



Uso Factory Method en Proyecto Ingenieria de Software I

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




1.1 Definición






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 .

1.2 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 😊, él 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

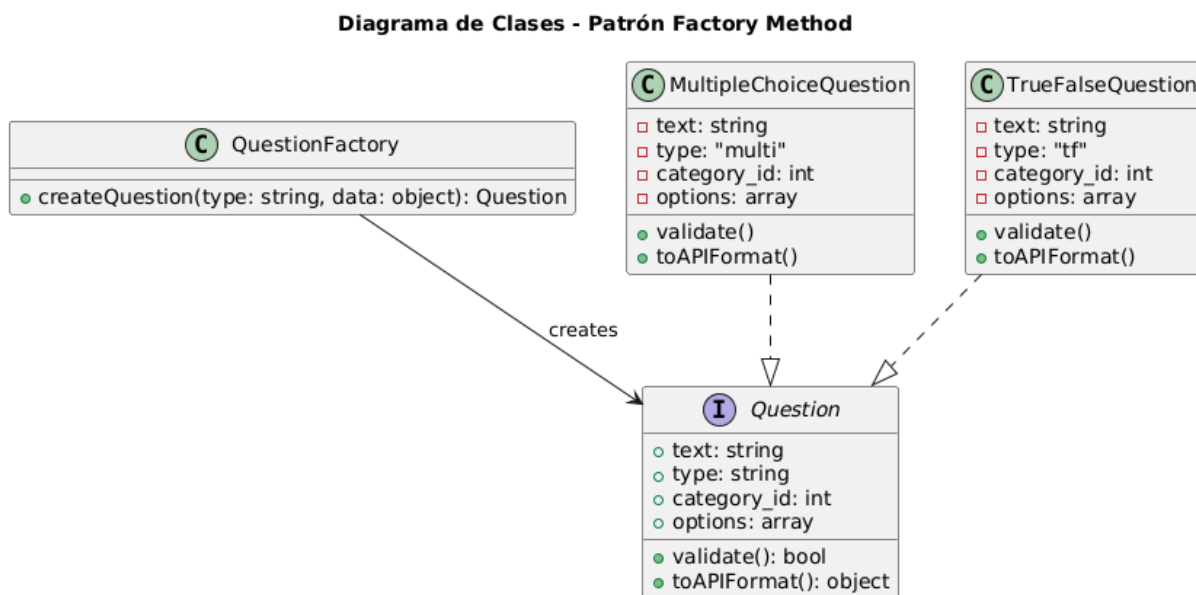


Diagrama UML, implementación Patrón Factory Method

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?

3.1 Simplicidad para los usuarios

Una de las razones principales por las que elegimos este patrón es que hace que nuestro sistema sea muy fácil de usar para las personas 🧑🧑. El sistema se encarga de todos los detalles técnicos ⚙️, permitiendo que se enfoque en lo realmente importante ⭐. El sistema guía automáticamente el proceso 🔄 y previene errores como crear una opción múltiple con solo una opción ❌.

3.2 Facilidad para agregar nuevos tipos

Nuestro sistema está diseñado para su escalabilidad 📈, si en un futuro quisiéramos agregar más tipos de preguntas ❓, con este patrón podremos agregar nuevos tipos de forma muy sencilla 😊.

Cuando queremos agregar un nuevo tipo de pregunta, solo necesitamos crear la “receta” específica para ese tipo 📄 y decirle a nuestro supervisor que existe este nuevo tipo 🧠. Y todo el resto del sistema seguirá funcionando exactamente igual ✅, sin necesidad de cambiar nada en las partes que ya funcionan 🔄.

3.3 Organización del código

Este patrón mantiene nuestro código muy organizado 🗂️, pues cada tipo de pregunta tiene su propio espacio en el código donde están todas sus reglas específicas 📌. Esto significa que si necesitamos cambiar algo sobre algún tipo de pregunta 🛠️, podremos hacerlo fácilmente y no correremos el riesgo de afectar nada en el sistema 🚫. Esto también hace muy fácil para nuevos desarrolladores entender el sistema 👤 y contribuir al proyecto 🤝.

3.4 Prevención de errores

El patrón nos ayuda a prevenir muchos tipos de errores 🚫 que se puedan presentar, ejemplo de esto puede ser que sea imposible crear una pregunta de verdadero o falso con tres opciones ? ✖️, o una pregunta de selección múltiple sin ninguna respuesta correcta 🔍. Esta validación automática 🔵 es especialmente importante, donde preguntas mal formadas pueden llegar a confundir a los usuarios 😬.

4 Conclusiones de beneficios para nuestro proyecto

La implementación del patrón Factory Method 🏢 en nuestro sistema trajo beneficios tanto para los usuarios 🙌 como para el equipo de desarrollo del proyecto 👤 👤. Esto se debe a que la experiencia de uso se volvió más sencilla ✨, pudiendo crear tipos de preguntas sin preocuparse por especificaciones, donde la interfaz intuitiva 💻 nos ayuda y con el método hacemos las validaciones ✅ para que el usuario sepa qué ocurre cuando está creando sus preguntas.

Por otro lado, para el equipo técnico, este patrón facilita el mantenimiento 🛠️ del sistema y la escalabilidad del mismo 📈, incluso permitiendo agregar nuevos tipos sin muchas complicaciones, al igual mejoró esta parte de la organización del código 🗂️.

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 ⚙️.