|  |
| --- |
|  |
| 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

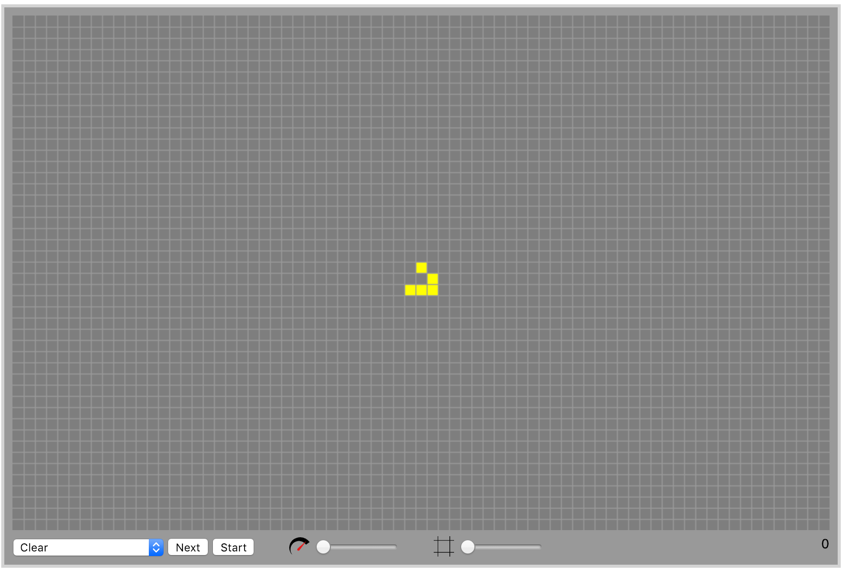
2. 김선재

* 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.

**ORIGINAL** : https://github.com/edwinm/Game-of-Life-in-Java/tree/710e8018729e764f7390fd5f8ba0e35b18e21865

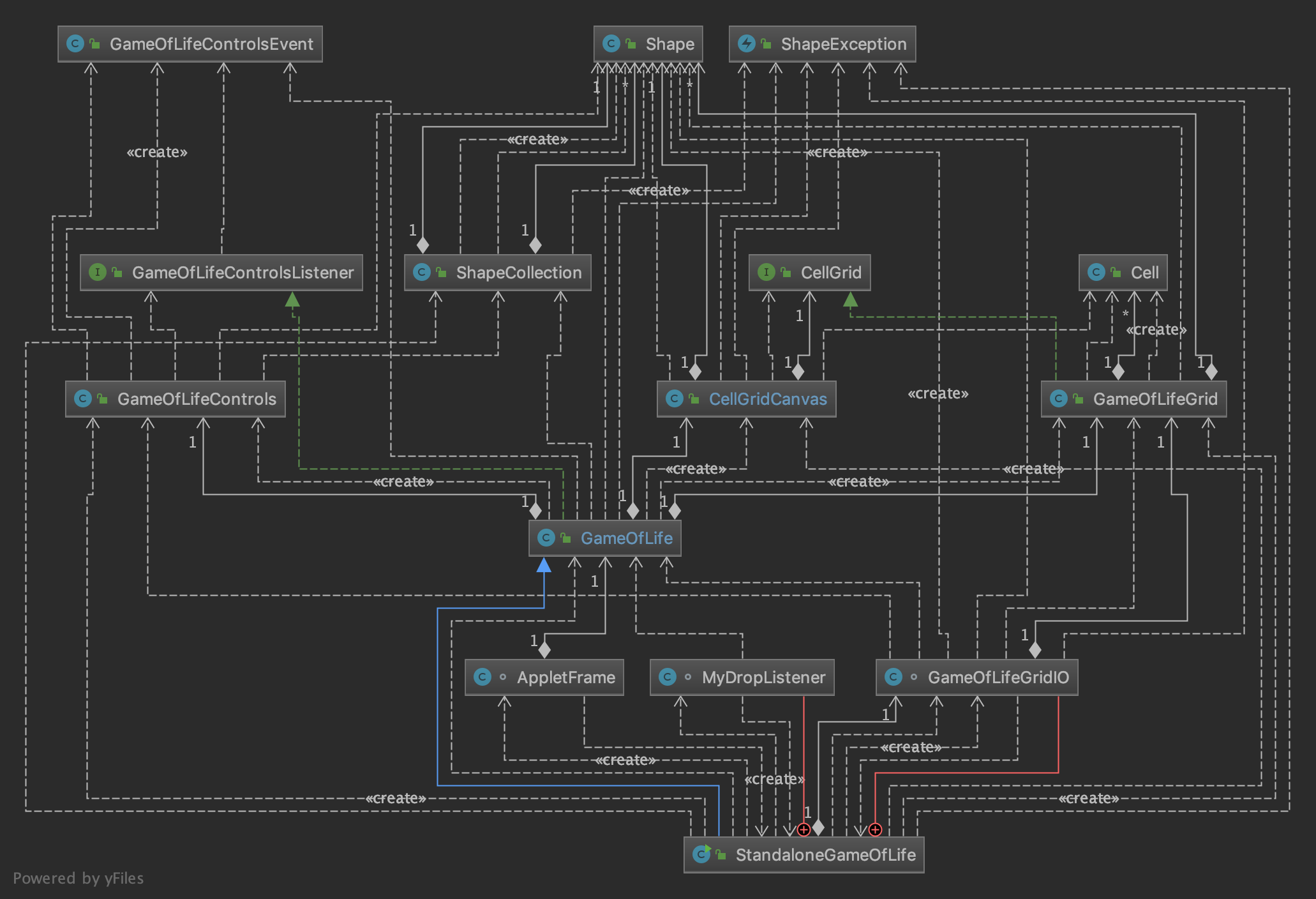
# 설계

**원본 프로그램과 한계**

  
< Fig 2. 원본 프로그램 UML >

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

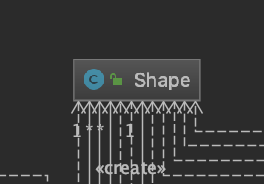
1. GameOfLife : 프로그램 UI와 로직
2. GameOfLifeGrid : 라이프게임의 규칙 로직
3. ShapeCollection : 특정 모양(Shape)의 세포 생성



< 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.

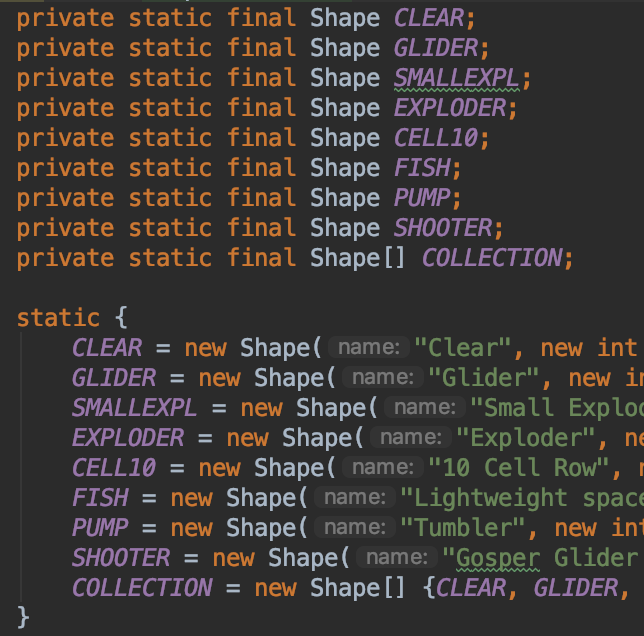
1. Shape class 의 의존성



< 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.

1. Hard coding



< 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**

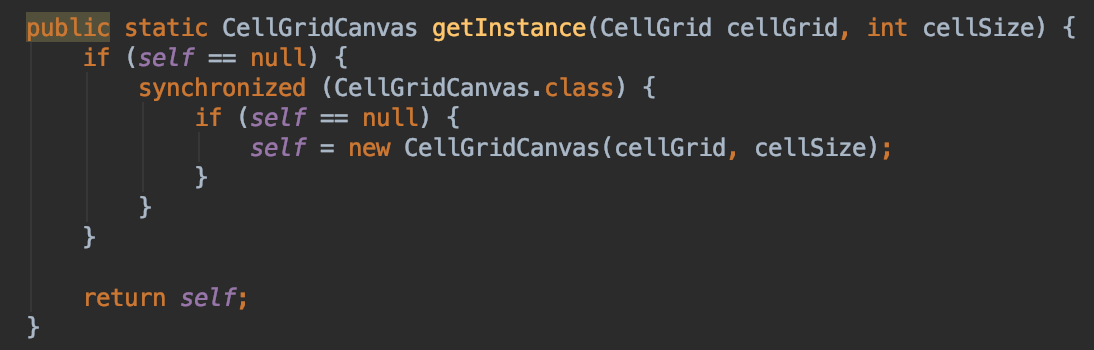
**1. Singleton 패턴**

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

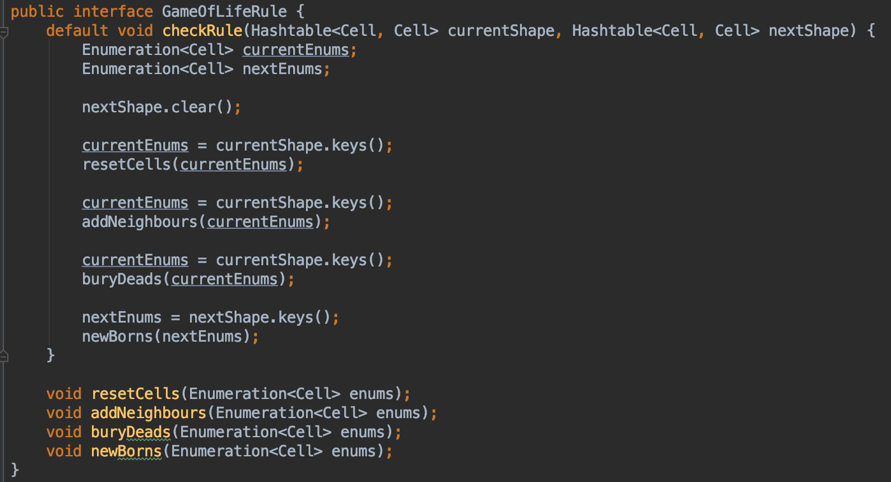
....

// 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 Method 패턴**

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

Shape 를 abstract class 로 만들고, 이를 Shapecell 이라는 concrete class 로 만들어 구현한다. 이는 shapeCollection 이라는 Factory 를 이용하여 생성되고, 생성되는 Method 는 static으로 선언한다(getShapeByName) .

또한, 원본 코드에서는 상대적으로 큰 shape object 를 전부 새로 생성하여 static 선언 뒤 사용하여 memory 사용량이 높았으나,, shape object 를 name, array 두개로 구성된 Shapeofyou object로 구성하여, static 을 사용하여 shared memory 의 이점을 누리고, 새로운 shape 를 적용할 시, interface 변화 없이 Shapeofyou object 추가로 쉽게 shape를 추가할 수 있게 된다. getShapeByName Method 에서는 new 를 통해 ShapeCell instance 를 생성하나,shape 가 바뀔시, reference 가 사라지도록 설계하여 GC 에 의해 memory reclaim 으로 정리된다.

ShapeCollection 등 여러 class name 은 코드 수정을 최소화 하고자 변경하지 않았다.

**[ 변경 전 소스 - Shape.java ]**

public class Shape { ... }

**[ 변경 전 소스 - ShapeCollection.java ]**

Public class Shapecollection{ ... } // shape 를 관리하는 class

public static Shape getShapeByName( String name )

// 이 method 에서 shape String 을 반환한다.

**[ 변경 후 소스 - Shape.java ]**

public abstract class Shape { ... }

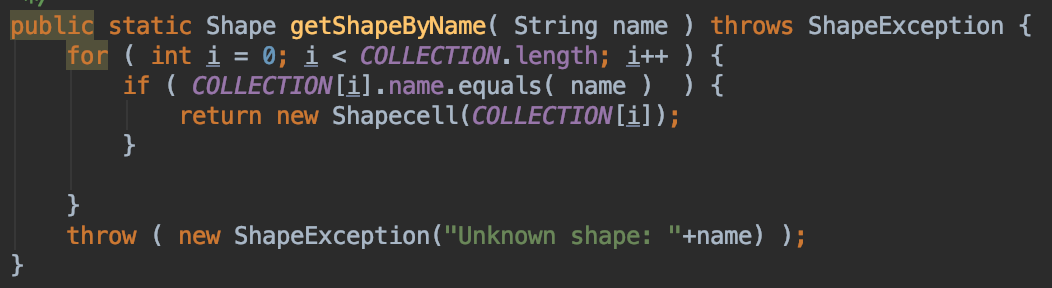
**[ 변경 후 소스 - ShapeCollection.java ]**

public static class Shapecell extends Shape { ... }

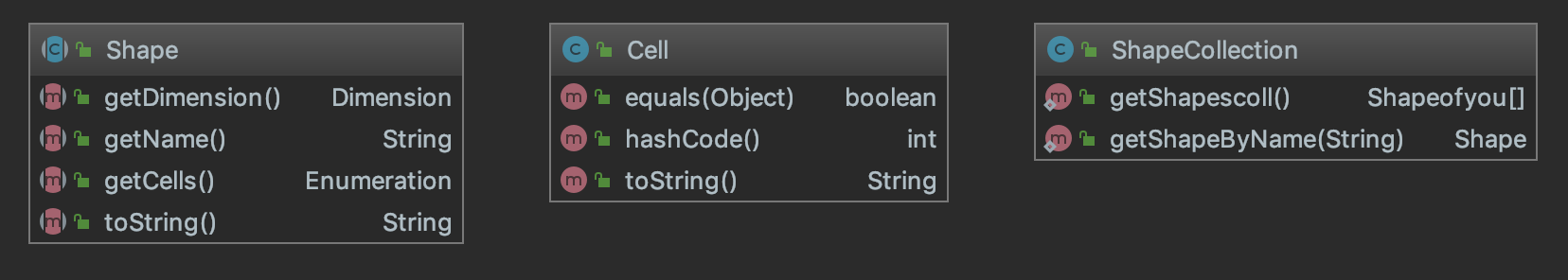
public static class Shapeofyou{…}

**[ 변경 후 소스 - GameOfLife.java ]**

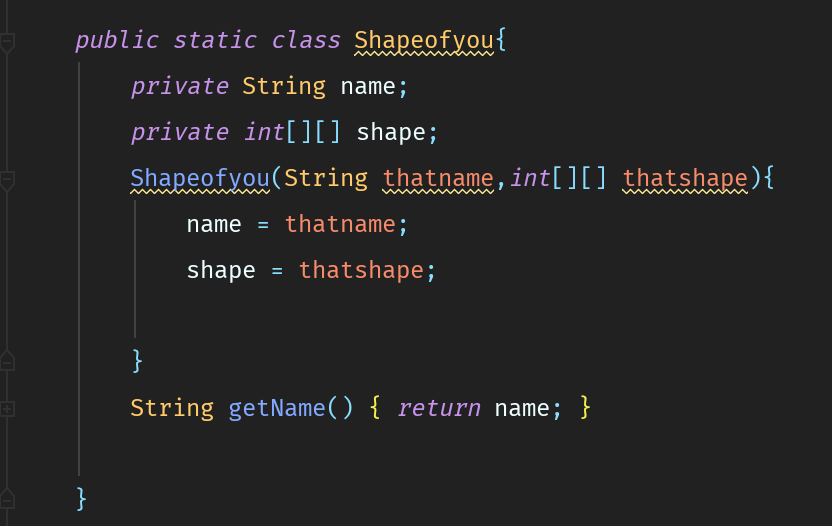
ShapeCollection.getShapeByName( "Glider" )



< Fig 6. ShapeCollection class의 getShapeByName >



< Fig 6. ShapeCollection class diagram >



< Fig 6. Shapeofyou nested class >

**4. State 패턴**

State pattern 은 다음과 같은 상황에서 적용한다.

1. 객체의 행동이 상태에 따라 달라지고, 런타임에 행동이 바뀜.
2. 객체에 따른 다중 분기 조건 처리가 많을 때.

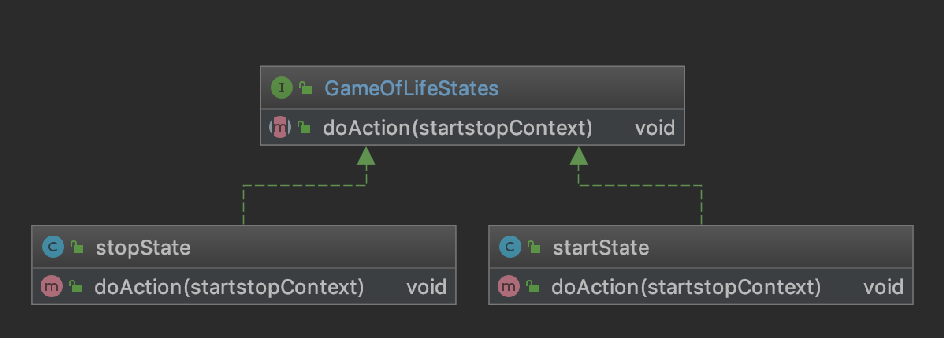
이 중 , Game of life 프로젝트 같은 경우에는 1번 경우에 해당한다. 게임의 start/stop 버튼에 따른 행동 변화가 있기 때문에 패턴을 적용하였다.

기본적으로 프로그램은 버튼을 눌러야 start state 에 진입하고, 초기 시작시에는 stop state에 있다. start state 없이는 next를 통해 다음 세대를 확인하며, start 버튼을 누를 경우 일정 속도로 자동으로 세대를 증가시켜준다. 위 두 상태를 관리하고 각기 다른 로직 구현을 위해 State 패턴을 적용했다.

기존 프로그램은 startStopButtonClicked (GameOfLife.java) 를 통해 state 를 관리하며, stop(), start2() method 를 통해 game state 를 관리한다.

이를 GameOfLifeStates interface 를 선언한 뒤, startState, stopState 두 concrete subclass 들이 구현하여 state를 구성한다. 그 후 startStopContext 가 doAction() method 를 통해 state 를 변경한다.

doAction은 context 를 통해 controls를 받아 현재 state에 대한method를 실행한 뒤 setState()를 통해 다음 state로 이전한다.



< Fig 6. State 패턴 다이어그램 >

기존 코드와 변경된 코드는 다음과 같다.

**[ 변경 전 소스 - GameOfLife.java ]**

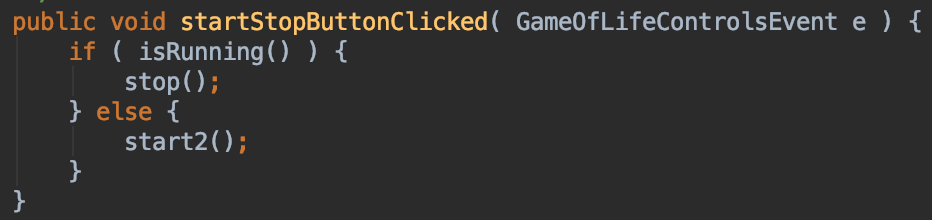
public void startStopButtonClicked(GameOfLifeControlsEvent e) { ... }.

**[ 변경 후 소스 – startstopContext.java ]**

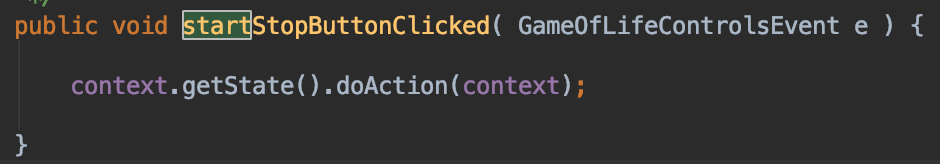
public class startstopContext { ... }

**[ 변경 후 소스 – GameOfLifeStates.java ]**

public interface GameOfLifeStates



< Fig 7. 상태 관리가 하드 코딩된, 변경 전 소스 >



< Fig 8. State 패턴이 적용된, 변경 후 소스 >

**5. Command 패턴**

Game of life 에서는 cell이 진화하는 속도와 진화 과정을 보여주는 디스플레이의 크기를 조절할 수 있다. 이는 GameOfLifeControls 의 listener 를 통하여 GameOfLife class의 speedChanged, zoomChanged 를 통하여 속도와 크기를 바꾸게 된다.

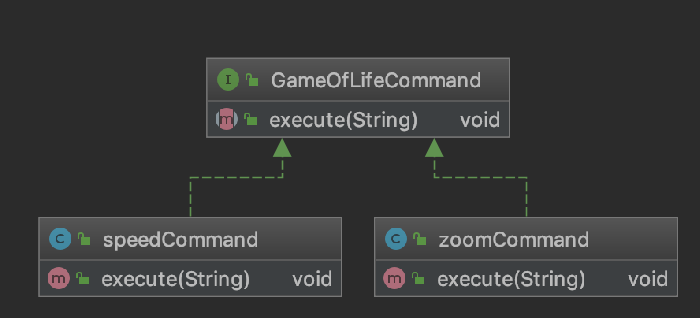
기존 프로젝트에서는 이를 if-else 의 다중 분기로 구현하였다. 또한, 속도와 크기를 final static 형식으로 GameOfLifeControls class 에 구현되어있다.

이런 구현체를 Command pattern 을 통해 refactoring 하여 요구사항을 객체로 캡슐화한다.

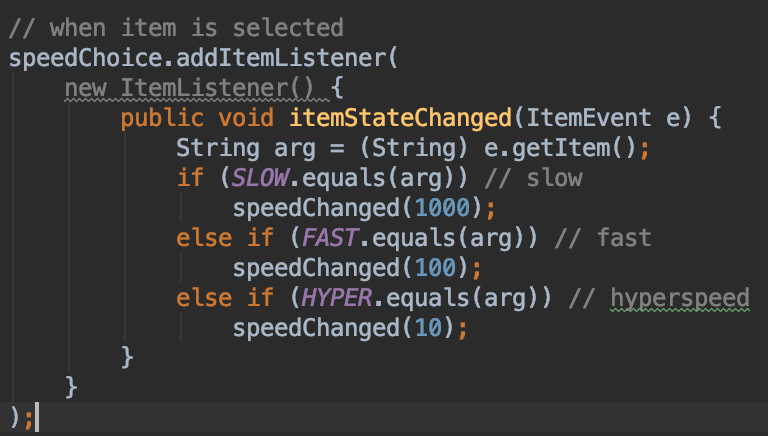
GameOfLifeCommand 인터페이스를 만들고,이를 구현한다. Speed, zoom command 를 concrete class 로 구현하고, 이를 GameOfLifeScrollboxControl.java 라는 Receiver 를 통해 speed,zoom 을 설정한다.

1. Client: GameOfLifeCommand class
2. Invoker GameOfLifeScrollboxControl class
3. Receiver SpeedCommand/zoomCommand class
4. Command commandpkg/GameOfLifeCommand interface

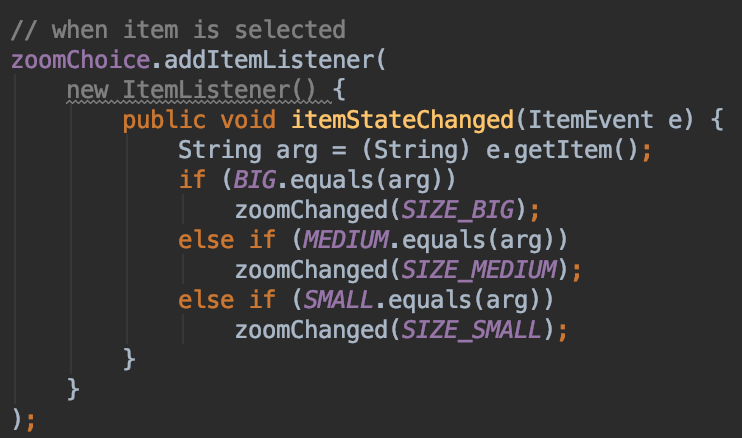
Command 패턴의 다이어그램은 다음과 같다.



< Fig 11. Command 패턴 다이어그램 >



< Fig 9. 하드코딩된, 속도 조절 소스 - GameOfLifeScrollboxControl.java >



< Fig 10. 하드코딩된, 격자 크기 조절 소스 - GameOfLifeScrollboxControl.java >

if ~ else 구문으로 메소드를 호출하고 있다. 이는 후에 변경될 가능성이 매우 크고 실수할 여지가 큰 소스다. Command 패턴을 적용한 소스 코드는 다음과 같다.

[ 변경 후 소스 - GameOfLifeScrollboxControl.java ]

String arg = (String) e.getItem();

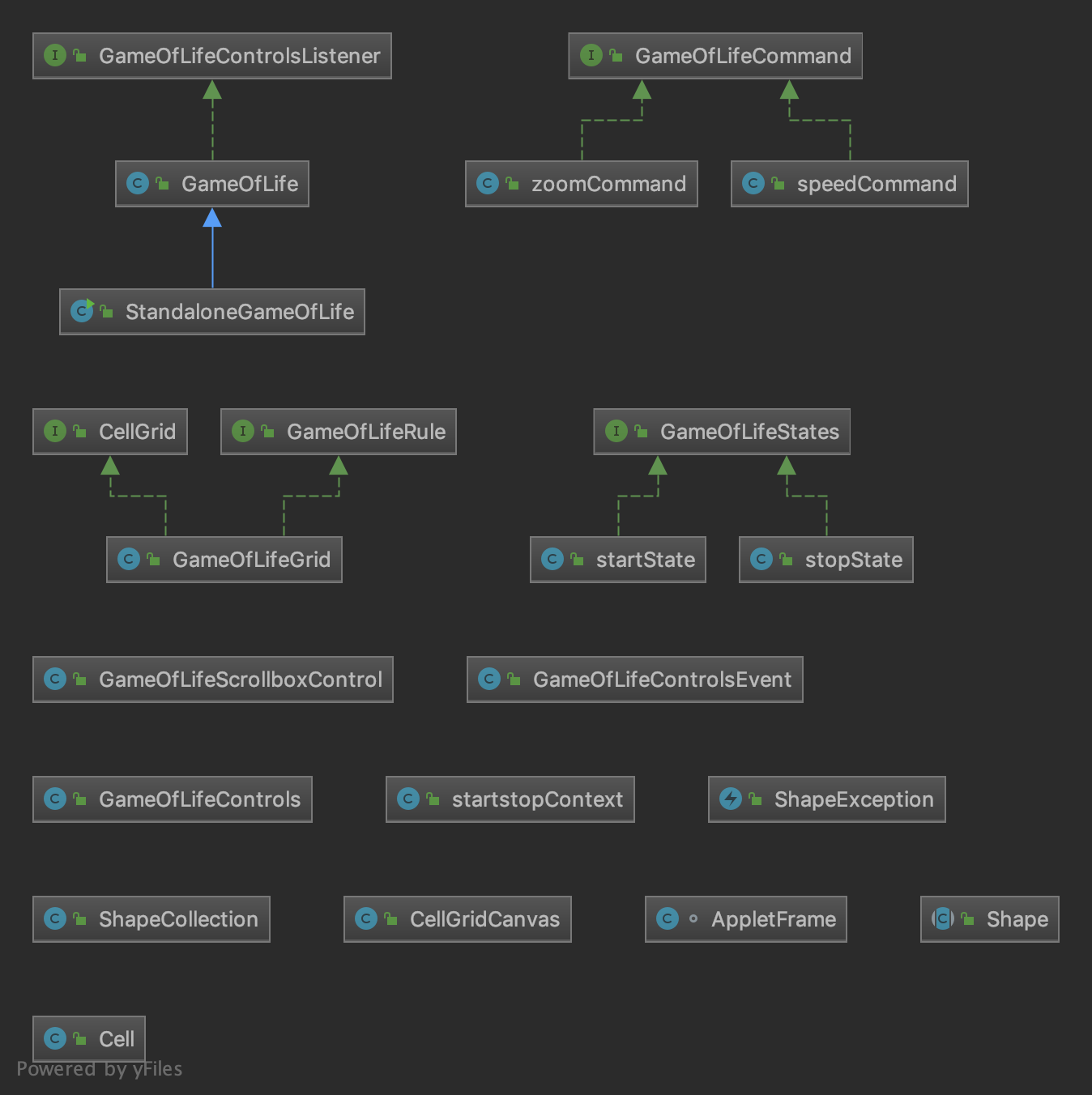
scControl.speedCommand(arg);

String arg = (String) e.getItem();

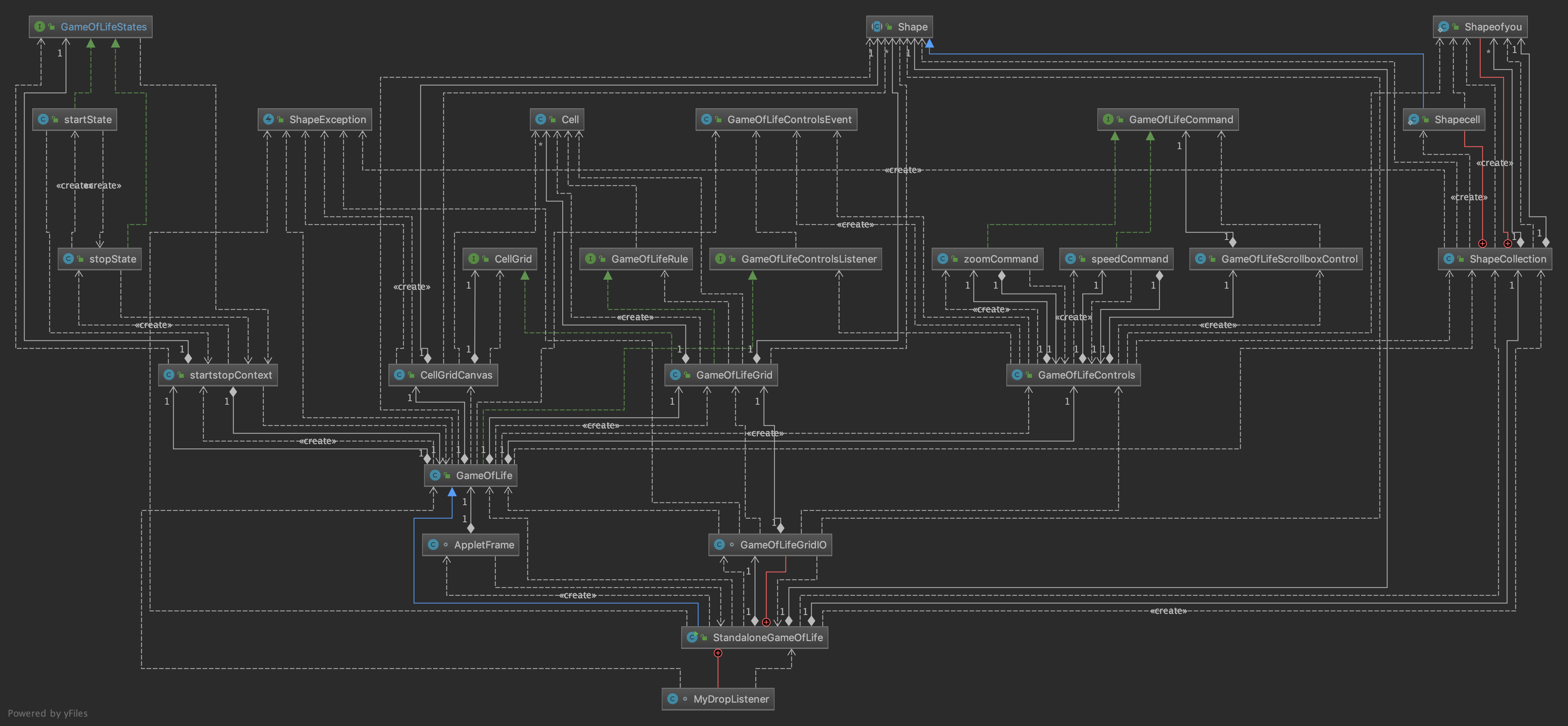
scControl.zoomCommand(arg);

**수정한 프로그램과 장단점**

패턴을 적용한 프로그램의 UML 다이어그램과 의존성 관계는 다음과 같다.

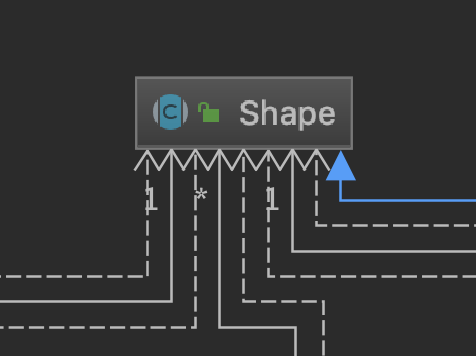


< Fig 12. 수정 프로그램 UML >

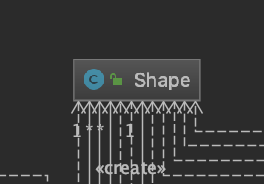


< Fig 13. 수정 프로그램 의존성 관계 >

처음에 제기한 의존성 문제를 해소했다. Shape 클래스를 의존하는 클래스가 줄은 것을 다음과 같이 확인할 수 있다.



< Fig 14. 변경 후 Shape 클래스의 의존성 >



< Fig 15. 변경 전 Shape 클래스의 의존성 >

원본 프로그램에 비해 클래스 수가 늘어난 것을 확인할 수 있다. 로직을 분리하는 패턴을 적용하여, 하나에 있던 여러 개의 책임을 분산시켰다. 그럼으로써, 객체 간의 결합도를 낮추고 확장에 열려있는 클래스 구조로 개선했다. 수정 프로그램의 장점과 단점은 다음과 같이 정리할 수 있다.

< 장점 >

1. 로직 코드 분리하여 결합도 낮춤
2. 확장에 용이(speed,zoom, cell shape)

< 단점 >

1. 관리해야 할 패키지 및 클래스가 많아짐(state,command pattern 적용)

**결론**

테스트 케이스

프로그램 사진

맺음말

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