    - Facilitar la reutilización

● Análisis de vulnerabilidades

2. Paradigmas de programación

- ¿Qué son los paradigmas de programación?

  Son una forma de clasificar los lenguajes de programación en base a sus funcionalidades. Se pueden clasificar en base a sus modelos de ejecución, a cómo está organizado el código o sobre el estilo/gramática. Los más comunes son:

  Imperativa: el programador indica cómo cambian los estados.

  - Programación por procedimientos (procedural).
  - Orientado a objetos (OOP).

  Declarativa: el programador declara propiedades del resultado, pero no los computa.

  - Programación Funcional.
  - Programación Lógica.
  - Programación Reactiva.
  - Optimización (mathematical programming)

1. Programación por procedimientos:

   Puede darse a alto o bajo nivel. Cuando se aplica como técnica a alto nivel se le conoce como programación funcional.

2. Prog. Orientada a Objetos:

   Los objetos emulan una instancia real del modelo del problema que se está resolviendo. Se basa en herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

   Duck typing hace referencia a los programas escritos en algún lenguaje orientado a objetos principalmente, en que los objetos pasados a una función o método soporta todo tipo de atributos en tiempo de ejecución.

   A diferencia del tipado estático, duck typing es dinámico y solo verifica el tipo en el momento de ser accedido.

3. Programación Reactiva:

Declarativo orientado al manejo asíncrono de streams de datos y la propagación de los cambios que introducen. Facilita la comunicación entre múltiples threads.

4. Programación multiparadigma:

   En general los lenguajes de alto nivel tienden a soportar más de un paradigma. De hecho, a pesar de que un lenguaje en su core no soporte algún paradigma, suelen existir librerías y/o frameworks que permiten a distintos lenguajes soportarlo. Por ejemplo hoy en día muchos lenguajes soportan programación reactiva gracias a librerías.

   Entonces da lo mismo cuál lenguaje use, si todo se puede importar → No.

   Dependerá de:
   - El tipo de implementación del lenguaje.
   - El tipo de soporte que entrega al paradigma que necesitas.
   - Al final los lenguajes de propósito general tienen sus ventajas y desventajas.

5. Domain Specific Languages (DSL):

   Si nos enfocamos en un modelo en particular, pueden existir lenguajes que representen de mejor y única
   manera ese modelo. Un Lenguaje Específico de Dominio (DSL) es un lenguaje de programación o
   especificación orientada a resolver problemáticas en un dominio muy acotado.

   ● DSL Interno (Embedded DSL): Se escriben dentro de un lenguaje existente que lo contiene.

   ○ Rake y Rspec son DSL internos de Ruby.
   ○ .NET también ofrece distintos tipos de DSL y herramientas de Extensibility.
   ○ Es común escribir DSL internos para problemas particulares, con tal de tener interfaces fluidas.

   ● DSL Externo: Tienen su propio lenguaje, pero necesitan ser interpretados por otros.

   ○ SQL, HTML, CSS, Sass, XML, UML, Latex.

   - Ventajas frente a los lenguajes de propósito general:

     ● Tienen mayor legibilidad
     ● Tienen menor verbosidad
     ● Su uso implica menor probabilidad de error

● Pueden ser analizados y usados por especialistas del dominio sin experiencia en programación

- Desventajas:

  ● Overhead extra para aprender
  ● Aumenta la complejidad
  ○ Diseñar un DSL interno puede ser complejo

Patrones de diseño

¿Qué son?

Son descripciones hacia objetos y clases personalizadas para resolver un problema general de diseño, pero determinado por un contexto específico.

Tipos de patrones

Los patrones pueden apuntar a solucionar problemas según un propósito (creacional, estructural o de comportamiento) y dentro de un scope determinado (clase u objeto).

		Propósito		
		Creacional	**Estructural**	**De comportamiento**
Scope	**Clase**	● Factory Method	● Adapter	● Interpreter ● Template Method
	Objeto	● Abstract Factory ● Builder ● Prototype ● Singleton	● Adapter ● Bridge ● Composite ● Decorator ● Facade ● Flyweight ● Proxy	● Chain of Responsibility ● Command ● Iterator ● Mediator ● Memento ● Observe ● State ● Strategy ● Visitor

Patrones creacionales

1. Factory method 🏭

   Scope: Clase

   Se define una interfaz o clase abstracta para la creación de objetos, pero son las subclases quienes deciden qué clase específica instanciar. (Como una fábrica 😉)

   Se utiliza cuando hay incertidumbre respecto a qué clases se deben instanciar o simplemente cuando las subclases son quienes se quiere que instancie los objetos.

2. Abstract Factory 🏭🏭

   Scope: Objeto

   A diferencia del patrón anterior, se define una interfaz para creación de familiar objetos que poseen relaciones, pero delegando la decisión de qué clases instanciar a las subclases.

   Se utiliza principalmente cuando un sistema debe ser independiente de los productos que crea, cuando se quiere forzar la utilización de familias de productos relacionadas en conjunto o cuando se quiere proveer de una librería de familias de productos pero sin revelar su implementación.

3. Builder 🏗️

   Scope: Objeto

   Se separa el proceso de construcción de un objeto en diferentes componentes (objetos) que permite la creación de diferentes representaciones del objeto.

   Se utiliza cuando el algoritmo para crear objetos es independiente de las partes del objeto y cómo se construyen o cuando se quiere permitir la diferentes representaciones del objeto.

4. Prototype 🦾

   Scope: Objeto

   Se definen los tipos de objetos a utilizar mediante prototipos (clases) y los nuevos objetos se crean como clones de prototipos preexistentes.

   Se utiliza cuando las clases necesarias se especifican durante la ejecución del programa, cuando se quiere crear objetos similares que es más sencillo desprender de los existentes o cuando los objetos de una clase tienes estados limitados (no

pueden variar demasiado), entonces se instancia una vez cada estado para utilizar copias de estos.

5. Singleton ❄️

   Scope: Objeto

   Es cuando se define una clase que únicamente puede ser instanciada una vez en todo el programa.

Patrones estructurales

1. Adapter 🔌

   Scope: Objeto y Clase

   Se define una interfaz para una clase que permita al cliente utilizar esta clase según el formato necesario. Además permite la interacción entre clases que originalmente no podrían interactuar.

   Se utiliza cuando se quiere utilizar una clase cuya interfaz es incompatible con los requerimientos del sistema.

2. Bridge 🌉

   Scope: Objeto

   Se busca desacoplar una abstracción de su implementación. Similar a builder la idea es que se pueda alterar la abstracción sin modificar la implementación o viceversa.

   Se utiliza cuando no se quiere unir permanentemente la abstracción con su implementación, o cuando se pretende que la abstracción e implementación sean extensibles o donde los cambios en alguna de ellas no tenga impacto en el otro.

3. Composite 🌿

   Scope: Objeto

   Los objetos se componen de objetos formando una jerarquía de árbol (como gráfos). De esta forma cada objeto puede ser tratado como una sub-jerarquía (objeto compuesto) o como un mismo objeto (objeto individual).

   Se utiliza cuando se requiere generar una jerarquía completa de objetos o cuando se busca que el cliente trate objetos compuestos y objetos individuales de igual forma.

4. Decorator 🎀
   Scope: Objeto

   Se busca agregar comportamientos adicionales a objetos de forma dinámica. Además de ser una alternativa flexible para agregar funcionalidad a subclases sin tener que re-definir.

   Se utiliza cuando se quiere agregar responsabilidades a objetos sin alterar los existentes, cuando se quiere agregar responsabilidades removibles o cuando la extensión mediante subclases es impracticable.

5. Facade 📱
   Scope: Objeto

   Se provee una interfaz unificada para operar interfaces dentro de un sub-sistema. Se busca crear una interfaz de alto nivel que sea más fácil de utilizar.

   Se utiliza cuando se quiere proveer una interfaz simple a un sistema complejo o cuando se quiere un sistema con sus sub-sistemas ordenados en capas accesibles mediante interfaces facade.

6. Flyweight 🗄️
   Scope: Objeto

   Utiliza valores por referencia para soportar grandes cantidades de objetos.

   Se utiliza principalmente cuando se tiene un gran número de objetos, los costos de almacenamiento (memoria) de los objetos son muy caros o cuando la aplicación no depende de la identidad exacta de los objetos.

7. Proxy 🔗
   Scope: Objeto

   Se provee una interfaz sustituta para alterar un objeto original.

   Se utiliza cuando se requiere una referencia más sofisticada que un puntero.

Patrones de comportamiento

1. Interpreter 💬
   Scope: Clase

Utilizando un lenguaje dado, el la clase interpreter define una representación de su gramática junto a un intérprete que utiliza dicha representación para interpretar frases en el idioma.

Se utiliza cuando la gramática del lenguaje es simple y no importa la eficiencia del programa.

2. Template Method 📑

Scope: Clase

Se define la estructura de un algoritmo delegando partes de este a las subclases. De esta forma se puede personalizar el algoritmos en los pasos establecidos sin alterar la estructura general.

Se utiliza para evitar la duplicación de código e implementar la parte común del algoritmo una única vez y dejando espacio a que cambien algunos pasos.

3. Chain of Responsibility ⏩

Scope: Objeto

Se encadena una secuencia de llamados al manejo de una consulta a través de varios objetos, donde alguno de ellos podría hacerse cargo, de esta forma se evita el acoplamiento entre el objeto emisor y el objeto receptor.

Se utiliza cuanto más de un objeto podría encargarse de la solicitud, pero no se sabe cual con exactitud o cuando el conjunto de objetos que podrían procesar la solicitud se define dinámicamente.

4. Command 📦

Scope: Objeto

Se encapsula una solicitud como un objeto para poder parametrizar diferentes solicitudes así como alterarlas de ser necesario.

Se utiliza cuando se necesita parametrizar objetos con una acción a realizar, cuando se quiere especificar, encolar y ejecutar acciones en momentos diferentes.

5. Iterator ♻️

Scope: Objeto

Permite acceder a un conjunto agregado de objetos sin necesariamente exponer su representación original.

Se utiliza cuando se quiere recorrer objetos agregador más de una vez o para proveer una interfaz uniforme para recorrer estructuras agregadas.

6. Mediator ⚖️

   Scope: Objeto

   Se define un objeto que contiene la lógica de interacción entre un conjunto de objetos. De esta forma se evita el acoplamiento al no referenciarlos directamente, sino que se comunican a través del objeto mediator. También permite modificar estas interacciones de forma independiente.

   Se utiliza cuando un conjunto de objetos se comunican de forma compleja pero bien definida o un comportamiento entre varias clases debe ser configurable sin generar muchas subclases.

7. Memento 📌

   Scope: Objeto

   Se captura y guarda el estado de un objeto para poder restablecerlo más tarde. Esto sin inclumplir el principio de ocultamiento.

   Se utiliza cuando se quiere restaurar un estado de un objeto o cuando una interfaz directa para obtener el estado expondría detalles de implementación, lo cual viola el principio de ocultamiento.

8. Observer 🔎

   Scope: Objeto

   Se define una relación entre uno (sujeto) y varios objetos (oyentes) mediante el observer, para notificar a los objetos oyentes cuando el objeto sujeto ha cambiado de estado.

   Se utiliza cuando un objeto depende de otro y se quiere desacoplar encapsulando estos aspectos en objetos diferentes comunicados por el observer. También cuando se busca que cambios en un objeto afecten a varios otros, sin saber cuántos.

9. State 🎛️

   Scope: Objeto

   Permite a un objeto cambiar su comportamiento cuando cierto estado cambia. Se define el estado como objeto, lo que permite encapsular la lógica correspondiente a cada variación en un objeto distinto.

Se utiliza cuando se quiere cambiar el comportamiento del objeto dependiendo del estado en tiempo de ejecución. También cuando el comportamiento de un objeto se vuleve complejo y dependiendo de varias condiciones, donde State permite separarlas en objetos distintos con sus lógicas respectivas.

10. Strategy 🗂️

    Scope: Objeto

    Se define una familia de algoritmos a través de un único objeto que permite intercambiar su comportamiento según se necesite.

    Se utiliza cuando se quiere diferenciar las variantes de un algoritmo o cuando se quiere ocultar estructuras complejas utilizadas por un mismo algoritmo o cuando una clase posee muchos comportamientos dependientes de condiciones.

11. Visitor 🧳

    Scope: Objeto

    Representa una operación que sea ejecutada en los elementos que componen la estructura de un objeto. Visitor permite definir una nueva operación sin cambiar las clases de los elementos en los cuales se aplica.

    Se utiliza cuando se tiene muchos objetos de diferentes clases dentro de una misma estructura y se quiere ejecutar operaciones entre estos objetos independiente de la clase.

Patrones de arquitectura

Arquitectura de software

La arquitectura de software brinda una visión general de lo que se busca lograr, haciendo referencia a las estructuras, interfaces y comportamientos sin entrar en detalles de implementación. También vela por los atributos de calidad del sistema, definiendo las prioridades de implementación.

¿Qué son los patrones de arquitectura?

Son el resultado de una serie de decisiones que llevan a un patrón común para resolver cierto problema de forma general.

Principios

- Encapsula lo que cambia
- Prefiere composición sobre herencia
- Programa interfaces, no implementaciones
- Apunta a diseñar interacciones entre objetos con bajo acoplamiento
- Las clases deben estar abiertas para extensión pero cerradas para modificación

Patrones

Modelo - Vista - Controlador (MVC)

Como el nombre lo indica este patrón se divide en 3 componentes:

- El modelo que contiene toda la información.
- Los controladores que se encargan de recibir las solicitudes y procesar la información.
- Por último las vistas que se encargan de mostrar la información al usuario.

Blackboard

Se utiliza en modelos no determinísticos, ya que acá todos los componentes pueden agregar objetos al “pizarrón”, el cual contiene las soluciones de diferentes fuentes.

Cliente - Servidor

En este patrón existe un servidor con la información centralizada, donde uno o más clientes se alimentan de él.

Microservicios y monolito

- **Microservicios:** Es un sistema compuesto por varios subsistemas o procesos donde cada uno cumple una función específica y se comunica con los demás.
- **Monolito:** Es un sistema donde la lógica, datos e interfaz se encuentran en un mismo proceso.

Big ball of mud

En un sistema descontrolado, el cual ha crecido sin supervisión, es decir, sin un diseño determinado y sin planificación.

CLEAN

¿Qué es?

CLEAN es un término propuesto por el co-autor del manifiesto ágil, el cual hace referencia a algunas buenas prácticas enfocadas en mantener un código “limpio”.

Algunas de estas prácticas son:

- Nombres con sentido
- The Scout Rule
- Funciones simples, claras y cortas
- DRY (Don’t Repeat Yourself)
- Tratar las excepciones y errores