객체지향개발론및실습

Laboratory 4. Factory Method 패턴

1. 목적

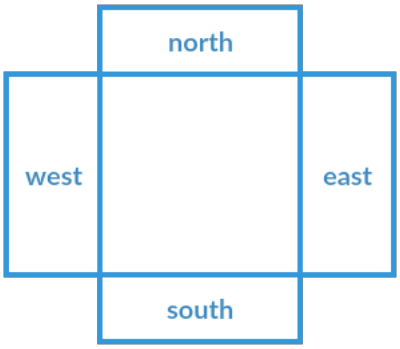
- 구체적인 객체의 생성을 하위 클래스로 위임하는 **팩토리 메소드** 패턴을 실습하여 본다.

2. 팩토리 메소드 패턴 실습

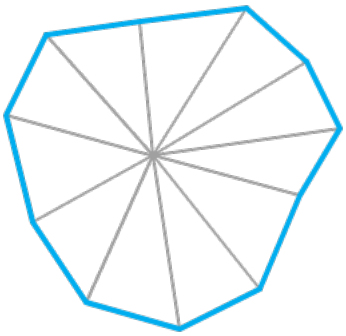
- 고전 게임 중에 1979년에 출시된 슈팅 게임인 Asteroid 게임이 있다. A 문자로 주인공 우주선을 나타내고, 적 우주선과 다양한 크기의 소행성을 피하거나 파괴하는 게임이다.
- 이 게임이 궁금하면 `http://www.freeasteroids.org`를 방문하면 이 게임을 실제 할 수 있다.
- 이 게임은 레벨이 올라감에 따라 큰 소행성이 등장하는 비율이 증가하며, 소행성의 움직이는 속도도 증가한다.
- 이 게임에서 소행성들이 지속적으로 랜덤한 위치에서 등장하며, 화면을 다양한 방향으로 가로질러 움직인다.
- 이 실습에서는 소행성을 생성하는 부분을 팩토리 메소드 패턴을 이용하여 구현해 본다.
- 팩토리 메소드 패턴은 상속을 이용하여 하위 클래스에게 생성을 위임하는 방식을 사용한다. 이 실습에서는 궁극적으로는 팩토리 메소드 패턴을 사용하지만 어떻게 구현하는 것이 효과적인지 다양한 방법을 검토해 본다.
- JavaFX와 PyQt6, Qt6를 이용하여 Asteroid 게임을 부분적으로 구현한다.
- 소행성 생성할 때 필요한 정보
  - 소행성의 크기 (3종류만 존재: 대, 중, 소)
  - 소행성의 이동 속도
- 이 정보는 현재 레벨에 따라 바뀐다.
- 기본적으로 **`Asteroid`**라는 클래스를 정의한다. 소행성은 다각형으로 표현되기 때문에 JavaFX에서는 **`Polygon`**을 상속받아 구현하고, C++에서는 **`QObject`**와 **`QGraphicsPolygonItem`**을 다중 상속하여 구현한다. 파이썬은 **`QObject`**만 상속하고 **`QGraphicsPolygonItem`**은 포함 관계로 모델링한다.
- 생성하는 부분이 복잡하기 때문에 추상화가 필요하다. 어떻게 설계하는 것이 가장 효과적일까? 몇 가지 설계 방법을 생각하여 보자.
  - 방법 1. **`Asteroid`** 클래스에 레벨 정보를 받는 생성자를 정의하고 해당 생성자에 모든 생성 로직을 추가한다.
  - 방법 2. 별도 클래스에 생성 로직을 구현한다.
  - 방법 3. 방법 2와 유사하지만 생성 로직을 가지고 있는 하나의 클래스를 만드는 대신에 각 레벨에 해당하는 클래스를 정의하고 각 레벨마다 생성할 때 사용하는 객체를 바꾼다.

  방법 1은 SRP 측면에서도 바람직하지 않으며, 유연성 측면에서도 좋은 방법이 아니다. 방법 3은 레벨이 많을 경우에는 클래스가 너무 많아지는 문제점이 있다.
- 방법 2를 더 효과적으로 설계 및 구현하기 위해 생성 과정에서 해야 하는 일을 단계별로 생각하여 보자.

- 단계 1. 주어진 레벨을 이용하여 크기를 결정한다. 앞서 언급한 바와 같이 이 게임에서 소행성의 크기는 3종류 밖에 없다. 하지만 각 종류마다 크기를 고정하지 않고 랜덤한 요소를 추가한다. 실제 최종 크기는 반지름 개념을 이용한다. 대, 중, 소의 기본 반지름은 10, 40, 60이다. 여기에 최대 10 정도의 랜덤 편차를 가질 수 있도록 한다. 레벨이 높을수록 큰 크기의 소행성이 등장할 확률이 높아야 한다.
- 단계 2. 주어진 레벨을 이용하여 속도를 결정한다. 속도는 화면을 가로지는 시간을 millisecond로 표현한다. 속도는 2000, 2500, 3000, 3500 4종류가 있으며, 여기에 200 정도의 랜덤 편차를 가질 수 있도록 한다. 레벨이 높을수록 빠른 속도의 행성이 등장할 확률이 높아야 한다.
- 단계 3. 움직임을 위한 시작 위치와 끝 위치를 결정한다. 소행성의 위치는 아래 그림에서 동서남북이 표시된 위치이다. 동서남북이 표시된 위치는 화면에서 보여지는 부분이 아니다. 시작 위치가 북이면 끝 위치는 남이 되고, 동이면 서가 되는 형태로 구현한다.



- 단계 4. 소행성의 각 꼭지점 좌표를 계산한다. 아래 그림과 같이 11개 꼭지점을 사용하기로 결정한 경우 원을 생각하고 중심에서 반지름 정도의 거리에 좌표를 계산한다.



- 위 단계 중 단계 4를 제외하고는 **AsteroidFactory**에 기본적으로 사용할 수 있는 메소드를 구현하고, 단계 4는 하위 클래스에게 위임하는 방법으로 구현하여 보자. **AsteroidFactory** 클래스의 기본 골격은 다음과 같다.

```
1  public abstract class AsteroidFactory{
2      private Direction startDirection;
3      protected int getSpeed(int level){}
4      protected AsteroidSize getRandomSize(int level){}
5      protected int getRadius(int level){}
6      protected Location getStartLocation(int radius){}
7      protected Location getDestLocation(int radius){}
8      public final Asteroid createAsteroid(int level) {
9          AsteroidSize size = getRandomSize(level);
10         int radius = getRadius(level);
11         Location startLoc = getStartLocation(radius);
12         Location destLoc = getDestLocation(radius);
13         Double[] points = createPoints(startLoc, radius);
14         return new Asteroid(size, level, startDirection,
15             startLoc, destLoc, getSpeed(level), points);
16     }
17     protected abstract Double[] createPoints(
18         Location centerLoc, int radius);
19 }
```

- 이 클래스를 상속하는 구체적인 생산자 클래스를 정의해야 하며, 이 클래스는 **createPoints**는 반드시 구현해야 한다.
- 자바에서 **Polygon** 클래스는 꼭지점을 나타내는 x 좌표와 y 좌표를 묶어 하나의 타입으로 처리하지 않고, 개별적으로 유지한다. 따라서 사각형이면 **points** 배열의 용량이 8이다. 반면에 파이썬은 **QPointF** 타입을 이용하여 각 꼭지점을 나타내고 유지한다. 파이썬는. 내부적으로 **QPointF**를 계속 이용하기 때문에 자바와 달리 별도. **Location**과 같은 클래스를 정의하여 사용하지 않는다. C++도 **QPointF**를 이용한다.
- 나머지 **getSpeed**, **getRadius**, **getStartLocation**, **getDestLocation**은 선택적으로 재정의할 수 있다.
- **Asteroid** 클래스, **Location**, **AsteroidFactory**, **AsteroidGame** 클래스 등 대부분의 클래스는 el를 통해 주어진다.
- 이 실습에서는 **AsteroidFactory** 클래스를 상속받은 구체적인 생산자 클래스인 **AsteroidRectangleFactory**와 **AsteroidPolygonFactory**를 정의해야 한다.
- **AsteroidDiamondFactory**는 주어져 있기 때문에 **AsteroidRectangleFactory**은 비교적 쉽게 정의할 수 있을 것이다. **AsteroidPolygonFactory**는 10개 이상의 꼭지점을 생성해야 한다. 기본 10개에 랜덤으로 5개 이내의 점을 추가하는 형태로 구현한다.
- **AsteroidsGame** 클래스에 팩토리를 변경하는 코드가 있다.
- 게임이 진행되면 우주선에서 미사일을 발사하여 소행성을 파괴할 수 있다. 대형 크기의 소행성이 미사일을 맞으면 두 개의 중형 크기로, 중형 크기의 소행성이 미사일을 맞으면 두 개의 소형 크기로 쪼개어진다. 소형 크기의 소행성은 맞으면 파괴되어 사라진다. 기존 행성을 두 개의 작은 크기의 행성으로 쪼개는 부분을 구현하기 위해 **createAsteroid** 메소드를 다음과 같이 다중 정의한다.

```
1 Asteroid createAsteroid(Asteroid asteroid,
2     Location startLoc, Location destLoc);
```

  이 메소드는 미사일 맞은 소행성을 인자로 받으며, **startLoc**은 미사일을 맞은 소행성의 현재 위치이며, **destLoc**은 미사일을 맞았을 때 새롭게 두 개의 목적 위치를 계산하여 전달한다. 따라서 소행성이 이 미사일을 맞으면 위 메소드를 두 번 호출하여 현재보다 작은 두 개의 소행성을 새롭게 생성한다.

2.1 게임 동작 설명

- 게임은 여섯 개의 키를 이용하여 조작한다.
  - 왼쪽, 오른쪽 화살표: 우주선이 각각 시계, 반시계 방향으로 회전한다.
  - 위 화살표: 우주선이 움직인다.
  - 아래 화살표: 우주선이 멈춘다.
  - shift 키: 임의의 위치로 이동한다.
  - 스페이스 키: 미사일을 발사한다.

2.2 PyQt6 주의사항

- 파이썬에서 객체를 생성하면 동적 생성이지만 지역 변수로 생성하면 그것의 수명이 함수가 종료되면 끝날 수 있다. 따라서 함수 수명보다 길어야 하는 것은 다르게 처리해 주어야 한다. 예를 들어 이 게임에서는 지속적으로 소행성 객체를 생성하는데, 특정 함수에서 그것을 생성하면 그것의 수명이 애니메이션을 통한 움직임이 완료되기 전에 끝날 수 있어 이상하게 동작할 수 있다. 이 때문에 멤버 변수로 소행성 목록을 만들고 소행성 목록에 생성한 소행성을 추가하는 형태로 프로그래밍하고 있다.
- 파이썬에서 리스트에 대해 반복하는 중간에 리스트를 수정하면 문제가 발생할 수 있다. 따라서 반복하는 과정에 수정이 필요하면 리스트를 복제한 후에 반복을 진행하고 수정은 원래 리스트에 대해 수정하는 형태로 프로그래밍해야 한다.