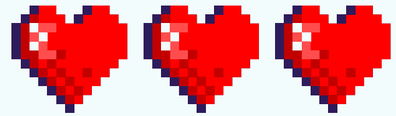
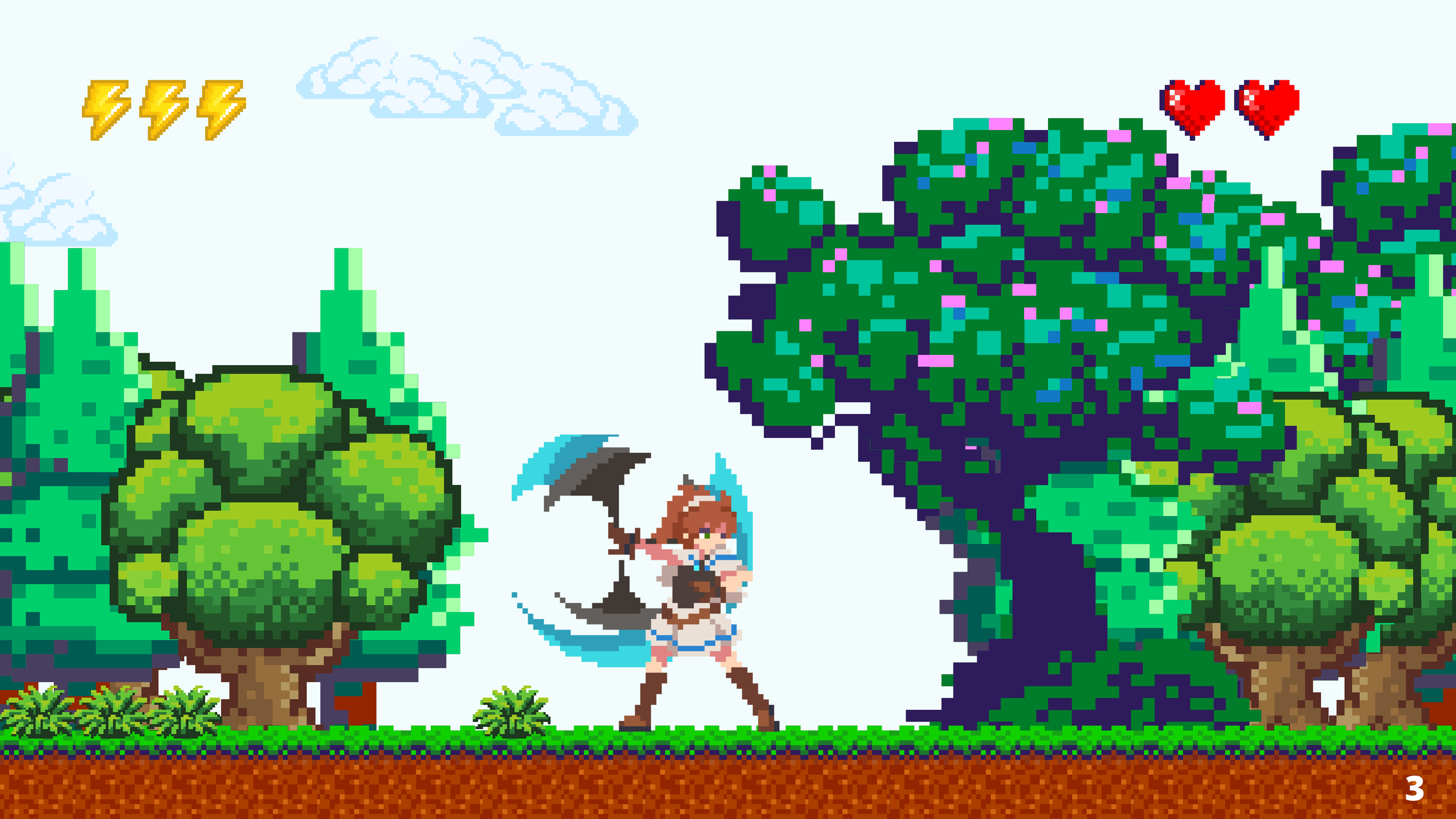


FLYWEIGHT

Luis Diego Barrantes B70994
Francisco Mora Díaz C05118

START

ÍNDICE



FLYWEIGHT PATTERN

Patrón de diseño estructural.

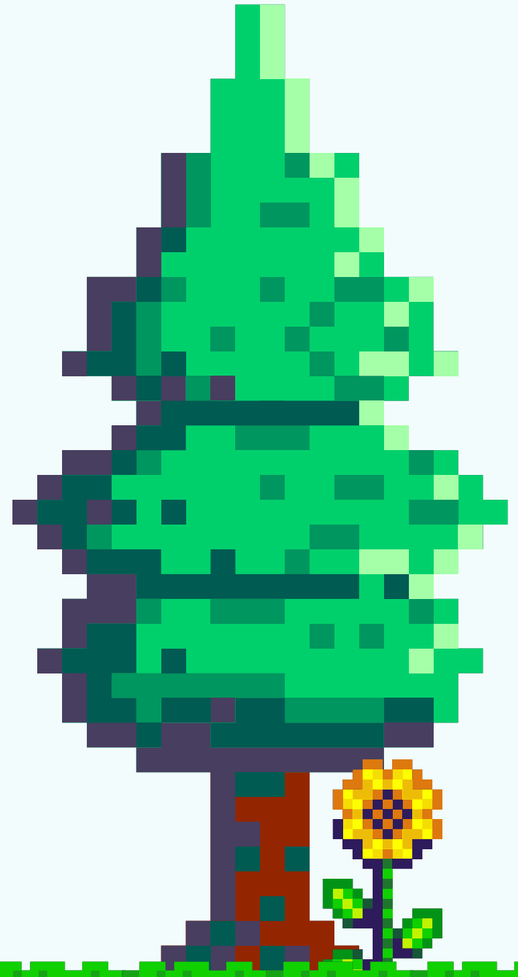
Procura un mejor aprovechamiento de la memoria.

Comparte partes comunes entre varios objetos por medio de Flyweights.

ESTADOS

INTRÍNSICOS

Elementos comunes que se pueden compartir.



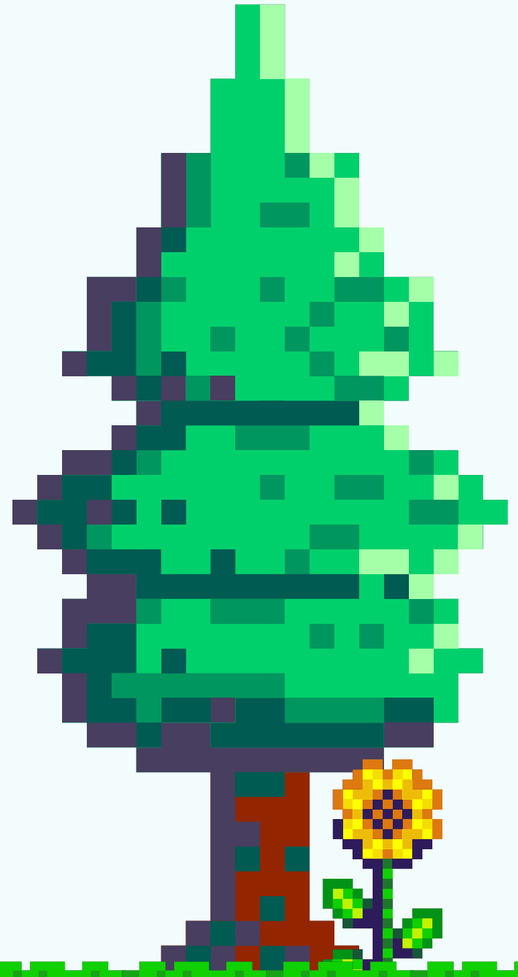
EXTRÍNSICOS

Elementos particulares.

ESTADOS

INTRÍNSICOS

Nombre
Textura
Color



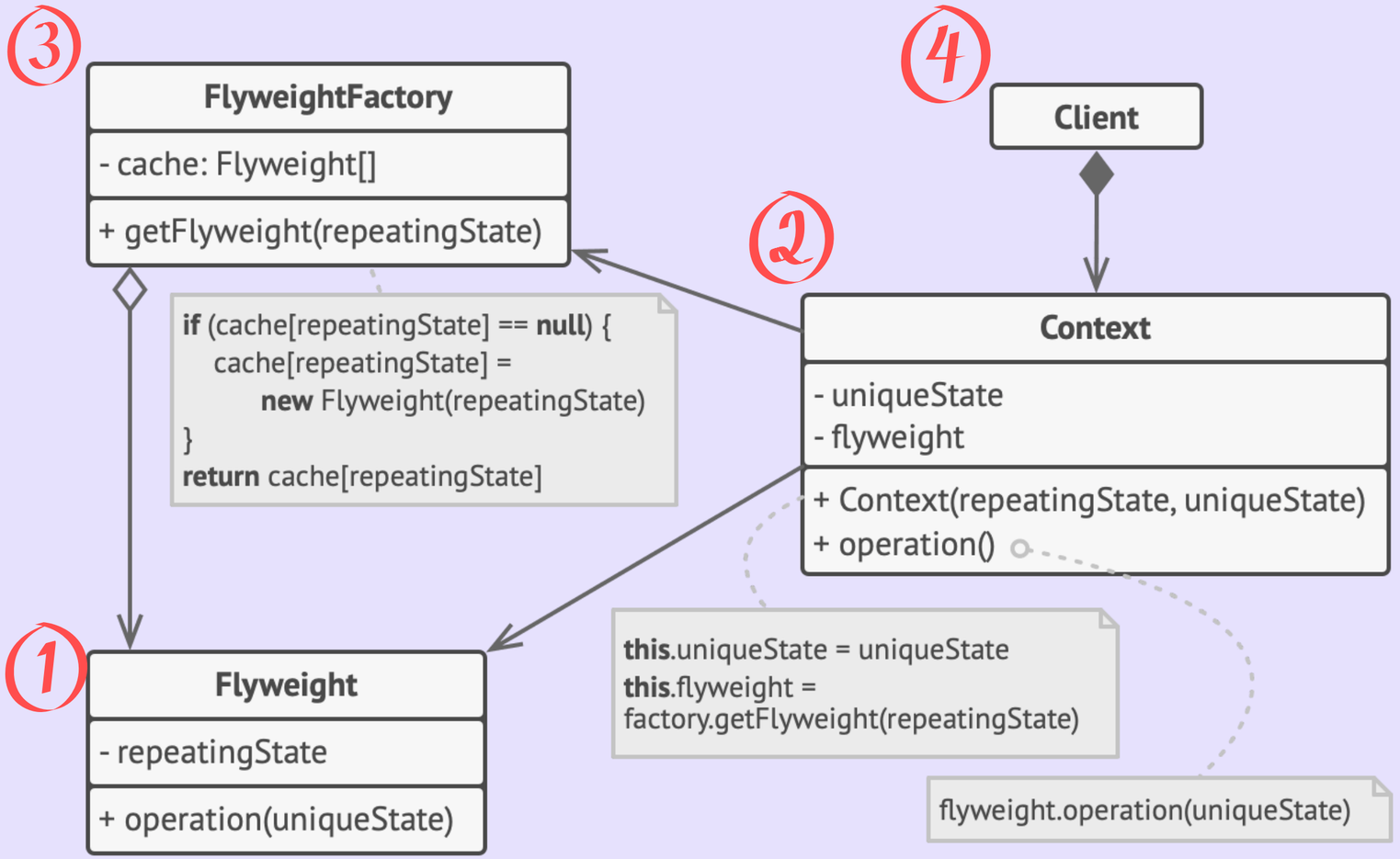
EXTRÍNSICOS

Ubicación
Tamaño

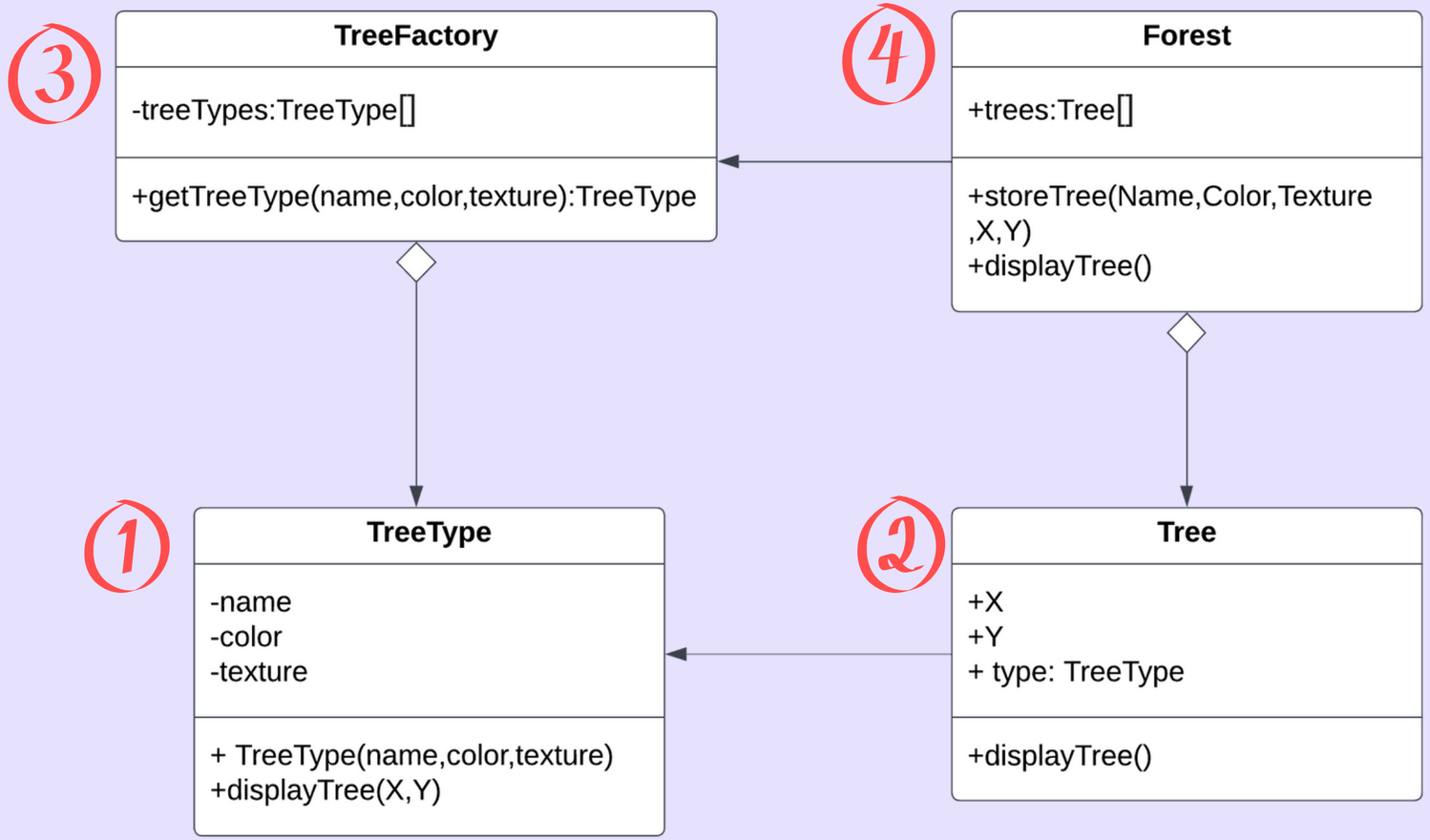
FLYWEIGHT

Objeto que contiene el estado intrínsico.

Es independiente del contexto.



1. **Flyweight:** Contiene el estado intrínsico.
2. **Context:** Contiene el estado extrínsico y el objeto flyweight.
3. **Flyweight Factory:** Administra reserva de flyweights. Únicamente crea un flyweight nuevo si en la reserva no existe uno con el estado intrínsico solicitado.
4. **Client:** Calcula o almacena estados extrínsicos.



1. **Flyweight:** Contiene el estado intrínsico.
2. **Context:** Contiene el estado extrínsico y el objeto flyweight.
3. **Flyweight Factory:** Administra reserva de flyweights. Únicamente crea un flyweight nuevo si en la reserva no existe uno con el estado intrínsico solicitado.
4. **Client:** Calcula o almacena estados extrínsicos.

① Flyweight

```
public class TreeType {

    private final String name;
    private final String color;
    private final String texture;

    public void displayTree(String name,double X, double Y) {
        System.out.println(name + " displayed at coordinates: X:" + X +" Y:"+Y);
    }
}
```

② Context

```
public class Tree {

    private final TreeType type;
    private final double X;
    private final double Y;

    public void displayTree() {
        type.displayTree(type.getName(), X, Y);
    }
}
```

③ Flyweight Factory

```
public class TreeFactory {

    private static final Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, String color, String texture) {
        if (treeTypes.get(name) == null) {
            treeTypes.put(name, new TreeType(name, color, texture));
        }
        return treeTypes.get(name);
    }
}
```

Client

```
public class Forest {

    private final List<Tree> trees = new ArrayList<>();

    public void storeTree(double X, double Y, String name, String color, String texture) {
        TreeType treeType = TreeFactory.getTreeType(name, color, texture);
        trees.add(new Tree(X, Y, treeType));
    }

    public void displayTrees() {
        trees.forEach((Tree) -> Tree.displayTree());
    }

}
```

PASOS

1. COMPROBAR EL RENDIMIENTO

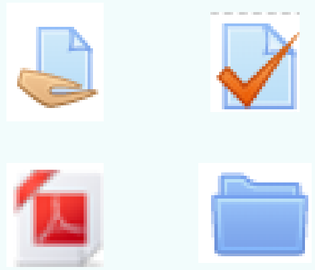
2. SEPARAR EL OBJETO EN SUS ESTADOS

3. CREAR FLYWEIGHT FACTORY

4. LIGAR EL CLIENTE CON LA FABRICA

FLYWEIGHT EN LA UCR

CLIENTE

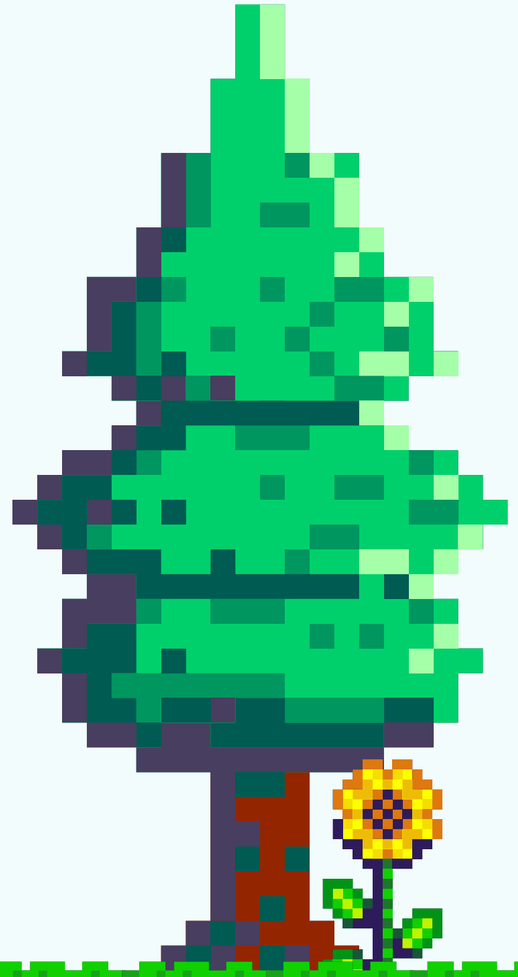


CURSO

CONSECUENCIAS

VENTAJAS

- Ahorro de memoria
- Mejoras en rendimiento
- Menos objetos



DESVENTAJAS

- Código complicado
- Intercambio de RAM por CPU

PATRONES RELACIONADOS

FACTORY METHOD

COMPOSITE

FACADE

REFERENCIAS

Refactoring.Guru. (2023). Flyweight. Refactoring.Guru. https://refactoring.guru/design-patterns/flyweight

Geekific. (2022, January 22). The Flyweight Pattern Explained and Implemented in Java | Structural Design Patterns | Geekific [Video]. YouTube. https://www.youtube.com/watch?v=qscOsQV-K14

GRACIAS

GAME OVER