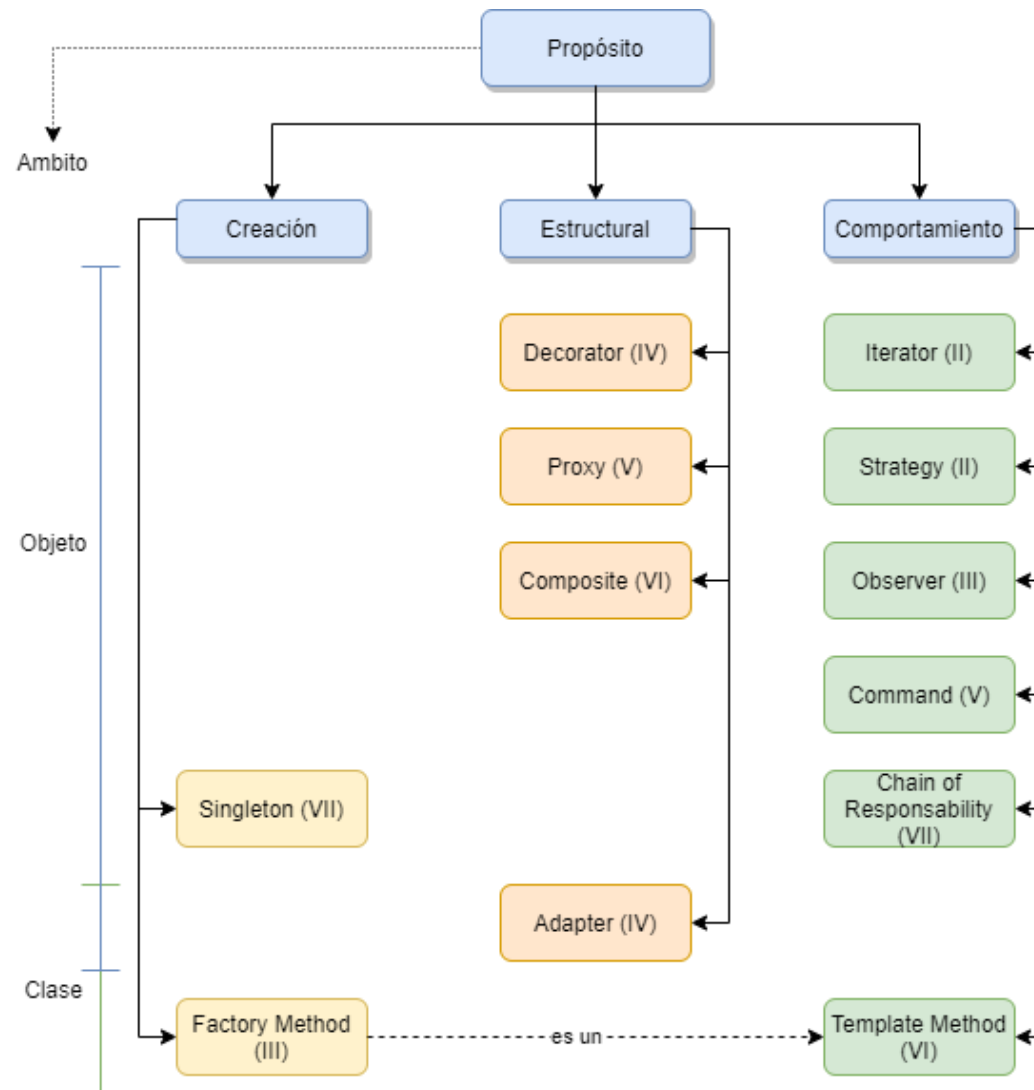


## Patrones de diseño

Los patrones de diseño representan soluciones que han ido evolucionando con el tiempo. Cada patrón describe un problema recurrente y explica un diseño con el fin de reutilizarlo. Permite resolver un problema particular adaptándose a nuevos requisitos (son flexibles).

*“Son descripciones de clases y objetos **relacionados** para resolver un problema de diseño general en un determinado contexto.” Gamma*



<b>Adapter</b>	Convierte la interfaz de una clase en otra que es la que esperan los clientes permitiendo que estas clases cooperen.	ESTRUCTURAL 4
<b>Chain of Responsibility</b>	Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto. Describe el modo de comunicación entre los objetos. Todos los métodos tienen el mismo comportamiento, reenviar la petición al sucesor.	COMPORTAMIENTO 7
<b>Command (Orden)</b>	Encapsula una petición en un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones. Define un objeto que actúa como elemento mágico que representa una petición que el cliente no llega a percibir.	COMPORTAMIENTO 6
<b>Composite</b>	Combina objetos en estructuras de árbol para representar jerarquías de parte – todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.	ESTRUCTURAL 6
<b>Decorator</b>	Anade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.	ESTRUCTURAL 4
<b>Factory Method</b>	Define una interfaz para crear un objeto, pero deja que las subclases decidan que clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.	CREACION 3
<b>Iterator</b>	Proporciona un modo de acceder secuencialmente a elementos de un objeto agregado sin exponer su representación interna. Encapsula el modo en que se accede y se recorren los componentes de un objeto.	COMPORTAMIENTO 2
<b>Observer</b>	Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él. El sujeto y el observador deben cooperar (comunicación distribuida).	COMPORTAMIENTO 3
<b>Proxy</b>	Proporciona un representante (funcionalidad mínima) de un objeto para controlar el acceso y la seguridad.	ESTRUCTURAL 5
<b>Singleton</b>	Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella. Le indica al lenguaje que impida el uso indiscriminado del operador new.	CREACION 7
<b>Strategy</b>	Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan. Divide la funcionalidad.	COMPORTAMIENTO 2
<b>Template Method</b>	La superclase define en una operación el esqueleto de un algoritmo, delegando y permitiendo que las subclases redefinan e implementen ciertos pasos del algoritmo sin cambiar su estructura.	COMPORTAMIENTO 6

**Creacional:** Soluciona problemas de la creación de instancias.

- **Factory Method (III)**
- Singleton (VII)

**Estructural:** Soluciona problemas de composición o agregación de clases y objetos.

- **Adapter (IV)**
- Decorator (IV)
- Proxy (V)
- Composite (VI)

**Comportamiento:** Soluciona problemas de interacción y responsabilidades entre clases y objetos.

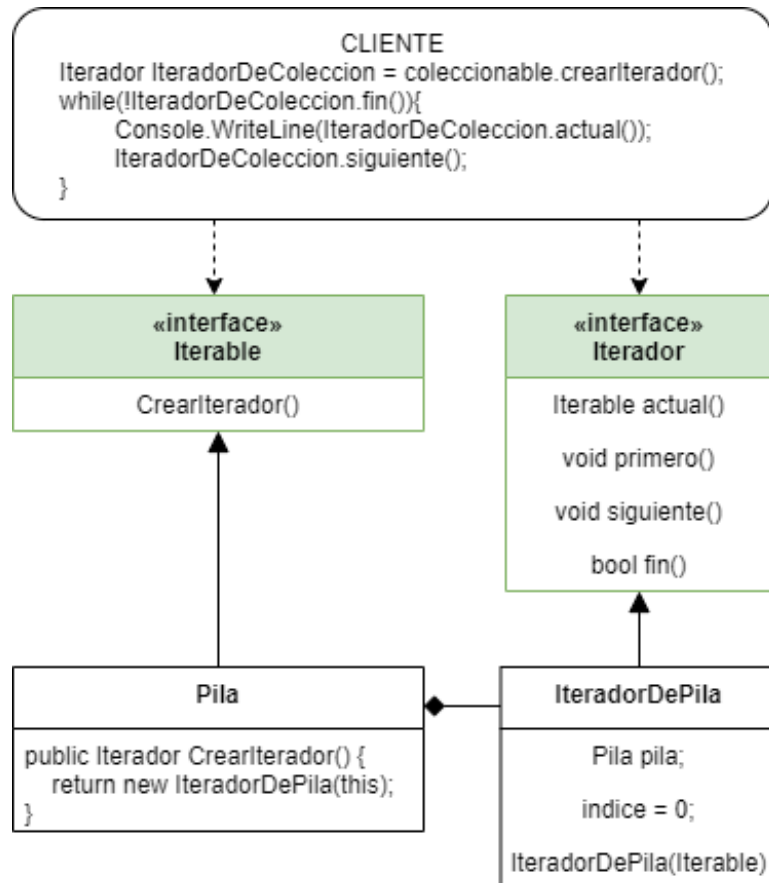
- Iterator (II)
- Strategy (II)
- Observer (III)
- Command (VI)
- Chain of Responsibility (VII)
- **Template Method (VI)**

El segundo criterio, denominado **ámbito**, especifica si el patrón se aplica principalmente a clases o a objetos. Los patrones de clases se ocupan de las relaciones entre las clases y sus subclases. Estas relaciones se establecen a través de la herencia, de modo que son relaciones estáticas —fijadas en tiempo de compilación—. Los patrones de objetos tratan con las relaciones entre objetos, que pueden cambiarse en tiempo de ejecución y son más dinámicas. Casi todos los patrones usan la herencia de un modo u otro, así que los únicos patrones etiquetados como “patrones de clases” son aquellos que se centran en las relaciones entre clases. Nótese que la mayoría de los patrones tienen un ámbito de objeto.

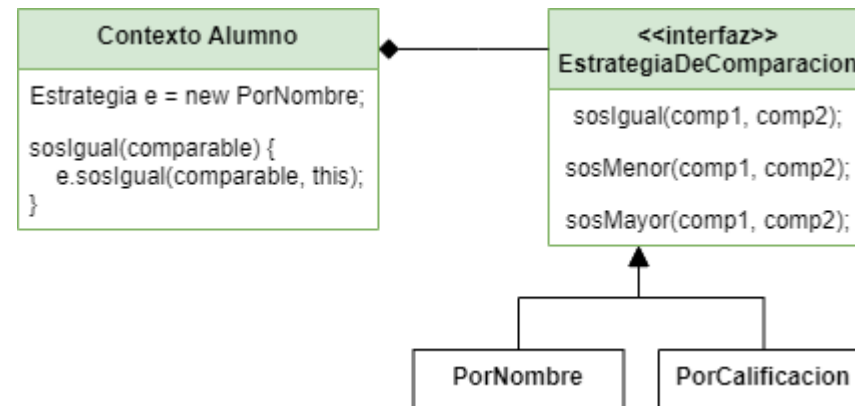
Los patrones de creación de clases delegan alguna parte del proceso de creación de objetos en las subclases, mientras que los patrones de creación de objetos lo hacen en otro objeto. Los patrones estructurales de clases usan la herencia para componer clases, mientras que los de objetos describen formas de ensamblar objetos. Los patrones de comportamiento de clases usan la herencia para describir algoritmos y flujos de control, mientras que los de objetos describen cómo cooperan un grupo de objetos para realizar una tarea que ningún objeto puede llevar a cabo por sí solo.

Hay otras maneras de organizar los patrones. Algunos patrones suelen usarse juntos. Por ejemplo, el Composite suele usarse con el Iterator o el Visitor. Algunos patrones son alternativas: el Prototype es muchas veces una alternativa al Abstract Factory. Algunos patrones dan como resultado diseños parecidos, a pesar de que tengan diferentes propósitos. Por ejemplo, los diagramas de estructura del Composite y el Decorator son similares.

## Clase 2 – Iterator



## Clase 2 – Strategy

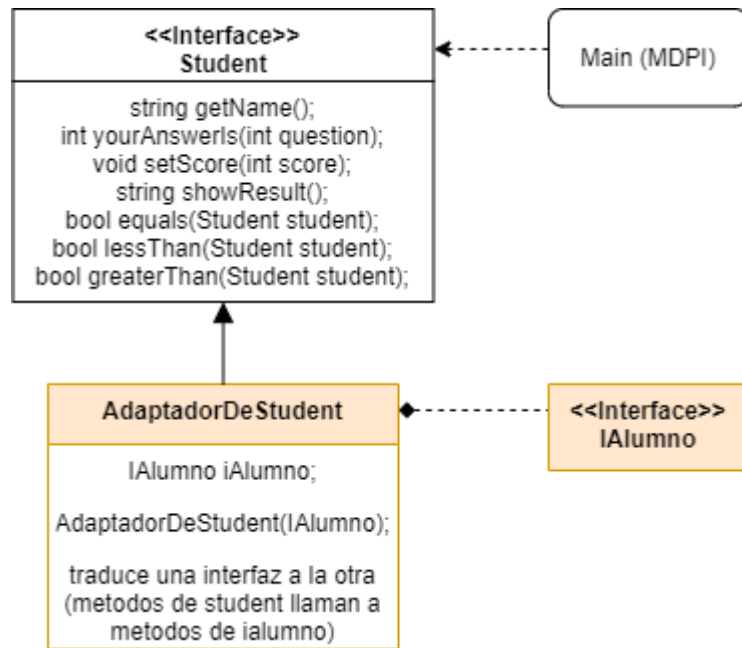


## Comportamiento

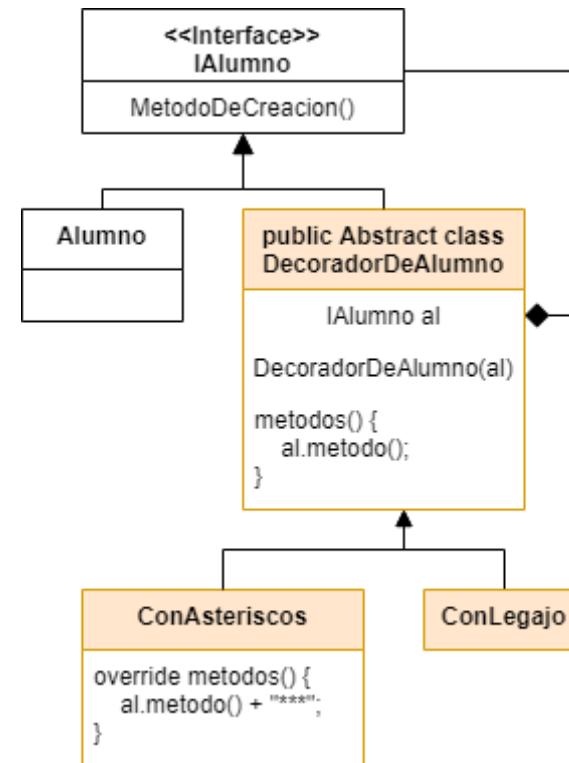
Soluciona problemas de interacción y responsabilidades entre clases y objetos.

Clase 3 – <i>Factory Method</i>	Clase 3 – <i>Observer</i>
<div data-bbox="208 140 797 488"> <p><b>abstract Class</b> <b>FabricaDeComparables</b></p> <pre> const opcion;  static Comparable CrearAleatorio(opcion) {     FabricaDeComparable fabrica= null;     switch(opcion)         case x: fabrica = new FabricaDeAlumno;     return fabrica.crearComparableAleatorio(); }  public abstract Comparable crearComparableAleatorio(op); </pre> </div> <div data-bbox="208 488 1093 724"> <pre> classDiagram     class FabricaDeComparables {         +const opcion         +static Comparable CrearAleatorio(opcion)         +public abstract Comparable crearComparableAleatorio(op)     }     class FabricaDeAlumnos {         +CrearComparable() : new Alumno     }     class FabricaDeNumeros {         +CrearComparable() : new Numero     }     class Alumno     class Numero     class Comparables      FabricaDeComparables &lt; -- FabricaDeAlumnos     FabricaDeComparables &lt; -- FabricaDeNumeros     FabricaDeAlumnos ..&gt; Alumno     FabricaDeNumeros ..&gt; Numero     Alumno --&gt; Comparables     Numero --&gt; Comparables </pre> </div>	<div data-bbox="1196 156 1937 700"> <pre> classDiagram     class Observado {         &lt;&lt;Observado&gt;&gt;         +AgregarObservador(o)         +QuitarObservador(o)         +Notificar()     }     class Observador {         &lt;&lt;Observador&gt;&gt;         +Actualizar(observado)     }     class Vendedor {         +List observadores         +notificar ()     }     class Gerente {         +Actualizar(observado)     }      Observado &lt; -- Vendedor     Observador &lt; -- Gerente     Vendedor ..&gt; Observado     Gerente ..&gt; Vendedor     Gerente --&gt; Observador </pre> </div>
<p style="text-align: center;"><b>Creacional</b></p> <p style="text-align: center;">Soluciona problemas de la creación de instancias.</p>	<p style="text-align: center;"><b>Comportamiento</b></p> <p style="text-align: center;">Soluciona problemas de interacción y responsabilidades entre clases y objetos.</p>

### Clase 4 – Adapter

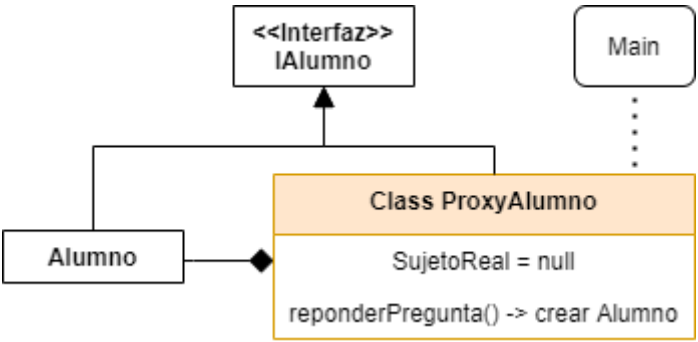
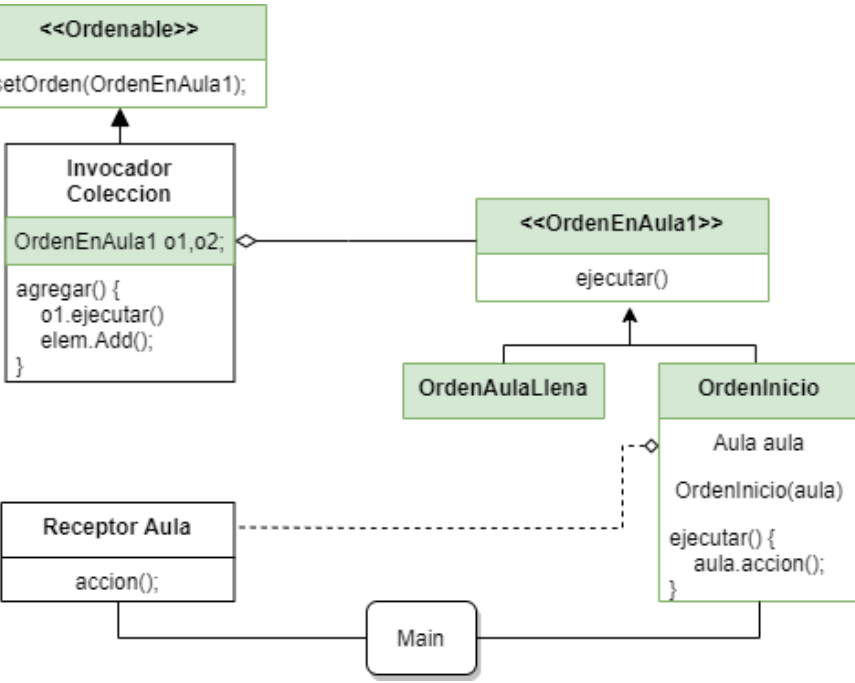


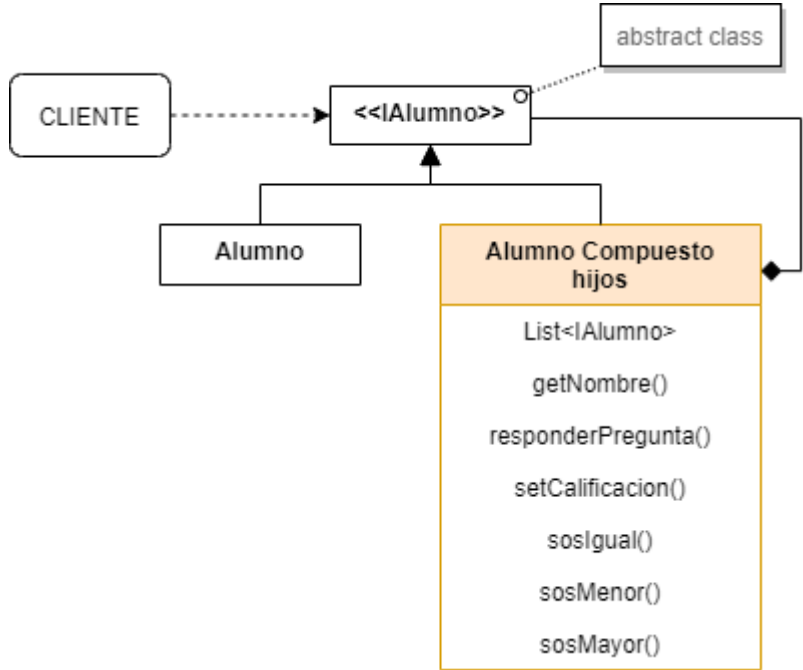
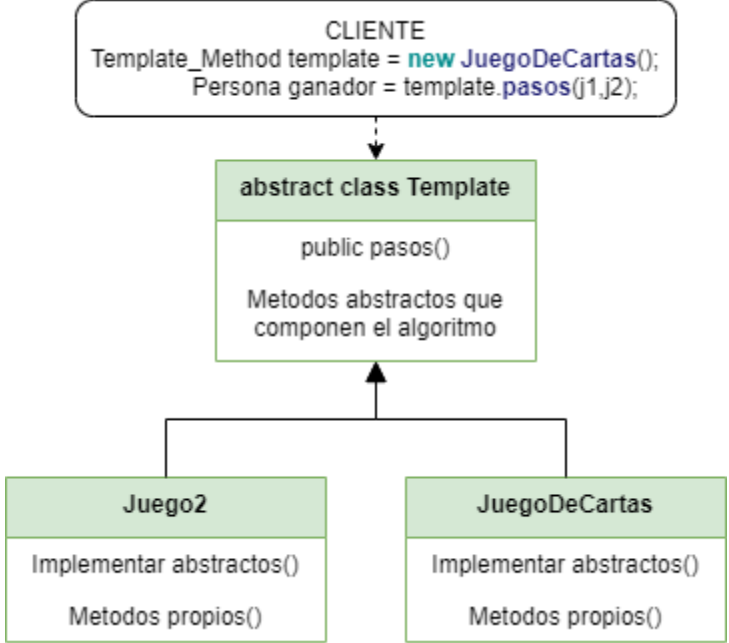
### Clase 4 – Decorator



### Estructural

Soluciona problemas de composición o agregación de clases y objetos.

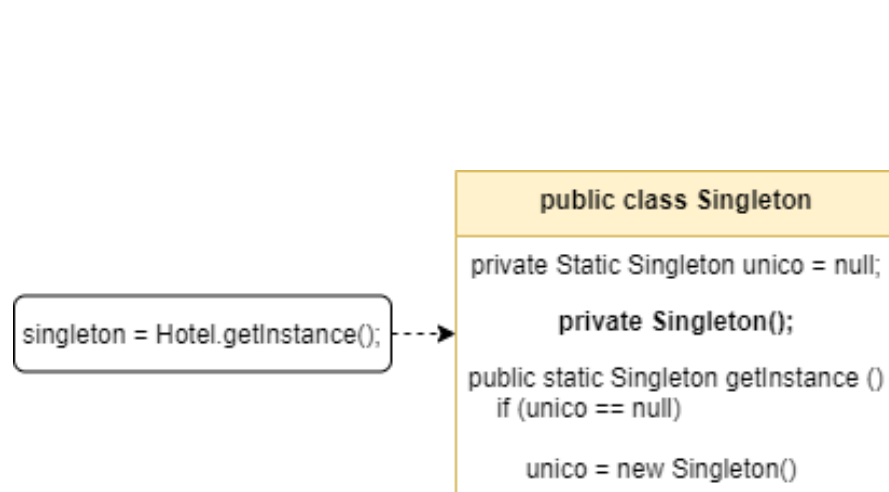
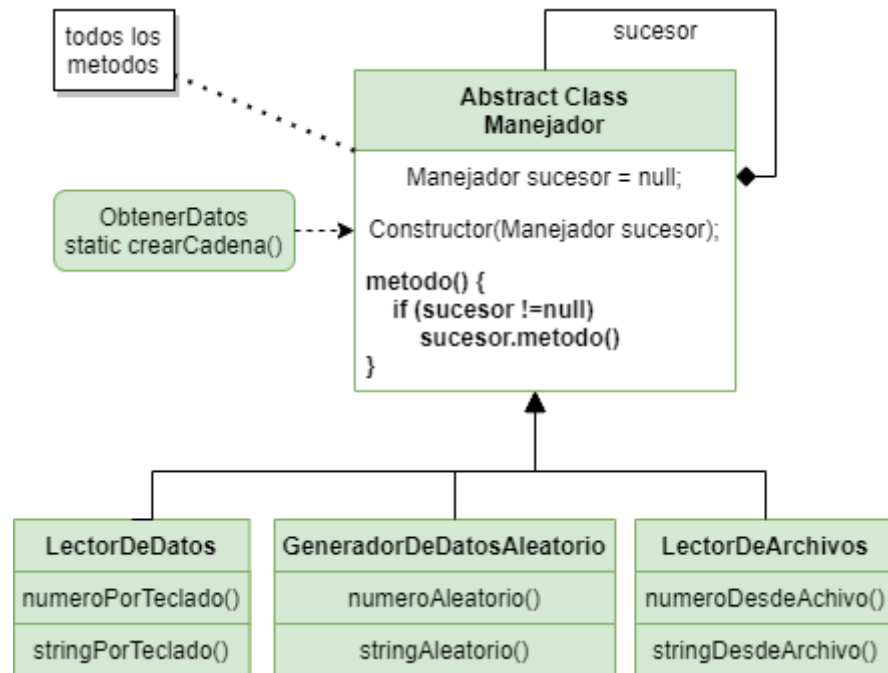
Clase 5 – Proxy	Clase 5 – Command
 <pre> classDiagram     class IAlumno {         &lt;&lt;Interfaz&gt;&gt;     }     class Alumno     class ProxyAlumno {         SujetoReal = null         reponderPregunta() -&gt; crear Alumno     }     class Main     IAlumno &lt; -- Alumno     IAlumno &lt; -- ProxyAlumno     Alumno *-- ProxyAlumno     Main ..&gt; ProxyAlumno </pre> <p>The diagram illustrates the Proxy pattern. It features an interface <code>&lt;&lt;Interfaz&gt;&gt; IAlumno</code> which is implemented by two classes: <code>Alumno</code> and <code>Class ProxyAlumno</code>. The <code>Class ProxyAlumno</code> contains a private attribute <code>SujetoReal = null</code> and a method <code>reponderPregunta() -&gt; crear Alumno</code>. A <code>Main</code> class is shown with a dashed line indicating its interaction with the <code>ProxyAlumno</code> class.</p>	 <pre> classDiagram     class Invocador {         Coleccion         OrdenEnAula1 o1,o2         agregar() {             o1.ejecutar()             elem.Add()         }     }     class ReceptorAula {         accion()     }     class OrdenEnAula1 {         &lt;&lt;OrdenEnAula1&gt;&gt;         ejecutar()     }     class OrdenAulaLlena     class OrdenInicio {         Aula aula         OrdenInicio(aula)         ejecutar() {             aula.accion()         }     }     class Main     Invocador &lt; -- ReceptorAula     Invocador &lt; -- OrdenEnAula1     Invocador o--&gt; OrdenEnAula1     OrdenEnAula1 &lt; -- OrdenAulaLlena     OrdenEnAula1 &lt; -- OrdenInicio     ReceptorAula ..&gt; OrdenInicio     Main --&gt; ReceptorAula     Main --&gt; OrdenInicio </pre> <p>The diagram illustrates the Command pattern. It features an interface <code>&lt;&lt;Ordenable&gt;&gt;</code> with a method <code>setOrden(OrdenEnAula1);</code>. Two classes, <code>Invocador Coleccion</code> and <code>Receptor Aula</code>, implement this interface. The <code>Invocador</code> class contains a collection of <code>OrdenEnAula1</code> objects and an <code>agregar()</code> method that calls <code>o1.ejecutar()</code> and <code>elem.Add()</code>. The <code>Receptor Aula</code> class has an <code>accion()</code> method. The <code>OrdenEnAula1</code> interface has an <code>ejecutar()</code> method. Two concrete classes, <code>OrdenAulaLlena</code> and <code>OrdenInicio</code>, implement the <code>OrdenEnAula1</code> interface. The <code>OrdenInicio</code> class has a private attribute <code>Aula aula</code> and an <code>OrdenInicio(aula)</code> constructor. The <code>ejecutar()</code> method in <code>OrdenInicio</code> calls <code>aula.accion()</code>. A <code>Main</code> class is shown with solid lines indicating its interaction with both the <code>Receptor Aula</code> and the <code>OrdenInicio</code> class.</p>
<p><b>Estructural</b></p> <p>Soluciona problemas de composición o agregación de clases y objetos.</p>	<p><b>Comportamiento</b></p> <p>Soluciona problemas de interacción y responsabilidades entre clases y objetos.</p>

Clase 6 – Composite	Clase 6 – Template Method
 <pre> classDiagram     class CLIENTE     class &lt;&lt;IAlumno&gt;&gt;     class Alumno     class AlumnoCompuesto["Alumno Compuesto hijos"] {         List&lt;IAlumno&gt;         getNombre()         responderPregunta()         setCalificacion()         sosIguar()         sosMenor()         sosMayor()     }     CLIENTE ..&gt; &lt;&lt;IAlumno&gt;&gt;     &lt;&lt;IAlumno&gt;&gt; &lt; -- Alumno     &lt;&lt;IAlumno&gt;&gt; &lt; -- AlumnoCompuesto     &lt;&lt;IAlumno&gt;&gt; o--&gt; AlumnoCompuesto </pre> <p>The diagram illustrates the Composite Design Pattern. It features a <b>CLIENTE</b> that interacts with an <b>&lt;&lt;IAlumno&gt;&gt;</b> interface. This interface is implemented by two classes: <b>Alumno</b> and <b>Alumno Compuesto hijos</b>. The <b>Alumno Compuesto hijos</b> class is highlighted in orange and contains a <code>List&lt;IAlumno&gt;</code> and several methods: <code>getNombre()</code>, <code>responderPregunta()</code>, <code>setCalificacion()</code>, <code>sosIguar()</code>, <code>sosMenor()</code>, and <code>sosMayor()</code>. A dashed arrow from the CLIENTE to the interface and a solid arrow from the interface to the composite class indicate the flow of interaction.</p>	 <pre> classDiagram     class CLIENTE {         Template_Method template = new JuegoDeCartas();         Persona ganador = template.pasos(j1,j2);     }     class Template {         &lt;&lt;abstract class&gt;&gt;         public pasos()         Metodos abstractos que componen el algoritmo     }     class Juego2 {         Implementar abstractos()         Metodos propios()     }     class JuegoDeCartas {         Implementar abstractos()         Metodos propios()     }     CLIENTE ..&gt; Template     Template &lt; -- Juego2     Template &lt; -- JuegoDeCartas </pre> <p>The diagram illustrates the Template Method Design Pattern. It shows a <b>CLIENTE</b> that uses a <b>Template_Method</b> to instantiate a <b>JuegoDeCartas</b> object and call its <code>pasos(j1,j2)</code> method. The <b>Template</b> is an abstract class that defines a <code>public pasos()</code> method and contains abstract methods that form the algorithm. Two concrete classes, <b>Juego2</b> and <b>JuegoDeCartas</b>, inherit from the <b>Template</b> and implement the abstract methods. The concrete classes are highlighted in green.</p>
<p align="center"><b>Estructural</b></p> <p>Soluciona problemas de composición o agregación de clases y objetos.</p>	<p align="center"><b>Comportamiento</b></p> <p>Soluciona problemas de interacción y responsabilidades entre clases y objetos.</p>



### Clase 7 – Chain of Responsibility

### Clase 7 – Singleton



### Comportamiento

Soluciona problemas de interacción y responsabilidades entre clases y objetos.

### Creacional

Soluciona problemas de la creación de instancias.