Capturas de requisitos antes las nuevas variantes del juego:

Colorised Game: Aquí identificamos que tendríamos que tener más tipos de células (una por cada color) y nuevas reglas. Dependiendo la variante del modo de juego Colorised que son las siguiente:

- Inmigration:
Tiene dos tipos células vivas, normalmente representados por celdas rojas y azules.
Y en las reglas, las células vivas conservan su color, de forma permanente, hasta que mueren debido a una sobrepoblación o una subpoblación (Como el modo de juego tradicional); Cuando nace una célula, adquiere el valor de color de la mayoría de sus tres vecinas.

- Quadlife:
Cuanta con cuatro tipo de celulas vivas, representadas por celdas de los colores: rojo, azul, verde y amarillo. Y en las reglas, cualquier configuración que consista en celdas de sólo dos colores se comporta de manera idéntica a Inmigración. Cuando se crean las células recién nacidas, adquieren el color mayoritario de sus vecinas. Sin embargo, existe otra posibilidad. Si los tres vecinos son de diferentes colores, la célula recién nacida adopta el color restante.

Generations Game: En esta versión del juego Game Of Life se implementaron dos variantes:

-Brains Brain:
En esta variante el principal cambio con respecto a las anteriores es que ahora las células poseen un nuevo estado intermedio (nuevo tipo de célula) entre estar viva y finalmente muerta, llamado “dying1”.

LivingCell → Dying1Cell → DeathCell

Con respecto a las reglas no encontramos un gran cambio con respecto a la versión tradicional del juego, se necesitan dos células vivas para nacer, ya no sobreviven, pasan directamente a un estado Dying1, para luego morir directamente: B2/S/C3. Este último C3 se refiere a la cantidad de estados que puede estar una célula, ahora 3.

-Stars Wars:
A diferencia de la variante anterior, ahora tendremos dos nuevos estados entre estar viva y muerta, de “dying1” a un estado “dying2” para luego si morir.

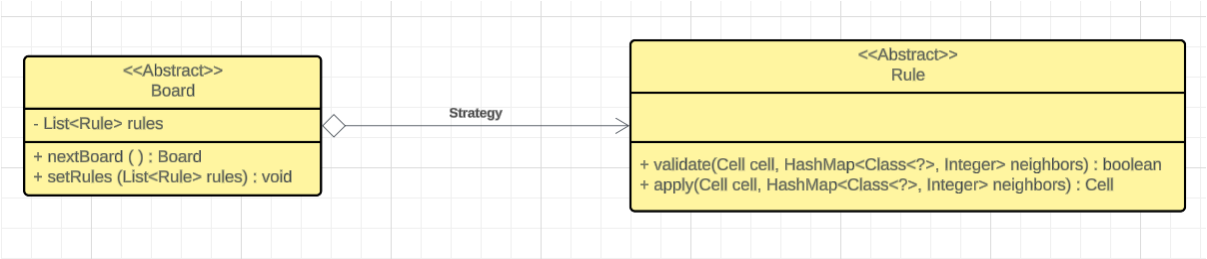
LivingCell → Dying1Cell → Dying2Cell → DeathCell

En cuanto a la reglas, a diferencia del Brains Brain ahora si podemos sobrevivir, se necesitan 2 células vivas para nacer, 3, 4 o 5 células vivas para sobrevivir, si no se pasa a un estado “Dying1”, para luego directamente pasar a “Dying2” y finalmente morir: B2/S345/C4.

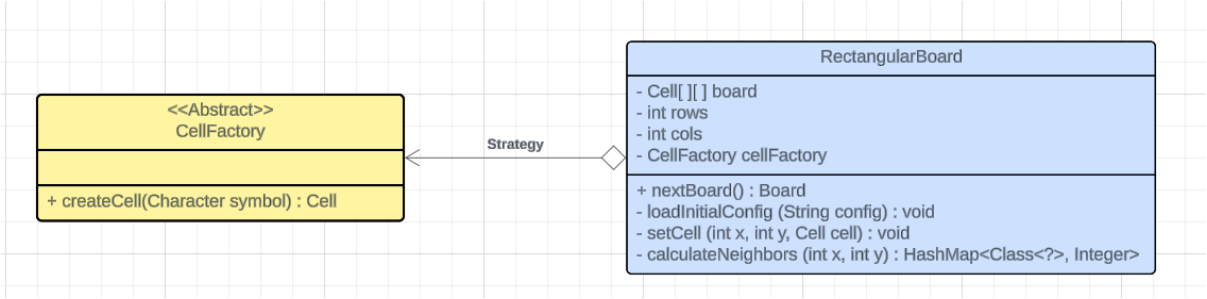
Patrones de diseño utilizados:

Para ambas versiones del juego se aplicaron los mismo patrones de diseño para lograr sus implementaciones, pues el diseño logrado permite manejar varias variantes del juego sin la necesidad de, por ejemplo, crear por separado y todo de cero, para cada nueva versión que se desee implementar.

En primer lugar identificamos un patrón Strategy en la clase abstracta Board con la clase abstracta Rule.



Asimismo identificamos el mismo patrón en la clase concreta Rectangular Board con la clase abstracta Cell Factory.



Finalmente identificamos Abstract Factorys Method:
Una necesaria para manejar la creación del tablero/células/reglas según el tipo/variante de juego que se desea jugar: la llamada Abstract Factory que crea las factorías de cada variante de juego.

En src/main/java/abstractfacotry/AbstracFacory.java

```
public interface AbstracFactory {

   CellFactory getCellFactory();
   List<Rule> createRules();

}
```

Así estas se encargan de crear las reglas según corresponda. La única otra creación que varía en cada modo de juego, es la creación de las células para el tablero, por esto también se cuenta con un método que nos devuelve la Cell Factory correspondiente, que luego le es pasada al método encargado de crear el tablero.

En src/main/java/abstractfacotry/TraditionalFactory.java

```java
public class TraditionalFactory implements AbstracFactory {

    public TraditionalFactory(){}

    @Override
    public CellFactory getCellFactory(){
        return new CellFactoryTraditionalGame();
    }

    @Override
    public List<Rule> createRules() {
        List<Rule> rules = new ArrayList<>();
        Rule rule1 = new BirthRule();
        Rule rule2 = new SurviveRule();
        Rule rule3 = new DeathRule();
        rules.add(rule1);
        rules.add(rule2);
        rules.add(rule3);
        return rules;
    }
}
```

En src/main/java/game/Game.java

```java
public void start() {
    Config config = new Config(nameConfigProperties);
    config.loadConfig();

    AbstracFactory factory = createFactory(config.gamemode);
    CellFactory cellFactory = factory.getCellFactory();
    Board board = new RectangularBoard(config.rows, config.cols, config.initialConfig,
cellFactory);
    List<Rule> rules = factory.createRules();
    board.setRules(rules);

    GameController gameController = createController(board, config.advanceMode);
    Output output = createOutput(gameController, config.output);
    gameController.start();
}
```

```java
private AbstracFactory createFactory (String gamemode) {
    switch (gamemode) {
      case "traditional":
        return new TraditionalFactory();
      case "traditionalHL":
        return new TraditionalHLFactory();
      case "colorisedIm":
        return new ColorisedImFactory();
      case "colorisedQL":
        return new ColorisedQLFactory();
      case "generationsBB":
        return  new GenerationsBBFactory();
      case "generationsSW":
        return  new GenerationsSWFactory();
    }
    return null;
}
```

He aquí la última abstract factory identificada, por cada variante de juego creamos una factoría de células correspondientes, así a la hora de levantar un tablero con un string inicial, podremos identificar y establecer qué célula crear.

En src/main/java/cell/CellFactory.java

```
public abstract class CellFactory {

 public abstract Cell createCell(Character symbol);

}
```

En src/main/java/cell/TraditionalGame/CellFactoryTraditionalGame.java

```
public class CellFactoryTraditionalGame extends CellFactory {

 public CellFactoryTraditionalGame() {}

 @Override
 public Cell createCell(Character symbol) {
   if (symbol.equals('■')) {
     return new LivingCell();
   }
   return new DeadCell();
 }

}
```

En src/main/java/board/RectangularBoard.java

```
private void loadInitialConfig (String config) {
 int count = 0;
 int x = 0;
 int y = 0;
 while (count < config.length()-1) {
   if (config.charAt(count) != '\n') {
     char character = config.charAt(count);
     if (character != ' ' && character != '\r') {
       board[x][y] = cellFactory.createCell(character);
       y++;
     }
   } else {
     y=0;
     x++;
   }
   count++;
 }
}
```

Asimismo se encontró, desde la primera versión del Game Of Life, la utilización del patrón Observer, para la notificación a los Output. Ahora dos para soportar salida por file, pero anteriormente para salida por terminal únicamente, encargados entonces de imprimir la board cada vez que se notifica que se actualizó a un nuevo estado (la siguiente generación).

En src/main/java/output/Observer.java

```
public interface Observer {

 public void update (Board board, Integer generation);

}
```

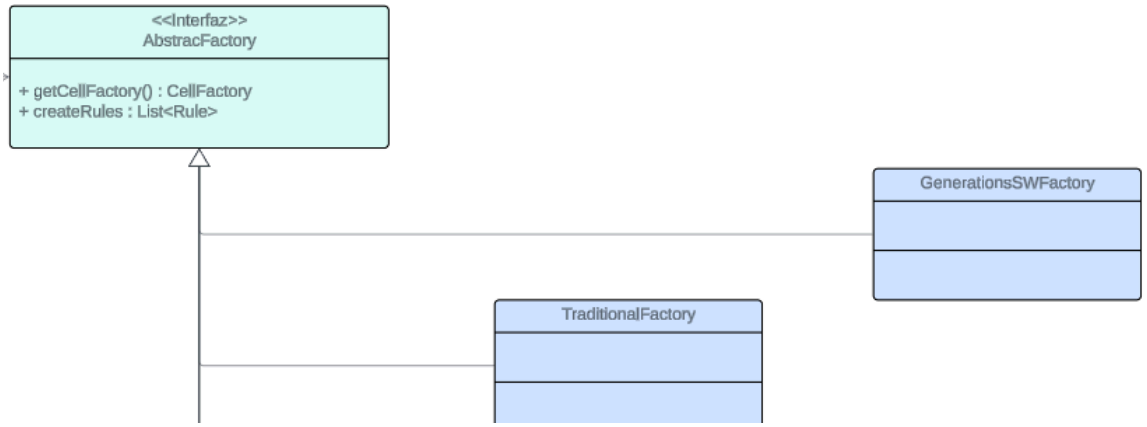
En src/main/java/output/TerminalOutput.java

```
public class TerminalOutput implements Output {
```

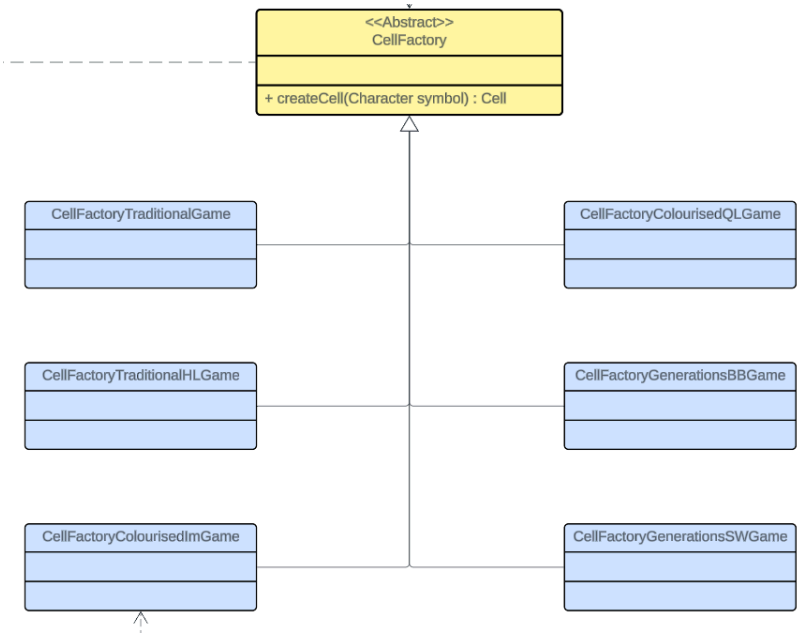
```
private GameController gameController;

public TerminalOutput (GameController gameController) {
  this.gameController = gameController;
  gameController.registerObserver(this);
}

@Override
public void update (Board board, Integer generation) {
  System.out.print("\033[H\033[2J");
  System.out.flush();
  String print = board.toString();
  System.out.println("---"+generation+"---");
  System.out.println(print);
}

}
```

Principios de diseños utilizados:

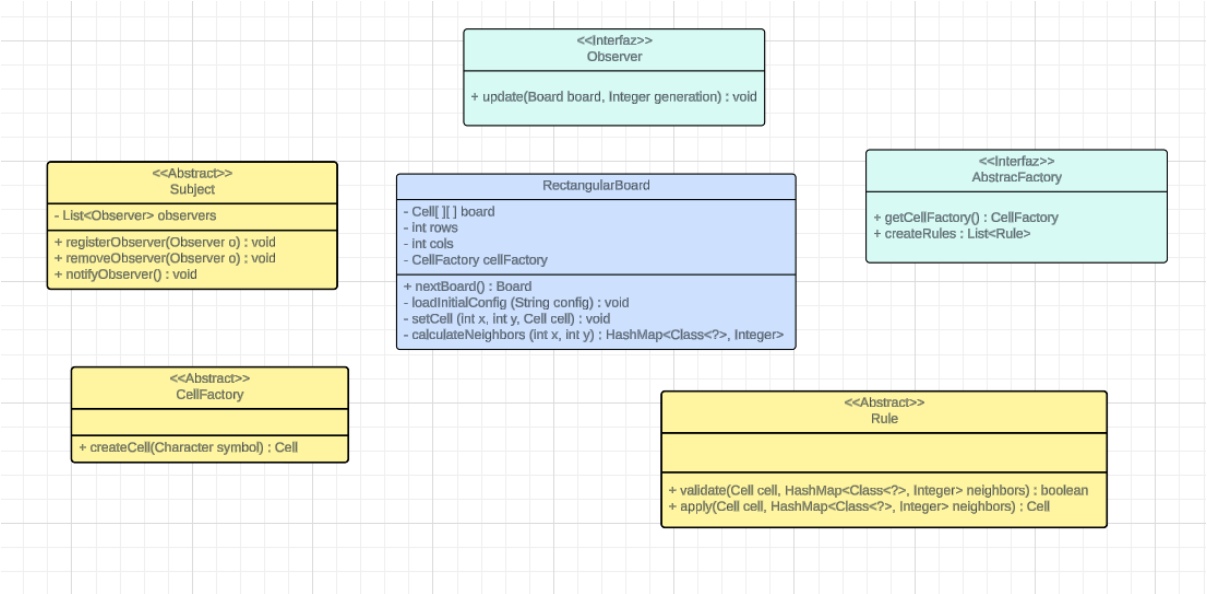
- Programar respecto de interfaces.



- Las clases están abiertas para extensión, pero cerradas para modificación.



- Cada clase tienen una única responsabilidad.



- Dependemos de abstracciones; no dependemos de clases concretas.