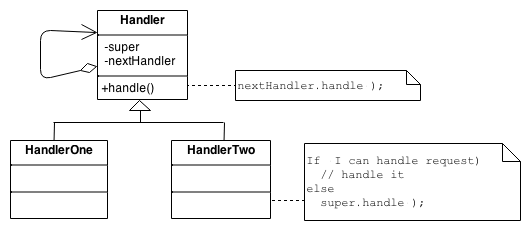
# Chain of Responsibility

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.

El 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");

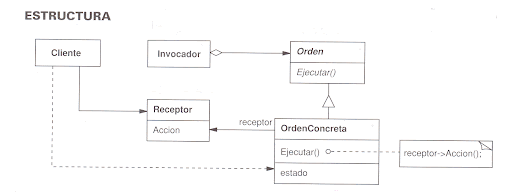
Client.ClientCode(squirrel);

}

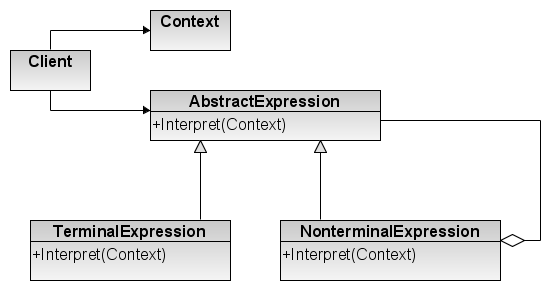
}

}

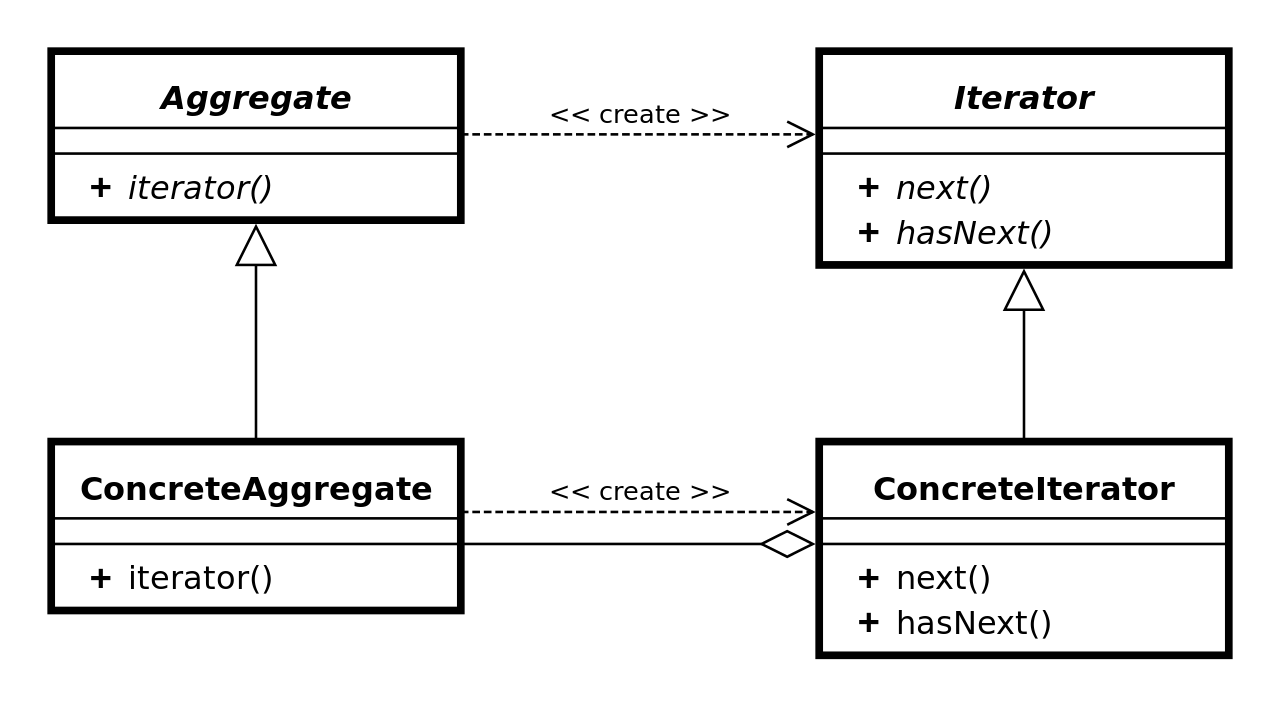
# Command



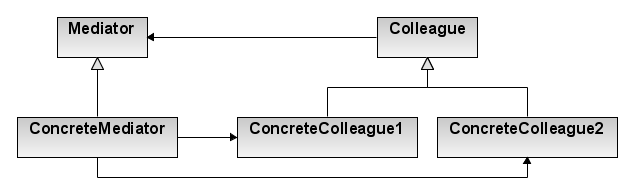
# Interpreter



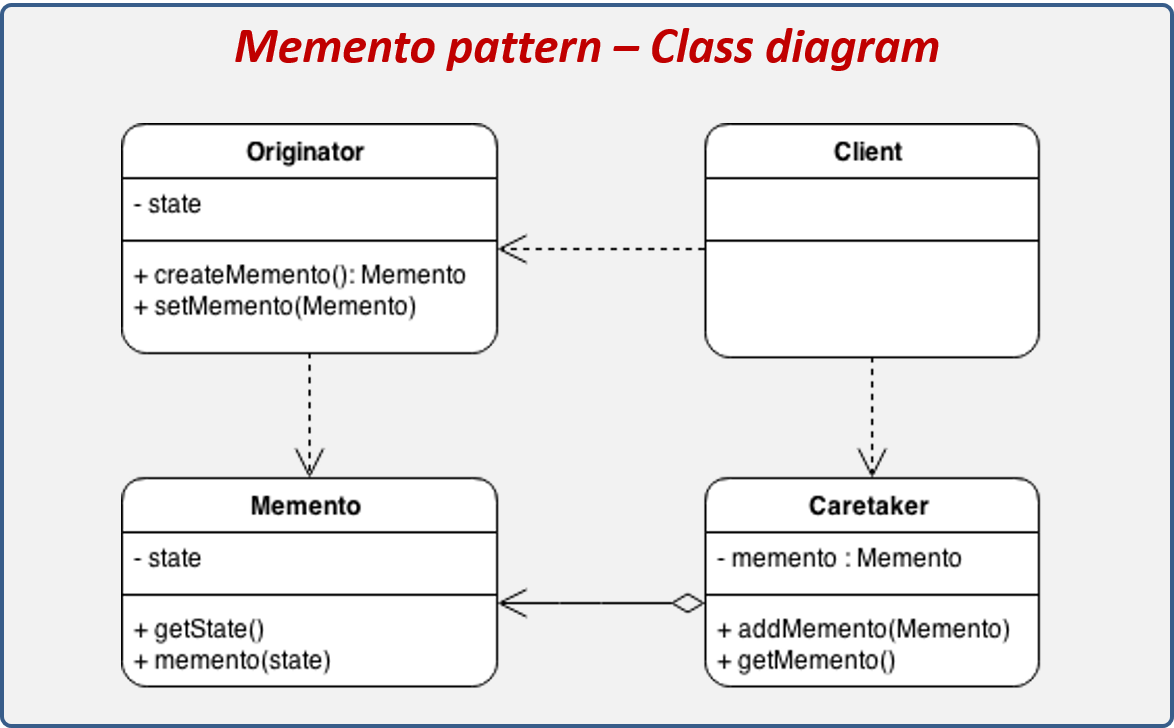
# Iterator



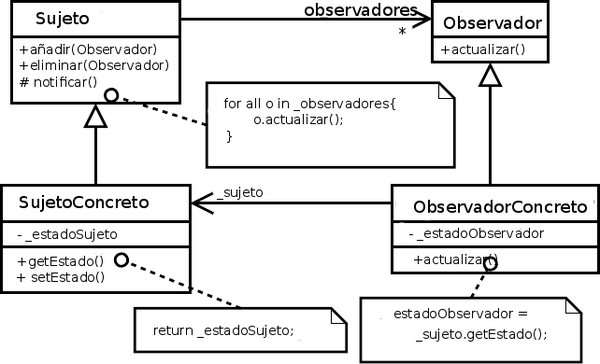
# Mediator



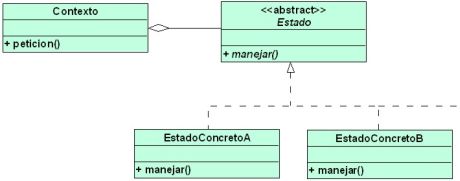
# Memento



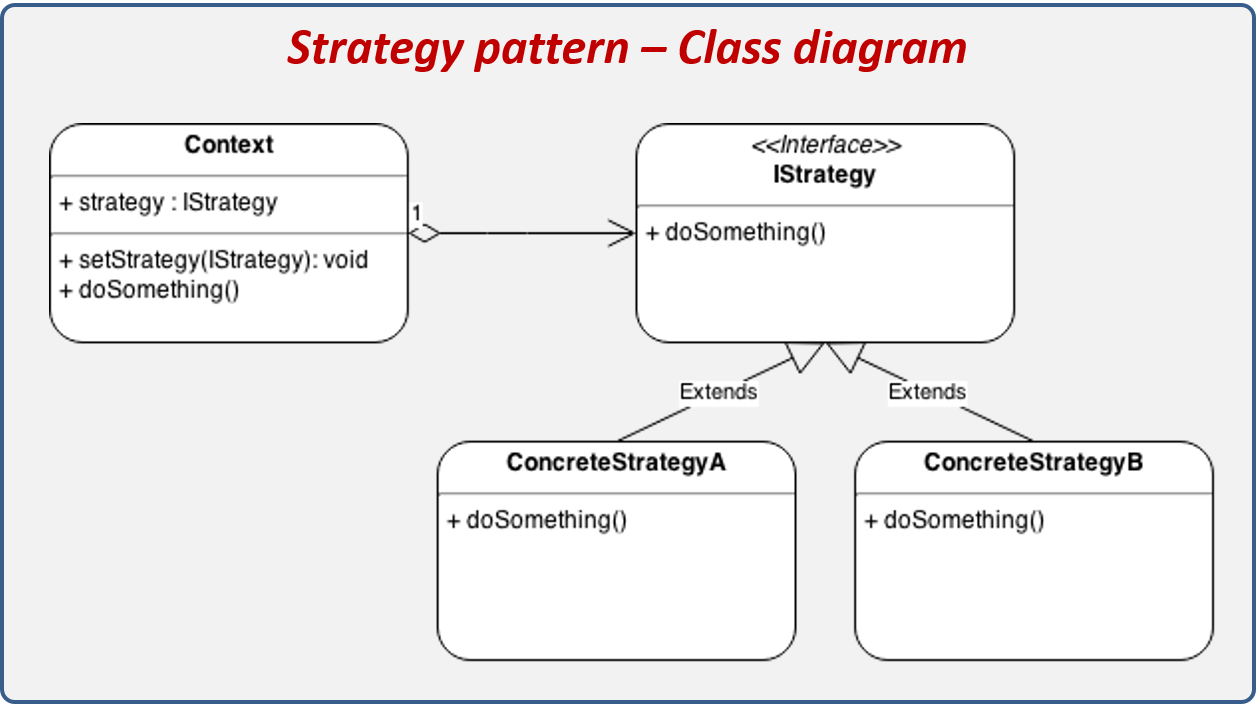
# Observer



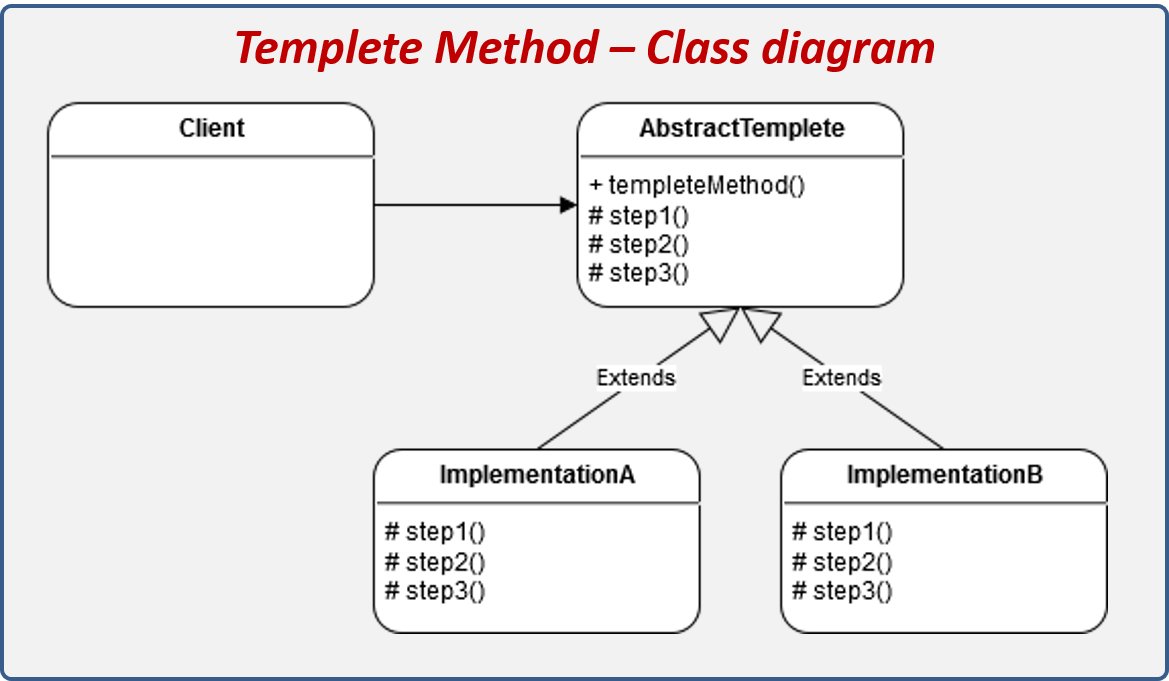
# State



# Strategy



# Template Method



# Visitor

