UNIVERSIDAD ANÁHUAC

UNIVERSIDAD ANÁHUAC
QUERÉTARO

Facultad de Ingeniería

Ingeniería Informática y Negocios Digitales

Ingeniería de Software II

Docente: Karla Vianney Ramos Garcia

Actividad 5

David Ceballos Mata - 00476577

08/02/2026

Análisis del problema y elección del patrón

Requerimientos funcionales principales

- El sistema debe mostrar una interfaz, la cual, tanto su UI y su funcionamiento deben de estar adaptados a la región. Por ejemplo, elegí hacer de México y Europa, en la interfaz europea el monto que se debe ingresar es en euros, mientras que en México es de pesos. Para hacer la diferencia más notoria, el impuesto que se aplica en los reportes financieros es diferente (no necesariamente son los de la vida real), en México establecí un 32% y en Europa un 35%.
- El sistema debe generar un reporte financiero.
- No debe mezclarse una interfaz europea con un reporte mexicano o viceversa.
- El sistema debe facilitar agregar nuevas regiones sin cambiar el código que ya existe.

¿El sistema necesita crear un solo tipo de objeto o familias de objetos relacionados?

El sistema requiere crear familias de objetos relacionados. No basta con crear un solo objeto, ya que la interfaz y el reporte pueden compartir normas regionales, por lo que la interfaz y el reporte cambian juntos. Esto define claramente familias de objetos y no objetos aislados.

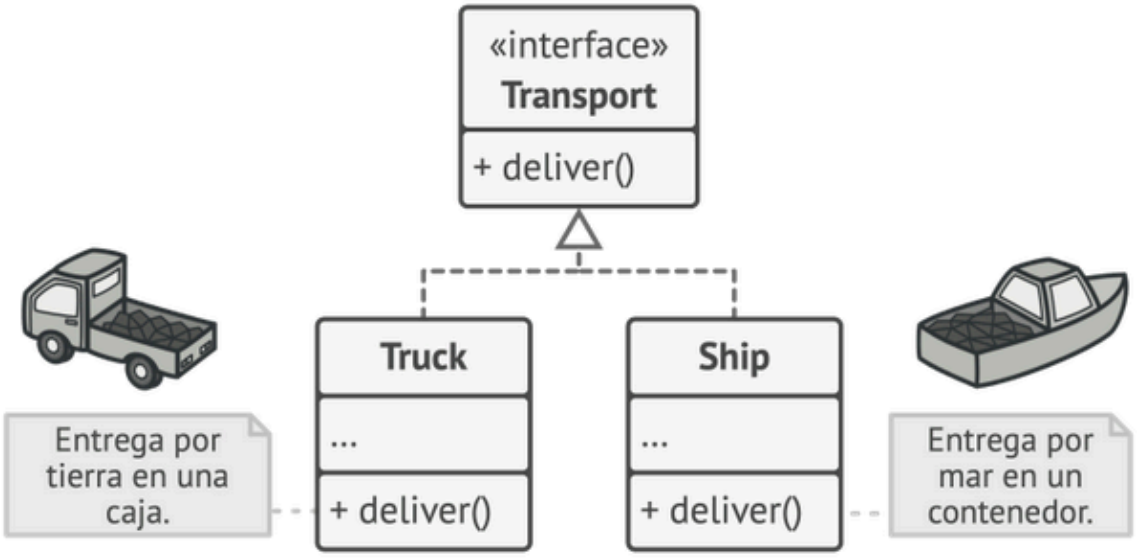
¿Es importante que los objetos creados mantengan consistencia entre sí?

Sí, en este caso, la consistencia entre objetos es fundamental para evitar inconsistencias como una interfaz europea con euros, pero una un reporte en pesos y con impuestos mexicanos. La consistencia entre objetos asegura que haya coherencia entre interfaz y reportes.

FactoryMethod:¿En qué casos se utiliza?

Este patrón de diseño creacional el cual proporciona una interfaz para crear objetos de una superclase, mientras que permite que las subclases alteren el tipo de objetos que se crean. Se emplea cuando el sistema necesita crear un solo tipo de objeto, se quiere delegar la creación de subclases, el cliente no puede conocer la clase concreta que se instancia y cuando la diferencia o variación está en una clase, no en un conjunto de

clases. Su principio clave es dejar que las subclases decidan qué objeto crear. Ejemplo:

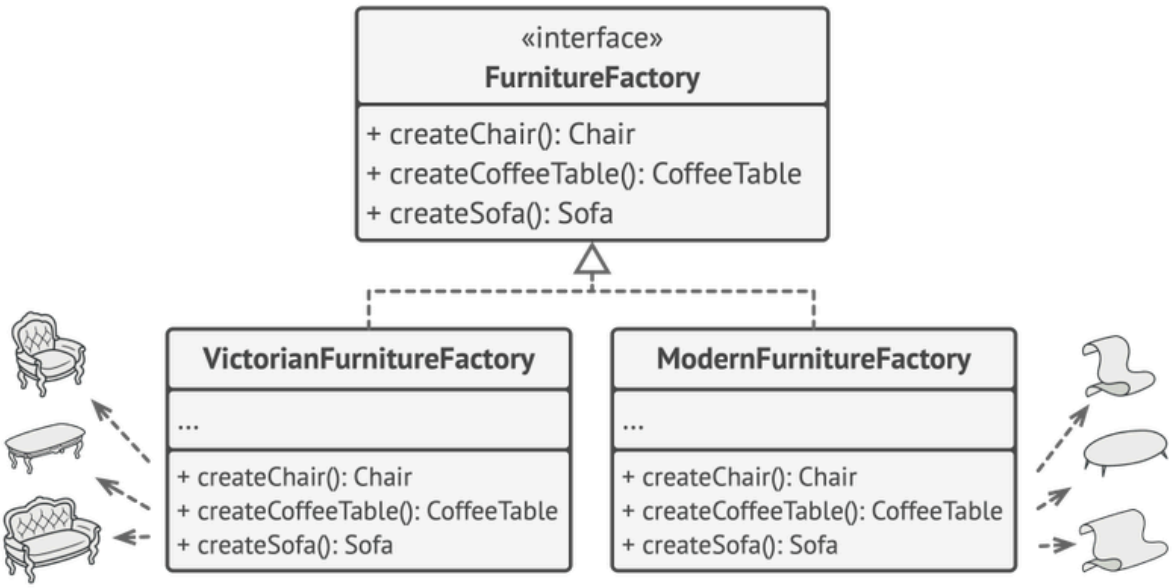


Todos los productos deben seguir la misma interfaz.

Donde un transporte puede ser un camión o un barco, ambos reparten, sin embargo, la forma en la que lo hacen es diferente.

Abstract Factory: ¿En qué casos se utiliza?

Este patrón de diseño creacional se usa cuando se requieren crear familias de objetos relacionados sin especificar sus clases concretas, dichos objetos deben de ser congruentes entre sí, y, así como en el patrón anterior, el cliente debe de ser independiente de las clases concretas.



Cada fábrica concreta se corresponde con una variante específica del producto.

En el ejemplo anterior lo único que cambiaba era la forma en la que los transportes reparten, aquí, pueden existir varios tipos de muebles con diferentes estilos. En esta situación conviene más utilizar Abstract Factory.

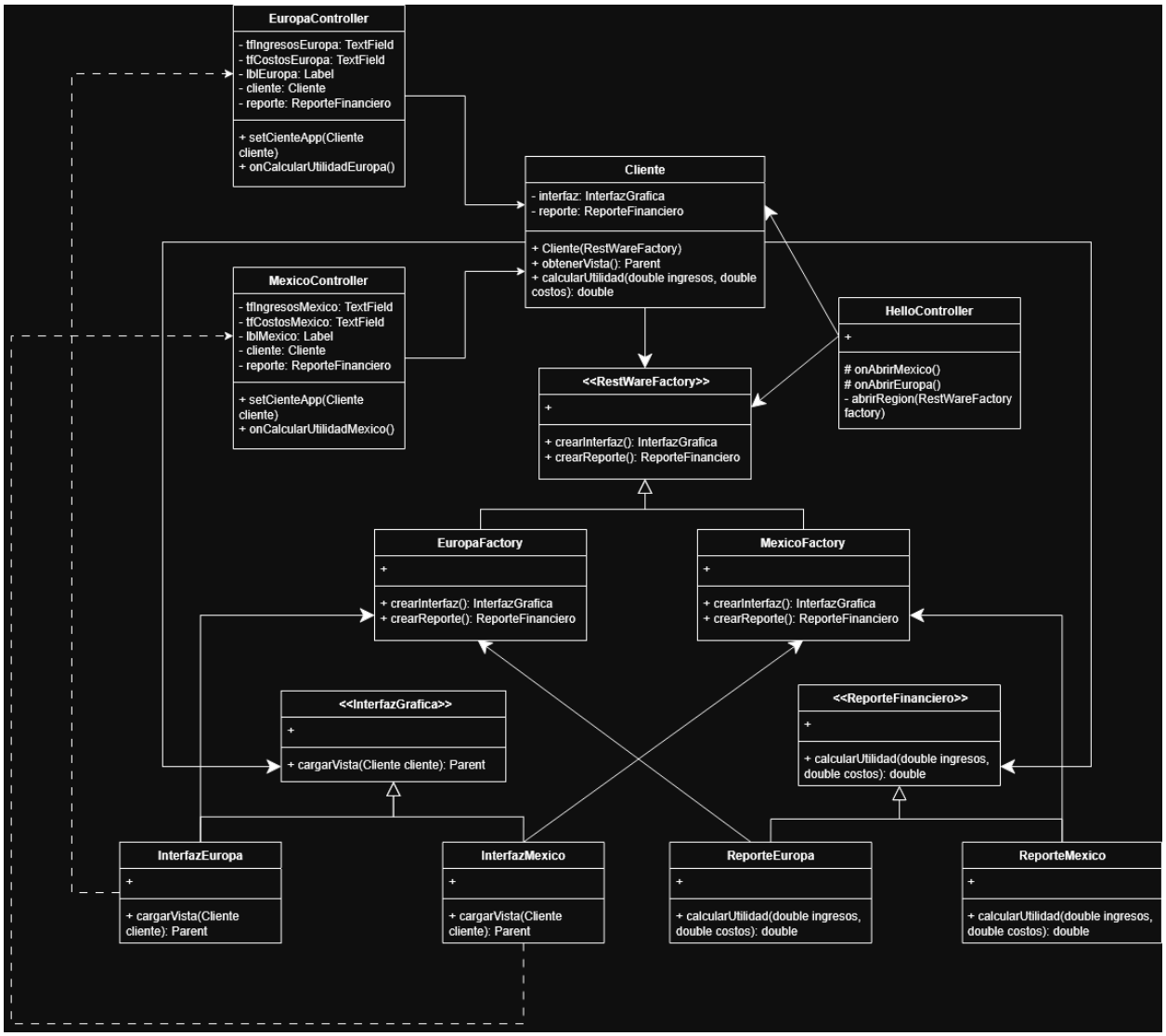
¿Cuál se ajusta mejor al problema planteado?

Considero que Abstract Factory es el patrón adecuado, ya que va a facilitar crear interfaces gráficas con reportes financieros, mantener consistencia regional, permitir agregar nuevas regiones sin modificar código existente. Por otro lado, Factory Method no es suficiente, ya que sólo crea un tipo de objeto y obliga a que la consistencia se maneje manualmente.

Elección de patrón

Elegí Abstract Factory, ya que Factory se utiliza cuando se necesita crear un solo tipo de objeto, delegando las decisiones a las subclases, por lo que no es suficiente para cumplir con los requerimientos. Abstract Factory, se emplea cuando hay familias de objetos relacionados y mantener consistencia entre ellos. En el caso que se plantea de RestWare, Abstract Factory se ajusta perfectamente, ya que permite crear interfaces gráficas y reportes financieros manteniendo coherencia con la región y facilita en gran medida la adición de nuevas regiones sin tener que modificar el código existente.

Diseño del sistema



Extensibilidad

¿Cómo se agregarían nuevas regiones?

Para añadir una nueva región, por ejemplo, “China”, simplemente se crean nuevos componentes que sigan el "contrato" ya establecido por las interfaces. El sistema está diseñado para que la lógica de creación esté aislada, lo que permite el programa maneje una nueva región conectando las nuevas piezas al inicio del programa, sin tocar una sola línea de la lógica de México o Europa.

Nuevas clases necesarias

- Una Fábrica Concreta: (ChinaFactory) que implemente RestWareFactory.
- Un Producto de Interfaz: (InterfazChina) que implemente InterfazGrafica y cargue sus propios archivos .fxml y .css.
- Un Producto de Reporte: (ReporteChina) que implemente ReporteFinanciero con las reglas fiscales de esa zona.
- Además, de un AsiaController, el cuál maneje los componentes de la UI como los botones, campos de texto y labels.

¿Por qué no es necesario modificar el código existente?

Gracias al acoplamiento. Las regiones actuales, tanto México como Europa, están encapsuladas en sus propias clases. Como el resto del programa (el Cliente y la lógica principal) interactúa únicamente con las interfaces y no con las clases concretas, lo cual hace que no importa cuántas regiones nuevas se agreguen. Al no haber sentencias if/else o switch, que decidan dependiendo del nombre de las regiones, el código que ya existe permanece completamente cerrado a cambios.

Reflexión Final

Se seleccionó este patrón, principalmente porque el problema requería crear familias de objetos relacionados. El patrón Abstract Factory garantiza la consistencia: asegura que un usuario de la región México nunca reciba por error un reporte con el impuesto de Europa. Esta consistencia se debe a que la fábrica coordina que todos los productos creados pertenezcan a la misma variante regional.

Me parece que en el mundo empresarial/profesional, este patrón aporta dos beneficios principales:
Mantenimiento y escalabilidad: Reduce considerablemente la deuda técnica. Si alguna ley fiscal o el impuesto llega a cambiar en México, solo se modifica la clase ReporteMexico. Se elimina el riesgo de romper otras partes del sistema, ya que los componentes se encuentran aislados.
Productividad del Equipo: Creo que permite que diferentes desarrolladores trabajen en regiones distintas al mismo tiempo sin llegar a generar conflictos, ya que cada equipo trabaja en sus componentes desacoplados, desarrollando implementaciones específicas bajo una estructura común y estándar.

Referencias

- Factory method. (s. f.). https://refactoring.guru/es/design-patterns/factory-method
- Abstract Factory. (s. f.). https://refactoring.guru/es/design-patterns/abstract-factory
- Abstract Factory en Java / Patrones de diseño. (s. f.). https://refactoring.guru/es/design-patterns/abstract-factory/java/example