|  |
| --- |
|  |
| 최종 보고서 |
|  |
| 설계 패턴. 팀프로젝트 |

**Team MEMBER ( happycoder TEAM )**

**20140073이태균 ( 컴퓨터공학부 )**

**20164542김선재 ( 전자전기공학부 )**

2018 December 7

최종 보고서

설계 패턴. 팀프로젝트

**목차**

개요

팀 이름

역할

원본 프로그램 소개

설계

원본 프로그램과 한계

적용한 패턴

수정한 프로그램과 장단점

결론

테스트 케이스

프로그램 사진

맺음말

# 개요

**팀 이름**

팀 이름은 HAPPYCODER 이다.

**역할**

1. 이태균

* Factory 패턴 적용
* State 패턴 적용
* Command 패턴 적용
* 프레젠테이션

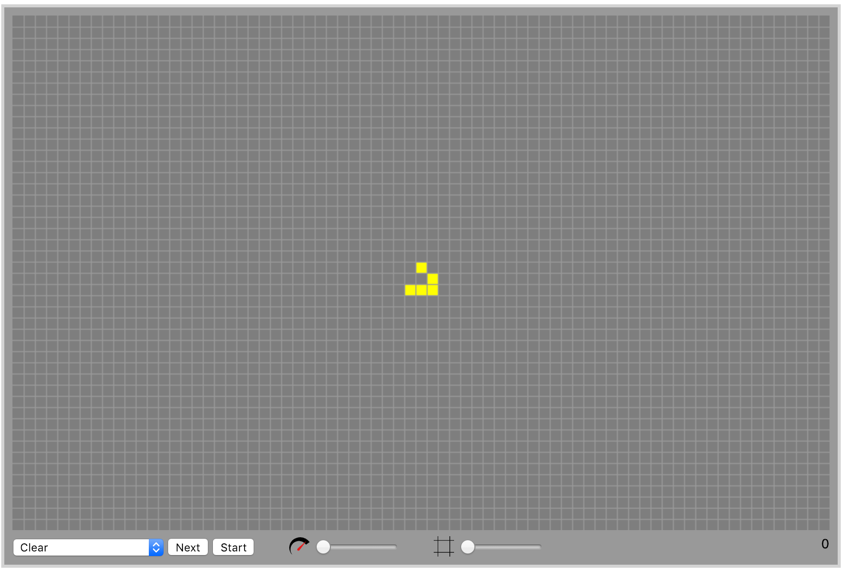
2. 김선재

* Singleton 패턴 적용
* Template Method 패턴 적용
* 레포트

**원본 프로그램 소개**

**라이프게임 (Game of Life)** : 영국의 수학자 존 호턴 코웨이가 고안해낸 세포 자동자의 일종이다. 무한히 많은 사각형(세포)으로 이루어진 격자 위에서, 각 세포들은 '죽음'과 '삶' 둘 중 하나의 상태를 가진다. 현재 세대의 세포들의 상태가 다음 세대의 세포의 상태를 결정한다.

* 죽은 세포의 이웃 중 세 개가 살아 있다면, 그 세포는 살아난다.
* 살아 있는 세포의 이웃 중 두 개 또는 세 개가 살아 있으면, 그 세포는 살아남는다. 그렇지 않다면 죽어버린다.



< Fig 1. 살아 있는 세포와 죽어있는 세포들,

현재의 상태가 다음 세대의 상태를 결정한다 >

Github 오픈소스[[1]](#endnote-1)의 첫 커밋이 프로젝트 주제다. 대부분의 소스코드가 각각 하나의 클래스로 하드코딩 되어있다. 변화할 가능성이 높은 코드를 선별하여 디자인 패턴을 적용했다. 다음 목차에서 그것들을 설명한다.

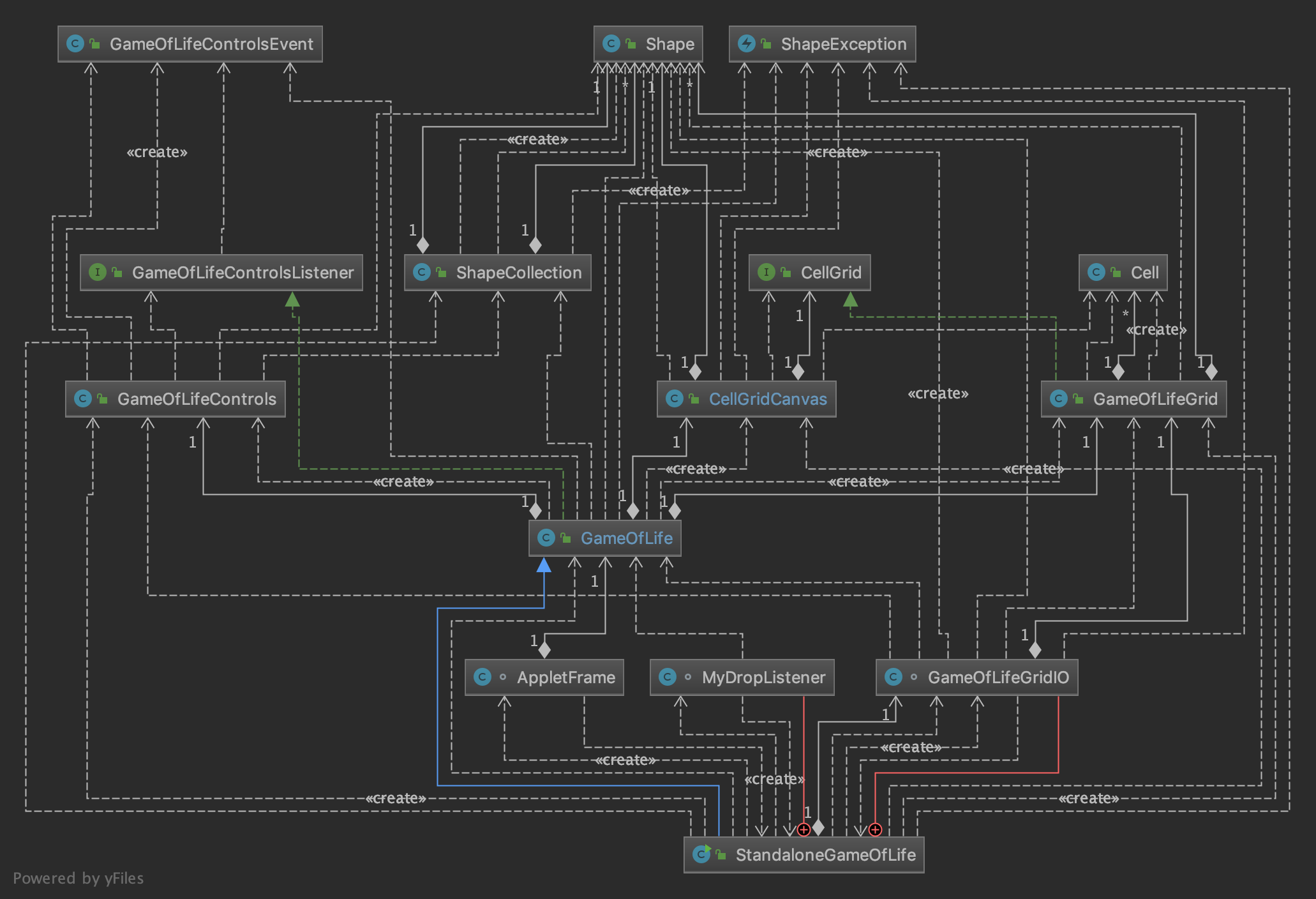
# 설계

**원본 프로그램과 한계**

  
< Fig 2. 원본 프로그램 UML >

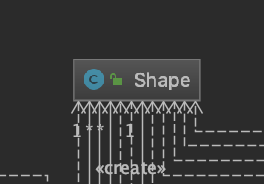
위 그림은 원본 프로그램의 UML 다이어그램이다. 주요 클래스는 아래와 같다.

1. GameOfLife : 프로그램 UI와 로직
2. GameOfLifeGrid : 라이프게임의 규칙 로직
3. ShapeCollection : 특정 모양(Shape)의 세포 생성



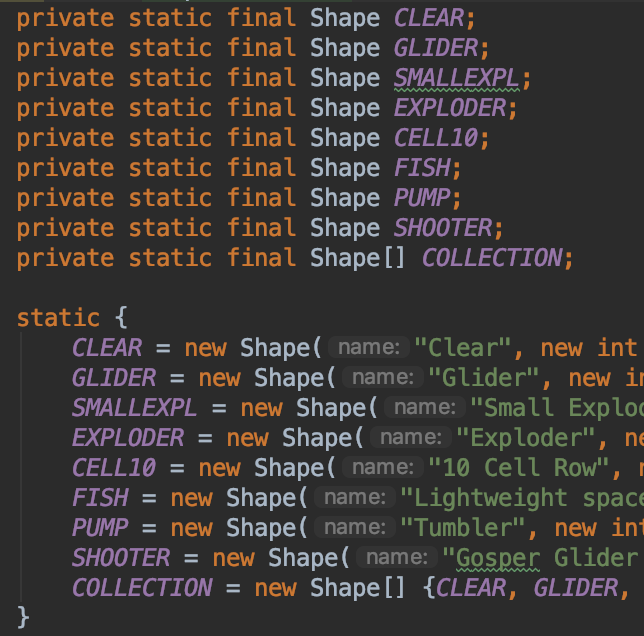
< Fig 3. 원본 프로그램 의존성 관계 >

위 그림은 원본 프로그램의 의존성 관계를 도식한 것이다. 다이어그램에서 보이는 의존성 문제와 개방-폐쇄 원칙(OCP)에 위반하는 한계점은 다음과 같다.



< Fig 4. Shape 클래스의 의존성 >

원본 프로그램에서 세포의 모양을 저장하는 Shape 클래스의 의존성이 강하다. 즉 Shape 클래스를 수정할 시, 변경해야 할 다른 클래스도 많다. 이러한 의존성을 줄여주는 디자인 패턴을 적용할 필요가 있다.



< Fig 5. 하드 코딩된 원본 소스 >

전체적으로 클래스 수가 적고, 한 개의 클래스에 기능들이 하드코딩돼있다. 그 기능들은 충분히 확장할 것으로 예상된다. 따라서 디자인 패턴을 적용하여 변화를 대비해야 한다.

**적용한 패턴**

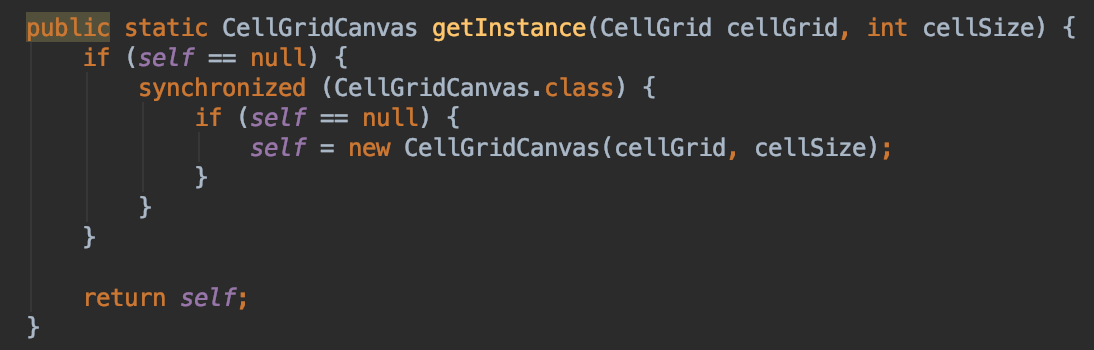
1. Singleton 패턴

CellGridCanvas 객체는 많은 사각형 격자를 만들고 살아있는 세포의 모양대로 화면에 그려준다. 그래픽 작업이 필요한 무거운 객체이므로 다시 생성되지 않도록 메모리를 효율적으로 관리해 줄 필요가 있다.

[ 변경 전 소스 ]

gameOfLifeCanvas = new CellGridCanvas(gameOfLifeGrid, cellSize);

[ 변경 후 소스 ]



gameOfLifeCanvas = CellGridCanvas.**getInstance**(gameOfLifeGrid, cellSize);

2. Template Method 패턴

GameOfLifeGrid 객체는 현재 세대 세포들의 상태로부터 다음 세대 세포들의 상태를 결정해준다. 특정 알고리즘에 따라 세포들의 생사를 결정한다. 한 메소드에 여러 로직이 하드코딩돼있다. 구현 코드를 분리시키고 변경에 유연해지기 위해 각 규칙을 추상화시킬 필요가 있다.

[ 변경 전 소스 - GameOfLifeGrid.java ]

public synchronized void next() {

// Reset Cells

...

// Add neighbours

....

// Bury the dead

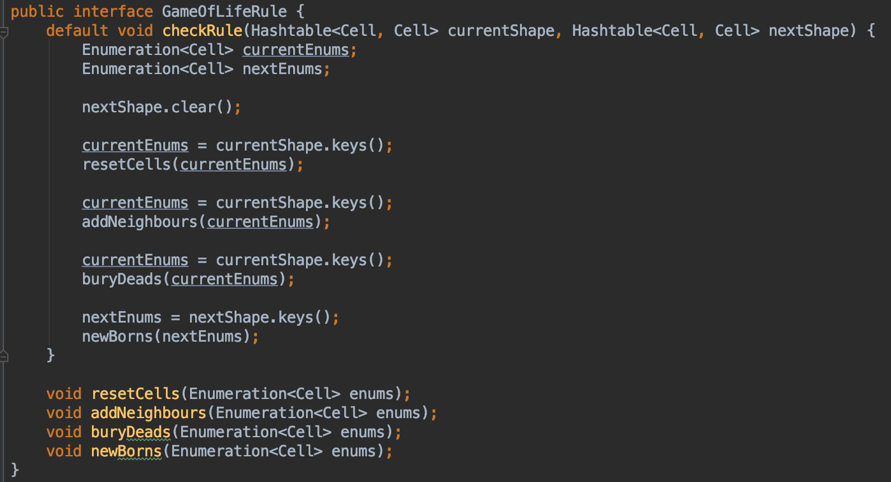
....

// Bring out new borns

....

}

[ 변경 후 소스 - GameOfLifeRule.java ]



인터페이스로 규칙을 추상화시켜 분리시켰다. 상세한 규칙 로직은, 이 인터페이스를 구현하는 클래스에게 책임을 위임시킨다.

[ 변경 후 소스 - GameOfLifeGrid.java implements GameOfLifeRule ]

public syncronized void **clear**() { ... }

public syncronized void **addNeighbours**() { ... }

public syncronized void **buryDeads**() { ... }

public syncronized void **newBorns**() { ... }

**checkRule**(currentShape, nextShape);

추상 클래스가 아닌 인터페이스로 템플릿 메소드 패턴을 구현한 이유는 다음과 같다. 추상 클래스로 상속받을 시, 이후에 다른 규칙 로직이 추가됐을 때 확장이 용이하지 않다. 다른 구현 로직 클래스를 상속받은 또 다른 GameOfLifeGrid.java 클래스를 구현해야 한다. 따라서 다른 로직 클래스의 상속이 아닌 인터페이스를 활용함으로써, 새 GameOfLifeGrid 클래스를 만들 필요를 제거했다.

3. Factory 패턴

앞서 설명했듯이, 원본 프로그램에는 Shape 클래스의 의존성이 컸다. 이 의존성을 줄여주기 위해 Shape 클래스를 추상 클래스로 수정했다. 그리고 하드코딩된 Shape 제공 로직을 Factory 패턴으로 관리한다.

[ 변경 전 소스 - Shape.java ]

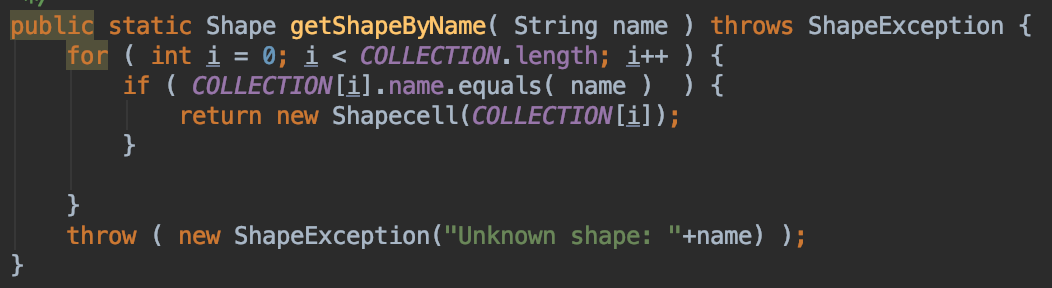
public class Shape { ... }

[ 변경 후 소스 - Shape.java ]

public abstract class Shape { ... }

[ 변경 후 소스 - ShapeCollection.java ]

public static class Shapecell extends Shape { ... }



[ 변경 후 소스 - GameOfLife.java ]

ShapeCollection.getShapeByName( "Glider" )

4. State 패턴

5. Command 패턴

**수정한 프로그램과 장단점**

Singleton 패턴

**결론**

테스트 케이스

프로그램 사진

맺음말

1. 주소 : https://github.com/edwinm/Game-of-Life-in-Java/tree/710e8018729e764f7390fd5f8ba0e35b18e21865 [↑](#endnote-ref-1)