Patrones Creacionales:

* Abstract Factory: Es una interfaz para crear familias de objetos relacionados sin tener que especificar sus clases concretas. Se utiliza cuando debo usar una o varias familias de productos relacionados. Ej: Swing
* Builder: Separa la construcción de un objeto complejo en diferentes partes. Se usa cuando tengo que crear un objeto de forma independiente de sus partes y de como se ensamblan. También puedo usarlo cuando el resultado de la construcción puede tener diferentes representaciones. Se resuelve con una clase constructor, que va creando las diferentes partes del objeto. Ej: Mecánico de autos.
* Factory Method: Se define una interfaz para crear un objeto, pero las implementaciones se las dejamos a las subclases. La instanciación la hace las subclases. Básicamente, es una clase abstracta con métodos abstractos y tiene hijos. Se usa cuando quiero delegar la creación de objetos a subclases específicas, cuando quiero variar el tipo de objeto de acuerdo a la clase que lo crea, y cuando no se puede determinar el tipo del objeto que se va a crear. Se resuelve con una clase creador, y subclases concretas donde cada una crea objetos de diferentes tipos. Ej: Traductor.
* Protoype: Crear clones de una instancia creada previamente. Se usa cuando quiero instancias clones de una existente, cuando el estado interno varia poco de una instancia base a otra, y para evitar fabricas de objetos similares. Se resuelve creando un método clon() en el objeto. Ej: Los Minions.
* Singleton: Tener una única instancia de una clase y tener un solo punto de acceso a ella. Se usa cuando quiero un cupo limitado de instancias de una clase. Se resuelve con un método de clase que devuelva la instancia creada, y si no esta creada, que la cree.

Patrones Estructurales:

* Adapter: Convertir una interfaz en otra, de modo que una clase que no pueda usar la original, pueda usarla a partir de la transformación. Se usa cuando tengo una clase que no puedo modificar y necesito conectarla con otra que no es compatible. Se resuelve usando herencia, hago subclase del adaptado. Ej: Documentos.
* Bridge: Desacopla una abstracción de su implementación para que puedan variar independientemente. Se usa cuando tengo que cambiar objetos concretos en run-time, cuando la abstracción y los objetos pueden ser extensibles de forma independiente y cuando el cambio en los objetos concretos no deben afectar al cliente. Se resuelve con clases abstractas que definan métodos abstractos que deban ser implementados por clases concretas. Ej: Afinar un instrumento musical.
* Composite: Construir objetos complejos a partir de objetos simples, usando la técnica de recursividad y una estructura en forma de árbol. Se usa cuando tengo jerarquías de objetos compuestos y cuando quiero tratar a los objetos compuestos igual que a los simples. Se resuelve con una jerarquía tipo árbol de componentes, con hojas y nodos que referencian a otro componente. Ej: Caja de Herramientas.
* Decorator: Agregar funcionalidad a un objeto sin tener que crear subclases que vayan heredando las funciones e incorporando nuevas funcionalidades. Se usa cuando tengo que agregar funcionalidad especifica a un objeto de forma dinámica, sin afectar a otros y cuando se hace imposible generar una nueva jerarquía por la cantidad de clases que generaría y la funcionalidad quedaría duplicada. Se resuelve con una jerarquía tipo árbol de componentes que tenga un componente concreto y un “decorado” que agrega la funcionalidad. Ej: Café Con Leche, Auto con Nafta.
* Facade: Punto de acceso único a clases y funcionalidades del sistema. Se usa cuando tengo un sistema complejo y quiero un punto de acceso único porque no quiero conocer como esta implementado el mismo y cuando quiero separar en capas. Se resuelve con una clase que conozca las demás clases del sistema y provea la interfaz necesaria al cliente. Ej: Mail.
* Flyweight: Eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos con información idéntica o similar. Se resuelve con una clase que resuelva si se puede crear un objeto o si ya hay uno igual.
* Proxy: Provee un objeto intermediario que controla el acceso a otro. Se usa cuando requiero acceso a un objeto a través de otro, cuando quiero controlar el acceso a un objeto, y cuando requiero creación de objetos bajo demanda. Se resuelve con una clase que reciba los requerimientos y provea el acceso al objeto real. Ej: Proxy De Internet.

Patrones De Comportamiento:

* Chain Of Responsibility: Evita que un emisor incorpore una petición a un receptor, dándole a más de un objeto la posibilidad de responder una petición. Se encadenan los receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto. Se usa cuando necesito filtrar requerimientos, y cuando puede haber varios objetos que cumplan el requerimiento y aun no se conocen. Se resuelve con una clase que tenga como sucesor a otro objeto de la misma clase.
* Command: Encapsula una petición como un objeto, permitiendo parametrizar clientes con diferentes métodos, encolar, registrar y deshacer eventos. Encapsula mensajes como un objeto. Se usa cuando tengo que realizar operaciones asincrónicas y cuando hay que desarrollar sistemas utilizando ordenes de alto nivel que se construyen con operaciones sencillas. Se resuelve con una jerarquía de clases Command que invoca a un método de un objeto receptor, en el momento que es invocado por un invocador. Ej: Ctrl+Z (Deshacer)
* Interpreter: Dado un lenguaje, define una representación para su gramática junto con un interprete del lenguaje. Se usa cuando tengo un lenguaje con una gramática especifica que tengo que interpretar. Se resuelve con un árbol de expresiones con hojas como expresiones terminales y nodos como expresiones que anidan otras. Ej: Calculadora.
* Iterator: Define una interfaz que permite recorrer una estructura de datos sin que sea necesario recorrer la estructura interna de la misma. Se usa cuando necesito varias formas de recorrer colecciones, y cuando necesito lograr la misma forma de iteración independientemente de la colección (Polimorfismo). Se resuelve con un iterador genérico e iteradores concretos para cada tipo de colección. Ej: Colecciones en Java, Control Remoto.
* Mediator: Define un objeto que encapsula la forma de interacción de un conjunto de objetos. Se usa cuando la interacción entre objetos es compleja, y cuando debo personalizar el comportamiento entre objetos. Se resuelve con una clase Mediator que se encarga de la comunicación entre objetos concretos. Ej: Juegos, Chat.
* Memento: Capturar el estado interno de un objeto para que este pueda ser restaurado a ese estado mas tarde. Se usa cuando se quiere restaurar el sistema desde estados pasados, y cuando se desea hacer y deshacer determinadas operaciones. Se resuelve con una clase Memento que solicita el estado al objeto concreto y lo almacena, pudiendo restaurar el estado de este objeto. Los terceros objetos acceden al estado del objeto, y no al Memento. Ej: Pausa.
* Observer: Define relaciones de dependencia (Uno a Muchos) entre objetos, dado que si el observado cambia de estado, los dependientes se actualizaran de forma automática. Se usa cuando el cambio de un objeto obliga la modificación de otros y no se saben cuáles son esos otros, y cuando necesito notificar a otros objetos de que algo sucedió y no se cuales son esos objetos. Se resuelve con una colección de objetos observadores que reciben la notificación de que el objeto ha cambiado. Ej: MVC, Sección Critica.
* State: Cambiar el comportamiento de un objeto cuando cambia su estado interno. Se usa cuando el comportamiento del objeto depende de su estado y debo cambiar ese comportamiento en ejecución. Se resuelve con una jerarquía de estados que manejan el requerimiento que se le hace al objeto. De acuerdo al estado es el comportamiento que toma el objeto. El objeto puede pasarse como parámetro al estado para ejecutar los métodos específicos del mismo. Ej: Maquina De Café, Alarma.
* Strategy: Determina como se debe realizar el intercambio de mensajes entre diferentes objetos para realizar una tarea. Permite mantener un conjunto de algoritmos los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades. Se usa cuando tengo muchas clases similares y que varían solo en algún comportamiento, y cuando necesito variantes de un algoritmo. Se resuelve con una jerarquía de algoritmos donde el objeto instancia el algoritmo concreto en el momento indicado. Ej: Login, Forma De Pago.
* Template Method: Define la estructura de un algoritmo, y con subclases especificas se implementan pasos del algoritmo sin cambiar la estructura. Se usa cuando tengo partes invariantes y partes dinámicas de un algoritmo, y para evitar duplicar código permitiendo tener las partes especificas en subclases. Se resuelve con una jerarquía de clases que definen operaciones (Probablemente Abstractas) que puedan implementar las subclases. Ej: Login.
* Visitor: Representa una operación que debe ser ejecutada sobre una estructura de objetos. Posibilita definir una nueva operación sin cambiar la estructura del objeto. Se usa cuando las operaciones sobre la estructura son diferentes, y cuando la estructura de objetos contiene muchas clases de objetos con diferentes interfaces y quiero ejecutar operaciones sobre esos objetos. Se resuelve con una jerarquía de visitadores que ejecutan operaciones específicas sobre una jerarquía de clases. El objeto a ser visitado debe aceptar la visita del Visitor. Ej: Vigía.