# JUSTIFICACION USO DE PATRONES

1. **Patrones Básicos.**
   1. Interface: No se usará. Si bien el uso de interfaces ayuda a clasificar las clases según el uso de los métodos; para éste caso la implementación de otros métodos que tienen acciones parecidas pero más definidas es más eficiente.
   2. Abstract Parent Class: Se usará. Para éste ejercicio en particular es muy funcional ya que se tienen objetos que heredan formas de hacer las cosas y características que son procesadas de forma diferente, además que se integra con el uso de patrones creacionales que son más completos, para el problema existen preguntas, pero tienen formas diferentes de configurar sus posibles respuestas.
   3. Privates Methods. No se usará. No se implementará para este problema métodos privados ya que no hay procesos propios de alguna clase que sea solo accesible desde sí mismo.
   4. Accessor Methods: Se usará. Es de utilidad ya que se tienen atributos privados y por este motivo los métodos de acceso para editar o devolver la información de estos atributos son importantes para la comunicación entre clases.
   5. Constant Data Manager: No se usará. Para este programa no se tienen datos de valor constante, por lo tanto el uso de un manager de datos constantes es innecesario.
   6. Immutable Object: No se usará. No se tendrán objetos cuyo instanciación no pueda ser cambiada, por lo tanto no será usado el patrón de objetos inmutables.
   7. Monitor: No se usará. No se tendrán objetos de uso sincronizado o por hilos.
   8. MVC: Se usará. Para éste caso es muy útil, ya que se pueden separar la implementación de la forma en que el usuario ve la información con un objeto controlando y comunicando las peticiones entre uno y el otro.
2. **Patrones Creacionales.**
   1. Abstract Factory: Se usará. Se tienen objetos parecidos, pero hacen ciertas acciones de forma diferente, por ejemplo: varios tipos de preguntas con varias formas de definir la(s) respuesta(s) correcta(s), pero quiero que cada tipo de pregunta sea elaborada por un objeto aparte para que las respuestas definidas sean hechas para el tipo de pregunta en cuestión.
   2. Builder: Se usará. Para elaborar un examen en una primera instancia se deben tener las preguntas hechas con su(s) respuesta(s) correcta(s) definidas, es así como builder me ayuda a realizar las cosas en el correcto orden para que la definición de un objeto examen sea correcto.
   3. Factory Method. No se usará. Si bien puede ser una opción parecida al Abstract Factory, éste no permite hacer familias de preguntas con familias de respuestas y asi garantizar la entrega del tipo de respuesta correcta para el tipo de pregunta definida.
   4. Singleton: Se usará. Para hacer conexión a la base de datos solo es necesario un objeto instanciado de tipo conexión por ejecución.
   5. Prototype: No se usará. No se necesita de modelo de objetos instanciados para el problema en cuestión, el ideal es generar preguntas de forma aleatoria y en el caso de las preguntas con opción múltiple siempre será necesario cambiar, por lo menos en un orden diferente, por lo tanto prototype no es funcional.
3. **Patrones Estructurales.**
   1. Adapter: No se usará. Como es de conocimiento éste patrón realiza la comunicación entre dos interfaces convirtiendo la interfaz de una clase en una nueva interfaz (nueva versión), es decir adaptar una aplicación que ya se encuentra en uso y mejorarla a una nueva versión según lo que el cliente este solicitando. En este caso de esta aplicación no es necesario implementarla ya que se está utilizando una única interfaz la cual tiene todos los componentes que el cliente necesita.
   2. Composite: No se usará. No es necesario para ésta aplicación ya que no se quiere representar jerarquías de objetos.
   3. Facade: Se usará. Para esta aplicación este patrón es funcional ya que se tomará el controlador como fachada para simplificar poder simplificar la comunicación entre el cliente y servidor. Es un beneficio ya que el cliente no tendrá que observar e interactuar con lo que está dentro de la aplicación, simplemente podrá observar una *interface* sencilla la cual podrá entender cómo manejarla.
   4. Flyweight: No se usará. Actualmente ésta aplicación no usará gran cantidad de información y el rendimiento de la memoria no se verá afectada por lo que no es necesario utilizar este patrón, ya que como se sabe lo que busca es reducir la redundancia de información idéntica liberando memoria para mejorar el rendimiento.
   5. Decorator: Se usará. Es funcional para la aplicación para agregar o eliminar de forma dinámica las funcionalidades de un objeto, esto lo implementamos para las preguntas ya que el usuario podrá ingresar nuevas opciones o a su vez eliminarlas, por lo que al realizar este proceso el contenido de las preguntas puede cambiar.
4. **Patrones comportamentales.**
   1. Chain of Responsability: Se usará. Debido a que habrán varios tipos de preguntas a realizar que no pueden ser ejecutas por uno solo, así que cadena de responsabilidades será útil para definir quien hará un tipo de pregunta en específico y si no puede, ceder esa responsabilidad a quien pueda hacerlo, informando lo que el anterior pudo hacer.
   2. Iterator: No se usará. Si bien vamos a almacenar los tipos de preguntas de alguna forma, en este caso, un arreglo, no tenemos que trabajar con ellas de forma constante o de forma donde el proceso requiera de una estructura en específico, así mismo no necesitamos una forma de iterar en esta estructura.
   3. Memento: No se usará. Para nuestro proyecto en específico no necesitaremos instancias almacenadas de nuestros objetos usados en un momento en especial, debido a que tanto la creación, eliminación de las preguntas, los exámenes e incluso los usuarios, no traen o concentran tanta información como guardar instantes validos de estos objetos.
   4. Visitor: No se usará. No necesitamos operar sobre una jerarquía de clases en específico sin modificar su estructura.
   5. Observer: No se usará. Al no tener un objeto que usen varias clases, no tengo la necesidad de notificar a todos sobre los cambios realizados.