

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 21: Patrones Creacionales

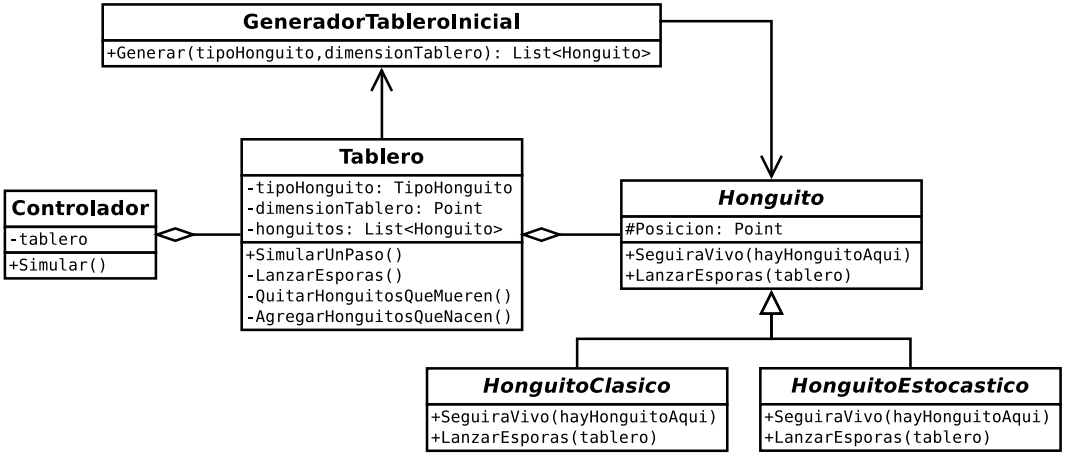
Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

21 de Octubre, 2025

Solución

Simulando la vida de un honguito



Simulando la vida de un honguito

De hecho, la clase `GeneradorTableroInicial` tiene este método:

```
2  private static void AgregarHonguitoSegunTipo(Point posicion) {
3    Honguito honguito;
4    if (tipoHonguito == TipoHonguito.Clasico)
5      honguito = new HonguitoClasico(posicion, dimensionTablero);
6    else if (tipoHonguito == TipoHonguito.Estocastico)
7      honguito = new HonguitoEstocastico(posicion, dimensionTablero);
8    else
9      throw new NotImplementedException("El honguito no existe.");
10   honguitos.Add(honguito);
11 }
```

Simulando la vida de un honguito

La situación en Tablero es aún peor:

```
2 private void AgregarHonguitoSiCorresponde(int x, int y){
3   if (tipoHonguito == TipoHonguito.Clasico)
4     AgregarHonguitoClasicoSiCorresponder(x, y);
5   else if (tipoHonguito == TipoHonguito.Estocastico)
6     AgregarHonguitoEstocasticoSiCorresponder(x, y);
7   else
8     throw new NotImplementedException("El honguito no existe.");
9 }
```

Simulando la vida de un honguito

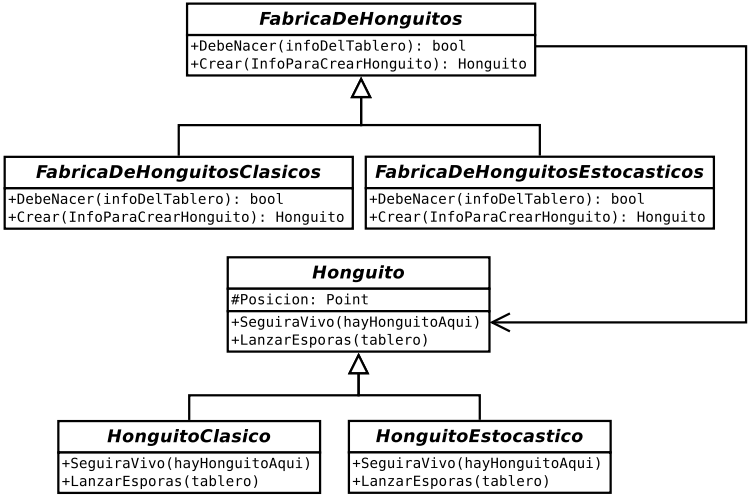
La situación en Tablero es aún peor:

```
2 private void AgregarHonguitoSiCorresponde(int x, int y){
3   if (tipoHonguito == TipoHonguito.Clasico)
4     AgregarHonguitoClasicoSiCorresponder(x, y);
5   else if (tipoHonguito == TipoHonguito.Estocastico)
6     AgregarHonguitoEstocasticoSiCorresponder(x, y);
7   else
8     throw new NotImplementedException("El honguito no existe.");
9 }
```

¿Cómo arreglarían esto?

Solución

Crear una fábrica de honguitos.



Solución

La fábrica abstracta define la interfaz de la fábrica:

```
2 public abstract class FabricaDeHonguitos{
3     public abstract bool DebeNacer(Point pos, bool[,] honguitos, int
      [,] esporas);
4     public abstract Honguito Crear(Point pos, Point dimTab);
5 }
```

Solución

La fábrica abstracta define la interfaz de la fábrica:

```
2  public abstract class FabricaDeHonguitos{
3      public abstract bool DebeNacer(Point pos, bool[,] honguitos, int
       [,] esporas);
4      public abstract Honguito Crear(Point pos, Point dimTab);
5  }
```

Luego cada fábrica construye objetos del tipo que le corresponda:

```
2  public class FabricaDeHonguitosClasicos : FabricaDeHonguitos {
3      public override bool DebeNacer(Point pos, bool[,] honguitos, int
       [,] esporas){
4          bool hayTresEsporas = esporas[pos.X, pos.Y] == 3;
5          bool laCasillaEstaVacia = !honguitos[pos.X, pos.Y];
6          return laCasillaEstaVacia && hayTresEsporas;
7      }
8      public override Honguito Crear(Point pos, Point dimTab)
9          => new HonguitoClasico(pos, dimTab);
10 }
```

Solución

La gracia es que después la aplicación crea honguitos usando la fábrica que le demos:

```
2 int dimX = 25;
3 int dimY = 10;
4 FabricaDeHonguitos fabrica = new FabricaDeHonguitosEstocasticos();
5 Point dimTab = new Point(dimX, dimY);
6
7 Controlador controlador = new Controlador(fabrica, dimTab);
8 controlador.Simular();
```

Solución

La gracia es que después la aplicación crea honguitos usando la fábrica que le demos:

```
2 int dimX = 25;
3 int dimY = 10;
4 FabricaDeHonguitos fabrica = new FabricaDeHonguitosEstocasticos();
5 Point dimTab = new Point(dimX, dimY);
6
7 Controlador controlador = new Controlador(fabrica, dimTab);
8 controlador.Simular();
```

... y el código ya no necesita esos ifs que dependen de cada tipo.

Solución

El código del `GeneradorTableroInicial`:

```
2  private static void AgregarHonguitoSegunTipo(Point posicion) {
3    Honguito honguito;
4    if (tipoHonguito == TipoHonguito.Clasico)
5      honguito = new HonguitoClasico(posicion, dimensionTablero);
6    else if (tipoHonguito == TipoHonguito.Estocastico)
7      honguito = new HonguitoEstocastico(posicion, dimensionTablero);
8    else
9      throw new NotImplementedException("El honguito no existe.");
10   honguitos.Add(honguito);
11 }
```

Solución

El código del `GeneradorTableroInicial`:

```
2 private static void AgregarHonguitoSegunTipo(Point pos) {
3   Honguito honguito = fabricaDeHonguitos.Crear(pos, dimTab);
4   honguitos.Add(honguito);
5 }
```

Solución

El código del GeneradorTableroInicial:

```
2 private static void AgregarHonguitoSegunTipo(Point pos) {
3   Honguito honguito = fabricaDeHonguitos.Crear(pos, dimTab);
4   honguitos.Add(honguito);
5 }
```

El código del Tablero:

```
2 private void AgregarHonguitoSiCorresponde(int x, int y){
3   if (tipoHonguito == TipoHonguito.Clasico)
4     AgregarHonguitoClasicoSiCorresponder(x, y);
5   else if (tipoHonguito == TipoHonguito.Estocastico)
6     AgregarHonguitoEstocasticoSiCorresponder(x, y);
7   else
8     throw new NotImplementedException("El honguito no existe.");
9 }
```

Solución

El código del GeneradorTableroInicial:

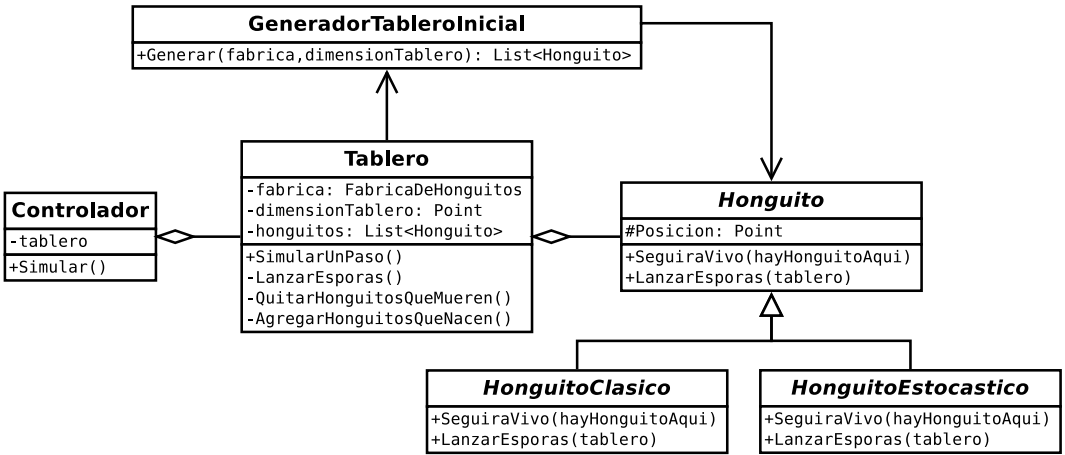
```
2 private static void AgregarHonguitoSegunTipo(Point pos) {
3   Honguito honguito = fabricaDeHonguitos.Crear(pos, dimTab);
4   honguitos.Add(honguito);
5 }
```

El código del Tablero:

```
2 private void AgregarHonguitoSiCorresponde(int x, int y){
3   Point pos = new Point(x, y);
4   if (fabricaDeHonguitos.DebeNacer(pos, hayHonguito, esporas))
5     honguitos.Add(fabricaDeHonguitos.Crear(pos, dimTab));
6 }
```

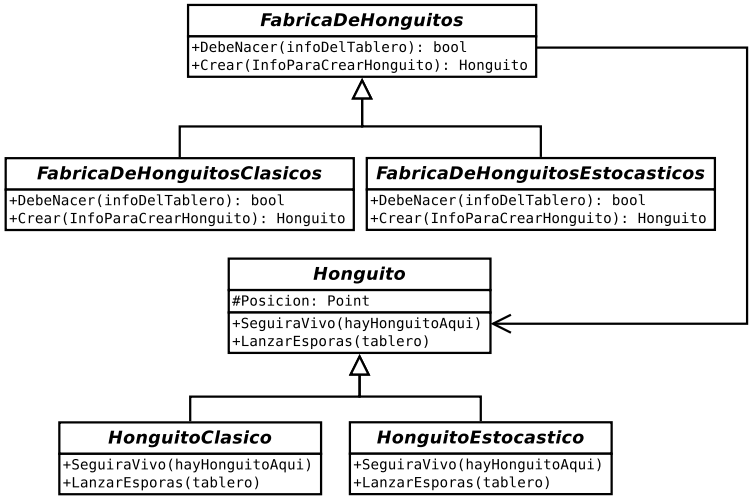
Solución

¿Qué pasa si ahora quiero crear un nuevo tipo de honguito?



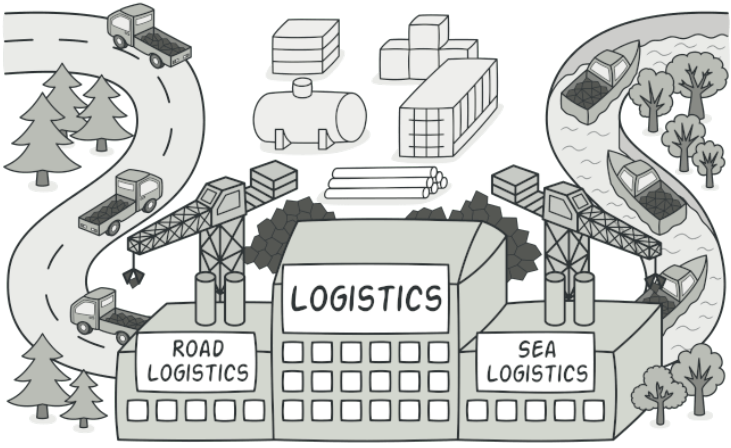
Solución

¿Qué pasa si ahora quiero crear un nuevo tipo de honguito?



Factory Method

Factory Method

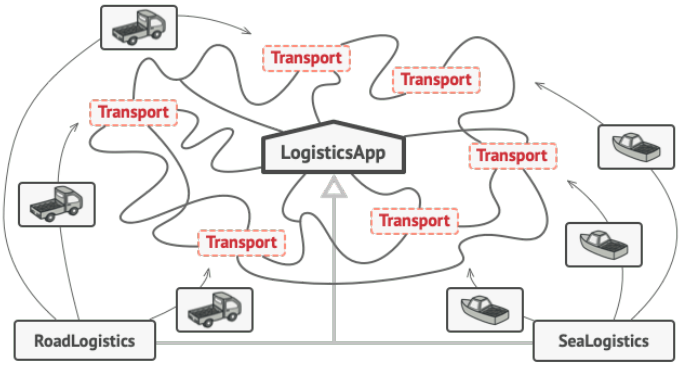


Factory Method

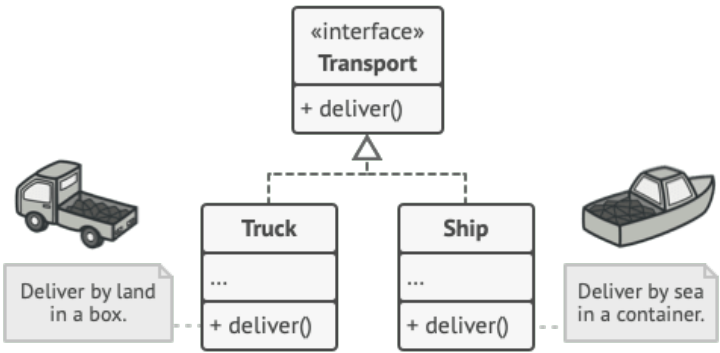




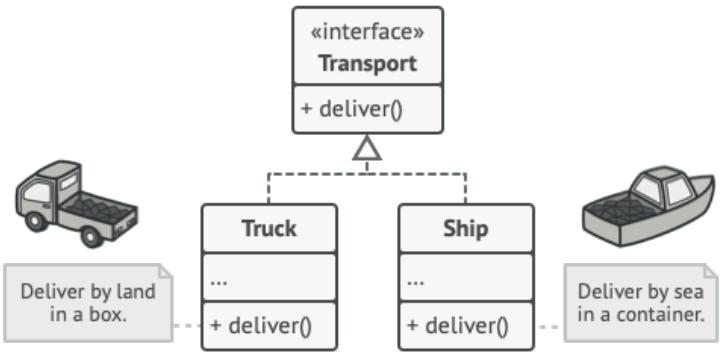
Factory Method



Factory Method



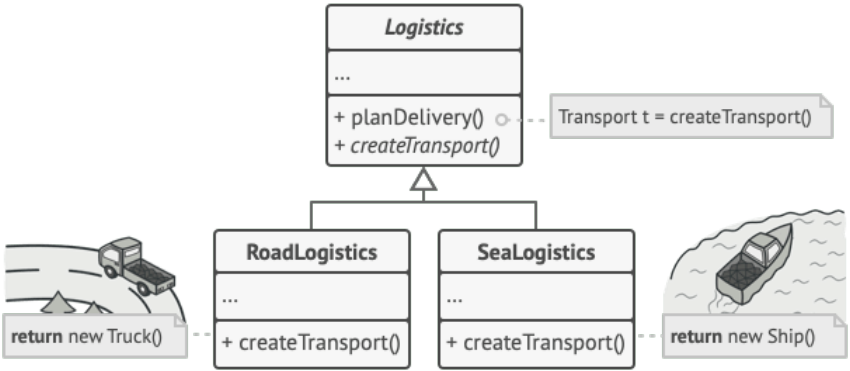
Factory Method



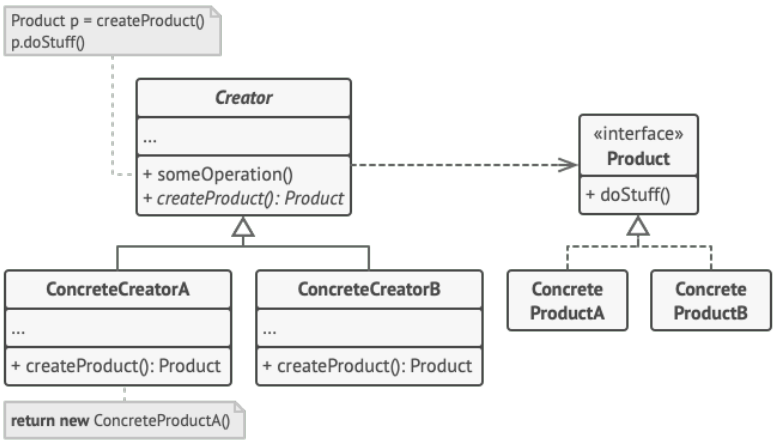
Problema: Cada `new Truck(...)` hay que cambiarlo por `new Ship(...)` 🤔

Factory Method

Solución: Agregar un método para crear el medio de transporte y sobreescribirlo.

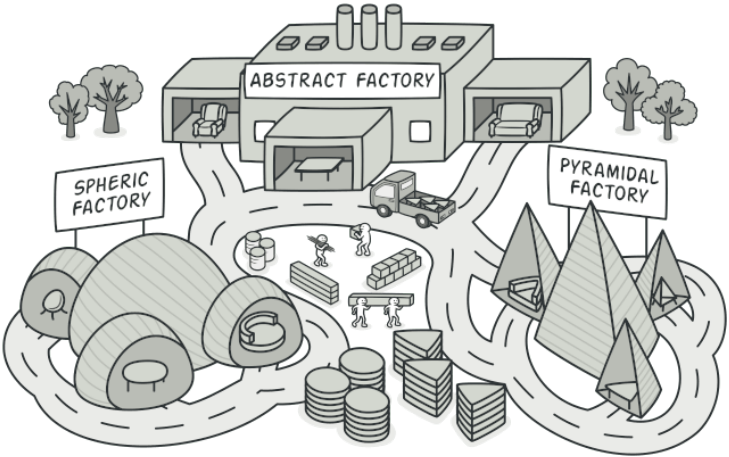


Factory Method

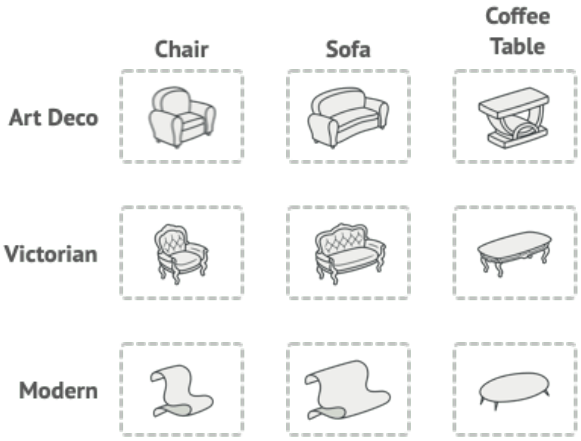


Abstract Factory

Abstract Factory



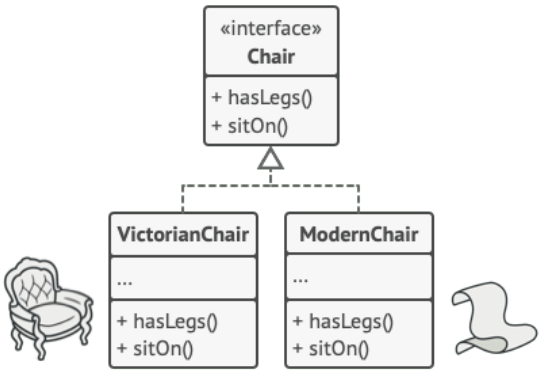
Abstract Factory



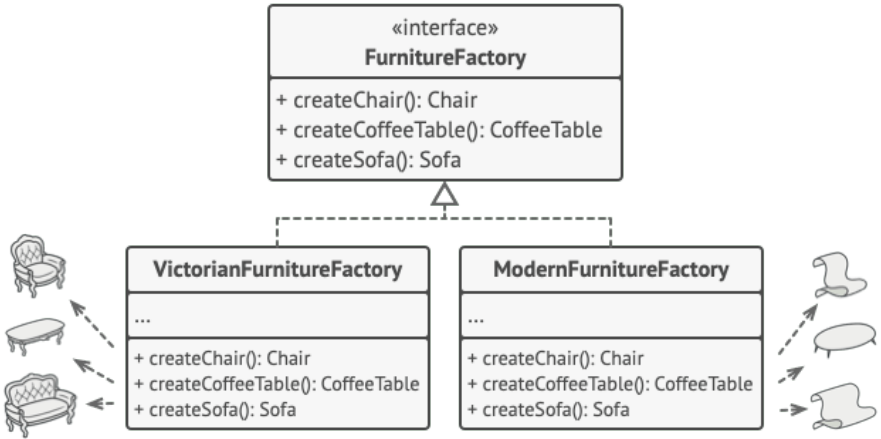
Abstract Factory



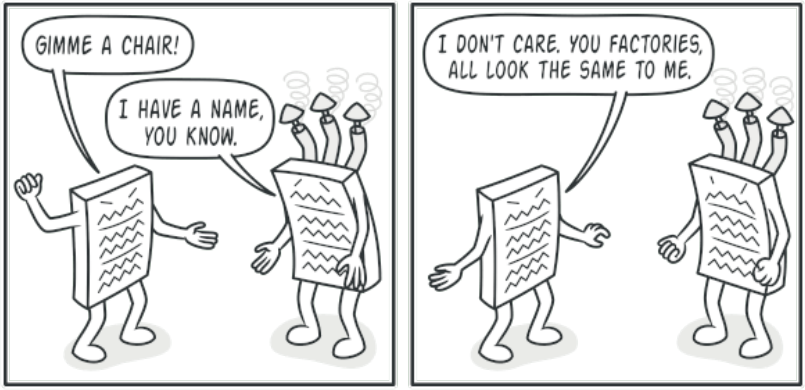
Abstract Factory



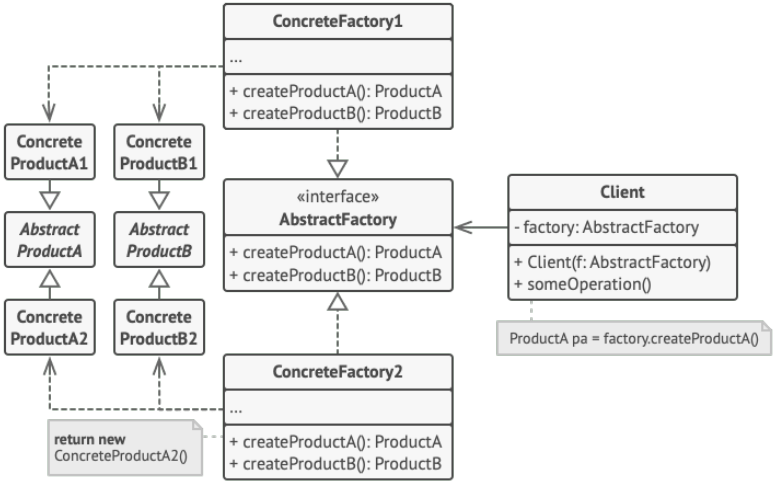
Abstract Factory



Abstract Factory



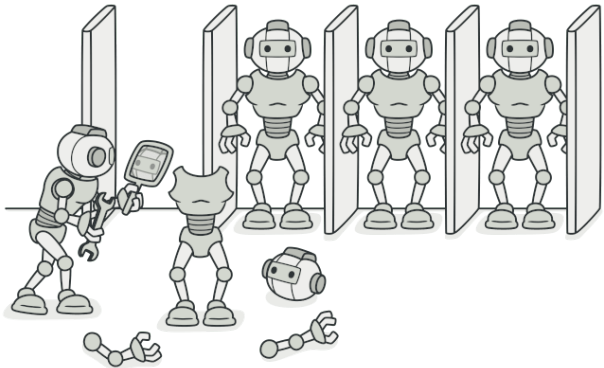
Abstract Factory



Prototype

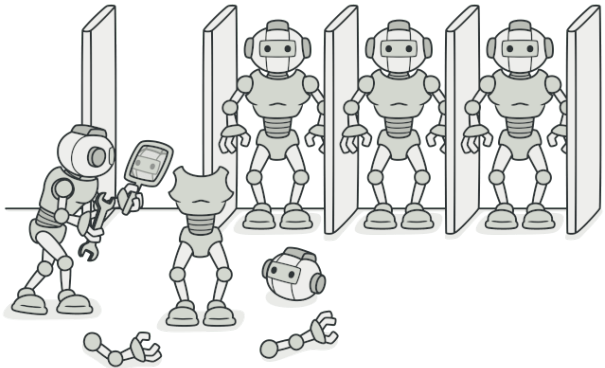
Prototype

Problema: A veces me gustaría poder clonar/copiar objetos en mi programa.



Prototype

Problema: A veces me gustaría poder clonar/copiar objetos en mi programa.



Solución: Agrega un método para clonar/copiar el objeto.

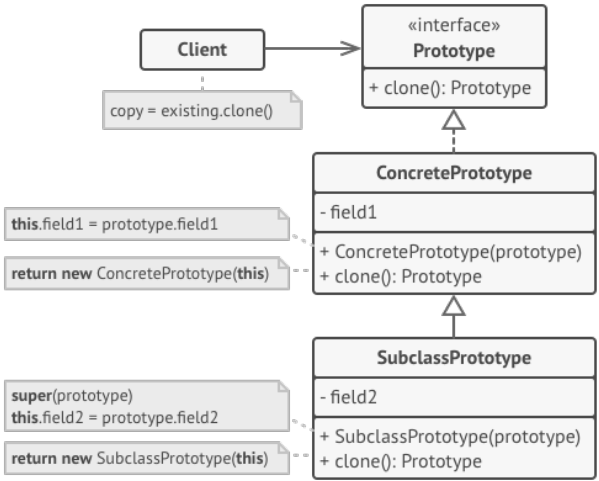
Prototype

Problema: A veces me gustaría poder clonar/copiar objetos en mi programa.



Solución: Agrega un método para clonar/copiar el objeto.

Prototype


Client
«interface»
Prototype
+ clone(): Prototype
copy = existing.clone()
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)

Prototype

Muchas clases vienen con este patrón ya implementado.

```
string imagePath = "Tests/Lenna.png";
Image<Rgb24> image = Image.Load<Rgb24>(imagePath);
Image<Rgb24> newImage = image.Clone();
newImage[20, 30] = new Rgb24(0, 255, 0);
```

Prototype

Muchas clases vienen con este patrón ya implementado.

```
2 string imagePath = "Tests/Lenna.png";
3 Image<Rgb24> image = Image.Load<Rgb24>(imagePath);
4 Image<Rgb24> newImage = image.Clone();
5 newImage[20, 30] = new Rgb24(0, 255, 0);
```

... no es la gran cosa, pero a veces resulta útil.

Prototype

Tengo un métodos que retorna la imagen en blanco y negro:

```
public Image<Rgb24> ConvertToBlackAndWhite(Image<Rgb24> image){
    for (int x = 0; x < image.Width; x++) {
        for (int y = 0; y < image.Height; y++) {
            Rgb24 px = image[x, y];
            Byte avg = (Byte)((px.R + px.G + px.B) / 3);
            image[x, y] = new Rgb24(avg,avg,avg);
        }
    }
    return image;
}
```

Prototype

Tengo un métodos que retorna la imagen en blanco y negro:

```
public Image<Rgb24> ConvertToBlackAndWhite(Image<Rgb24> image){
    for (int x = 0; x < image.Width; x++) {
        for (int y = 0; y < image.Height; y++) {
            Rgb24 px = image[x, y];
            Byte avg = (Byte)((px.R + px.G + px.B) / 3);
            image[x, y] = new Rgb24(avg,avg,avg);
        }
    }
    return image;
}
```

Para llamar al método el código es el siguiente:

```
string imagePath = "Lenna.png";
Image<Rgb24> image = Image.Load<Rgb24>(imagePath);
Image<Rgb24> BlackAndWhite = ConvertToBlackAndWhite(image);
```

Prototype

Tengo un métodos que retorna la imagen en blanco y negro:

```
2  public Image<Rgb24> ConvertToBlackAndWhite(Image<Rgb24> image){
3      for (int x = 0; x < image.Width; x++) {
4          for (int y = 0; y < image.Height; y++) {
5              Rgb24 px = image[x, y];
6              Byte avg = (Byte)((px.R + px.G + px.B) / 3);
7              image[x, y] = new Rgb24(avg,avg,avg);
8          }
9      }
10     return image;
11 }
```

Para llamar al método el código es el siguiente:

```
2 string imagePath = "Lenna.png";
3 Image<Rgb24> image = Image.Load<Rgb24>(imagePath);
4 Image<Rgb24> BlackAndWhite = ConvertToBlackAndWhite(image);
```

¿Alguien ve el problema?

Prototype

Solución 1: Cuando puedes cambiar el método.

```
2  public Image<Rgb24> ConvertToBlackAndWhite(Image<Rgb24> origImage){
3      Image<Rgb24> image = origImage.Clone();
4      for (int x = 0; x < image.Width; x++)
5          for (int y = 0; y < image.Height; y++) {
6              Rgb24 px = image[x, y];
7              Byte avg = (Byte)((px.R + px.G + px.B) / 3);
8              image[x, y] = new Rgb24(avg,avg,avg);
9          }
10     return image;
11 }
```

Prototype

Solución 1: Cuando puedes cambiar el método.

```
2  public Image<Rgb24> ConvertToBlackAndWhite(Image<Rgb24> origImage){
3      Image<Rgb24> image = origImage.Clone();
4      for (int x = 0; x < image.Width; x++)
5          for (int y = 0; y < image.Height; y++) {
6              Rgb24 px = image[x, y];
7              Byte avg = (Byte)((px.R + px.G + px.B) / 3);
8              image[x, y] = new Rgb24(avg,avg,avg);
9          }
10     return image;
11 }
```

Solución 2: Cuando no puedes cambiar el método.

```
2  string imagePath = "Lenna.png";
3  Image<Rgb24> image = Image.Load<Rgb24>(imagePath);
4  Image<Rgb24> BlackAndWhite = ConvertToBlackAndWhite(image.Clone());
```

Singleton

Singleton

Motivación: Queremos agregar un logger a nuestro código.

Singleton

Motivación: Queremos agregar un logger a nuestro código.

```
2  public class Logger {
3      private const string Path = "log.txt";
4      private StreamWriter writer;
5
6      public Logger()
7          => writer = new StreamWriter(Path);
8
9      public void Log(string text)
10         => writer.WriteLine(text);
11
12     public void Close()
13         => writer.Close();
14 }
```

... este código se ve razonable.

Singleton

Problema

Queremos poder escribir en el logger desde cualquier parte en nuestro código.

Singleton

Problema

Queremos poder escribir en el logger desde cualquier parte en nuestro código.

Opción 1: Crear nuevos loggers cada vez que queremos usar uno.

```
2 Logger logger = new Logger();
3 logger.Log("Todo bien");
```

Singleton

Problema

Queremos poder escribir en el logger desde cualquier parte en nuestro código.

Opción 1: Crear nuevos loggers cada vez que queremos usar uno.

```
2 Logger logger = new Logger();
3 logger.Log("Todo bien");
```

... esto falla porque cada `new Logger()` terminamos sobreescribiendo el log.

Singleton

Problema

Queremos poder escribir en el logger desde cualquier parte en nuestro código.

Opción 1: Crear nuevos loggers cada vez que queremos usar uno.

```
2 Logger logger = new Logger();
3 logger.Log("Todo bien");
```

... esto falla porque cada `new Logger()` terminamos sobreescribiendo el log.

Opción 2: Crear un logger en el `main` y pasarlo como argumento en todos lados.

Singleton

Problema

Queremos poder escribir en el logger desde cualquier parte en nuestro código.

Opción 1: Crear nuevos loggers cada vez que queremos usar uno.

```
2 Logger logger = new Logger();
3 logger.Log("Todo bien");
```

... esto falla porque cada `new Logger()` terminamos sobreescribiendo el log.

Opción 2: Crear un logger en el `main` y pasarlo como argumento en todos lados.

... esto funciona, pero es una lata andar pasando el logger

Singleton

Problema

Queremos poder escribir en el logger desde cualquier parte en nuestro código.

Opción 1: Crear nuevos loggers cada vez que queremos usar uno.

```
2 Logger logger = new Logger();
3 logger.Log("Todo bien");
```

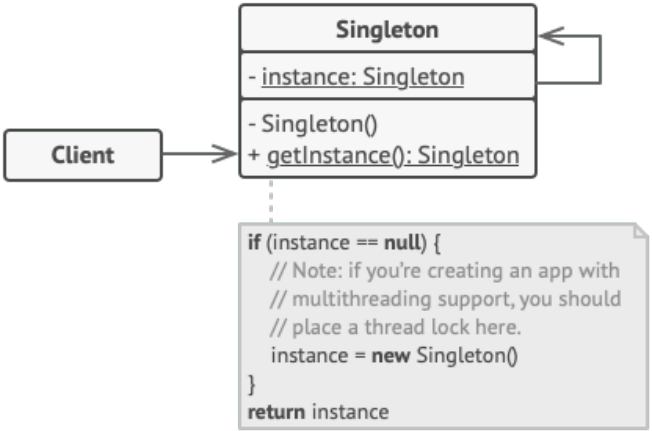
... esto falla porque cada `new Logger()` terminamos sobreescribiendo el log.

Opción 2: Crear un logger en el `main` y pasarlo como argumento en todos lados.

... esto funciona, pero es una lata andar pasando el logger

Opción 3: Usar el patrón *Singleton*.

Singleton



Esta patrón sirve cuando queremos que una sola instancia del objeto pueda ser creada (y nos gustaría acceder a esa instancia desde cualquier módulo).

Singleton

```
2  public class Logger{
3      private const string Path = "log.txt";
4      private StreamWriter writer;
5      private static Logger instance;
6
7      public static Logger GetInstance() {
8          if (instance == null)
9              instance = new Logger();
10         return instance;
11     }
12     private Logger()
13         => writer = new StreamWriter(Path);
14     public void Log(string text)
15         => writer.WriteLine(text);
16     public void Close()
17         => writer.Close();
18 }
```

Singleton

Ahora, puedes llamar al mismo logger desde donde quieras:

```
2 Logger logger = Logger.GetInstance();
3 logger.Log("Todo bien");
```

Singleton

Ahora, puedes llamar al mismo logger desde donde quieras:

```
Logger logger = Logger.GetInstance();
logger.Log("Todo bien");
```

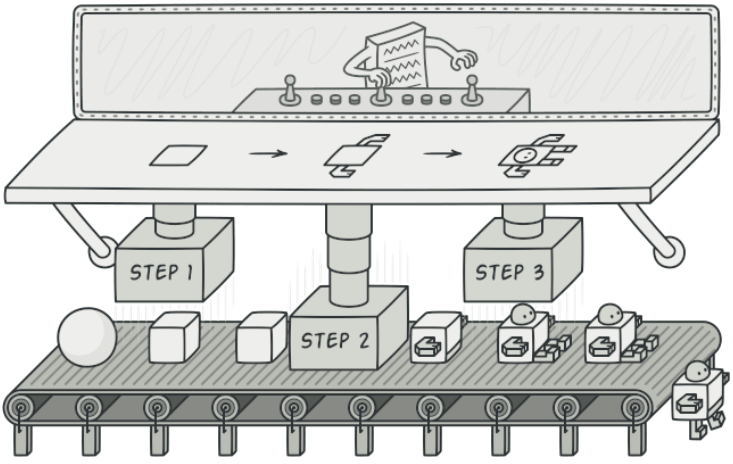
¿Cómo se diferencia esto de hacer la clase estática?

```
public static class Logger {
    private const string Path = "log.txt";
    private static StreamWriter writer = new StreamWriter(Path);

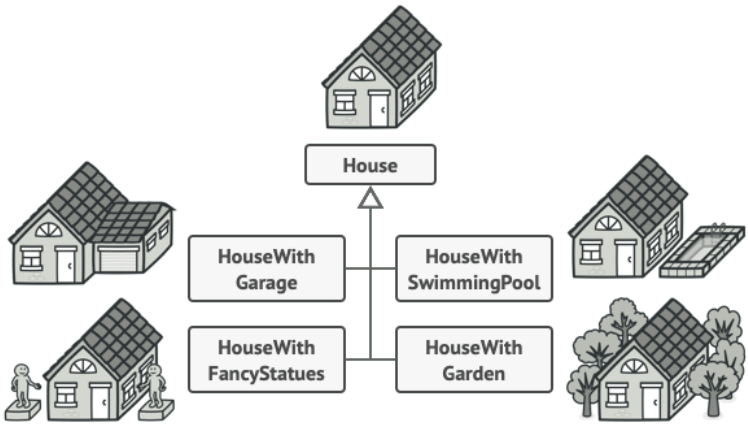
    public static void Log(string text)
        => writer.WriteLine(text);
    public static void Close()
        => writer.Close();
}
```

Builder

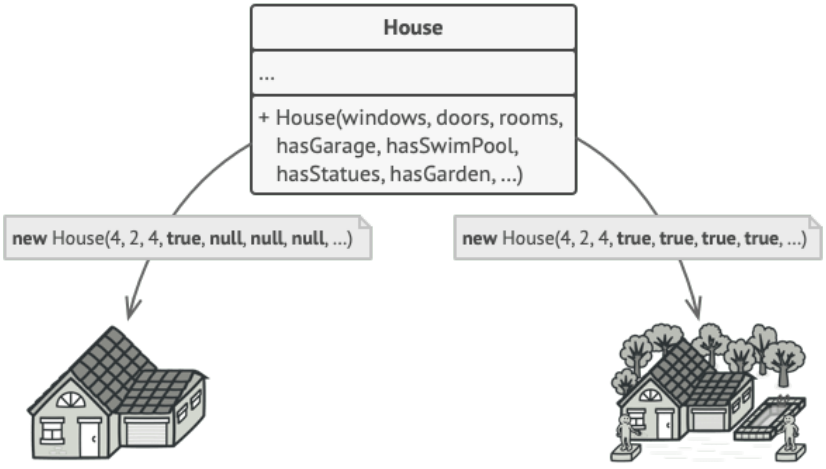
Builder



Builder


House
HouseWith Garage
HouseWith SwimmingPool
HouseWith FancyStatues
HouseWith Garden

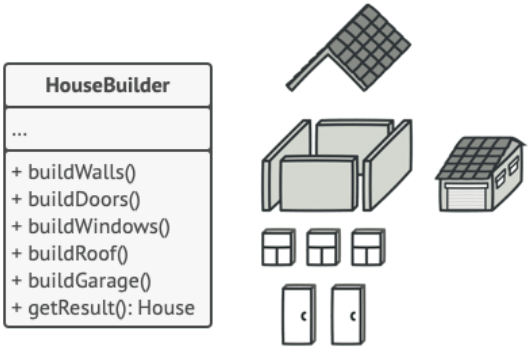
Builder


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

Builder

```
import plotly.express as px
df = px.data.iris()
df["e"] = df["sepal_width"]/100
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species",
                 error_x="e", error_y="e")
fig.show()
```

Builder


HouseBuilder
...
+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House

Builder

```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

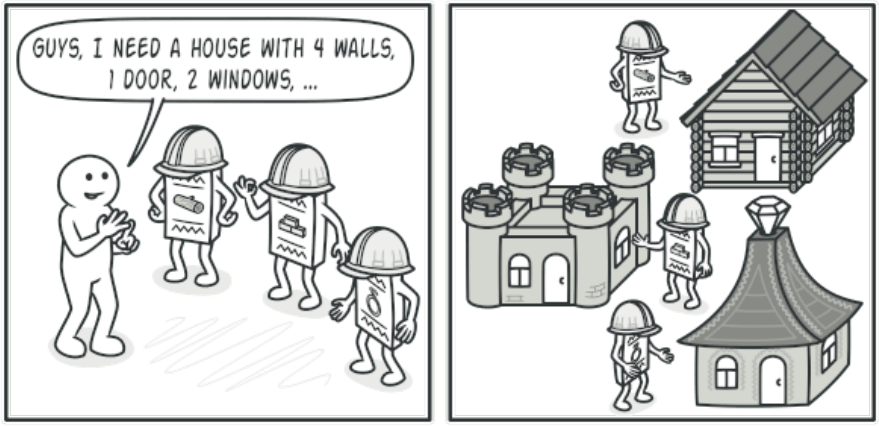
fruits = ['apple', 'blueberry', 'cherry', 'orange']
counts = [40, 100, 30, 55]
bar_labels = ['red', 'blue', '_red', 'orange']
bar_colors = ['tab:red', 'tab:blue', 'tab:red', 'tab:orange']

ax.bar(fruits, counts, label=bar_labels, color=bar_colors)

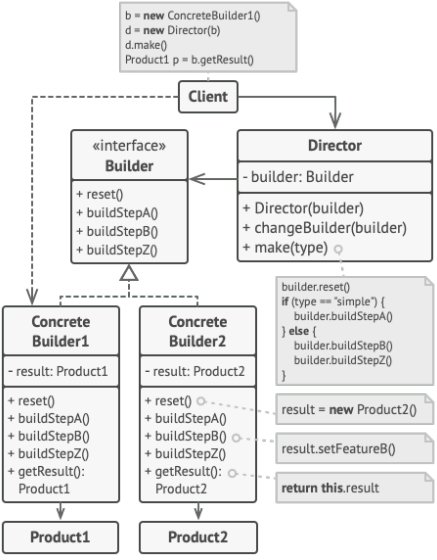
ax.set_ylabel('fruit supply')
ax.set_title('Fruit supply by kind and color')
ax.legend(title='Fruit color')

plt.show()
```

Builder



Builder


b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
Client
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
Concrete
Builder1
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
Concrete
Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
result = new Product2()
result.setFeatureB()
return this.result
Product1
Product2

Resumen

The Open Closed Principle (OCP)

Mensaje: Debemos encapsular las clases que crean objetos y separar su lógica de las clases que usan los objetos. Esto permite que OCP funcione en la práctica.

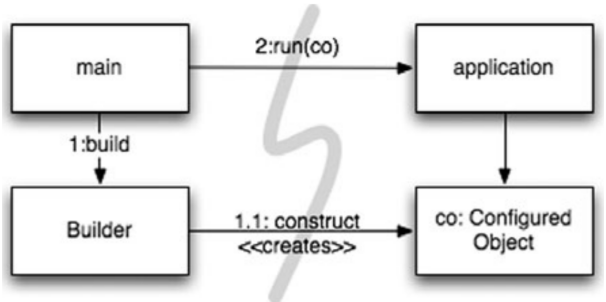


Figure 11-1

Separating construction in `main()`

The Open Closed Principle (OCP)

Mensaje: Debemos encapsular las clases que crean objetos y separar su lógica de las clases que usan los objetos. Esto permite que OCP funcione en la práctica.

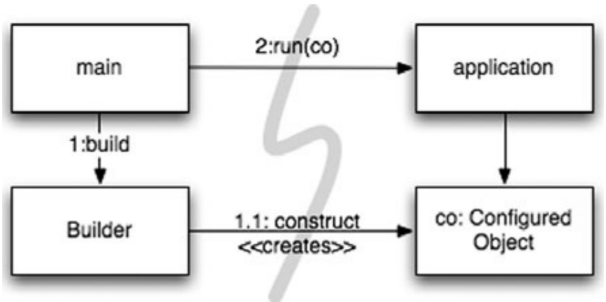


Figure 11-1

Separating construction in `main()`

Los patrones creacionales nos ayudan a separar/encapsular la lógica para crear objetos.

Actividad

Actividad

Examen: Es una pregunta sobre patrones con tres partes:

- Usa un patrón ya implementado por nosotros.
- Extiende el patrón implementado por nosotros correctamente.
- Agrega un patrón sobre nuestro patrón.

Actividad: Resuelve parte a) y b) del examen 2023-2.