Design Pattern

최종 레포트

CHUNG-ANG UNIVERSITY

Subject	설계패턴
Professor	이찬근 교수님
Team name	삼삼하조
Team members	김나정 20131667
	서상원 20134563
	최은정 20132621
	최현경 20142167

목차

Ⅰ. 프로젝트 개요

1. 프로젝트 배경 및 목표

"설계에 있어 완벽함 이란 더 이상 추가할 것이 없을 때 이루어지는 것이 아니라 더 이상 버릴 것이 없을 때 이루어진다" 라는 에릭 레이몬드의 소프트웨어 핵심 설계의 명언이 있다.

객체지향 프로그래밍에는 유지보수와 확장이 쉬운 시스템을 만들고자 SOLID 5 가지의 설계 기본원칙이 존재 한다. 이미 3 학년 1 학기에 수강한 소프트웨어 공학과목에서 통칭 SOLID 라고 불리는 5 개의 설계원칙인 OOD 를 배웠다. 또, 이런 설계원칙들을 이용한 설계패턴들을 이번 수업을 통해 배울 수 있었다.

한 학기를 마무리하면서, 설계 패턴들을 실제 프로젝트에 적용하는 시간을 가지려고 한다. 주어진 라이프 게임 프로젝트를 분석하고, 이 프로젝트에 적용할 수 있는 패턴들을 구상하여 적용하는 것이 목표다. 최종적으로는 수업시간에 배운 여러 설계패턴들을 적용해 라이프 게임 프로젝트를 확장 및 개선하고자 한다.

2. 개발환경

구분		항목
개발환경	OS	Windows 10
	개발 언어	JAVA
	JAVA JDK	1.8.0
	IDE	IntelliJ IDEA 2017.2
	VCS	OneDrive
		GitHub
	문서 비교 Tool	WinMerge, SimpleWinMerge

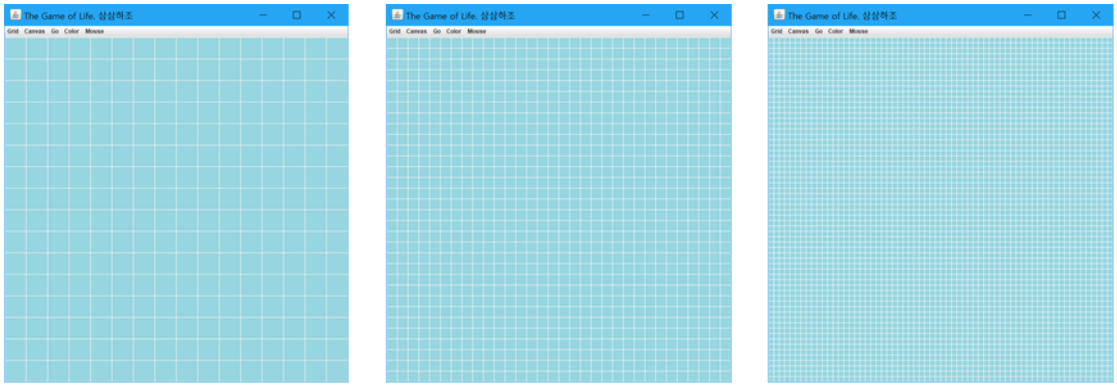
3. 팀원 소개 및 역할

학번	이름	역할
20131667	김나정	문서, Speed
20134563	서상원	문서, Mouse, Visitor
20132021	최은정	문서, Canvas
20142167	최현경	문서, Theme

Ⅱ. 프로그램 디자인

1. 확장한 기능

1) 캔버스



<Small> <Medium> <Large>

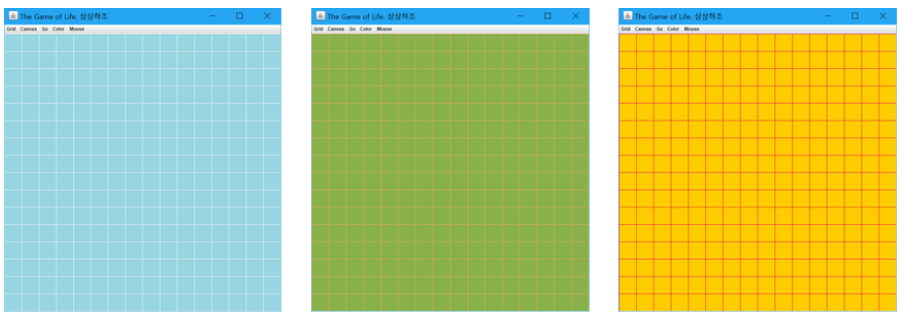
Figure 1: 캔버스 크기 조절 기능

메뉴에서 캔버스 크기를 조절 가능하다. Small, Medium, Large로 캔버스 크기가 커질수록 셀의 크기는 작아져 많은 셀들을 다룰 수 있게 된다.

2) 속도

기존에 프로그램에서 제공하고 있는 속도가 아닌 다른 속도를 추가하고 싶을 때, 그 속도에 대한 정보가 담긴 클래스를 추가하고 MySpeed에 클래스를 덧붙이면 쉽게 속도를 추가할 수 있다.

3) 컬러

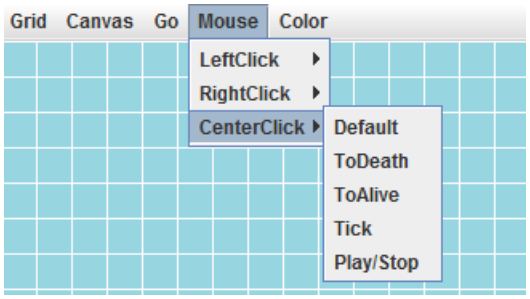


<Blue> <Green> <Red>

Figure 2: 테마 변경 기능

사용자가 게임의 테마를 선택할 수 있다. Blue, Green, Red 3가지의 테마가 있으며, 속도와 마찬가지로 테마를 더 추가하고 싶으면 테마 정보를 입력한 클래스의 추가로 확장이 용이하다.

4) 마우스



기능	살아있는 셀	죽어있는 셀
Defalt	Death	Alive
ToAlive	Alive	Alive
ToDeath	Death	Death

Figure 3: 마우스 메뉴

사용자가 마우스를 이용해 클릭할 때, 오른쪽 버튼을 클릭 할 때, 왼쪽 버튼을 클릭할 때, 휠을 클릭할 때의 기능을 지정할 수 있다. 각 버튼 당 지정할 수 있는 기능은 Default, ToAlive, ToDeath, Tick, Play/Stop이다. Default의 경우에는 원래 라이프 게임에서 클릭(살아있는 셀을 클릭하면 죽고, 죽어 있는 셀을 클릭하면 살아있는 상태로 만드는 클릭)할 때와 동일하다. ToAlive는 클릭할 때 같은 셀을 두 번을 클릭하든 3번을 클릭하든 셀을 살아있는 상태로 만든다. ToDeath는 ToAlive와 반대로 셀을 클릭할 때 셀을 죽어 있는 상태로 만든다. Tick의 경우에는 Go메뉴의 Tick 기능과 동일한 역할을 수행한다. 마우스 버튼 클릭만으로 Tick의 기능을 수행할 수 있다. 또한 우리가 추가한 기능인 play/stop 기능을 커맨드로 추상화 하였다. 재생 중이면 정지, 정지상태면 재생을 하도록 하는 기능이다.

5) 메뉴 Visitor

메뉴 생성을 Visitor 패턴을 사용하여 메뉴생성에 필요한 클래스를 visit만 하면 자동으로 메뉴가 생성되도록 하여 유연성을 제공함.

2. 패턴 구현 방법과 적용 이유

1) 캔버스

적용한 패턴

Singleton Pattern

이유

기존의 소스코드에서는 Universe에서 gridSize와 cellSize를 final 변수로 정하여 size를 정하고 이후 변경되지 못하도록 설정하고 있다. 하지만 우리가 진행하는 프로젝트는 다양한 size를 제공하기 위해서 defaultSize라는 class를 이용하여 size에 관련한 정보를 setting한다. size의 정보는 clear가 실행되는 경우에도 사용자가 지정한 사이즈로 남아 있어야 된다. 그렇기 때문에 이것을 일반적인 변수로 지정을 하여 시도하면 문제가 생길 수 있다. 그래서 size에 관련된 정보를 singleton으로 구현하여 항상 하나의 instance를 통해 하나의 정보만 가지고 있을 수 있게 하였다.

구현방법

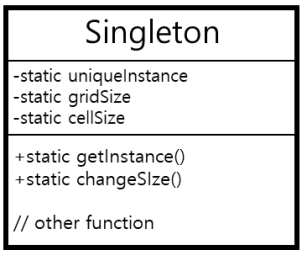


Figure 4: 싱글톤

gridSize와 cellSize를 defaultSize class에서 저장한다. 만약 패러미터가 없이 defualSize가 실행이 되면 8, 8으로 각각 지정된다. getInstance를 실행하는 함수는 synchronized를 이용하여 오직 1번만 실행되게 한다. 사용자가 사이즈를 변경하기를 원할 때 changeSize함수가 실행되는데 이때 instance가 null이 아니면 이를 null로 변경하고 instance를 다시 생성한다. 사이즈는 3단계로 구분되는데 2,8 4,8 8,8로 이를 지정하였다.

2) 속도

적용한 패턴

Strategy Pattern

이유

속도를 구현할 때, 가장 핵심 목표는 추가 속도를 구현하게 될 때 개발자 입장에서 쉽게 확장할 수 있게 만드는 것이다. 속도의 경우, Clock 클래스에서 변화하는 부분이다. 이 변화하는 부분을 변화하지 않는 부분에서 분리해 캡슐화 하여 필요할 때 교체할 수 있게 만든다면 새로운 속도를 추가하더라도 쉽게 추가할 수 있을 것이다. 이런 점이 Strategy 패턴에 적합하다고 판단하였다.

구현방법

Strategy 패턴을 적용할 때 가장 먼저 고려해야 할 부분은 변화하는 부분을 뽑아내는 것이다. 속도를 Strategy 패턴으로 구현하기 위해서 Speed 패키지를 만들었다. Speed 패키지에 각각의 속도들이 implement할 Speed라는 interface를 만들었다. Speed interface는 속도의 이름을 리턴하는 getName()과 Clock의 Tick함수에 들어갈 주기를 리턴하는 getPeriod()를 함수로 갖는다. Strategy 패턴에 대입해보면 Strategy인 Speed를 implement할 Concrete Strategy를 기존 Life Game에서 제공하고 있는 속도인 Agonizing, Slow, Medium, Fast를 Clock 클래스에서 분리해 구현하였다.

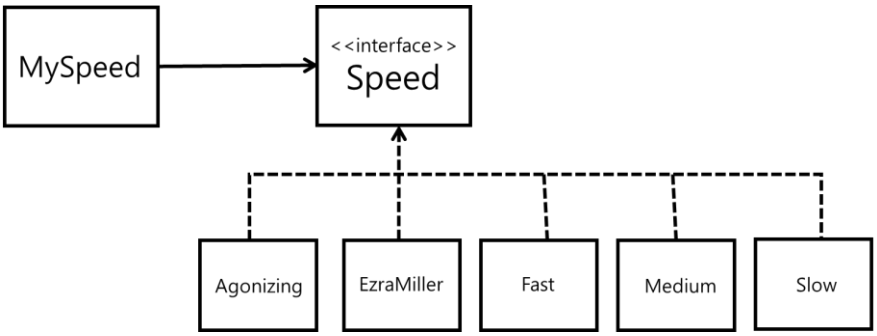


Figure 5: Strategy Pattern

Speed 기능에서 Strategy 패턴은 위와 같이 적용되어 있다. Speed가 Strategy이고, Concrete Strategy는 Agonizing, Fast, Medium, … 등이 있다. MySpeed가 Speed를 쓰는 Context이다.

3) Theme

적용한 패턴

Bridge Patter: 구조적, 객체패턴

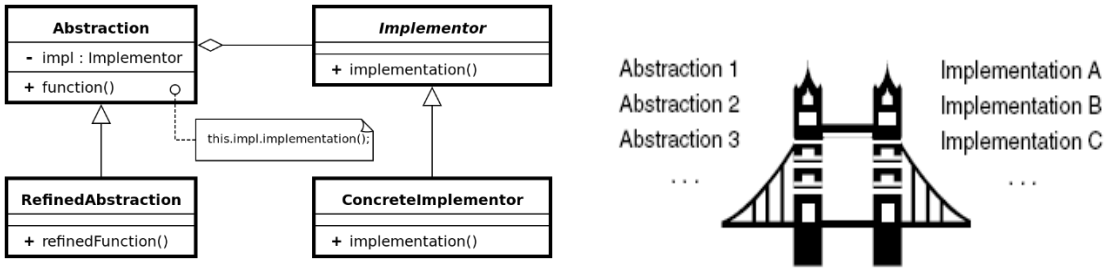


Figure 6 : Bridge

구현(Implementation)으로부터 개념(Conceptual, 수업시간에 Abstraction보다 Conceptual이 맞는 표현이라고 하였음) 계층을 분리하여 이 두 계층이 서로 독립적으로 변화할 수 있도록 한다.

Abstraction - 추상 인터페이스를 정의한다. Implementor에 대한 레퍼런스를 유지
RefinedAbstraction - Abstraction에 의해 정의된 인터페이스를 확장(extends)
Implementor - 구현 클래스를 위한 인터페이스를 정의
ConcreteImplementor - Implementor 인터페이스를 구현

이유

Universe.java, Resident.java, Neighborhood.java 이 세가지 소스코드에 색깔을 설정하는 코드들이 분산되어 있다. 이것을 하나로 뭉치기 위하여 Bridge Pattern을 사용하였다. 클라이언트의 요청을 받아 색상을 수정할 경우 많은 부분의 클래스를 수정하여야 한다. 이 부분을 해결하기 위하여 개념적 계층과 구현 부분 계층을 분리하였다.

Bridge 패턴의 장점

구현부를 인터페이스(개념,추상)에 완전히 결합시키지 않았기 때문에 구현과 추상화(개념적인 부분)된 부분을 분리시킬 수 있다. -> 독립적으로 확장 가능
사용자의 요구사항에 맞게 Theme 변경 및 추가가 용이하다.

Bridge 패턴의 활용

추상화(개념적)과 구현이 컴파일 타임에 묶여서 실행되면 안될 때
인터페이스와 실제 구현부를 서로 다른 방식으로 변경해야 하는 경우에 유용하게 사용
하지만 디자인이 복잡해진다는 단점이 있다.

구현방법

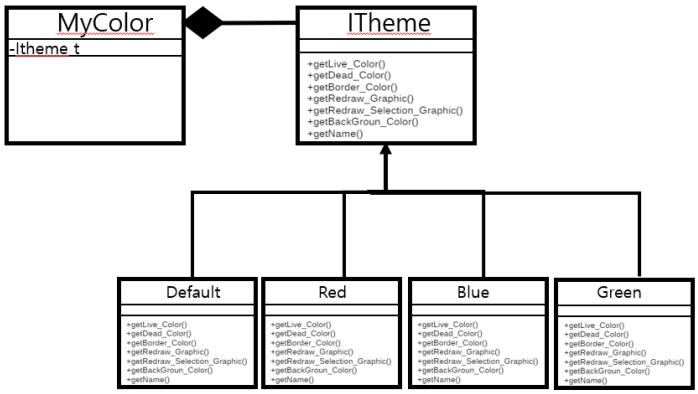


Figure 7: Bridge Pattern

MyColor -> Abstraction 부분은 추상인터페이스를 정의. Implementor인 ITheme 레퍼러스를 유지

RefinedAbstraction - Abstraction에 의해 정의된 인터페이스를 확장(extends)

ITheme -> Implementor

```
+getLive_Color()
+getDead_Color()
+getBorder_Color()
+getRedraw_Graphic()
+getRedraw_Selection_Graphic()
+getBackGroun_Color()
+getName()
```

이런 함수들을 제공한다. 이 부분은 색깔과 테마이름을 유연하게 설정할 수 있게 해준다.

ConcreteImplementor - Implementor 인터페이스를 구현하는 부분이다. 현재 구현된 Theme색은 Default, Green, Blue, Red 부분이다.

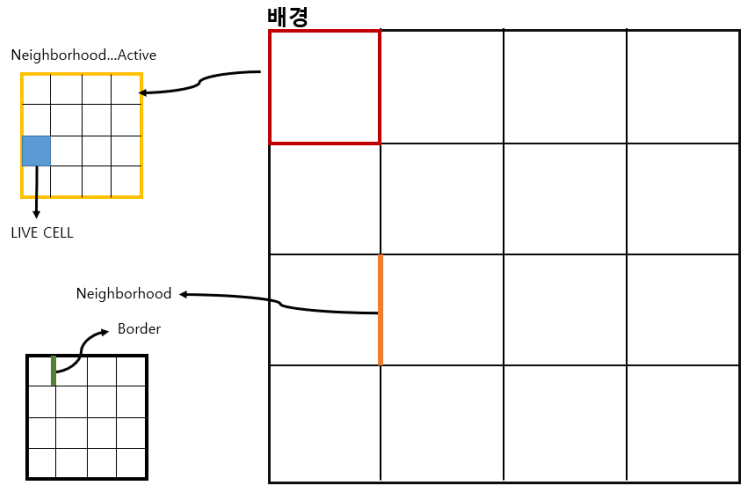


Figure 8: Color UI

Neighborhood.java 에 g.setColor()와 g.setColor //Active 상태면 한 영역이 색깔을 지정하는 형식 Resident.java 에서 역시 dead_cell, live_cell, 하나의 이웃 영역안에 cell영역을 표시하는 border의

선을 조절하는 부분이 있다. Background Frame은 결국 dead_cell로 덮여서 코드를 구현할 때 (테마를 적용시) 동일한 색깔로 적용하였다.

```
public interface ITheme {
    String getName();

    //cell
    public Color getLive_Color (); //세포색깔
    public Color getDead_Color (); //죽은 세포색깔
    //cell 영역안에 있는 줄 색깔 !!!
    public Color getBorder_Color (); //게임판색깔

    //Neighborhood 색깔 지정 //-> cell의 영역표시하는 줄 색깔 지저
    //Border색상과 동일하게
    public Color getRedraw_Graphic();
    public Color getRedraw_Selection_Graphic();

    //universe
    public Color getBackGround_Color ();

}
```

```
public class MyColor {

    //default조정

    private static MyColor ourInstance = new MyColor();

    public static MyColor getInstance() { return ourInstance; }

    private ITheme t;
    private MyColor() { t =  new DefaultTheme(); }


    //set theme 부분
    public void setTheme(ITheme t) {
        this.t = t;
        Universe.instance().repaint();

    }
```

Figure 9: 코드 부분

적용한 부분의 일부

```
        //g.setColor(  BORDER_COLOR);
//혁녕 추후 적용
        g.setColor(  MyColor.getInstance().getT().getBorder_Color() );
```

Figure 10: 패턴 적용 예시

```
getBorder_Color()                    Color
getRedraw_Graphic()                  Color
getRedraw_Selection_Graphic()        Color
getBackGround_Color()                Color
getDead_Color()                      Color
getLive_Color()                      Color
getName()                           String
```

Figure 11: GetXXX함수

이제 색상을 바꾸는 부분은 다 MyColor.getInstance().getT().get~~~ 식을 이용하여 변경하였다. Bridge 패턴을 이용하여 Color을 설정하는 개념적인 부분을 분리하였다. 이제 클라이언트(유저)의 요구사항에 맞게 Cell 색깔, Border, Neighborhood, Active 색깔을 기존에는 여러 클래스에서 하나하나 작업을 했다면 하나의 색상 클래스 추가로 쉽게 extend 할 수 있다.

4) 마우스

적용한 패턴

Command Pattern

이유

마우스 각각에 대한 요청(ex. 재생, 셀 활성화)들을 라이프 게임의 새로운 기능으로 추가하였다. 이때, 요청들을 동적으로 변경하는 것이 필요하다. 그러므로, 요청들을 캡슐화하는 Command Pattern을 적용하기로 하였다. 요청들을 캡슐화하여 요청들이 동적으로 변경 가능케 했다. 또한, Command Pattern의 적용으로 새로운 요청들에 대해서도 쉽게 추가할 수 있다.

구현방법

Command interface를 implement하는 요청들을 Concrete Command 클래스로 생성한다. Concrete Command로는 AlwaysAliveCommand, AlwaysDieCommand, DefaultCommand 등이 있다. 각각의 클래스의 execute() 함수에 각각의 기능들을 구현하였다. 각각 클래스가 하는 요청들은 다음과 같다.

AlwaysAliveCommand –셀을 클릭 할 때, 항상 셀이 활성화

AlwaysDieCommand – 셀을 클릭 할 때, 항상 셀이 비활성화

DefaultCommand – 셀을 클릭 할 때, 셀이 비활성화 된 상태일 경우엔 활성화, 활성화된 상태일 땐 비활성화

PlayNStopCommand – MySpeed 클래스에서 Play가 되어있는 경우엔 멈추고, 정지된 상태에서는 play를 하는 getTickTime()이란 함수를 생성하였다. 이 요청을 객체화한 클래스

TickCommand – Clock 클래스의 셀들을 Tick 하는 요청을 객체화한 클래스

우리는 Command Pattern을 사용하여 위의 요청들을 객체화 하였다. 이 Concrete Command들을 실행하는 것이 MouseCommandManager이다.

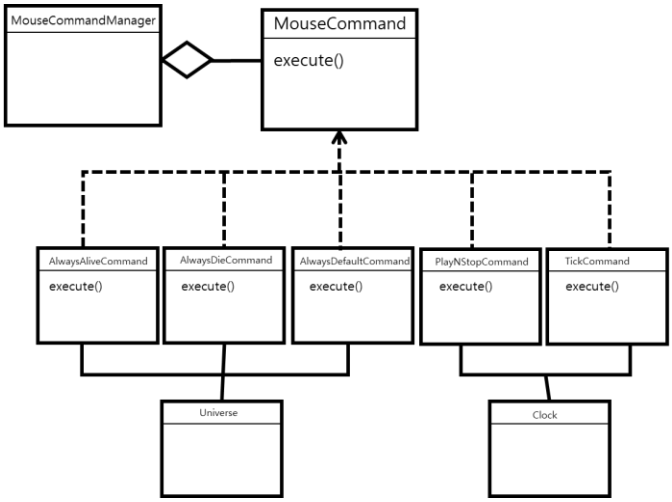


Figure 12: Command Pattern

마우스 구현에서 Command Pattern은 위의 그림과 같이 적용되었다. MouseCommand는invoker 역할을 하고 있으며, Universe와 Clock은 ConcreteCommand의 Receiver이다.

5) Menu Visitor

적용한 패턴

Visitor Pattern

이유

기존의 Life Game은 메뉴를 추가하는데 불편함이 있었다. 메뉴를 생성할 때 메뉴에 넣어야 하는 클래스가 직접 Menusite 클래스를 호출하여야 하는 것이다. 비록 Façade로 간략하게 메뉴를 추가하게 되지만, 메뉴가 많아질수록 이를 관리하는 부분에서 불편함이 발생하게 된다. 그래서 Visitor 패턴을 사용하여 Menu에 추가할 함수를 visit만 하면 메뉴를 자동으로 생성하도록 변경하였다. 이로 인해 개발자는 메뉴에 올리는 부분을 통합하여 관리할 수 있어서 Aspect-oriented programming의 원칙을 따르기 위해 만들었다.

구현방법

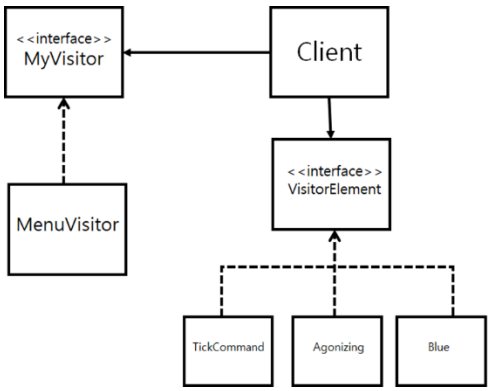


Figure 13: Visitor Pattern

Visitor 패턴의 Visitor 인터페이스는 MyVisitor 파일에 작성하였다. 이를 구현한 Concrete Visitor은 MenuVisitor이다. Visitor 패턴의 Element에 해당하는 부분은 VisitorElement이다. 이 VisitorElement를 구현한 concrete element들은 Default command, TickCommand, PlayNStopCommand, Agonizing등 기존에 있던 메뉴들 중 클래스로 빠진 부분들에 적용하였다. 각각의 element들은 accept함수를 통해서 vistor의 접근을 허가하게 된다. 각각의 visit 행위에 대해서는 MyVisitor 클래스에서 구현하였다.

Ⅲ. 진행 과정

1. 개발 일정

요일	날짜
11/04 ~ 11/10	팀 설정 및 개발 환경 설정
11/11 ~ 11/19	요구사항명세서 작성 및 주석작업
11/20 ~ 11/30	적용 패턴 논의 및 개발
12/01 ~ 12/03	구현 및 테스트
12/04	문서 정리

2. GitHub

https://github.com/kyung2/holub_byeCenter

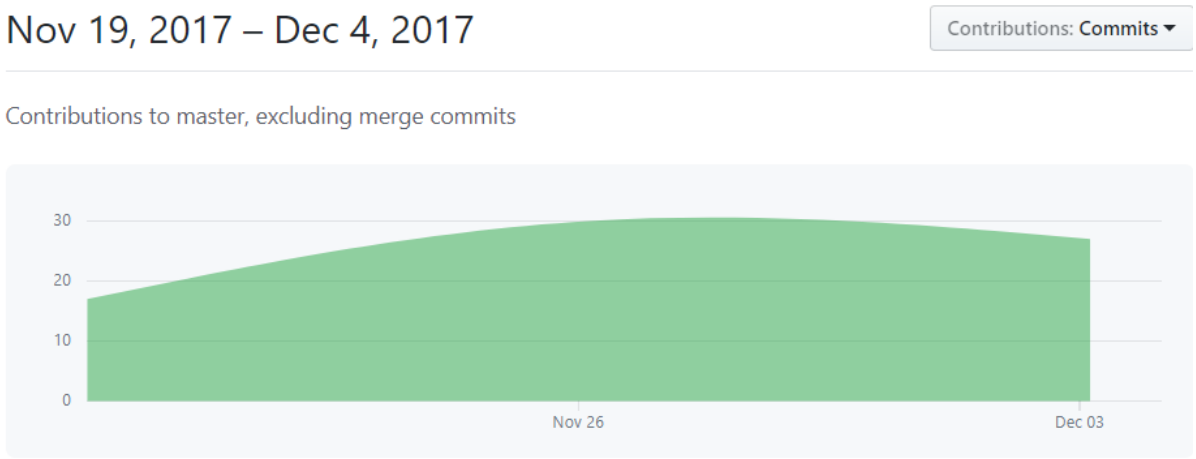


Figure 14: github 그래프

Ⅳ. 소스 코드 전 후 차이

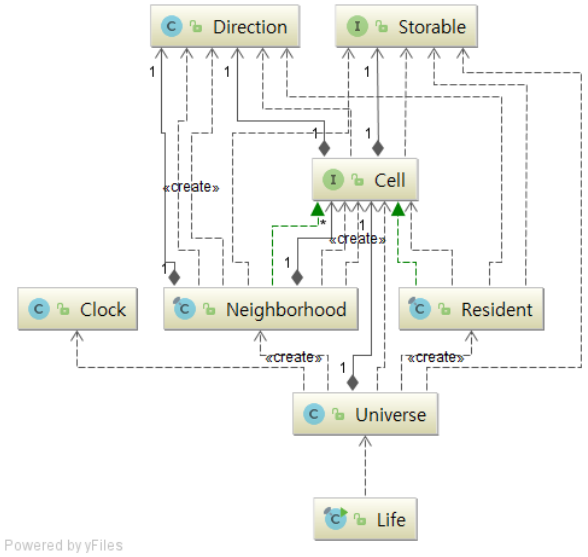


Figure 15: 원래 코드의 다이어그램 life부분 diagram

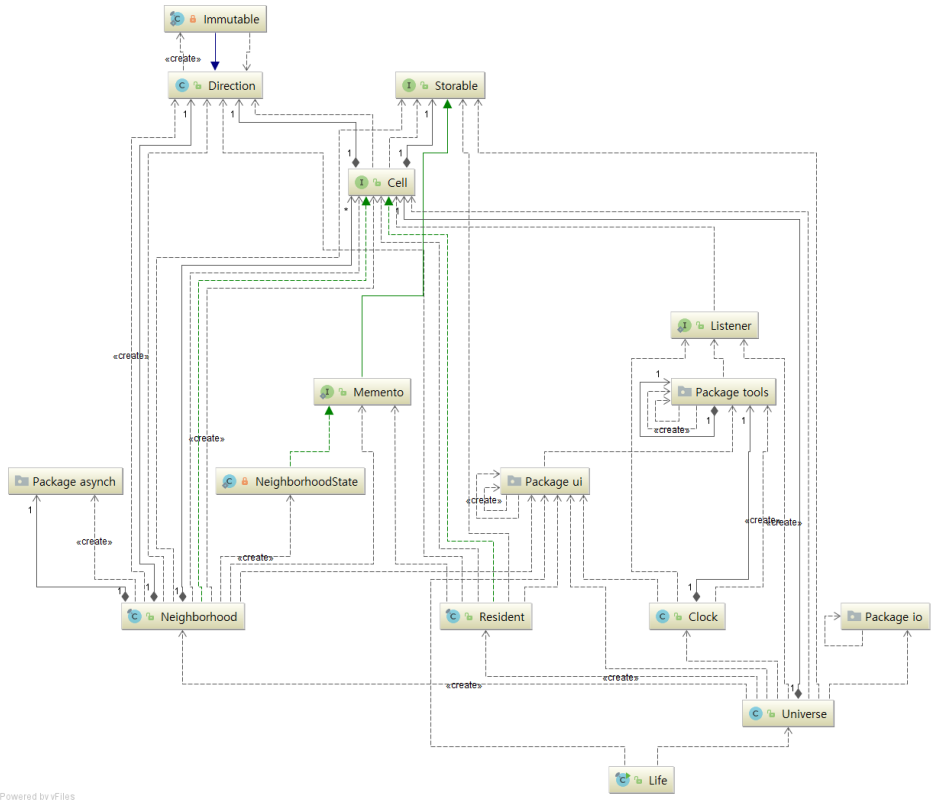


Figure 16: 전체 diagram

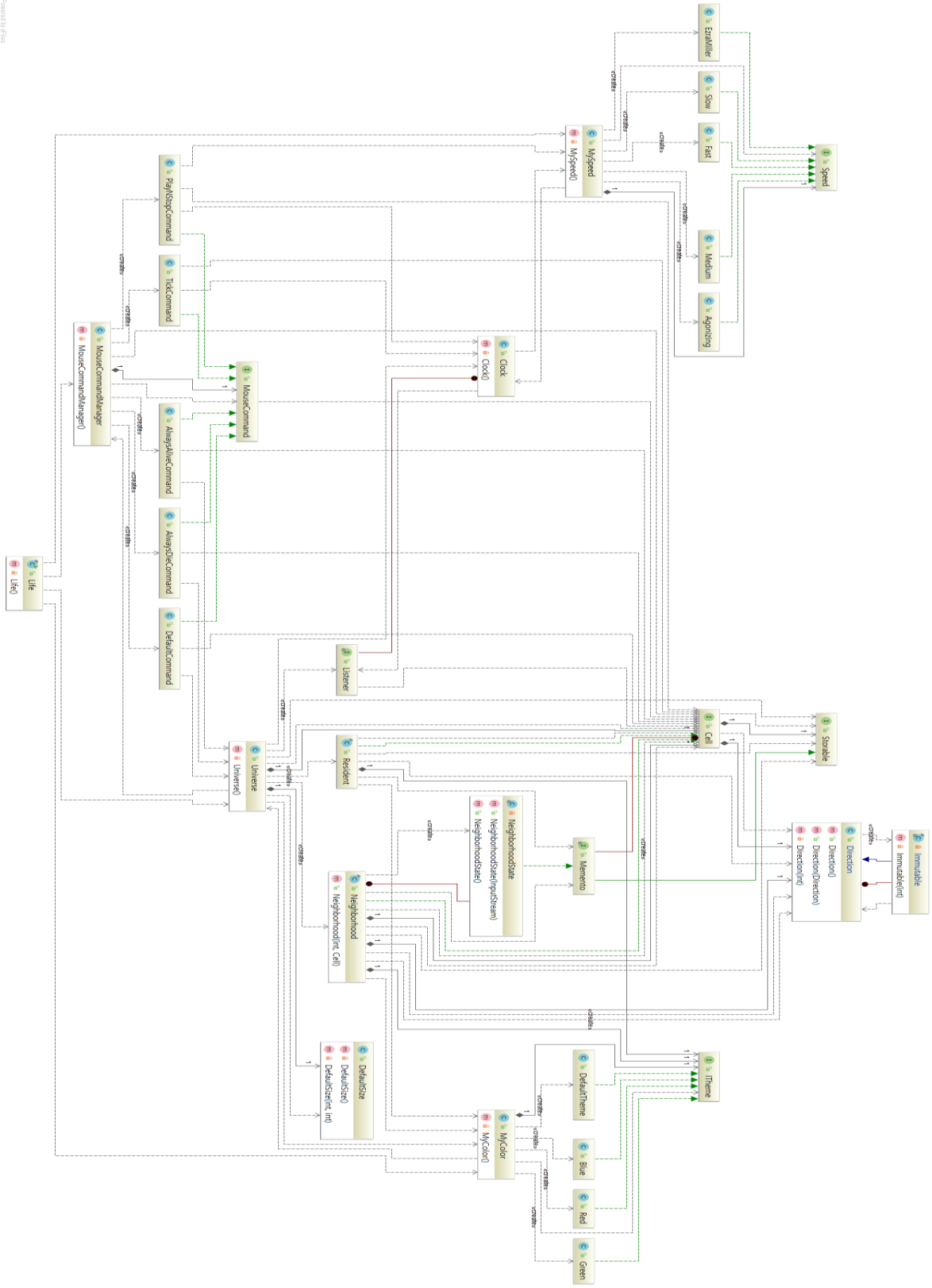


Figure 17: 최종 다이어그램(life 부분)

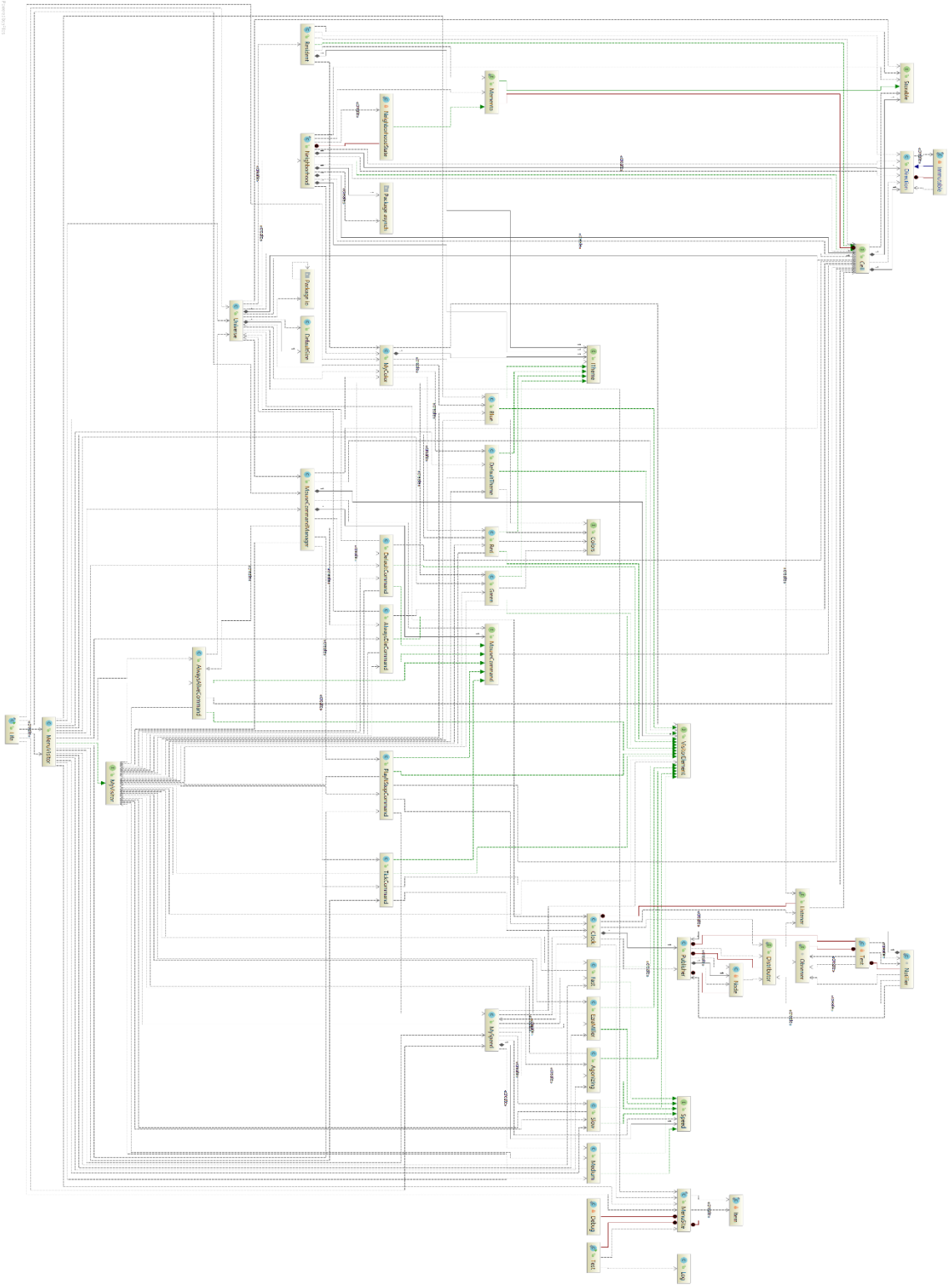


Figure 18: 최종다이어그램 (전체)

모든 변화를 다 비교사진을 올리는 것에 한계가 있어 중요한 부분은 사진으로 남기고 비교내용은 소스코드 비교폴더에 html파일을 참고하면 됩니다.

1. 기존코드 비교

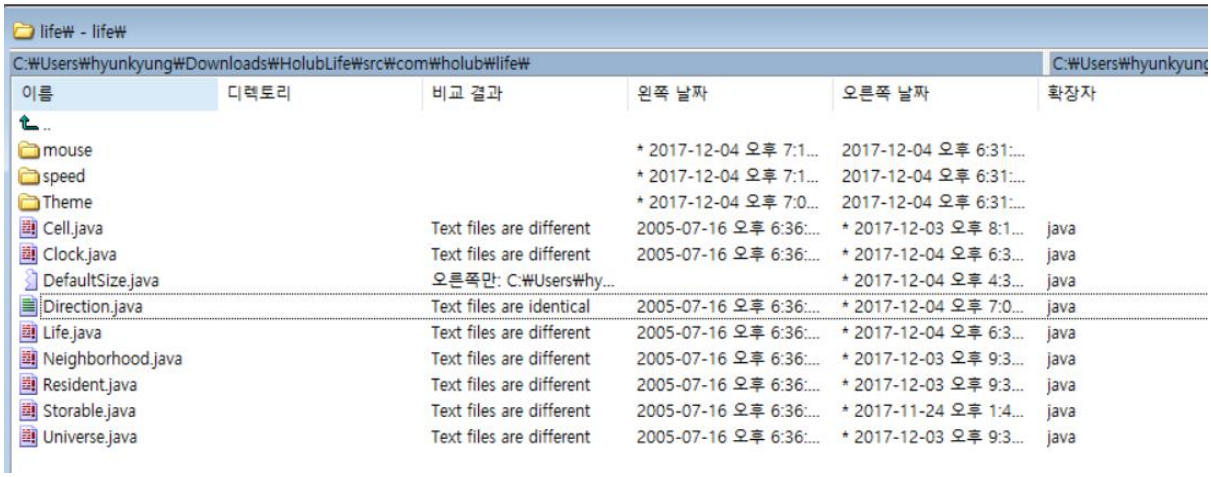


Figure 19: 전체코드 폴더

또한 저희는 Mouse,Speed,Theme를 Package로 설정하여 각각의 패턴을 적용하였습니다.
그래서 비교가능한 java파일 부분은 cell, clock, life, universe, neighborhood, resident 부분입니다.
(Storable 패턴같은경우 코드 분석하면서 주석부분이 비교처리 되어있습니다)

Cell.java

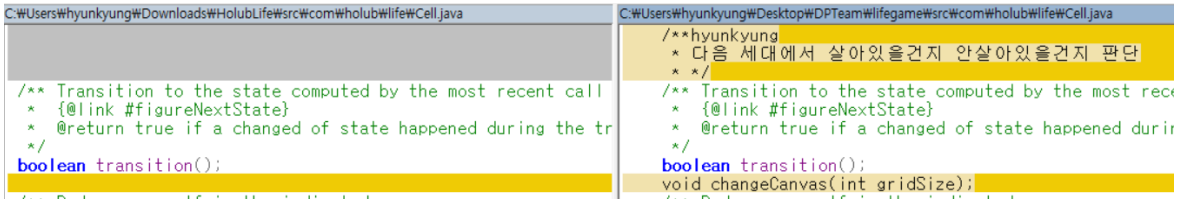


Figure 20: void changeCanvas

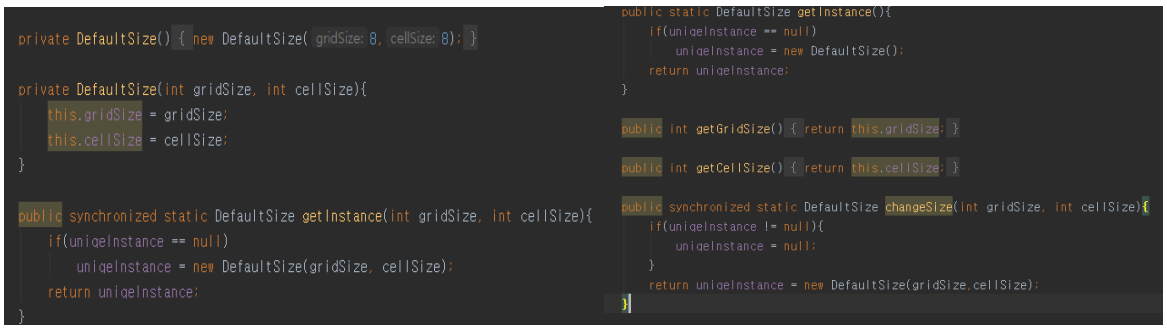


Figure 21: Size조절

grid사이즈와 cell사이즈를 class를 이용하여 지정을 하고 cell의 size가 변경되고 이를 clear하더라

도 사용자가 설정한 size가 나올 수 있도록 설정되어 있습니다.. getinstance()함수를 이용하여 한 번만 instane를 생성하는데 만약 사용자가 캔버스 사이즈를 변경하는 액션을 하여 changeCanvas() 함수가 실행되면 기존의 instance는 NULL로 바꾸고 다시 instance를 생성한다.

Clock.java

```
public void stop()
{   startTicking( 0 );

}
```

```
    public void stop()
    {
        startTicking( 0 );
        MySpeed.getInstance().setPlaying(false);
    }
```

Figure 22: Clock

```
    MenuSite.addLine(this,"Go","Halt",              modifier);
    MenuSite.addLine(this,"Go","Tick (Single Step)",modifier);
    MenuSite.addLine(this,"Go","Agonizing",         modifier);
    MenuSite.addLine(this,"Go","Slow",              modifier);
    MenuSite.addLine(this,"Go","Medium",            modifier);
    MenuSite.addLine(this,"Go","Fast",              modifier); // {=endSetup}
}   //{=endCreateMenus}
```

```
        MenuSite.addLine(this,"Go","Halt",              modifier);
        MenuSite.addLine(this,"Go","Tick (Single Step)",modifier);
        MenuSite.addLine(this,"Go","Play/Stop",modifier);
// {=endSetup}
}   //{=endCreateMenus}
```

Figure 23: Clock

Life.java

```
C:₩Users₩hyunkyung₩Downloads₩HolubLife₩src₩com₩holub₩life₩Life.java
        // Must establish the MenuSite very early in case
        // a subcomponent puts menus on it.
        MenuSite.establish( this );     //{=life.java.establish}

        setDefaultCloseOperation    ( EXIT_ON_CLOSE         );
        getContentPane().setLayout  ( new BorderLayout()    );
        getContentPane().add( Universe.instance(), BorderLayout.CENTER); //{=life.

        pack();
        setVisible( true );
    }
}
```

```
* C:₩Users₩hyunkyung₩Desktop₩DPTeam₩lifegame₩src₩com₩holub₩life₩Life.java
        // Must establish the MenuSite very early in case
        // a subcomponent puts menus on it.

        MenuSite.establish(this);       //{=life.java.establish}

        setDefaultCloseOperation(EXIT_ON_CLOSE);

        getContentPane().setLayout(new  BorderLayout());
        getContentPane().add(Universe.instance(), BorderLayout.CENTER); //{=life.j

        MouseCommandManager manager = MouseCommandManager.getInstance();
        MenuVisitor visitor = new MenuVisitor();
        manager.addMenus(visitor);
        MySpeed.getInstance().addMenus(visitor);
        MyColor.getInstance().addMenus(visitor);
        setPreferredSize(new Dimension(900, 800));
        pack();
        setVisible(true);
    }
//center에 넣을려고함
    public static void centerWindow(Window frame) {

        Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (int) ((dimension.getWidth() - frame.getWidth()) / 2);
        int y = (int) ((dimension.getHeight() - frame.getHeight()) / 2);
        frame.setLocation(x, y);
    }
}
```

Neighborhood.java

```

```

```
    public void changeCanvas(int gridSize){
        this.gridSize = gridSize;

    }
```

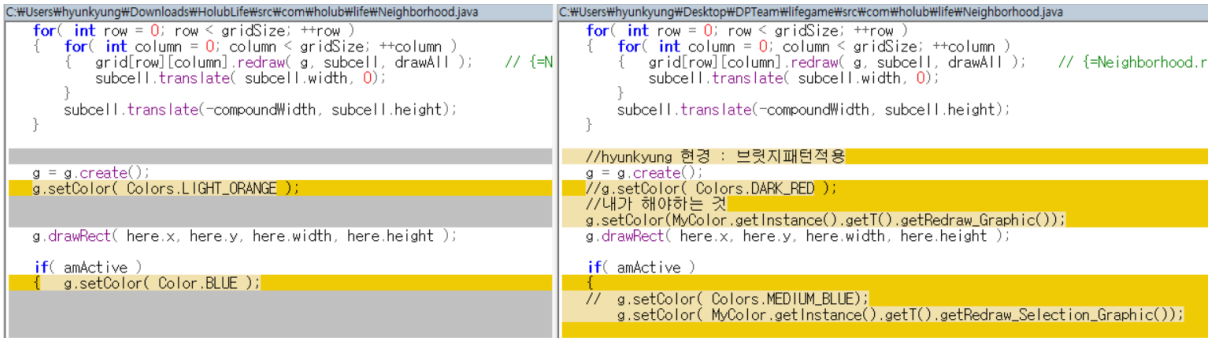


Figure 24: Neighborhood.java Bridge Pattern 브릿지 패턴을 적용하는 부분

Resident.java

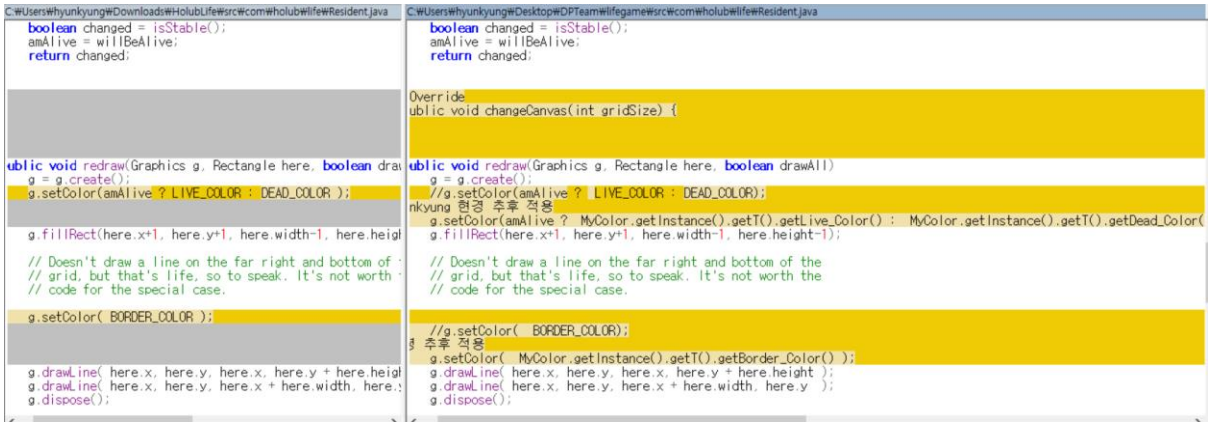


Figure 25: Resident.java Bridge

Universe.java

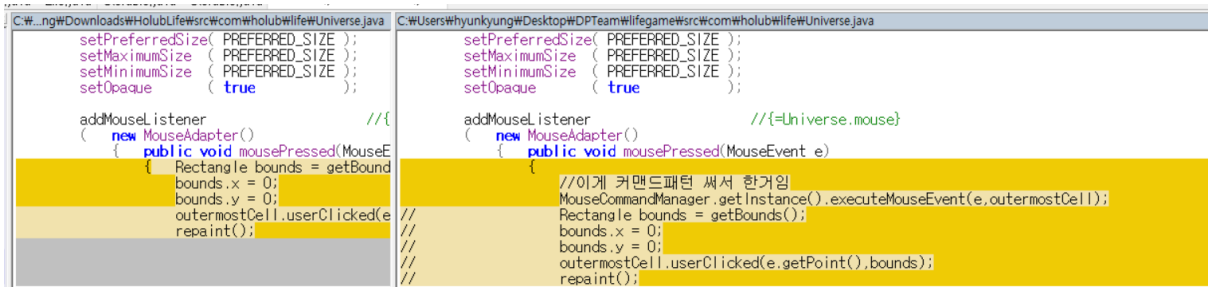


Figure 26: Universe.java 커맨드패턴

Color.java

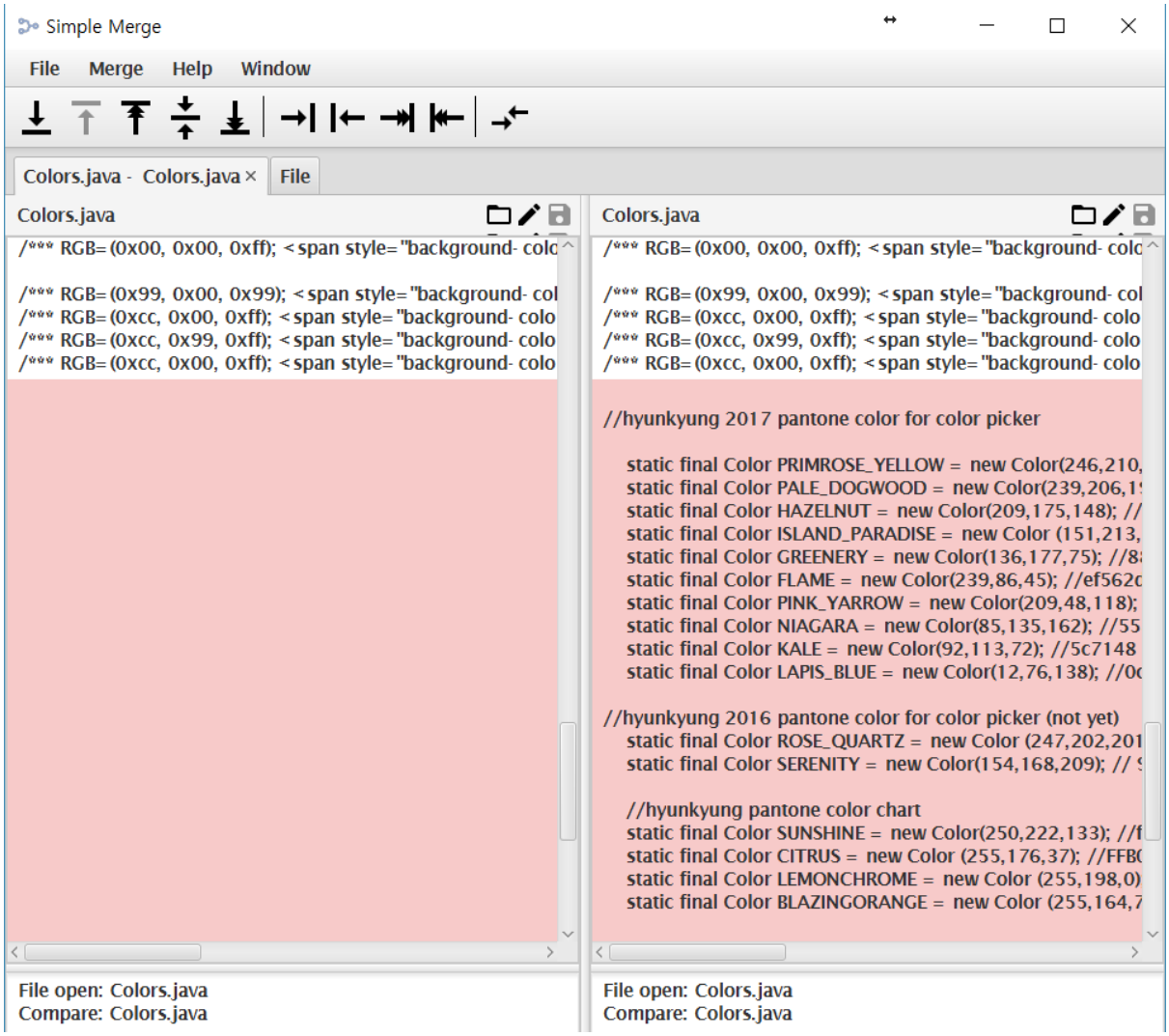


Figure 27 : Color 비교

Colors.java 부분에는 Color 만 설정하는 부분이라서 3학년 소프트웨어공학 과목에서 구현한 현경 SimpleWinMerge을 사용하였다. 나머지 부분은 폴더비교가 가능한 winmerge을 사용하였다.

2. 추가 클래스 코드부분

<Theme Package>

```
public interface ITheme {
    String getName();

    //cell
    public Color getLive_Color (); //세포색깔
    public Color getDead_Color (); //죽은 세포색깔
    //cell 영역안에 있는 줄 색깔 !!!
    public Color getBorder_Color (); //게임판색깔

    //Neighborhood 색깔 지정 //-> cell의 영역표시하는 줄 색깔 지저
    //Border색상과 동일하게
    public Color getRedraw_Graphic();
    public Color getRedraw_Selection_Graphic();

    //universe
    public Color getBackGround_Color ();

}
public class MyColor {

    //default조정

    private static MyColor ourInstance = new MyColor();

    public static MyColor getInstance() { return ourInstance; }

    private ITheme t;
    private MyColor() { t =  new DefaultTheme(); }


    //set theme 부분
    public void setTheme(ITheme t) {
        this.t = t;
        Universe.instance().repaint();

    }

    public ITheme getT() { return t; }

    ArrayList<VisitorElement> getSpeeds(){
        ArrayList<VisitorElement>elements = new ArrayList<>();
        elements.add(new DefaultTheme());
        elements.add(new Blue());
        elements.add(new Green());
        elements.add(new Red());
        return elements;
    }
    public void addMenus(MyVisitor visitor){
        for(VisitorElement element:getSpeeds()){
            element.accept(visitor);
        }
    }
```

Figure 28: package 코드 부분

DefaultTheme - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
package com.holub.life.Theme;

import com.holub.ui.Colors;
import com.holub.ui.MyVisitor;
import com.holub.ui.VisitorElement;

import java.awt.*;

public class DefaultTheme implements ITheme,VisitorElement {

    //cell
    Color c;
    public String getName() {
        return "Defualt";
    }
    public Color getLive_Color() {
        return Color.RED;
    }

    public Color getDead_Color() {

        return Colors.LIGHT_YELLOW;
    }
    public Color getBorder_Color() {

        return Colors.DARK_YELLOW;
    }
    //Neighborhood 색깔 지정
    public Color getRedraw_Graphic() {

        return Colors.LIGHT_ORANGE;
    }
    public Color getRedraw_Selection_Graphic() {
       // c= new Color(178,57,88);

        return Color.BLUE;
    }
    //universe //backgroud 는 어차피 덮히니까..
    public Color getBackGround_Color (){
        return Color.white;
    };
    @Override
    public void accept(MyVisitor visitor) {

        visitor.visit(this);
    }
    @Override
    public String getMenuName() {
        return "Defualt";
    }
}
```

Figure 29: Default 그림

대표적으로 default.java코드만 보여주었다.

Ⅴ. 결론

pdf 파일을 참고하여 코드 분석을 진행하면서 다양한 패턴이 프로젝트에 적용된 것을 확인할 수 있었다. 이를 통해서 우리가 배웠던 패턴들이 코드에 어떻게 적용되어 실제로 활용되는지 알아볼 수 있었고 또 직접적으로 추가함으로 많은 경험을 얻을 수 있었다. 패턴이 복잡하게 얽혀 있어서 아무것도 추가하지 못할 것이라고 생각했는데 패턴이 잘 적용되어 있어서 클래스끼리 결합도가 적어 우리가 조금이라도 수정할 수 있었던 것 같다. 수업시간에 배운 것을 예제로 말고 직접 구현해보니까 더 자세히 공부할 수 있는 계기가 되었다. 특히 bridge 패턴을 구현하면서 객체, 클래스 단위와 행동, 구조, 생성 부분으로 비슷한 패턴들을 더 자세히 이해하게 된 계기가 되었다. 전략패턴과 브릿지 패턴 등 같은 다이어그램을 가진 패턴들의 차이에 대해서 한 번 더 이해하게 된 계기가 되었고, 무엇보다 OOP디자인 원칙에 대해서 다시 한 번 생각하였다. 한 학기 동안 설계패턴을 수강하면서 배웠던 여러 설계 패턴들을 분석하고, 적용하는 시간이었다. 패턴들을 개념으로만 배웠을 때보다 실제로 프로젝트에서 사용하는 것들을 보니 설계패턴에 대해 확실히 와 닿을 수 있었다. 한 학기 동안 공부한 것들을 마무리하는 시간을 가지게 된 것 같다. 이 프로젝트를 Holub on Patterns에서 본 가장 멋진 문장으로 마무리 하겠다.

"라이프게임에서 '삶(life)'이란 '우주(universe)'안에서 '이웃(neighborhood)'과 더불어 '살아가는(Resident)' 것이다."