Design Pattern

요구사항명세서

Subject	설계패턴
Professor	이찬근 교수님
Team name	삼삼하조
Team members	김나정 20131667
	서상원 2013456
	최은정 20132621
	최현경 20142167

목차

Requirement for Life Game (Holub)

프로젝트 목표

주어진 Life Game 소스코드를 디자인 패턴을 적용하여 확장 및 개선

"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 설계 의 명언이 있다.
이미 3학년 1학기에 수강한 소프트웨어 공학과목에서 통칭 SOLID라고 불리는 5개의 설계원칙인 OOD을 배웠다.
객체지향 프로그래밍에는 유지보수와 확장이 쉬운 시스템을 만들고자 **SOLID** 5가지의 설계 기본 원칙이 존재 한다. 즉 이 원칙들은 소프트웨어 작업에서 프로그래머에게 소스코드의 가독성을 높이고, 확장성이 보장 될 때까지 리팩토링을 하여 Code Smell을 제거하기 위해 적용되는 방식이다. 이 방식들은 애자일 소프트웨어개발의 전략 중 일부이다. 주어진 life Game 소스코드를 분석하여 적용할 수 있는 패턴 및 기능을 추가하는 것이 목표이다.

라이프 게임이란?

배경

라이프게임(생명 게임)은 영국의 수학자 존 호튼 콘웨이가 고안해낸 세포 자동차 일종.
미국의 과학잡지 사이언티픽 아메리칸 1970년 10월호 중 마틴 가드너의 칼럼 "수학게임" 란을 통하여 대중들에게 소개되었다.
이 게임은 단순한 규칙 몇가지로 복잡한 패턴을 만들어 낼 수 있다는 점에 많은 관심과 반응을 일으켰다.

게임 설명:

라이프 게임은 자신의 의지로 게임을 진행하는 일반적인 게임과 다르다. 오로지 처음 입력된 초기값으로 완전히 게임방향이 결정된다.
라이프게임은 많은 사각형(혹은 cell)로 이루어진 격자(Grid)위에서 돌아간다.
각각의 세포주위에는 인접해 있는 **8개**의 "이웃 세포"가 존재한다.
각 세포의 상태는 "죽어" 있거나 "살아"있는 두가지 상태 중 한가지 상태를 가진다.
격자를 이루는 세포의 상태는 연속적이 아닌 **이산적**으로 변한다.
즉, 현재 세대의 세포들 전체의 상태가 다음 세대의 세포 전체의 상태를 결정한다.

게임규칙

1. 격자무늬 바탕에 각 칸마다 "세포"이 채워져 있다.
2. 세포는 살거나 죽은 상태 둘 중에 하나의 상태만 가진다.
3. 각 세포 주변에는 8개의 세포(상하좌우, 대각선4방향)을 "**이웃**"이라 한다.
4. 정확히 3개의 이웃이 살아 있다면, (죽어 있는) 세포는 살아난다.
5. 2개의 이웃이 살아 있다면 살아있는 세포 은 다음세대에도 산다.
6. 1개 이하 또는 4개이상의 이웃이 살아있으면, 살아있는 세포 은 외로워서 또는 질식해서 죽는다.

패턴 예시

라이프 게임에는 전혀 변화가 없는 고정된 패턴 (정물, still life)와 일정한 행동을 주기적으로 반복하는 패턴 (진동자, oscillator), 한쪽 방향으로 계속 전진하는 패턴(우주선, spaceship)등 여러 패턴 등이 존재한다.

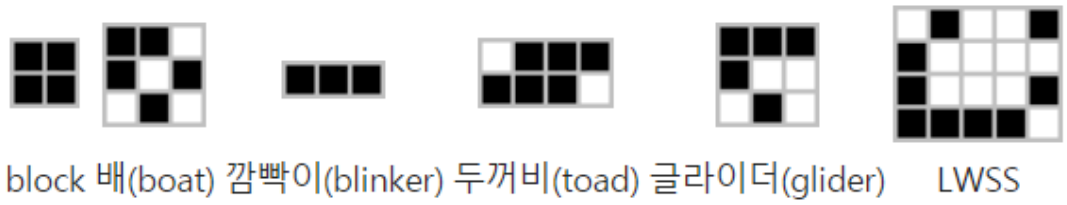

Figure 1패턴의 예시(위키피디아)

Block 과 boat는 정물이고 blinker와 toad는 진동자, 글라이더는와 LWSS(경량급 우주선)은 우주선 패턴이다.

참고 할 만한 사이트

온라인 시뮬레이션

https://bitstorm.org/gameoflife/

Requirements 정리

1) 팀원들이 프로젝트에서 어떤 역할을 했는지 최종보고서에 작성되어 있어야한다.
2) 깃허브 주소와 진행과정이 있어야 한다.
3) 추가하거나 확장한 패턴에 대한 설명이 있어야 한다
   1. 어떤 패턴을 사용하였고 왜 그 패턴을 선택했는지에 대해 설명 해야 한다.
   2. 원래 프로그램에서 어떤 기능 또는 구조가 확장 되었는가 설명되어야 한다.
   3. 제공된 코드와 개선한 코드를 비교하여 설명해야 한다.

Design Principle

SRP: Single Responsibility Principle 단일 책임의 원칙

"객체는 하나의 책임만을 맡아야 한다"

한 클래스는 하나의 책임을 가져야 하며, 하나의 기능에 집중해야 한다.

높은 응집도 => *스트레이트지 (전략) 패턴*에 응용

DIP: Dependency Inversion Principe 의존 관계 역전의 원칙

"클라이언트는 구체 클래스가 아닌 인터페이스나 추상 클래스에 의존 해야 한다"

팩토리 패턴, LSP을 지켜야함

ISP: Interface Segregation Principle 인터페이스 분리의 원칙

"클라이언트에 특화된 여러 개의 인터페이스가 하나의 범용 인터페이스보다 낫다"

여러 개의 구체적 인터페이스 설계

서로 다른 성격의 인터페이스 분리

LSP: Liskov Substitution Principle 리스코프 대체 원칙

"기반 클래스는 파생 클래스로 대체 가능해야 한다"

프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다. 상위 타입의 객체를 하위 타입의 객체로 치환해도 상위타입을 사용하는 프로그램은 정상적으로 동작해야 한다.

OCP: Open-Closed Principle 개방폐쇄원칙

"변화에 닫혀 있고 확장엔 열려 있어야 한다"

변경을 위한 비용은 최대한 줄이고, 확장을 위한 비용은 가능한 늘려야 한다는 원칙이다. 기능을 변경하거나 확장 할 수 있으면서(상속) 그 기능을 사용하는 코드는 수정하지 않는다. 개방-폐쇄원칙은 주로 다운 캐스팅을 하거나 instanceOf 를 사용 할 시 깨진다.

헤드퍼스트의 디자인 원칙

애플리케이션에서 달라지는 부분을 찾아 내고, 달라지지 않는 부분으로부터 분리시킨다.

바뀌는 부분을 따로 뽑아서 캡슐화한다.

패턴의 목적은 시스템의 일부를 다른 부분과 독립적으로 변화 시킬 수 있는 방법 제공

상속보다 구성을 활용한다.

"A is B"보다 "A has B" 가 더 좋다.

상속 단점 : 서브클래스에서 코드 중복 발생 가능

: 실행시 특징(행동, 실제 인스턴스의 타입)을 바꾸기 힘들다.

: 코드변경시 다른 클래스에 의도치 않은 영향을 준다.

구현이 아닌 인터페이스 맞춰서 프로그래밍한다.

인터페이스에 맞추게 되면 하나의 메소드만 수정하면 된다.

서로 상호작용하는 객체사이에서는 가능하면 느슨하게 결합하는 디자인을 사용해야 한다.

두 객체가 느슨한 결합상태라는 서로 상호작용을 하지만 서로에 대해 잘 모른다는 것을 의미. 변경 사항이 생겨도 유연하게 객체지향 시스템을 구축할 수 있다. 객체 사이의 상호의존성 최소화 한다.

클래스는 확장에 대해서 열려 있어야 하지만 코드 변경에는 닫혀 있어야 한다. OCP

요구사항이나 조건이 바뀌면 클래스를 확장하고 행동을 마음대로 추가 할 수 있어야 한다.

코드 변경 및 수정에는 닫혀 있어야 한다.

추상화된 것에 의존하도록 만들어라. 구상 클래스에 의존하지 않도록 한다. DIP

구상클래스처럼 구체적인 것이 아닌 추상 클래스나 인터페이스와 같이 추상적인 것에 의존하는 코드를 만들어야 한다. 이 원칙은 고수준 모듈과 저 수준 모듈에 모두 적용

정말 친한 친구하고만 이야기하라(최소 지식원칙) = Law of Demeter

1. 객체자체
2. 메서드에서 매개변수로 전달된 객체
3. 메서드에서 생성하거나 인스턴스를 만든 객체
4. 그 객체에 속하는 구성 요소

먼저 연락하지 마세요. 저희가 연락 드리겠습니다. (할리우드 원칙)

의존성 부패(Dependency rot)을 방지 할 수 있다.

의존성 뒤집기 원칙은 될 수 있으면 구상 클래스 사용을 줄이고 대신 추상화된 것을 사용해야 한다는 원칙이다.

헐리우드 원칙은 저수준 구성요소가 고수준 계층사이에 의존성을 만들어내지 않도록 프레임워크 또는 구성요소 구축하기 위한 기법을 제공

클래스를 바꾸는 이유는 한 가지 뿐이어야 한다. SRP

디자인패턴의 분류

기능적 분류

생성 관련 패턴 (Creational Pattern)

객체 인스턴스 생성을 위한 패턴으로, 클라이언트와 그 클라이언트에서 생성해야 할 객체 인스턴스 사이의 연결을 끊어주는 패턴

ex) 싱글턴, 팩토리 메소드, 추상 팩토리, 프로토타입, 빌더 패턴

행동 관련 패턴 (Behavioral Pattern)

클래스와 객체들이 상호작용하는 방법 및 역할을 분담하는 방법과 관련된 패턴

ex) 스트래티지, 옵저버, 스테이트, 커맨드, 이터레이터, 템플릿 메소드, 미디에이터, 비지터

구조 관련 패턴 (Structural Pattern)

클래스 및 객체들을 구성을 통해서 더 큰 구조로 만들 수 있게 해 주는 것과 관련된 패턴

ex) 데코레이터, 어댑터, 컴포지트, 퍼사드, 프록시, 브리지

클래스와 객체 단위로 패턴 분류

클래스 패턴 (Class Pattern)

클래스 사이의 관계가 상속을 통해서 어떤 식으로 정의되는지를 다룬다. 클래스 패턴은 컴파일시에 관계가 결정.

템플릿 메소드, 팩토리 메소드, 어댑터,

객체 패턴 (Object Patterns)

객체 사이의 관계를 다루며, 객체 사이의 관계는 보통 구성을 통해서 정의된다. 객체 패턴에서는 일반적으로 실행 중에 관계가 생성되기 때문에 더 동적이고 유연

스트래티지, 옵저버 ,데코레이터, 프록시, 컴포짓&이터레이터, 스테이트, 추상팩토리,브리짓,미디에이터,빌더

디자인 패턴의 종류

- 스트래티지 패턴 (strategy pattern)
교환 가능한 행동을 캡슐화하고 위임을 통해서 어떤 행동을 사용할지 결정
- 옵저버 패턴 (observer pattern)
상태가 변경되면 다른 객체들한테 연락을 돌릴 수 있게 한다.
- 데코레이터 패턴 (decorator pattern)
객체를 감싸서 새로운 행동을 제공
- 팩토리 패턴 (factory pattern)
생성할 구상 클래스를 서브클래스에서 결정
- 추상 팩토리 패턴 (AbstractFactory pattern)
클라이언트에서 구상 클래스를 지정하지 않으면서도 일군의 객체를 생성 가능하게 함
- 싱글턴 패턴 (singleton pattern)
딱 한 객체만 생성하도록 함
- 커맨드 패턴 (command pattern)
요청을 객체로 감싼다.
- 어댑터 패턴 (adaptor pattern)
객체를 감싸서 다른 인터페이스를 제공
- 퍼사드 패턴 (facade pattern)
일련의 클래스에 대해서 간단한 인터페이스를 제공
- 템플릿 메소드 패턴 (template method pattern)
알고리즘의 개별 단계를 구현하는 방법을 서브클래스에서 결정
- 이터레이터 패턴 (iterator pattern)
컬렉션 구현을 숨기면서 컬렉션 내에 있는 객체에 대해 반복 작업 처리
- 컴포지트 패턴 (composite pattern)
클라이언트에서 객체 컬렉션과 개발 객체를 똑같이 다룰 수 있도록 함
- 스테이트 패턴 (state pattern)
알고리즘의 개별 단계를 구현하는 방법을 서브클래스에서 결정
- 프록시 패턴 (proxy pattern) 쁘락치 패턴
객체를 감싸서 그 객체에 대한 접근을 제어
- 컴파운드 패턴 (compound pattern)
반복적으로 생길 수 있는 일반적인 문제를 해결하기 위한 용도, 2개 이상의 패턴을 결합 사용
- 브리지 패턴 (bridge pattern)
구현 뿐만 아니라 추상화(개념적 부분)된 부분까지 변경시켜야 하는 경우
- 미디에이터 패턴 (mediator pattern)
서로 관련된 객체 사이의 복잡한 통신과 제어를 한 곳으로 집중시키고자 할 때
- 비지터 패턴 (visitor pattern)
다양한 객체에 새로운 기능을 추가해야 하는데 캡슐화가 별로 중요하지 않은 경우