



Uso del Patrón de Diseño “Factory Method” en el Proyecto “Pretty Exam”

Grupo 2 (Los Pretty Boys)

Juan David Buitrago Salazar – jbuitragosa@unal.edu.co

Luis David Garzón Morales – lgarzonmo@unal.edu.co

Juan David Cárdenas Galvis – jcardenasgal@unal.edu.co

Deibyd Santiago Barragán Gaitán – dbarragang@unal.edu.co

Docente

Oscar Eduardo Álvarez Rodríguez

Universidad Nacional de Colombia




Facultad de Ingeniería



Departamento de Ingeniería de Sistemas e Industrial


Bogotá D.C., Colombia

2025




1 Definición del Patrón Factory Method






El patrón Factory Method (o "método fábrica")  es una forma organizada de crear objetos en programación. En vez de usar directamente un new para crear algo (como cuando decimos new Question()), este patrón propone usar un método especial que se encarga de construir el objeto por nosotros . Esto ayuda a que nuestro código sea más flexible y fácil de mantener .

Entonces imaginemos que tenemos una fábrica de preguntas para los exámenes de nuestro proyecto . En lugar de ser capaces de recordar cómo hacer cada tipo de pregunta, tenemos un supervisor  que se encarga de esto. Nosotros solamente le diríamos qué tipo de pregunta necesitamos y él se encarga de crearla con sus características que la diferencian de las demás, esto lo hace en la fábrica.

El patrón Factory Method funciona exactamente así en nuestro sistema de exámenes. Como tener un supervisor que sabe cómo crear cada tipo de pregunta sin que tengamos que recordarlo .



1.1 Propósito y Objetivos del Patrón en el Proyecto

En nuestro proyecto, este patrón nos ayuda a manejar la creación de las preguntas de manera organizada y simple , haciendo esto escalable incluso si queremos agregar nuevos tipos de preguntas  .




El patrón se encarga de todos los detalles complicados, como verificar que las preguntas de opción múltiple tengan al menos dos opciones  , o que las preguntas de verdadero/falso siempre tengan exactamente esas dos opciones  . Entonces, este patrón nos permite tratar todas las preguntas de manera similar, pero manejando sus diferencias específicas internamente .

2 Cómo lo Usamos en Nuestro Proyecto




2.1 QuestionFactory

En el proyecto tenemos una clase llamada QuestionFactory  que actúa como este supervisor inteligente que crea las preguntas en la fábrica. Esta clase tiene como función principal createQuestion  que funciona como un punto central para crear cualquier tipo de pregunta.

```
class QuestionFactory {  
  static createQuestion(type, data) {  
    switch (type) {  
      case 'multiple_choice':  
        return new MultipleChoiceQuestion(data)  
      case 'true_false':  
        return new TrueFalseQuestion(data)  
      default:  
        throw new Error(`Unknown question type: ${type}`)  
    }  
  }  
}
```

Cuando alguien necesita crear una pregunta , simplemente le dice a este supervisor  qué tipo quiere y él crea este tipo de pregunta específica con las reglas y características apropiadas de la misma .

2.2 Preguntas de Opción Múltiple

Nuestro proyecto maneja preguntas de opción múltiple , estas preguntas tienen unas características únicas en nuestro sistema: Pueden tener múltiples opciones de respuesta, pero se necesitan al menos dos opciones para que sea válida , adicionalmente debe tener al menos una respuesta correcta .

```
class MultipleChoiceQuestion {
  constructor(data) {
    this.text = data.text
    this.type = 'multiple_choice'
    this.category_id = data.category_id || 1
    this.options = data.options || [{ text: '', is_correct: false }]
  }

  validate() {
    if (!this.text?.trim()) throw new Error('Question text is required')
    if (this.options.length < 2) throw new Error('At least 2 options required')
    if (!this.options.some(opt => opt.is_correct)) {
      throw new Error('At least one correct option required')
    }
    return true
  }
}
```

Cuando se crea una pregunta de este tipo 🤖, el sistema automáticamente verifica que cumpla todas las condiciones ✅, si algo está mal ❌, le dice exactamente al usuario que necesita corregir 🔧.

2.3 Preguntas de Verdadero/Falso

Las preguntas de verdadero o falso ✅❌ son más simples, pero tienen sus propias reglas diferentes a las de selección múltiple. Siempre tienen exactamente dos opciones: “Verdadero” 👍 y “Falso” 👎, y el usuario debe seleccionar cuál es la respuesta correcta.

```
class TrueFalseQuestion {
  constructor(data) {
```

```

    this.text = data.text
    this.type = 'true_false'
    this.category_id = data.category_id || 1
    this.options = data.options
    ? [
      {
        text: 'Verdadero',
        is_correct: !!data.options.find(o => o.text ===
'Verdadero')?.is_correct,
      },
      { text: 'Falso', is_correct: !!data.options.find(o => o.text ===
'Falso')?.is_correct },
    ]
    : [
      { text: 'Verdadero', is_correct: false },
      { text: 'Falso', is_correct: false },
    ]
  }
}

```

Lo inteligente de nuestra implementación es que puede manejar tanto la creación de preguntas nuevas como la edición de preguntas existentes. Si se crea una pregunta y se está editando y se tiene marcado “Verdadero” como correcto, el sistema mantiene esa selección.

2.4 Cómo Funciona en la Práctica

Cuando una persona usa nuestro sistema para crear una pregunta 📝, el proceso es muy sencillo 😊, el usuario selecciona el tipo de pregunta en la interfaz 🖱️, escribe el texto de la pregunta, y proporciona las opciones de respuesta. Detrás de escena nuestro supervisor (QuestionFactory) 🧠 toma esa información y crea automáticamente el tipo correcto de pregunta con todas las características específicas. El sistema se encarga de esto y de hacer la retroalimentación si algo necesita de corrección ✅.

2.5 Diagrama UML

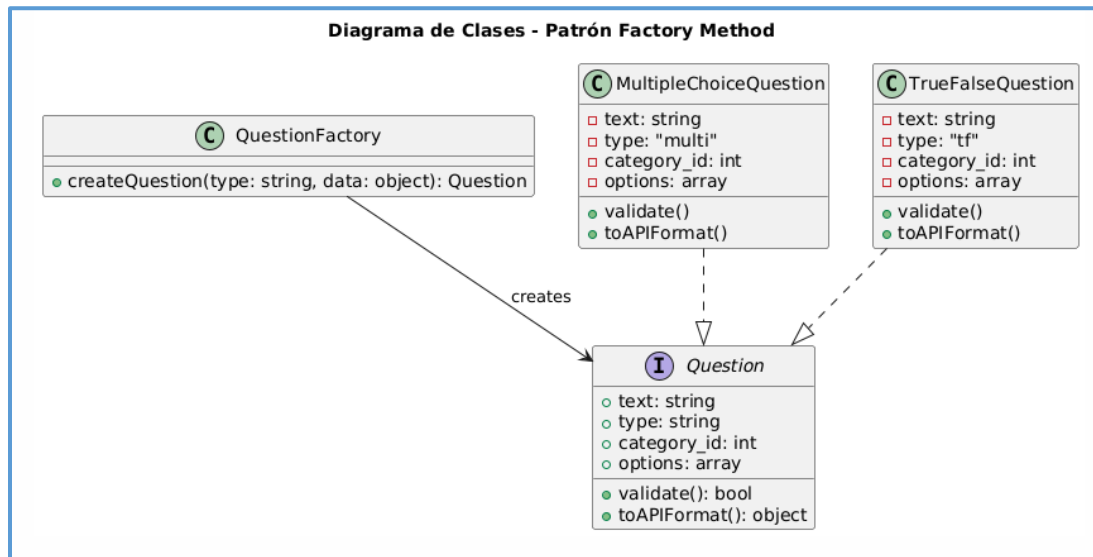


Diagrama UML, implementación Patrón Factory Method

Este diagrama muestra cómo se organiza el sistema 🧩 para crear diferentes tipos de preguntas en el proyecto. Son tres componentes principales: **QuestionFactory** que es la clase creadora, la interfaz **Question** 📄 que define el contrato común que deben cumplir todas las preguntas, y los productos concretos que tenemos hasta ahora que son **MultipleChoiceQuestion** y **TrueFalseQuestion** ✅ ❌.

La clase **QuestionFactory** actúa como el punto central de creación utilizando su método `createQuestion(type, data)` 🛠️ para determinar qué tipo de pregunta instanciar. Dependiendo del parámetro `type` recibido, la fábrica crea objetos de las clases concretas **MultipleChoiceQuestion** y **TrueFalseQuestion**. Ambos tipos de preguntas implementan la misma interfaz **Question**, garantizando que tengan una integridad de los datos 🔒.

3 ¿Por qué Elegimos este Patrón?

Una de las principales razones por las que elegimos este patrón es porque hace que nuestro sistema sea muy fácil de usar para las personas 🧑. Se encarga de todos los detalles técnicos,

permitiendo que los usuarios se concentren en lo verdaderamente importante 🌟. Además, guía automáticamente el proceso y previene errores comunes, como crear una pregunta de opción múltiple con una sola opción ❌.

Por otro lado, el sistema está diseñado para ser escalable. Si en el futuro quisiéramos agregar nuevos tipos de preguntas ❓, este patrón nos permitirá hacerlo de forma sencilla. Solo necesitamos crear la “receta” correspondiente y notificar al supervisor que existe un nuevo tipo 🗨️. Todo lo demás seguirá funcionando sin necesidad de modificar las partes ya implementadas 🔧.

Este enfoque también mantiene nuestro código bien organizado 📁. Cada tipo de pregunta tiene su propio espacio con todas sus reglas específicas, lo que facilita hacer cambios sin afectar otras partes del sistema 🔧🚫. Esto también ayuda a que nuevos desarrolladores comprendan el funcionamiento y puedan contribuir fácilmente al proyecto 🤝.

Además, el patrón nos permite prevenir diversos errores 🚫. Por ejemplo, evita la creación de preguntas de verdadero o falso con más de dos opciones ❌, o preguntas de selección múltiple sin ninguna respuesta correcta. Esta validación automática es fundamental, ya que las preguntas mal formuladas pueden confundir a los usuarios.

En conclusión, la implementación del patrón Factory Method 🏢 ha traído beneficios tanto para los usuarios como para el equipo de desarrollo 👤. Por un lado, mejora la experiencia de uso gracias a una interfaz intuitiva y validaciones claras ✅. Por otro, facilita el mantenimiento, mejora la organización del código y permite escalar el sistema con facilidad 📁. A nivel general 🌐, este patrón se convirtió en una base sólida para el crecimiento del proyecto. Aprendimos que invertir en una buena estructura desde el inicio permite adaptarse fácilmente a nuevas necesidades en las preguntas sin afectar la estabilidad del sistema ⚙️.