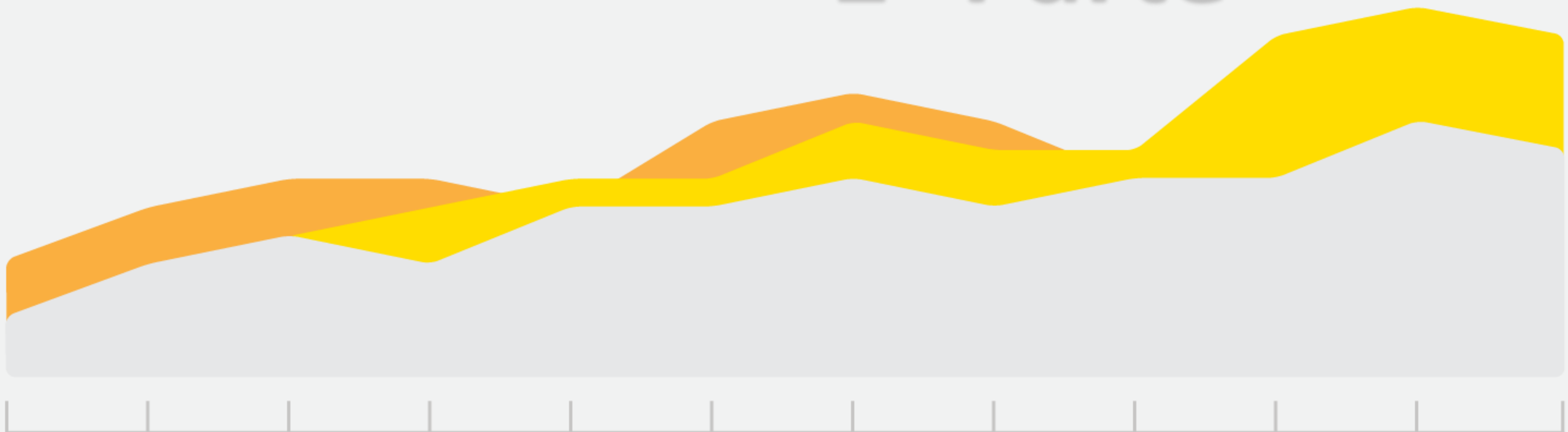
oldemar rodríguez
CONSULTOR en MINERIA DE DATOS

Patrones de Diseño Orientados a Objetos 1° Parte



Idea detrás del Diseño de Patrones

- "...describa un problema que sucede una y otra vez en nuestro entorno, y luego describa el núcleo de la solución a ese problema, de tal forma que pueda utilizar esa solución un millón de veces más, sin siquiera hacerlo dos veces de la misma manera.”
- *Christopher Alexander*

Origen del Diseño de Patrones

- El avance reciente más importante en el diseño orientado a objetos es probablemente el movimiento de los patrones de diseño, inicialmente presentado en "Design Patterns", por Gamma, Helm, Johnson y Vlissides AddisonWesley, 1995, que suele llamarse el libro de la "Banda de los Cuatro" (en inglés, GoF: Gang of Four). El GoF muestra 23 soluciones para clases de problemas muy particulares.

El Concepto de Patrón

- En principio, puede pensar en un patrón como una manera especialmente inteligente e intuitiva de resolver una clase de problema en particular. ***Parece que un equipo de personas han estudiado todos los ángulos de un problema y han dado con la solución más general y flexible*** para ese tipo de problema.
- Este problema podría ser uno que usted ha visto y resuelto antes, pero su solución probablemente no tenía la clase de completitud que verá plasmada en un patrón.
- Es más, ***el patrón existe independientemente de cualquier implementación particular*** y puede implementarse de numerosas maneras.

El Concepto de Patrón

- El concepto básico de un patrón puede verse también como el concepto básico del diseño de programas en general: ***añadir capas de abstracción***. Cuando se abstrae algo, se están aislando detalles concretos, y una de las razones de mayor peso para hacerlo es separar las cosas que cambian de las cosas que no.
- Por lo tanto, el objetivo de los patrones de diseño es ***encapsular el cambio***. Si lo enfoca de esta forma. Por ejemplo, la herencia podría verse como un patrón de diseño (aunque uno implementado por el compilador).

Clasificación de los Patrones

- ***Creacional:*** Cómo se puede crear un objeto. Habitualmente esto incluye aislar los detalles de la creación del objeto, de forma que su código no dependa de los tipos de objeto que hay y por lo tanto, no tenga que cambiarlo cuando añada un nuevo tipo de objeto.
- Los más conocidos son los patrones de ***Instancia Única*** (Singleton) y ***Fábricas Abstractas*** (Factories).

Clasificación de los Patrones

- ***Estructural:*** Esto afecta a la manera en que los objetos se conectan con otros objetos para asegurar que los cambios del sistema no requieren cambiar esas conexiones.
- Los patrones estructurales suelen imponerlos las restricciones del proyecto.
- Entre los Patrones estructurales tenemos:
  - Modelo Vista Controlador – MVC
  - Adaptador - Adapter.
  - Puente - Bridge.
  - Decorador o Envoltorio - Decorator.

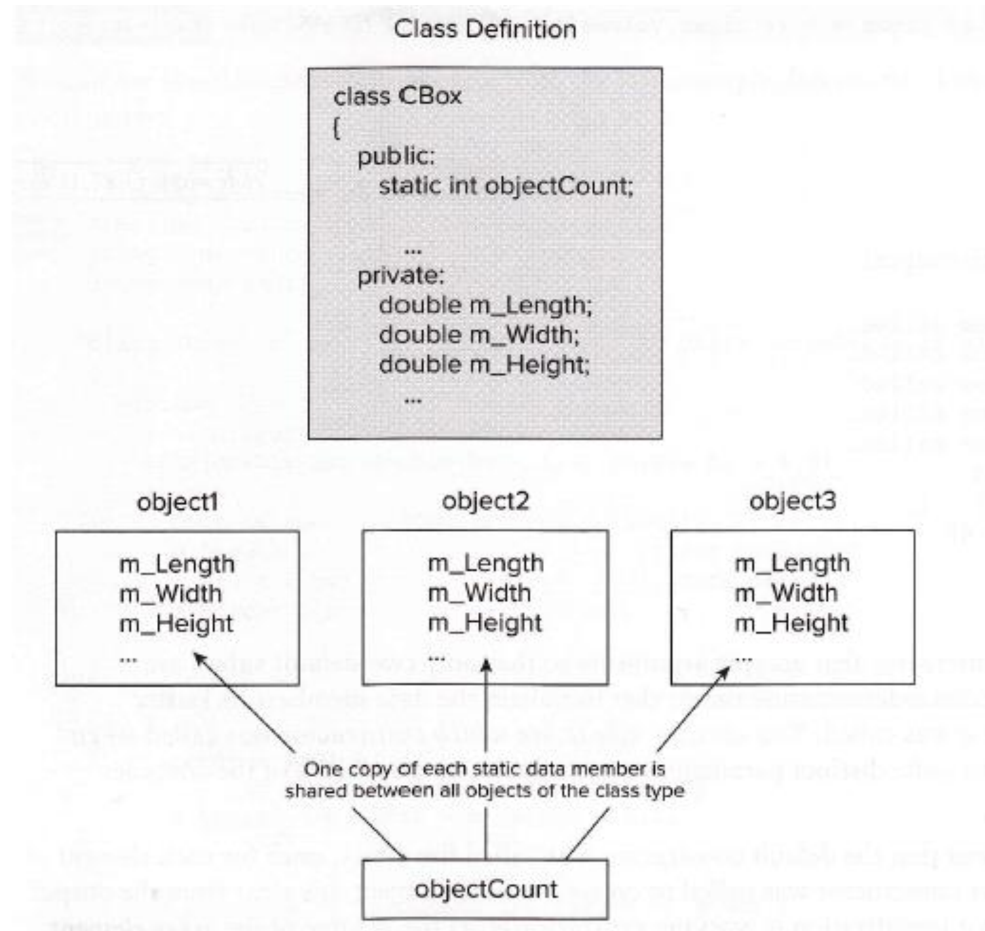
oldemar rodríguez
CONSULTOR en MINERÍA DE DATOS

Clasificación de los Patrones

- **Comportacional:** Objetos que manejan tipos particulares de acciones dentro de un programa. Éstos encapsulan procesos que quiere que se ejecuten, como interpretar un lenguaje, completar una petición, moverse a través de una secuencia (como en un iterador) o implementar un algoritmo.
- Entre los Patrones estructurales tenemos:
  - Comando (Command),
  - Método Plantilla (Template Method),
  - Estado (State),
  - Estrategia (Strategy),
  - Cadena de Responsabilidad (Chain of Responsibility),
  - ***Observador (Observer),***
  - Despachador Múltiple (Multiple Dispatching)
  - Visitador (Visitor).

Recordemos: Miembros Estáticos en una Clase

<Ej2.cpp>



oldemar rodríguez
CONSULTOR en MINER14 DE D4T0S

Instancia Única - Singleton

- Posiblemente, el patrón de diseño más simple del GoF es el Singleton, que es una forma de asegurar una única instancia de una clase.
- La clave para crear un Singleton es evitar que el programador cliente tenga control sobre el ciclo de vida del objeto. Para lograrlo, declare todos los constructores privados, y evite que el compilador genere implícitamente cualquier constructor.
- **Ver Ej1.CPP**
- Fíjese que el constructor de copia y el operador de asignación (que intencionadamente carecen de implementación alguna, ya que nunca van a ser llamados) están declarados como privados, para evitar que se haga cualquier tipo de copia.

Ejercicio 4: Tarea 8

- Programe las siguientes 2 clases
  - VentasEmpresa que tiene como atributos la cantidad de unidades vendidas por la empresa y total de ventas en colones que esto representa, lo anterior como una clase "Singlenton".
  - Vendedor que tiene como atributos unidades vendidas por él y total de ventas en colones que esto representa, además cuando una instancia de Vendedor vende este debe actualizar la clase Singlenton VentasEmpresa.
- Programe en el main() 5 instancias de Vendedor para probar el programa.

oldemar rodríguez
CONSULTOR en MINERIA DE DATOS

Fábricas: Encapsular la creación de objetos

- Cuando se descubre que en un programa se necesitan añadir nuevos tipos a un sistema, el primer paso ***más sensato es usar polimorfismo*** para crear una interfaz común para esos nuevos tipos. Así, se separa el resto del código en el sistema del conocimiento de los tipos específicos que se están añadiendo.
- Por ejemplo, agregar en nuestra jerarquía de *Personas* un nuevo "tipo" ***EstudianteHistoria***

Fábricas: Encapsular la creación de objetos

- La solución es forzar a que la creación de objetos se lleve a cabo a través de una ***Fábrica común***, en lugar de permitir que el código creacional se disperse por el sistema.
- Si todo el código del programa debe ir a esta ***Fábrica*** cada vez que necesita crear uno de esos objetos, todo lo que hay que hacer para añadir un objeto es modificar la Fábrica.

Fábricas: **Encapsular la creación de objetos**

- Los tipos nuevos pueden añadirse sin "molestar" al código existente, o eso parece. A primera vista, podría parecer que hace falta cambiar el código únicamente en los lugares donde se hereda un tipo nuevo, ***pero esto no es del todo cierto***.
- Todavía hay que crear un objeto de este nuevo tipo, y ***en el momento de la creación hay que especificar que constructor usar***.
- Ejemplo → Asistente A; EstCompu E;

oldemar rodríguez
CONSULTOR en MIN3R14 D3 D4T0S

Fábricas: Encapsular la creación de objetos

- **Ver Ej3.cpp**

```
FormaGeomerica* FormaGeomerica::Fabrica(const string& type)
  throw(FormaGeomerica::BadShapeCreation) {
  if(type == "Circulo")
      return new Circulo;
  if(type == "Cuadrado")
      return new Cuadrado;
  throw BadShapeCreation(type);
}
```

Fábricas: Encapsular la creación de objetos

- El método ***Fabrica()*** toma un argumento que le permite determinar qué tipo de figura crear. Aquí, el argumento es una cadena, pero podría ser cualquier conjunto de datos. El método ***Fabrica()*** es el único código del sistema que hay que cambiar cuando se añade un nuevo tipo de Figura.
- Para asegurar que la creación sólo puede realizarse con el método ***Fabrica()***, los constructores de cada tipo específico de Figura se hacen privados, y la clase ***FormaGeomerica*** se declara como "friend" de forma que el método ***Fabrica()*** tiene acceso a los mismos.

oldemar rodríguez
CONSULTOR en MINERÍA DE DATOS

Modelo general de una Fábrica Abstracta

Cliente

ProductoAbstractoA

FabricaAbstracta

CrearProductoA()
CrearProductoB()

ProductoA2

ProductoA1

ProductoAbstractoB

Fabrica1

CrearProductoA()
CrearProductoB()

Fabrica2

CrearProductoA()
CrearProductoB()

ProductoB2

ProductoB1

oldemar rodríguez
CONSULTOR en M1NƐR14 DƐ D4T0S

Fábricas polimórficas

- La función estática ***Fabrica()*** en el ejemplo anterior fuerza que las operaciones de creación se centren en un punto, de forma que sea el único sitio en el que haya que cambiar código.
- Sin embargo, el GoF enfatiza que la razón de ser del patrón "Fábricas" es que diferentes tipos de fábricas se puedan derivar de la fábrica básica.
- Así tendremos Fábricas polimórficas.

Fábricas: Encapsular la creación de objetos

- **Ver Ej4.cpp**

```
class FabricaFormaGeometrica {
  virtual FormaGeometrica* Crear()=0;
  static map<string, FabricaFormaGeometrica*> Fabricas;
public:
  virtual ~FabricaFormaGeometrica() {}
  friend class InicializadorFabricaFormasGeometricas;
  static FormaGeometrica* CrearFormaGeometrica(const string& id) throw(MalaCreacionFormaGeometrica) {
     if(Fabricas.find(id) != Fabricas.end())
        return Fabricas[id]->Crear();
     else
       throw MalaCreacionFormaGeometrica(id);
  }
};
```

Ejercicio 5: Tarea 8

- Modifique la Jerarquía de Persona, Estudiante, EstCompu y Asistente de manera que las Instancias de estas clase sean creadas mediante una Fábrica Abstracta (la más versión simple sin archivos ni listas).
- Escriba un programa de prueba con un par de instancias de cada tipo.

oldemar rodríguez
CONSULTOR en MINERIA DE DATOS

Gracias….

oldemar rodríguez

CONSULTOR en M1NƐR14 DƐ D4TØS