Taller 5 – Patrones de diseño. Patrón seleccionado: “Flyweight”

Hecho por: Felipe Rueda, Mariana Ruíz y Gabriel Villabon

1. **Punto 1- Información general del proyecto:**
   El proyecto que seleccionamos para este análisis es “Hephaest/Minesweeper”, fue elaborado por la “Miao Cai”. Este proyecto de código abierto consiste en una recreación del famoso juego “Busca Minas” para PC de Windows. En la siguiente imagen se puede ver un ejemplo del proyecto.

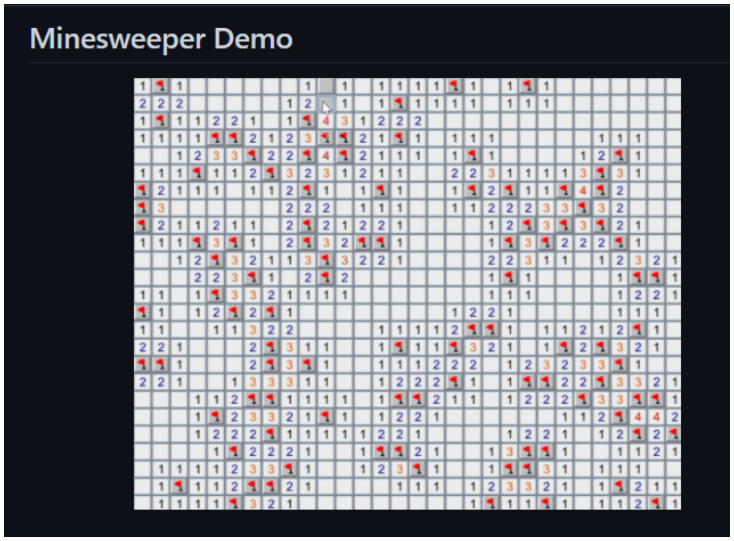


Figura 1. Imágenes del proyecto. Tomado de: Hephaest/Minesweeper: A minesweeper game created by Java. (Java 实现经典扫雷游戏) (github.com)

El link del proyecto es: Hephaest/Minesweeper: A minesweeper game created by Java. (Java 实现经典扫雷游戏) (github.com)

2. **Punto 2 - Información y estructura del fragmento del proyecto donde aparece el patrón.**

   La interfaz del juego se compone de 6 clases como se puede apreciar en la figura 2:

Figura 2. Imágenes de las clases de la interfaz. Tomado de: Hephaest/Minesweeper: A minesweeper game created by Java. (Java 实现经典扫雷游戏) (github.com)

Menu: Este proyecto funciona principalmente por su interfaz, se compone de un Ventana principal llamada “Menu.java” donde el jugador puede seleccionar la dificultad del juego e iniciar el mismo.
GameBoard: Después está la clase “GameBoard.java” donde se actualiza el panel de la ventana para comenzar el juego, en esta aparecen muchos widgets con varias imágenes compartidas y sus propios “action listeners”.
GameSquare: Es una clase abstracta que se encarga de mostrar lo que se verá en los cuadros de cada casilla y solo proporciona métodos básicos.
SmartSquart: Muestra la información dentro de las casillas de los cuadrados como el número de bombas alrededor y, la bandera roja o el signo dependiendo del tipo de click que haga el usuario. Además, la clase “SmartSquart” extiende de la clase “GameSquare”.
Checksquard: Comprueba el resto de las plazas sin excavar y muestra el número de bombas que rodean a la seleccionada. Si el número es 0, el programa comprobará el número de bombas rodeadas en los alrededores. La condición terminada es que el objetivo haya sido atravesado o que la posición del objetivo está fuera del límite.

Diagrama de interfaz:

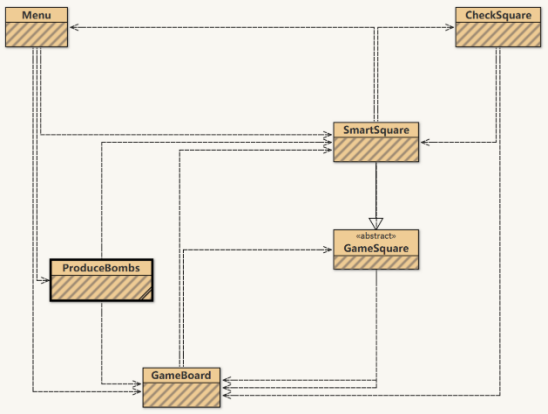


Figura 3. Imágenes del diagrama de la interfaz. Tomado de: Hephaest/Minesweeper: A minesweeper game created by Java. (Java 实现经典扫雷游戏) (github.com)

3. Punto 3 - Información general sobre el patrón de “Flyweight”
Un Flyweight sucede cuando un objeto o entidad que es compartido se usa en múltiples contextos simultáneamente y de forma independiente a cada situación.
Los Flyweights no pueden hacer suposiciones sobre el contexto en el que operan.
Este patrón tiene un único propósito: minimizar el consumo de memoria. Si tu programa no tiene problemas de escasez de RAM, no es necesario usar este patrón.
Ventajas:

- Más eficiencia
- Menor consumo de memoria
- Más recursividad
- Reducción de objetos innecesarios
- Menos complejidad

Desventajas de no usarlo:

- Posibilidad de tener objetos duplicados
- Demasiado gasto de recursos
- Consumo de memoria excesivo
- Más complejidad

Ejemplo de este patrón:

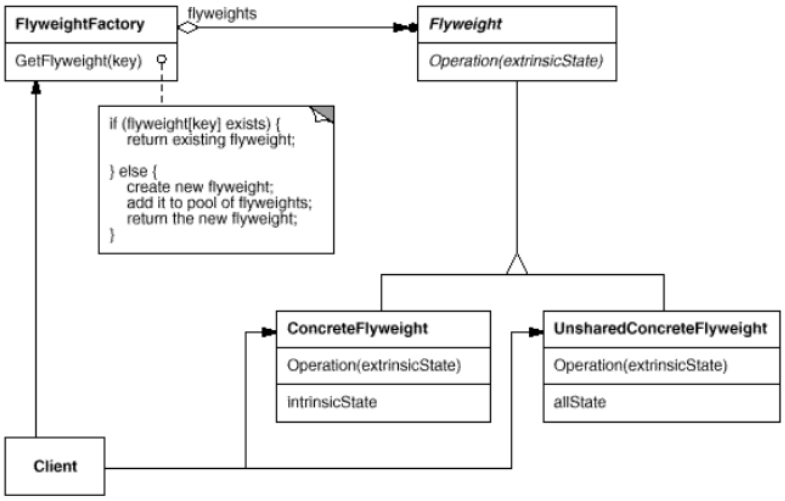
Flyweights modelan conceptos o entidades que normalmente son abundantes para representarse con objetos.

En este ejemplo se hablará sobre la representación de muchos árboles dentro de un videojuego. Cada árbol será representado por su propio objeto, que tiene cierto estado (coordenadas, textura, etcétera). Aunque el programa hace su trabajo principal, naturalmente consume mucha memoria RAM.

Crear un objeto individual por cada árbol con sus propias características consumiría demasiados recursos. Por esa misma razón es bueno usar el patrón Flyweight. Ahora, en lugar de almacenar la misma información en miles de objetos Tree (Árbol), vamos a referenciar uno de los objetos flyweight con un grupo particular de valores.

El usuario no se dará cuenta de nada, ya que la complejidad de reutilizar objetos flyweight queda enterrada dentro de una fábrica flyweight ahorrando recursos.

Estructura:



Aplicabilidad:

- Un programa que usa una gran cantidad de objetos repetidos
- Cuando se intenta reducir el costo de memoria
- Cuando una gran cantidad de objetos pueden ser reemplazados entre sí con una relativa facilidad

- Aplicaciones que no dependan de la identidad de un objeto

4. Punto 4 - Información del patrón aplicado al proyecto.

Este patrón se ve principalmente en la clase “SmartSquare” ya que ahí es dónde se concentra la mayoría de las imágenes que se utilizan en la aplicación. Cada vez que se va a actualizar una casilla de juego cargando una imagen, si cumplen el mismo caso, todas son referenciadas por una misma imagen para actualizar, en vez de que cada casilla tenga su propia imagen independiente. Es evidente el uso de ese patrón en todo momento que se carga una imagen en el programa ya que reutiliza los mismos recursos para no tener imágenes o recursos duplicados. En la siguiente imagen vemos como los listeners cumplen una misma acción independientemente con qué casilla estén tratando dentro del juego.

```
if(thisSquareHasBomb)
{
        /*
         * If this square contains a bomb, show the bomb image.
         * Then display the selection window.
         */
        setImage(SmartSquare.class.getResource("/bombReveal.png"));
        long costTime = System.currentTimeMillis() - ((SmartSquare) board.getSquareAt(0, 0)).
        cq.showBomb(xLocation, yLocation);
        window("You used " + TimeChecker.calculateTime(costTime) +". Do you want to try again
                        new ImageIcon(SmartSquare.class.getResource("/failFace.png")));
} else{
        thisSquareHasTraversed = false;
        /*
         * If this square doesn't contain a bomb, calculate its surrounding bombs.
         * If this square has zero bombs in its surrounding squares,
         * expanding into empty space until the surrounding of the space has at least one bom
         * or the space touches the window's boundary.
         */
        cq.countBomb(xLocation, yLocation);

        if (cq.isSuccess()) {
                long costTime = System.currentTimeMillis() - ((SmartSquare) board.getSquareAt
                cq.showBomb(xLocation, yLocation);
                window("You win this game in " + TimeChecker.calculateTime(costTime) +
        "! Do you want to try again?","Congratulations",
                                new ImageIcon(SmartSquare.class.getResource("/passFace.jpg"))
        }
}
```

Figura 4. Imagen reutilización de recursos. Tomado de: Hephaest/Minesweeper: A minesweeper game created by Java. (Java 实现经典扫雷游戏) (github.com)

5. Punto 5 - ¿Por qué tiene sentido usar ese patrón en ese punto del proyecto? ¿Qué ventajas tiene?

Al usar este patrón se ahorra bastantes recursos de memoria. En este juego no es necesario que la imagen de cada casilla sea diferente si comparten el mismo estado, por lo que, si tenemos un tablero de 10x10 casillas, no tiene mucho sentido que referenciamos 100 imágenes diferentes para todas las casillas que hay si cumplen el mismo caso, se gasta más memoria de la que se necesita. Por esta razón, el autor de este proyecto decidió usar el patrón de “Flyweight” referenciando una solo imagen para cada situación, independiente de las casillas dentro del tablero de juego. De esta

forma, se consigue un programa mucho más eficiente sin necesidad de tener imágenes duplicadas.

6. **Punto 6 - ¿Qué desventajas tiene usar ese patrón, en ese punto del proyecto?**

   Por culpa de este patrón, sería bastante complicado sobrescribir el código en dado momento donde el autor quiera cambiar la aplicación para que ciertas casillas aisladas dentro del tablero tengan un comportamiento diferente a las demás. Tendría que modificar gran parte del código y abandonar este patrón ya que "Flyweight" haría todo lo contrario, tiene un solo método que es el encargado de proporcionar el mismo comportamiento para varias entidades, las casillas en este caso.

7. **Punto 7 - ¿De qué otras formas se pudieron haber solucionado, en este caso los problemas que resuelve el patrón?**

Usando los patrones prototype o observer, prototype es un patrón de tipo creación en el cual se copia un objeto sin la necesidad que el código dependa de las clases, debido a que el buscaminas es un juego en el cual se tiene un campo hecho a base de objetos es posible copiar un mismo objeto tanta veces sea necesario para poder tener el tablero completo. Observer es un patrón de tipo comportamental en el cual se informa de una acción realizada en una parte del código y a partir de esta acción tomar una decisión, esto nos permite que al tocar una parte del campo minado la acción a tomar es el cambio de imagen a una donde se vea un número, la mina, una bandera o excavar una mina. Con estos patrones podemos simular el patrón realizado y solucionar el problema del gasto de memoria ram excesiva.

Referencias:

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993, July). Design patterns: Abstraction and reuse of object-oriented design. In the European Conference on Object-Oriented Programming (pp. 406-431). Springer, Berlin, Heidelberg. Andy en Stackoverflow. (2012, 14 agosto). How ListView's recycling mechanism works. https://stackoverflow.com/questions/11945563/how-listviews-recycling-mechanism-works

Cai, M. (2019, 8 julio). *Hephaest/Minesweeper*. Github.

https://github.com/Hephaest/Minesweeper