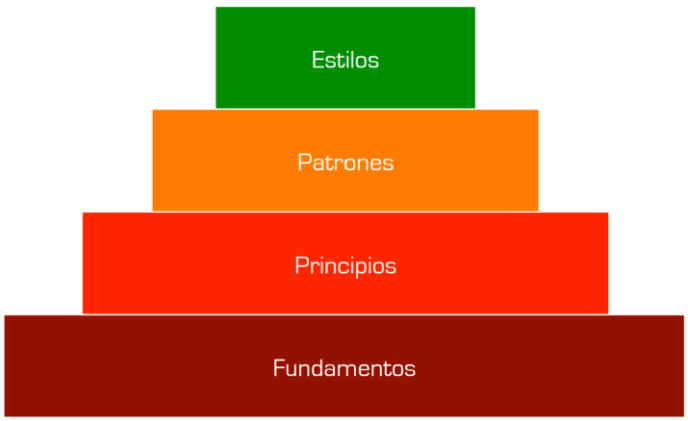
Polimorfismo: (varias formas diferentes) es la relajación del sistema de tipos, de manera que si se hace referencia a una clase se aceptan objetos de dicha clase y de sus clases derivadas. Permite que un objeto de una subclase pueda ser considerado y utilizado como si fuese un objeto de la superclase.

Mueble mueble = new silla();

Vinculación dinámica: mecanismo por el cual se escoge el método que responderá a un determinado mensaje.

Ejemplo:

```
interface Person {
  void imprimir();
}

class Padre implements Person {
 @Override
 public void imprimir() {
  System.out.println("Imprimir en Padre");
 }
}

class Hijo implements Person {
  @Override
  void imprimir() {
    System.out.println("Imprimir en Hijo");
   }
}

class Principal {

  public static void main(String args[]) {

     Padre miPadre = new Padre(); // Objeto de la clase Padre
     Hijo miHijo = new Hijo();    // Objeto de la clase Hijo

     Person unObjeto;

     unObjeto = miPadre;
     unObjeto.imprimir();  // El resultado sería "Imprimir en Padre"

     unObjeto = miHijo;
     unObjeto.imprimir();  // El resultado sería "Imprimir en Hijo"

 }

}
```



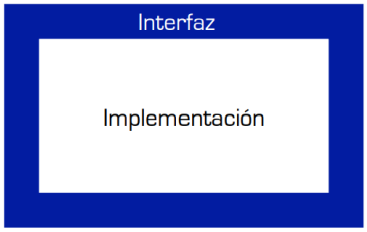
Fundamentos del diseño de software: características que tiene todo software

- **Modularidad**: propiedad que permite subdividir una aplicación en partes más pequeñas, donde cada una debe ser tan independiente como sea posible de las otras partes. También es una estrategia de desarrollo de software en la que los componentes o módulos son recombinables para construir diferentes aplicaciones de una misma familia, reduce la complejidad del desarrollo y permite que varias personas puedan colaborar en equipo.
- **Abstracción**: se aísla toda aquella información que no resulta relevante (se eliminan los detalles para la simplificación)
- **Acoplamiento**: es la medida de la interconexión entre los módulos de un programa.
  A menor acoplamiento, mejor: Se reduce la propagación de errores y se fomenta la reutilización de los módulos
- **Cohesión**: módulo que ejecuta una **tarea sencilla** de un procedimiento de software.
  A mayor cohesión, mejor: El módulo será más sencillo de diseñar. La cohesión es baja cuando el componente realiza un conjunto de tareas que se encuentran débilmente relacionadas entre sí.

Encapsulación: no se puede acceder a información de otro módulo

Módulo: partes de un programa que resuelve uno de los subproblemas en que se divide el problema original. Cada módulo tiene una tarea bien definida para la que puede necesitar a otros módulos.

Interfaz: módulo actúa como una caja negra para diferenciar la parte externa (interfaz) de la interna (implementación)



Sustituibilidad: El diseño debe realizarse permitiendo usar cualquier subtipo de una clase. La subclase puede responder a los métodos de manera diferente a la clase padre (por el principio del polimorfismo), pero el invocador no debe preocuparse por la diferencia.

Encapsulación: Los módulos se diseñan de forma que la información contenida dentro de un módulo sea inaccesible a otros módulos que no necesiten la información.

Principios del software: tienen el objetivo de que el software sea de calidad, que tolere cambios, que sea fácil de entender.

Principios de diseño SOLID:

S - Principio de responsabilidad única: una clase debe tener solo una responsabilidad, si una clase tiene múltiples responsabilidades hay riesgo de modificar una responsabilidad, pero alterar otra al estar ambas en el mismo sitio.

O - Principio abierto-cerrado: Una clase debe estar abierta a extensiones, pero cerrada a las modificaciones

L - Principio de sustitución de Liskov: Las funciones que referencian clases base deben ser capaces de usar objetos de clases derivadas. Si S es un subtipo de T, entonces los objetos de tipo T pueden ser sustituidos por objetos de tipo S sin alterar ninguna de las propiedades del programa

```
Mueble mueble = new Mueble();
mueble = new silla();
```

I – Principio de Segregación de la Interfaz: Muchas interfaces específicas son mejores que una única interfaz de propósito general, ya que ningún código debería verse forzado a depender de métodos que no utiliza

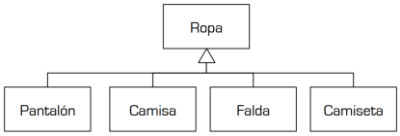
D - Principio de Inversión de Dependencia: Los módulos de alto nivel no deberían depender de los módulos de bajo nivel: ambos deberían depender de abstracciones (interfaces)

Otros Principios de diseño:

Diseño por contrato: Los componentes se deben diseñar asumiendo que con ciertas condiciones de entrada se deben garantizar ciertas condiciones de salida.

Ley de demeter: un objeto no debe conocer la estructura interna de los objetos que manipula, ni obtener otros objetos a través de ellos. Reduce el acoplamiento entre clases.

Generalización:



Dependencia: módulo requerido por otro módulo para poder funcionar correctamente

Independencia funcional: Evitar la excesiva interacción con otros módulos

Dependencias circulares: son nocivas ya que:

- se reduce o incluso hace imposible la prueba o la reutilización de un módulo
- un cambio local en un módulo se puede extender a otros módulos
- pueden causar recursiones infinitas y/o filtraciones de memoria

Reglas de dependencias:

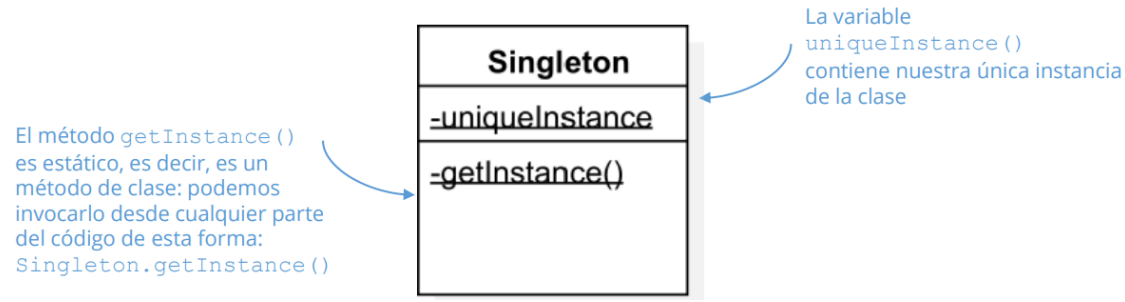
1. Los módulos de alto nivel no deben instanciar sus componentes. Debe ser un módulo externo el que le indique (inyecte) sus componentes

2. Las clases de alto nivel no deben depender de los componentes que implementen los detalles

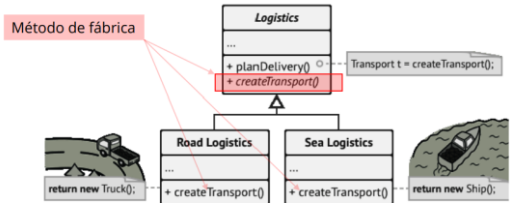
3. Las clases de alto nivel se ejecutan a partir de sus generalizaciones por módulos de bajo nivel

Patrones de diseño: conjunto de conocimientos de expertos del software con los cuales se puede dar solución a problemas

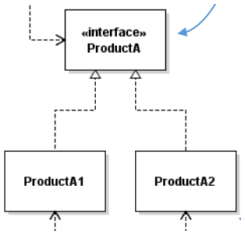
- Singleton: garantiza que una clase solo tenga una instancia y le proporcione un único punto de acceso global. Se pone el constructor privado y cuando se carga la clase se crea la única instancia que esta tendrá.



- Factory: la clase abstracta delega la creación de las instancias en un método de la clase. La llamada al constructor se realiza desde el método. Problema: todos los objetos que se crean deben tener una interfaz común.

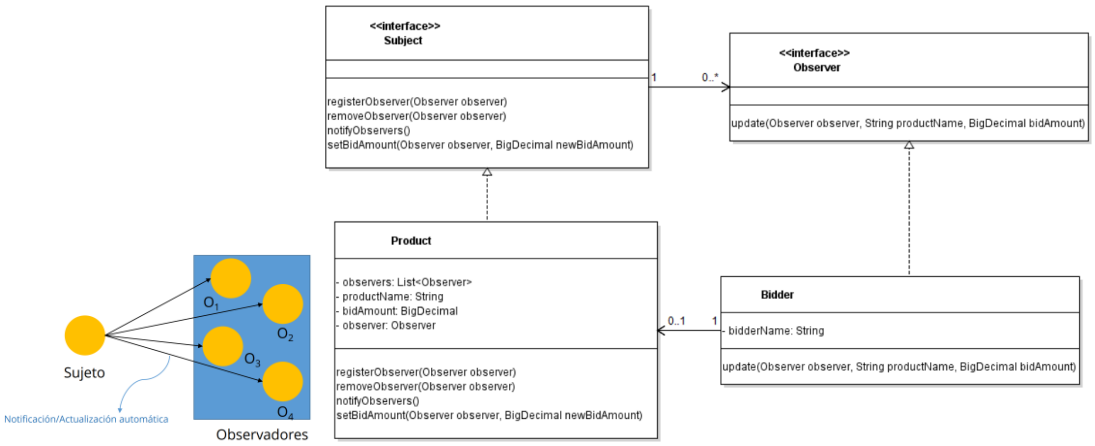


- Abstract Factory: permite producir familias de objetos relacionados o dependientes sin especificar sus clases concretas. Utiliza una interfaz para productos diferentes, pero de la misma familia. (pido una silla y me devuelve una silla del ikea o una silla de elmueble)

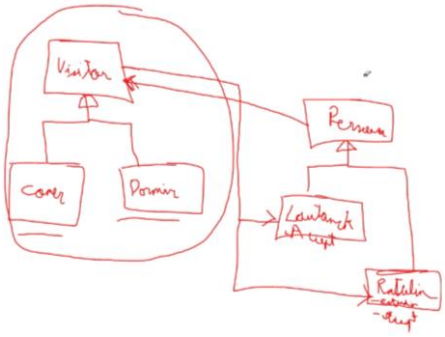


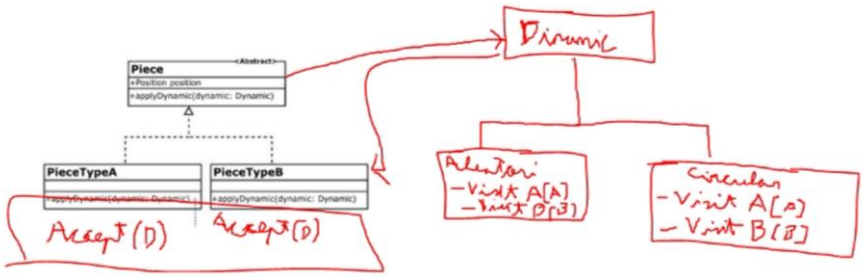
- Observer: define una dependencia "uno a muchos" entre los objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

  Bajo acoplamiento: El sujeto no necesita saber nada acerca de los observadores concretos



- Listener
- Command: transforma un comando en un objecto y está basado en el principio de diseño de inversión del control
- Iterator: proporciona una forma de acceder a los elementos de un objeto de una colección de forma secuencial sin saber cómo se representan las cosas internamente (independientemente de su implementación)
- Visitor: añade funcionalidad a un objeto sin cambiar su clase





- **Decorator**: permite añadir funcionalidades a objetos colocándolos dentro de objetos encapsuladores que contienen estas especialidades.
- **Composite**: objetos en estructura de árbol, donde todos los objetos de la jerarquía cuentan con la misma interfaz

Estilo de diseño: organización del software

- Model View Controller (MVC)
- Model View Presenter (MVP)

¿Por qué cambiamos el modelo model view controller (MVC) por el model view presenter (MVP)? Para que la responsabilidad se quede dentro del modelo y no en la vista