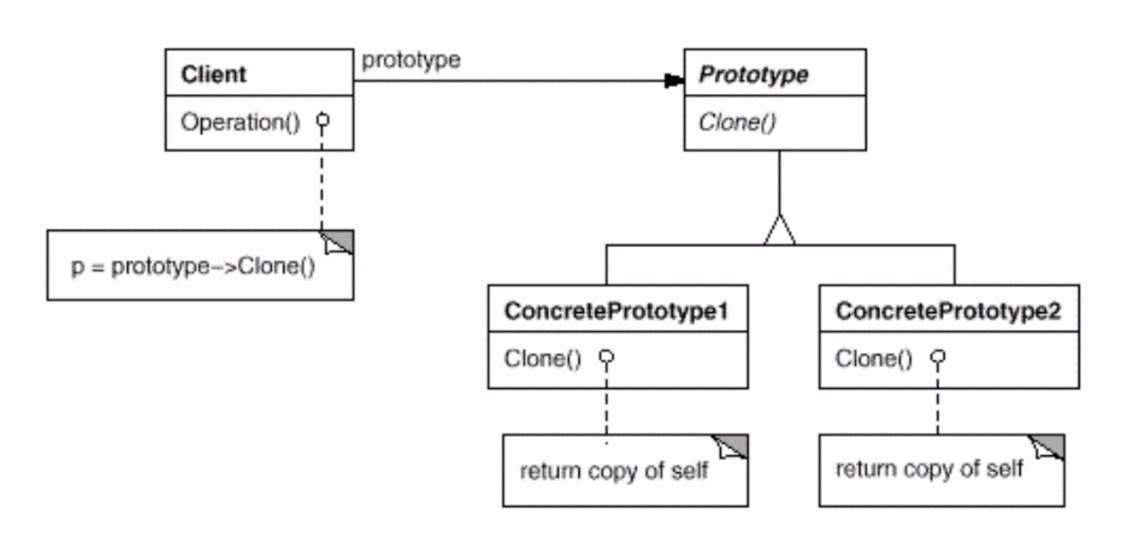
|  |  |
| --- | --- |
| Universidad Tecnológica Metropolitana | Texto  Descripción generada automáticamente |
| Arquitectura de Software  Patrones de diseño en C#  Maestro: M.G.T.I. Uh Pérez Oscar Josué | |
| Integrantes:   * Peniche Estrada Ronald Eduardo * Suarez Jiménez Jonathan Enrique * Quijano Vázquez Carlos Eduardo | |
| Cuatrimestre: 7mo  Ingeniería en Desarrollo y Gestión de Software | |
| Fecha de entrega: 30/11/2021 | |

Antes de empezar el enlace al repositorio de github en donde encontrará el resto de los patrones es el siguiente: <https://github.com/Soldier-2790/patronesCSHARP.git>

Patrones de diseño:

Creacional.

PROTOTYPE

Sirve para clonar información de un objeto existente a uno nuevo del mismo tipo. Cuando se desean pasar datos de un objeto a otro, hacer una asignación del tipo Objeto1 = Objeto2 no es efectivo y asignar los datos, variable por variable es tedioso y puede dar problemas como cuando hay campos privados. Se escogió Prototype porque es problema que resuelve resulto de interés y bastante útil. Un ejemplo de su aplicación seria cuando se tiene que hacer copias de un archivo, como podría ser un archivo de Word en donde se puede retroceder con Control Z a versiones anteriores, esas versiones podrían crearse usando Prototype.

Captura de pantalla de computadora

Descripción generada automáticamenteEjemplo:

Se tienen dos clases (Persona y Address), la primera tiene un objeto de la segunda.

Captura de pantalla de computadora

Descripción generada automáticamenteSe crea un objeto Persona llamado Juan con sus datos y luego es clonado con otro objeto con mismo tipo con el nombre JuanManuel. Los datos mostrados en consola son idénticos en ambos.

Captura de pantalla de computadora

Descripción generada automáticamentePosteriormente se modifican algunos datos al objeto clonado JuanManuel y al comparar los datos, solo han cambiado en la copia

Captura de pantalla de computadora

Descripción generada automáticamenteResultados finales en consola. Al modificar los datos en la clonación, estos no cambian en la original puesto que son dos objetos distintos y no referencias en la memoria.

Estructural.

ADAPTER

Este patrón permite a un objeto que tiene una base parecida, pero composición o acciones diferentes ser adaptado a un grupo de clases que tiene como base una clase padre de tipos abstracta, en simples palabras permite que las acciones de una clase tengan coherencia tanto con el padre como con sus hermanos.

Diagrama

Descripción generada automáticamenteEste patrón es muy interesante de analizar, ya que permite relacionar una clase que tiene características similares, pero con una presentación diferente, por ejemplo, podemos tener un programa en la que se está dando a conocer las acciones de un motor de diferentes tipos. Su diagrama puede ser el siguiente:

Texto

Descripción generada automáticamenteEn la parte de clases tenemos los siguientes:

Motor es la clase padre y los motores Diesel, Naftero y ElectricoAdatpter son sus hijos, la clase motor electrico es la que no comparte las mismas caracteriscas que ka clase motor, estos se componen así de la siguiente manera.

Texto

Descripción generada automáticamenteLa clase abstracta Motor se compone de lo siguiente:

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteLa clase hijo MotorDiesel, MotorNaftero y MotorElectricoAdapter, se componen de lo siguiente:

Como se pueden dar cuenta estas tres clases están relacionadas con la clase motor.

Texto

Descripción generada automáticamentePero la clase motor eléctrico es una adaptación de otra que no tiene el mismo dictamen que la clase motor.

Texto

Descripción generada automáticamenteA lo que voy es que en la clase motor eléctrico tiene el mismo concepto pero sus accions son diferentes por lo que la clase motor eléctrico adapter es la que lo relaciona con la otra. Para que se tenga un contexto más alla de un uso aplicativo, sería el de poner a los usuarios y de ahí poner los roles por medio de lo que debe hacer y así. El resultado del programa es el siguiente sería el siguiente:

De comportamiento.

CHAIN OF RESPONSIBILITY

Es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Al igual que muchos otros patrones de diseño de comportamiento, el Chain of Responsibility se basa en transformar comportamientos particulares en objetos autónomos llamados manejadores. Cada comprobación debe ponerse dentro de su propia clase con un único método que realice la comprobación. La solicitud, junto con su información, se pasa a este método como argumento.

Diagrama

Descripción generada automáticamenteEl patrón sugiere que se vinculen esos manejadores en una cadena. Cada manejador vinculado tiene un campo para almacenar una referencia al siguiente manejador de la cadena. Además de procesar una solicitud, los manejadores la pasan a lo largo de la cadena. La solicitud viaja por la cadena hasta que todos los manejadores han tenido la oportunidad de procesarla. Un manejador puede decidir no pasar la solicitud más allá por la cadena y detener con ello el procesamiento.

Código

using System;

using System.Collections.Generic;

namespace RefactoringGuru.DesignPatterns.ChainOfResponsibility.Conceptual

{

public interface IHandler

{

IHandler SetNext(IHandler handler);

object Handle(object request);

}

abstract class AbstractHandler : IHandler

{

private IHandler \_nextHandler;

public IHandler SetNext(IHandler handler)

{

this.\_nextHandler = handler;

return handler;

}

public virtual object Handle(object request)

{

if (this.\_nextHandler != null)

{

return this.\_nextHandler.Handle(request);

}

else

{

return null;

}

}

}

class MonkeyHandler : AbstractHandler

{

public override object Handle(object request)

{

if ((request as string) == "platano")

{

return $"Mono: voy a comer el {request.ToString()}.\n";

}

else

{

return base.Handle(request);

}

}

}

class SquirrelHandler : AbstractHandler

{

public override object Handle(object request)

{

if (request.ToString() == "Bellota")

{

return $"Ardilla: voy a comer la {request.ToString()}.\n";

}

else

{

return base.Handle(request);

}

}

}

class DogHandler : AbstractHandler

{

public override object Handle(object request)

{

if (request.ToString() == "Alboniga")

{

return $"Perro: voy a comer la {request.ToString()}.\n";

}

else

{

return base.Handle(request);

}

}

}

class Client

{

public static void ClientCode(AbstractHandler handler)

{

foreach (var food in new List<string> { "Bellota", "Platano", "Taza de cafe" })

{

Console.WriteLine($"Cliente: Quien quiere {food}?");

var result = handler.Handle(food);

if (result != null)

{

Console.Write($" {result}");

}

else

{

Console.WriteLine($" {food} Quedo sin pedir.");

}

}

}

}

class Chain

{

static void Main(string[] args)

{

var monkey = new MonkeyHandler();

var squirrel = new SquirrelHandler();

var dog = new DogHandler();

monkey.SetNext(squirrel).SetNext(dog);

Console.WriteLine("Cadena: Mono > Ardilla > Perro\n");

Client.ClientCode(monkey);

Console.WriteLine();

Console.WriteLine("SubCadena: Ardilla > Perro\n");

Client.ClientCode(squirrel);

}

}

}

Enlace al repositorio en donde se encuentran los demás patrones: <https://github.com/Soldier-2790/patronesCSHARP.git>