Universidad César Vallejo

Pregrado

SESIÓN 07:
GRASP parte 1: - Experto en información

Patrones GRASP

- Los patrones GRASP (General Responsibility Assignment Software Patterns) son un conjunto de principios y patrones de diseño utilizados en la ingeniería de software para asignar responsabilidades adecuadas a las clases y objetos en un sistema.
- Estos patrones proporcionan pautas prácticas para la asignación de responsabilidades, ayudando a lograr un diseño más robusto, flexible y fácil de mantener.

Patrones GRASP

- **Expert (Experto):** Asigna la responsabilidad a la clase que tiene la información y el conocimiento necesarios para realizar una determinada tarea.
- **Creator (Creador):** Define qué objeto debe ser responsable de la creación de instancias de otro objeto, evitando acoplamientos innecesarios.
- **Controller (Controlador):** Designa una clase o un objeto como punto de entrada centralizado para coordinar y controlar las operaciones de un sistema.
- **Low Coupling (Bajo Acoplamiento):** Promueve la independencia y la modularidad del sistema al minimizar las dependencias entre las clases o los módulos.

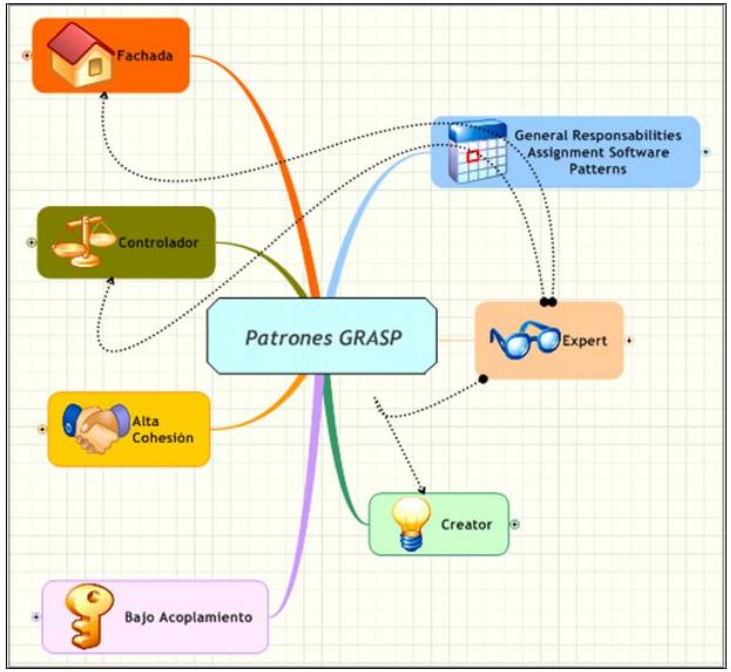
Patrones GRASP

- **High Cohesion (Alta Cohesión):** Agrupa las responsabilidades relacionadas de manera que los miembros de un mismo grupo tengan fuertes vínculos internos y realicen tareas relacionadas.
- **Polymorphism (Polimorfismo):** Utiliza el polimorfismo para permitir que objetos de diferentes clases respondan a un mensaje de manera específica, lo que aumenta la flexibilidad y extensibilidad del sistema.
- **Indirection (Indirección):** Introduce un intermediario o un objeto de enlace para desacoplar clases o componentes que tienen dependencias directas.

Patrones GRASP

Estos patrones proporcionan un marco conceptual para abordar el diseño orientado a objetos y son ampliamente utilizados en el desarrollo de software.

Sin embargo, es importante tener en cuenta que la elección y aplicación de los patrones GRASP dependerá del contexto y los requisitos específicos del proyecto.

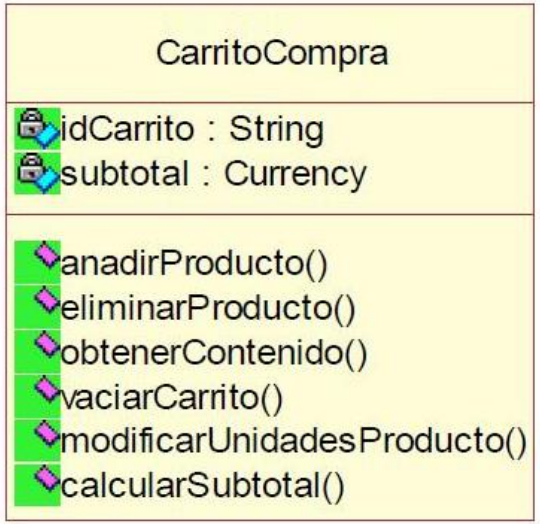


Patrón Experto

- El patrón "Experto" (Expert en inglés) se utiliza para asignar la responsabilidad de una determinada tarea o información a la clase que posee el conocimiento necesario para realizarla de manera adecuada.
- En otras palabras, este patrón busca asignar la responsabilidad a la clase que tenga la información más relevante y el contexto adecuado para llevar a cabo una operación específica.
- El principio subyacente del patrón Experto es que una clase o objeto debe ser responsable de realizar una tarea si tiene la mayor cantidad de información y conocimiento necesarios para llevarla a cabo de manera coherente y eficiente.
- Al asignar la responsabilidad a la clase experta, se promueve un diseño más cohesivo y se minimiza el acoplamiento entre clases.

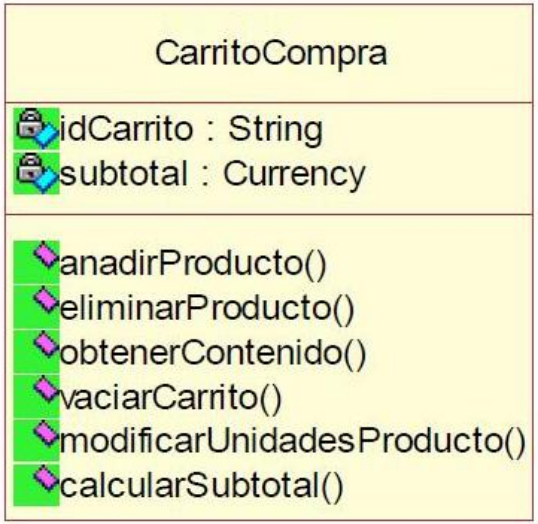
Patrón Experto

- Algunos ejemplos de aplicación del patrón Experto podrían ser:
  - Asignar la responsabilidad de cálculos matemáticos complejos a una clase especializada en matemáticas.
  - Designar una clase de "Carrito de compras" como el experto para agregar, eliminar y calcular el total de los productos en un sistema de comercio electrónico.
  - Asignar la responsabilidad de validaciones de datos a una clase especializada en verificaciones y validaciones.



Patrón Experto

En resumen, el patrón Experto busca asignar la responsabilidad a la clase que posee la información y el conocimiento necesario para realizar una tarea específica de manera experta. Esto ayuda a lograr un diseño más coherente, modular y de fácil mantenimiento.

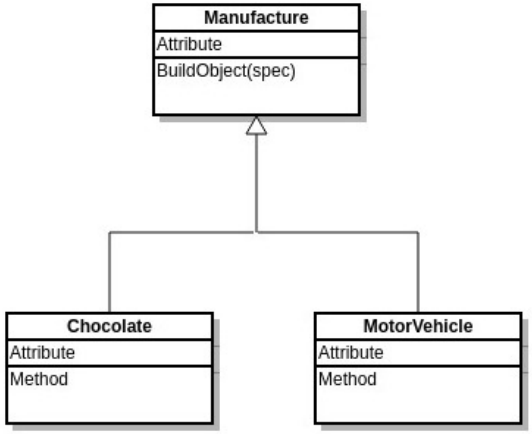


Patrón Creador

- El patrón "Creador" (Creator en inglés) se utiliza para determinar qué objeto o clase debe ser responsable de la creación de instancias de otro objeto.
- Este patrón ayuda a evitar acoplamientos innecesarios y asegura que la responsabilidad de la creación de objetos se asigna adecuadamente.
- El principio subyacente del patrón Creador es que una clase debe ser responsable de crear instancias de otras clases cuando tenga la información necesaria para hacerlo de manera adecuada.
- En lugar de que un objeto externo cree instancias directamente, se asigna esta responsabilidad a una clase específica.

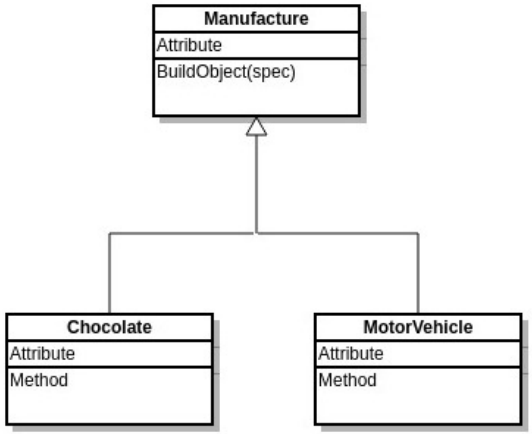
Patrón Creador

- Algunos ejemplos de aplicación del patrón Creador podrían ser:
  - Una clase "Factory" que se encarga de crear y retornar instancias de diferentes tipos de objetos según los parámetros proporcionados.
  - Una clase "Builder" que se utiliza para crear objetos complejos paso a paso, permitiendo configuraciones flexibles.
  - Una clase "Controller" que es responsable de crear y gestionar instancias de diferentes objetos relacionados en un sistema.



Patrón Creador

- Al utilizar el patrón Creador, se promueve un diseño más flexible, modular y con menor acoplamiento entre las clases.
- Además, este patrón permite delegar la responsabilidad de creación de objetos a una clase específica, evitando que otras clases tengan que conocer los detalles internos de la creación de instancias.



Patrón Bajo Acoplamiento

- El patrón GRASP "Bajo Acoplamiento" (Low Coupling en inglés) se enfoca en reducir las dependencias y el acoplamiento entre clases y componentes en un sistema de software.
- El acoplamiento se refiere al grado en que una clase o módulo depende de otros para su funcionamiento, y un bajo acoplamiento busca minimizar estas dependencias para lograr un diseño más flexible, mantenible y extensible.
- El principio fundamental del patrón Bajo Acoplamiento es que las clases y componentes deben depender lo menos posible de otros elementos.
- Al reducir las dependencias, se facilita la modificación y evolución de las clases de forma aislada, sin afectar a otras partes del sistema.
- Además, se promueve la reutilización de componentes, ya que estos pueden ser extraídos y utilizados en diferentes contextos sin tener fuertes dependencias con otros elementos.

Patrón Bajo Acoplamiento

- Algunas técnicas para lograr un bajo acoplamiento son:
  - Abstracción: Utilizar interfaces, clases base o contratos para interactuar con otras clases, en lugar de depender directamente de implementaciones concretas.
  - Inyección de dependencias: Permitir que las dependencias sean proporcionadas externamente, en lugar de crearlas internamente en una clase. Esto facilita la configuración y reemplazo de las dependencias.
  - Eventos o mensajes: Utilizar eventos o mensajes para comunicarse entre componentes en lugar de depender directamente de instancias de otras clases.

Patrón Bajo Acoplamiento

- Algunas técnicas para lograr un bajo acoplamiento son:
  - Principio de Responsabilidad Única (SRP): Asegurarse de que cada clase o componente tenga una única responsabilidad, lo que ayuda a reducir las dependencias y facilita el mantenimiento y la comprensión del código.
  - Uso de interfaces: Depender de interfaces en lugar de implementaciones concretas, lo que permite la sustitución de las implementaciones sin afectar a otras clases que las utilizan.
- Al aplicar el patrón Bajo Acoplamiento, se obtiene un sistema más flexible, fácil de modificar y adaptar a cambios futuros.
- Además, el código se vuelve más modular y reutilizable, ya que las clases y componentes pueden ser aislados y utilizados en diferentes contextos sin introducir efectos colaterales inesperados.
- En resumen, el patrón Bajo Acoplamiento busca minimizar las dependencias y el acoplamiento entre clases y componentes, promoviendo un diseño modular, flexible y mantenible.

Patrón Alta Cohesión

- El patrón "Alta Cohesión" (High Cohesion en inglés) se refiere a la organización de las responsabilidades de una clase de manera que las tareas relacionadas estén agrupadas y exista una fuerte conexión interna entre los miembros de ese grupo.
- El objetivo es lograr que cada clase o componente tenga una única responsabilidad claramente definida y que sus miembros trabajen juntos de manera coherente y orientada a un propósito común.
- El principio fundamental del patrón Alta Cohesión es que una clase debe tener una alta coherencia interna y su funcionalidad debe estar enfocada en una tarea o responsabilidad específica.
- Esto ayuda a mejorar la claridad del código, facilita su mantenimiento y reduce los efectos colaterales al realizar cambios.

Patrón Alta Cohesión

- Algunas formas de lograr una alta cohesión son:
  - **División en clases:** Separar las responsabilidades relacionadas en diferentes clases, de modo que cada clase se ocupe de un aspecto específico.
  - **Métodos coherentes:** Agrupar los métodos relacionados en la misma clase, de manera que trabajen juntos para cumplir con una responsabilidad común.
  - **Principio de Responsabilidad Única (SRP):** Asegurarse de que cada clase tenga una única razón para cambiar, es decir, una única responsabilidad.
  - **Encapsulación:** Ocultar la implementación interna de una clase y proporcionar una interfaz clara y coherente para interactuar con ella.
  - **Comunicación efectiva:** Garantizar una comunicación adecuada entre las clases, permitiendo que interactúen de manera coherente y enfocada en su objetivo común.

Patrón Alta Cohesión

- La alta cohesión facilita la comprensión del código, mejora la mantenibilidad y promueve la reutilización, ya que las clases se vuelven más autónomas y menos dependientes de otras partes del sistema.
- Además, permite cambios más controlados y reduce el impacto de modificaciones en otras partes del código.
- En resumen, el patrón Alta Cohesión busca agrupar las responsabilidades relacionadas en una clase y promover una fuerte conexión interna entre sus miembros. Esto conduce a un diseño más claro, mantenible y adaptable, mejorando la calidad y la eficiencia del código.

Patrón Controlador

- El patrón "Controlador" (Controller en inglés) se utiliza para designar una clase o un objeto como punto de entrada centralizado para coordinar y controlar las operaciones de un sistema.
- El Controlador actúa como un intermediario entre las diferentes clases y componentes, facilitando la comunicación y la coordinación de las acciones.
- El objetivo del patrón Controlador es separar la lógica de coordinación y control del sistema de la lógica específica de las clases individuales.
- Esto ayuda a mantener un bajo acoplamiento y una alta cohesión en el diseño, al tiempo que permite un manejo más eficiente de las interacciones y flujos de trabajo del sistema.

Patrón Controlador

- Algunas características y responsabilidades del Controlador son:
  - **Punto de entrada:** El Controlador sirve como punto de entrada principal para las solicitudes y eventos del sistema. Recibe las solicitudes y las dirige a las clases y componentes correspondientes.
  - **Coordinación de acciones:** El Controlador se encarga de coordinar y orquestar las acciones necesarias para cumplir con una solicitud o evento. Puede interactuar con diferentes clases y componentes para lograrlo.
  - **Manipulación de flujo de trabajo:** El Controlador gestiona el flujo de trabajo del sistema, decidiendo qué acciones se deben ejecutar en función de la lógica y las condiciones establecidas.

Patrón Controlador

- Algunas características y responsabilidades del Controlador son:
  - **Comunicación entre objetos:** El Controlador facilita la comunicación y la interacción entre diferentes objetos y componentes del sistema, actuando como un intermediario para el intercambio de información y la realización de tareas específicas.
  - **Enrutamiento de solicitudes:** El Controlador determina qué clase o componente es responsable de manejar una solicitud en particular y direcciona la solicitud a ese objeto.

Patrón Controlador

- El patrón Controlador es comúnmente utilizado en arquitecturas de software como el patrón Modelo-Vista-Controlador (MVC), donde el Controlador es responsable de manejar las interacciones del usuario y coordinar la lógica del sistema.
- En resumen, el patrón Controlador se utiliza para designar una clase o un objeto que actúa como punto de entrada centralizado y coordina las acciones del sistema.
- Proporciona una capa de abstracción entre las clases y componentes, mejorando el acoplamiento y facilitando la coordinación y el control del flujo de trabajo del sistema.

Patrón Comando

- El patrón "Comando" (Command en inglés) se refiere a asignar la responsabilidad de una solicitud o comando a un objeto separado que encapsula la acción a realizar.
- El objetivo del patrón Comando en GRASP es lograr un diseño más flexible, modular y extensible al separar las solicitudes de los objetos que las ejecutan.
- En el patrón Comando, se identifica una clase o un objeto que representa una solicitud o acción específica. Este objeto actúa como un intermediario entre el objeto que solicita la acción (cliente) y el objeto que la ejecuta (receptor). El objeto Comando encapsula los detalles de la acción y proporciona un método para ejecutarla.

Patrón Comando

- Algunas características y responsabilidades del patrón Comando en GRASP son:
  - **Comando:** Es la abstracción del comando que encapsula la solicitud o acción a realizar. Puede tener métodos como ejecutar() para realizar la acción y deshacer() para deshacer la acción si es necesario.
  - **Cliente:** Es el objeto que solicita la acción y crea el objeto Comando correspondiente. El cliente conoce el comando y puede invocarlo según sea necesario.
  - **Receptor:** Es el objeto que realiza la acción real asociada con el comando. El comando encapsula la llamada al método del receptor para ejecutar la acción.
  - Invocador: Es el objeto que invoca el comando y se encarga de llamar al método ejecutar() del comando cuando sea apropiado. El invocador puede mantener una lista de comandos ejecutados o manejar el deshacer/rehacer de acciones.

Patrón Comando

- Un ejemplo práctico del patrón Comando en GRASP podría ser un sistema de edición de documentos en el que se tienen comandos como "Copiar", "Pegar" y "Deshacer". Cada comando sería implementado como un objeto Comando separado, y el cliente (por ejemplo, una interfaz de usuario) crearía y ejecutaría los comandos según las interacciones del usuario.
- En resumen, el patrón GRASP Comando se utiliza para asignar la responsabilidad de una solicitud o acción a un objeto separado que encapsula la acción a realizar. Proporciona un diseño más flexible, modular y extensible al separar las solicitudes de los objetos que las ejecutan. Esto permite agregar nuevos comandos y facilita el manejo de acciones deshacer/rehacer.

Ejercicio 1

Ejercicios 2

Universidad **César Vallejo**

Licenciada por Sunedu
para que puedas salir adelante