

발사체 궤적에 대한 연산시간과 슈팅로봇의 다수 객체 생성에 대한 메모리 사용량 개선을 위한 소프트웨어 설계 기법에 대한 연구

A Study on Software Design Approach for improving Trajectory Computation Time
and Memory Usage for Generating A Large Number of Shooting-Robots's Objects

Prohibition of Unauthorized Use : Chung HWG 2001

I. 서론(연구목적/연구필요성, 현황파악)

1. 연구배경 : 소개

요즘 청소년들에게 게임의 위상은 부정적으로 생각되었던 예전과 달리 학업 스트레스를 풀거나 여가 시간을 보내기 좋은 수단 중 하나로 부상하고 있다. 미래에 사람들에게 도움이 되는 로봇과 소프트웨어를 만들고 싶은 연구자 본인도 공부에 지쳐 복잡해진 마음을 치유하기 위해 게임 자체를 즐기기도 하지만, 개인적으로 직접 프로그래밍을 통해 원하는 게임이나 검증 소프트웨어를 만드는 과정 또한 즐기기도 한다.

얼마 전 슈팅 게임을 즐기던 중, 게임 속 로봇이 발사한 다수의 발사체가 특정 개수를 넘어서면 전체 게임 동작이 심각하게 느려지거나 심지어 몇몇 기능들은 정상적으로 동작하지 않기도 하는 상황이 발생하고는 하였다. 이에 해당 성능 저하 문제점을 인식하고 소프트웨어 관점에서 관련된 분석을 시도해 보고자 한다.

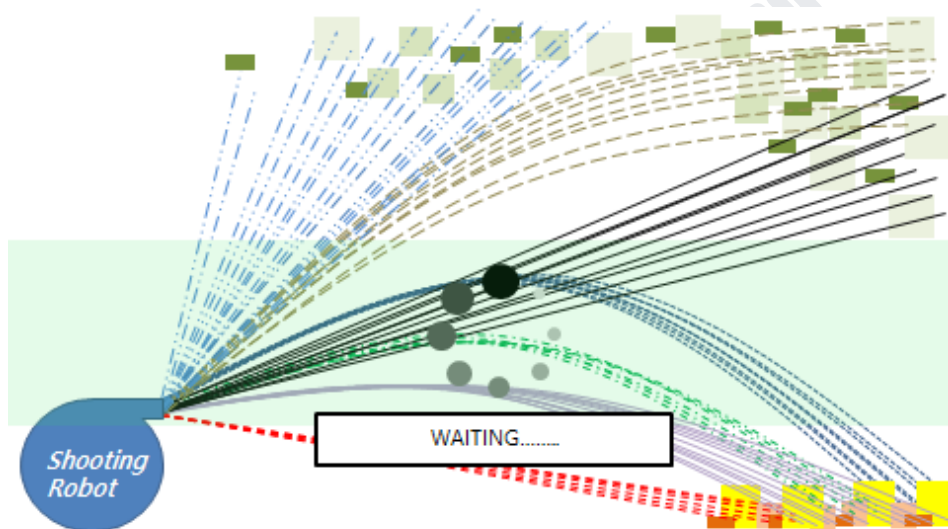


그림1. 슈팅로봇의 다수 발사체로 인한 속도/성능 저하 문제

2. 연구목적(의의/공헌점) : 품질 문제 정의 - 효율성/성능 - 발사체 궤적 연산시간, 메모리 사용량

국제표준에서 정의하는 여러 소프트웨어 품질 특성[] 중에서 본 연구 주제인 성능 저하로 생기는 문제는 소프트웨어 효율성과 연관이 있다. 소프트웨어 응용 분야에서 소프트웨어 효율성, 즉 소프트웨어 성능에 영향을 미치는 요인에 대한 여러 연구가 있었다[1, 2, 3]. 일반적으로 소프트웨어의 효율성 품질에 가장 영향을 크게 미치는 주요 요인은 연산 또는 메모리 자원이다. 또한, 현재 컴퓨터 하드웨어 사양이 발전하면서 고성능의 기기에서 동작하는 게임 플랫폼 또한 복잡해지는 경향을 보이고 있다. 게임 플랫폼 환경의 고급화와 컴퓨터 하드웨어 사양의 고급화는 서로 영향을 주어 더욱 고사양을 원하는 소프트웨어들이 계속 나오고 있다. 하지만 소프트웨어

자체는 스마트폰과 같은 모바일 기기나 컴퓨터 내 자원들에 대한 사용 제약이 존재하기 때문에 연산이나 메모리의 영향이 클 수 밖에 없다. 즉, 대기시간이나 반응속도에 대한 품질을 보장할 수 있는 효율적인 연산 기법과 메모리 관리 방안이 필요하다.

앞서 슈팅 게임에서 문제에 대한 가설을 설정해 보면 다음과 같은 고려 사항을 가진다.

첫째, 발사체 개체들이 생성되면서 메모리를 차지한다. 즉, 다수의 발사체 개체의 생성에 대한 메모리 사용량이 로봇 동작에 영향을 줄 수 있다.

둘째, 각 발사체 개체들이 궤적에 대한 연산시간을 필요로 한다. 즉, 발사체 개체의 궤적에 대한 연산시간이 로봇 동작에 영향을 줄 수 있다.

결과적으로 다수 발사체 개체들에 할당된 메모리와 각 발사체 궤적 연산시간을 확인해보면 문제 분석과 검증이 가능하리라 판단된다. 즉, 게임 내에 다양한 종류의 수많은 개체들이 존재하므로 그 개체들을 표현하기 위한 효율성, 특히 연산시간이나 메모리에 대한 효율성이 주 연구 대상이다. 따라서, 본 연구에서는 게임 플랫폼에 독립적인 소프트웨어 관점에서 발사체 궤적 연산시간, 발사체 객체 생성 메모리 사용량을 확인하고 가능하다면 개선 방안을 찾아 보다 나은 소프트웨어를 설계/구현해보고자 한다.

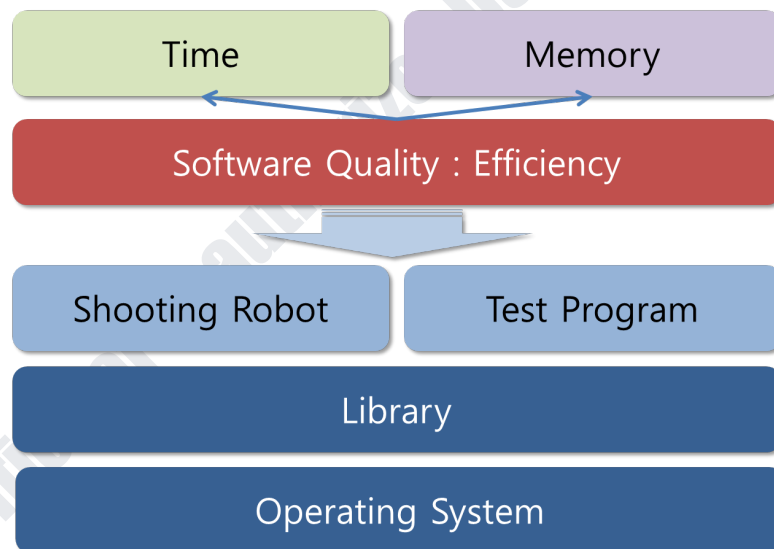


그림2. 효율성 품질을 고려하는 소프트웨어 관점 대상 환경

그림2는 이 연구의 대상 환경을 나타낸다. 소프트웨어 설계 관점의 접근을 위해 슈팅 로봇은 다수의 발사체를 생성할 수 있고, 생성된 발사체들은 여러 조건들에 따라 발사체의 궤적 연산 과정을 거쳐 발사된다. 본 연구의 검증을 위해 GUI는 구현하지 않을 것이고 발사체 과 발사장치를 가진 슈팅로봇, 다른 속성을 가질 수 있는 발사체, 발사체를 발사하기 위한 발사장치(예. 총), 연산함수와 테스트를 위한 시간, 메모리 측정함수 등을 설계/구현하고자 한다.

3. 연구구성 : 관련연구-분석/개선전략-설계/구현/테스트/실험-평가-결론/향후

본 연구의 구성은 다음과 같다. 먼저 2장에서는 관련 연구를 기술하고, 3장에서는 객체 지향 프로그래밍 설계 관점에서 일반적인 개체 생성과 연산 방법을 분석하여 연산이나 메모리에 대한 효율성을 고려한 다수의 발사체 개체 생성 전략을 도출하고, 4장에서는 실제 개선을 검증할 발사체 개체, 궤적 연산 함수와 메모리 사용 함수를 설계/구현하고, 5장에서는 일반적인 설계와 개선 설계의 연산시간과 메모리 사용량의 변화를 측정하여 비교/분석한 후 개선 효율성을 검증/평가해보고자 한다. 6장에서는 결론 및 향후 개선과제를 제안한다. 또한, 연구 관련 용어집 정리와 관련 자료 리서치는 연구 중에 지속적으로 수행하여 추후 본 연구를 통한 추가 연구의 기반을 마련하고자 한다.

II. 관련연구(이론적배경/선행연구 : 연구가설과 관련된 정보) - 발사체 궤적 논문, 객체 메모리 사용 논문

이 장에서는 먼저 소프트웨어 품질에 대한 연구를 통해 소프트웨어 효율성의 조건과 특성을 확인하고, 발사체 궤적 연산과 메모리 관련 선행 연구를 분석하여 본 연구에서 활용 가능한 접근 방법 또는 전략 수립을 위한 기반 지식을 쌓고, 실제 설계/구현을 위해 객체지향설계와 디자인 패턴에 대한 자료들을 활용하여 본 연구에 적용한다.

1. 소프트웨어 품질

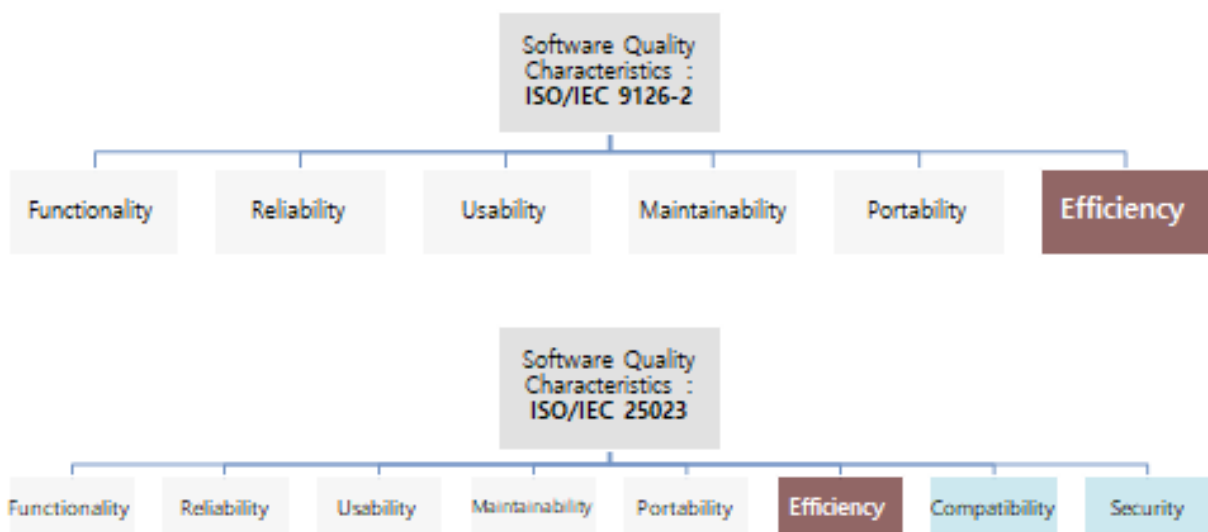


그림3. ISO/IEC 9126-2, ISO/IEC 25023 기반 품질 특성 모델

Efficiency	Subcharacteristic	
	Time Behaviour	Response Time
		Throughput
		Turnaround Time
		...
	Resource Utilization	I/O Device Utilization
		Memory Utilization
		CPU Utilization
		...
	Capacity	No. of online requests
		...

표1. ISO/IEC 표준 기반 효율성 테스트

그림은 국제표준에서 정의하는 소프트웨어 품질 평가 모델이다. 국제표준(ISO/IEC 9126-2와 ISO/IEC 25023)에서는 소프트웨어 품질 특성으로 보안성과 상호운용성을 포함하는 기능성, 신뢰성, 사용성, 유지보수성, 이식성, 효율성을 정의한다. 그 중 본 연구와 연관있는 소프트웨어 효율성 품질은 시간과 자원(메모리, CPU, I/O등) 사용이 주요 평가 항목이다.

또한, 게임 소프트웨어의 품질 중 사용자 요구사항이 가장 많은 부분은 규정된 조건에서 사용되는 자원의 양에 따라 요구된 성능을 제공하는 소프트웨어의 능력인 효율성[1]이다.

본 연구에서는 효율성에 영향을 주는 주요인인 궤적 연산 시간과 메모리 사용 기법들에 대해 구체적인 분석을 통해 실제 소프트웨어로 구현하여 검증해보고자 한다.

2. 발사체 궤적 연산

발사체켓의 궤적은 미적분에 의해서 계산할 수 있으며[2], 세계적으로 유명한 게임인 '앵그리버드'에서 적용된 미적분과 간단한 탄도학에 대한 지식[3]은 본 연구와 병행했던 '미적분의 원리가 적용되어 속도, 방향 등을 제어할 수 있는 탄도를 구현할 수 있을까?' 라는 다른 연구[4]에서 활용할 수 있었다.

해당 연구[4]에서는 포물선(발사체의 궤적) 운동을 수직운동과 수평운동으로 구분하고, 각 운동들에 대해 미분, 적분, 삼각함수 등 수학적 개념을 활용하여 발사체의 궤적에 대한 x 좌표, y 좌표의 위치함수를 유도하였다. 또한, 위치함수를 유도하기 위하여 물체의 예상 도달거리, 예상 도달시각, 발사각을 계산하였고, 발사되는 발사체의 궤적 그래프를 엔트리 프로그래밍으로 표현하였다.

자세히 살펴보자면, 우선 발사체의 궤적 운동은 공기저항을 무시한 조건에서 수평운동은 등속운동이라고 가정, 수직운동은 등가속운동이며 중력의 영향을 받는다고 가정하였다. 발사체의 속도를 시간에 대하여 적분하면 발사체의 위치를 나타내는 x, y 좌표값이 나오는데 발사체를 발사한 당시의 초기속도를 고정된 값으로 사용하여 시간의 흐름에 따라 x, y 값을 좌표평면에 표시하면 발사체의 궤적이 된다. 목표물은 정해져 있으므로 예상 도달거리는 항상 입력해야 한다는 가정을 하였다.

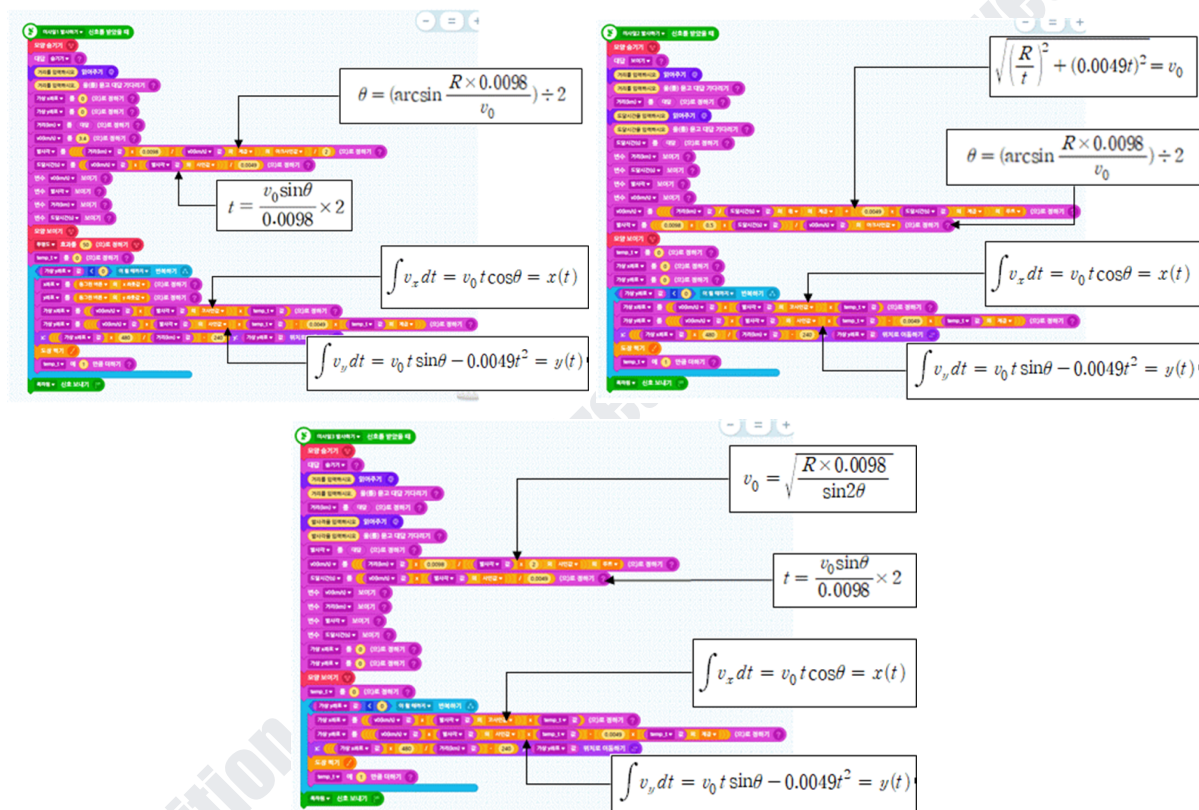


그림4. 발사체의 궤적 연산을 위한 계산 방법들

관련 연구[4]의 발사체 궤적 연산을 위한 계산 방법들을 활용하여 본 연구의 설계와 구현에 사용되는 계산식들을 도출하여 최적의 연산시간을 가지는 조건을 판단하고자 한다.

3. 메모리 사용

자원에 대한 사용 제약이 존재하는 환경에서 게임 소프트웨어의 질을 높일 수 있는 효과적인 메모리 관리 및 활용 기법이 필요하다. 이를 통해 메모리와 밀접한 관계에 있는 게임 성능 역시 효과적으로 높일 수 있다.[5]

본 연구에서는 소프트웨어 설계 시 자원, 즉 메모리 효율성을 고려하고자 정적 메모리 관리 기법[6]에 기반한 설계를 진행한다.

4. 객체지향설계와 디자인 패턴

소프트웨어 생산성과 품질을 높이기 위해 게임을 만들 때 객체지향개념을 충분히 활용할 수 있는 설계기법이 유용하다. 그림 5는 소프트웨어 개발 프로세스 내에서 디자인 패턴 활용 방안을 나타낸다.[9]

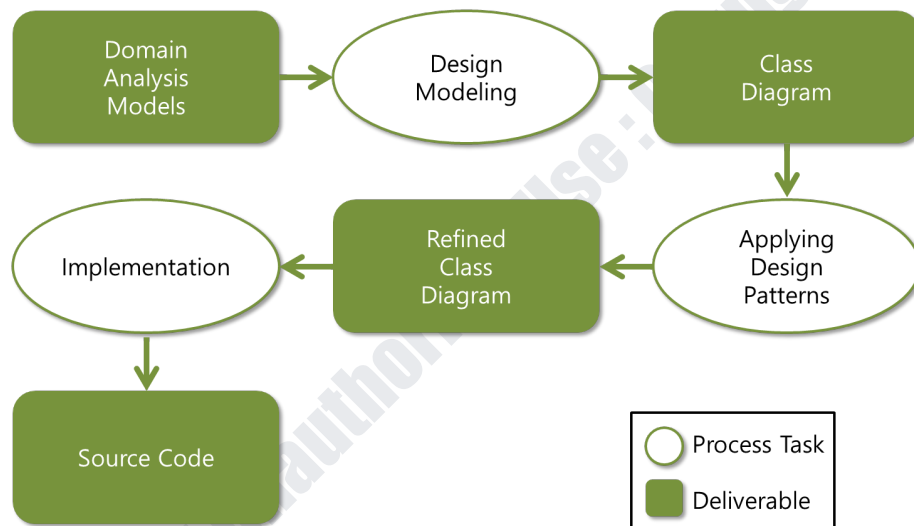


그림5. 소프트웨어 개발 프로세스 내 디자인 패턴 활용

본 연구에서는 객체 지향 디자인 패턴 중 발사체 개체의 생성절차를 추상화하기 위한 생성패턴과 더 큰 구조로 확장하는 구조패턴에 대해서 분석하여 활용하고자 한다. 일반적으로 다음과 같은 5가지 생성패턴과 7가지 구조패턴이 존재한다.[7]

생성패턴:

- (1) 추상 팩토리 : 동일한 분류의 다른 팩토리들을 묶어줌
- (2) 빌더 : 생성과 표현을 분리하여 복잡한 객체를 생성
- (3) 팩토리 메서드 : 생성할 객체의 클래스를 특정하지 않고 객체 생성 가능
- (4) 프로토타입 : 기존 객체를 복제함으로써 객체 생성
- (5) 싱글턴 : 하나의 클래스에 하나의 객체 인스턴스만 존재

구조패턴:

- (1) 어댑터(Adapter) : 클래스 인터페이스를 다른 인터페이스로 분리하여 다른 클래스가 이용 가능
- (2) 브리지(Bridge) : 구현과 추상을 분리하여 각각 독립적으로 확장가능
- (3) 컴포지트(Composite) : 트리 형태로 객체들 관계를 구성하여 단일객체와 복합 객체 모두 활용 가능
- (4) 데코레이터(Decorator) : 한 객체에 다른 객체를 특정 상황이나 용도에 맞게 추가 가능
- (5) 퍼싸드(Facade) : 하위 인터페이스들을 통합하여 한 Wrapper 인터페이스 제공 가능
- (6) 플라이웨이트(Flyweight) : 작은 크기의 다수 객체를 무작정 생성하지 않고 가능한 공유하여 메모리 절약 가능
- (7) 프록시(Proxy) : 객체의 Surrogate나 Placeholder 제공하여 객체로 접근 제어 가능

실제 상용 게임에서 다양한 디자인 패턴들을[8,9] 활용하여 객체지향 설계를 지향한다.

본 연구에서도 다양한 발사체 개체들 중 특정 개체가 생성 또는 변경되어도 프로그램 구조에 영향을 주지않도록, 실제 설계에 적용할 생성 기법을 도출하기 위해 소프트웨어 효율성 품질 특성을 고려한 디자인 패턴들을 이용한다.

III. 분석/전략(연구내용/가설설정/연구방법, 전략 설정) - 발사체 궤도 연산 기법과 발사체 개체 생성 방안 분석 및 도출

1. 기본설계

```
METHOD ShootingRobot::main()
```

```
BEGIN
```

```
Ammunition : Object that are shot from Weapon
```

```
AmmoFactory : Factory Object that generate Specific Ammunition
```

```
Gun : One Object of Ranged Weapons that load with Ammunition
```

```
numOfAmmo : Number of Ammunition
```

```
LOOP numOfAmmo
```

```
Generate a Ammunition from AmmoFactory
```

```
Set Specific Properties of the Ammunition
```

```
Load Gun with the Ammunition
```


Compute a ballistic trajectory of the Gun's Ammunition
Fire the Ammunition on the ballistic trajectory

END

그림6. 기본 알고리즘

그림6은 슈팅로봇의 기본 알고리즘을 나타낸다. 루프를 돌면서 발사체를 생성하여 특성을 설정한 후 무기에 장전하여 발사하는데, 이 때 발사체의 궤적 연산을 수행한다.

본 연구에서 사용하는 주요 용어에 대한 정의는 다음과 같다.

용어정의1.

발사체 (Ammunition) : 탄, 탄환, 포탄, 총알, 화살

용어정의2.

개체(Entity) : 실체/실생활

객체(Object) : 객체 프로그래밍

용어정의3.

탄도 : 발사체 궤적

2. 발사체 궤적 연산 기법 도출

슈팅 게임에서 슈팅로봇이 발사체를 발사하면 발사체의 경로를 계산할 필요가 있는데 이 때 복잡한 연산을 수행한다. 구체적으로 거리를 입력받아 각도, 도달시간, 발사체의 궤적 등을 계산하는데 미분과 적분 개념을 이용한 연산이 필요하고[3], 이는 전체 프로그램의 시간이나 속도에 대한 효율성에 영향을 미친다.

병행연구[4] 내용을 기반으로 본 연구에서는 발사체의 궤적을 연산하려면 발사각과 초기속도, 도달시간, 도달거리가 필요한데 4가지 조건 중 도달거리는 슈팅로봇이 알고있다고 가정한다. 따라서, 도달거리를 제외한 3가지 조건 중 2 이상의 조건이 주어진다면 3가지 발사체 궤적 좌표를 구하기 위한 계산식들을 (삼각함수를 이용하여) 도출 가능하다.

발사체의 운동을 수평운동과 수직운동으로 나누어 수평운동은 등속 운동

$v_x = v_0 \cos\theta$, 수직운동은 등가속운동이며 중력의 영향을 받는다고 보고

$v_y(t) = v_0 \sin\theta - 0.0098t$ 라고 하였다. 여기서 중력가속도 $g = 9.8m/s^2$ 이지만 탄두

속도의 단위는 $Mach(1 Mach = 0.34km/s)$ 이므로 $g = 0.0098km/s^2$ 이다.

본 연구에서는 3가지 발사체 궤적 좌표를 구하기 위한 계산식들의 연산량(연산시간)을 비교하여 가장 최적의 식을 선택하고자 한다.

탄도 궤적을 구하기 위해서

$$x(t) = v_0 \cos\theta t$$

$$y(t) = v_0 \sin \theta t - 0.0049 t^2$$

와 같이 t 에 관한 x, y 좌표의 함수(이하 발사체 궤적 좌표함수)를 얻을 수 있다.

이 때, 도달거리 R 이 고정되어 있다고 가정한다면 도달시각 T , 발사각 θ , 초기속도 v_0 는 계산식의 변수가 되어 다음과 같은 3가지 계산식이 도출된다.

첫번째 식은 도달거리 R 을 입력받아 예상 도달시각 T 와 발사각 θ 를 계산하여 x, y 좌표를 리턴한다. 여기서 초기속도 v_0 은 $10Mach(3.4km/s^2)$ 로 세팅하였다. 먼저 y 의 최댓값인 최고 고도 H 를 구할 필요가 있고, y 의 극값을 찾아보기 위해 y 의 미분계수를 0으로 만드는 t_H 과 최고 고도 H 를 계산하였다.

$$\frac{dy}{dt} = v_0 \sin \theta - 0.0098 t = 0$$

를 만족하는 t 는 최고 고도 H 를 갖게 하는 변수이므로 t_H 이라 하자.

$$t_H = \frac{v_0 \sin \theta}{0.0098}, \quad H = \frac{v_0^2 \sin^2 \theta}{0.0049}$$

일반적으로 포물선 운동에서 최고 고도까지 가는 시간은 포물선의 운동체의 전체 운동 시간의 절반이므로 운동체의 도달거리까지 걸리는 시간 T 는 최고 고도 H 까지 가는 시간인 t_H 의 2배라 하겠다.

$$\therefore T = 2t_H = \frac{v_0 \sin \theta}{0.0049}$$

또한, 삼각함수의 배각공식($\sin 2\theta = 2 \sin \theta \cos \theta$)을 사용하여 도달거리 R 은 다음과 같이 표현한다.

$$R = x(T) = \frac{v_0^2 \sin \theta \cos \theta}{0.0049} = \frac{v_0^2 \sin 2\theta}{0.0098}$$

한편 발사각 θ 는 위의 도달거리 R 을 구하는 식에 삼각함수의 역함수를 이용하여 다시 정리하였다.

$$\therefore \theta = \left(\arcsin \frac{R \times 0.0098}{v_0^2} \right) \div 2$$

이상 구해진 T, θ 를 발사체 궤적 좌표함수에 대입하여 시간 t 에 관한 x, y 좌표의 함수를 다음과 같이 얻을 수 있다.

$$x(t) = v_0 \cos \theta t = v_0 \cos \left\{ \left(\arcsin \frac{R \times 0.0098}{v_0^2} \right) \div 2 \right\} \times t$$

$$y(t) = v_0 \sin \theta t - 0.0049 t^2 = v_0 \sin \left\{ \left(\arcsin \frac{R \times 0.0098}{v_0^2} \right) \div 2 \right\} \times t - 0.0049 \times t^2$$

두번째 식은 도달거리 R 과 도달시각 T 를 입력받아 초기속도 v_0 와 발사각 θ 를 계산하여 x, y 좌표를 리턴한다. 발사각 θ 는 첫번째 식에서 유도된 결과를 사용한다.

$$\therefore \theta = \left(\arcsin \frac{R \times 0.0098}{v_0^2} \right) \div 2$$

도달시각 t 는 최고고도 H 까지 가는 시각의 2배이므로

$$T = \frac{v_0 \sin \theta}{0.0098} \times 2 = \frac{v_0 \sin \theta}{0.0049} \quad \therefore v_0 \sin \theta = 0.0049 \times T$$

이 식을 이용하여

$$R = \frac{v_0^2 \sin \theta \cos \theta}{0.0049} \quad \text{에 대입하여} \quad v_0 \cos \theta = \frac{R}{T} \quad \text{이다.}$$

$$\left(\frac{R}{T} \right)^2 + (0.0049 T)^2 = v_0^2 \quad (\because v_0^2 \sin^2 \theta + v_0^2 \cos^2 \theta = v_0^2)$$

$$\therefore v_0 = \sqrt{\left(\frac{R}{T} \right)^2 + (0.0049 T)^2}$$

이상 구해진 θ, v_0 를 발사체 궤적 좌표함수에 대입하여 시간 t 에 관한 x, y 좌표의 함수를 다음과 같이 얻을 수 있다.

$$x(t) = v_0 \cos \theta t = \sqrt{\left(\frac{R}{T} \right)^2 + (0.0049 T)^2} \times \cos \left\{ \left(\arcsin \frac{R \times 0.0098}{v_0^2} \right) \div 2 \right\} \times t$$

$$y(t) = v_0 \sin \theta t - 0.0049 t^2 = \sqrt{\left(\frac{R}{T}\right)^2 + (0.0049 T)^2} \times \sin \left\{ \left(\arcsin \frac{R \times 0.0098}{v_0^2} \right) \div 2 \right\} \times t - 0.0049 t^2$$

세번째 식은 도달거리 R 과 발사각 θ 를 입력받아 초기속도 v_0 와 예상 도달시각 T 를 계산하여 x, y 좌표를 리턴한다. 첫번째 식에서

$$R = \frac{v_0^2 \sin \theta \cos \theta}{0.0049} = \frac{v_0^2 \sin 2\theta}{0.0098} \quad \text{이므로}$$

$$\therefore v_0 = \sqrt{\frac{R \times 0.0098}{\sin 2\theta}}$$

도달시각 T 는 첫번째 식에서 유도된 결과를 사용한다.

$$T = \frac{v_0 \sin \theta}{0.0098} \times 2 = \frac{v_0 \sin \theta}{0.0049}$$

이상 구해진 v_0, T 를 발사체 궤적 좌표함수에 대입하여 시간 t 에 관한 x, y 좌표의 함수를 다음과 같이 얻을 수 있다.

$$x(t) = v_0 \cos \theta t = \sqrt{\frac{R \times 0.0098}{\sin 2\theta}} \times \cos \theta t$$

$$y(t) = v_0 \sin \theta t - 0.0049 t^2 = \sqrt{\frac{R \times 0.0098}{\sin 2\theta}} \times \sin \theta t - 0.0049 t^2$$

본 연구에서는 발사체 궤적 연산을 위하여 위에서 도출한 3가지 계산식을 활용한다.

3. 개체 생성 방안 분석 및 선정

소프트웨어 개발 시, 가장 기본적인 개체 생성 방식은 발사체 개체 갯수를 입력받아 해당 개수의 발사체 개체를 만드는 것이다. 다수의 발사체를 만들게 되면 메모리 사용량도 그에 비례하여 증가하게 된다.

기본적인 개체 생성방식의 문제 개선 전략을 마련하기 위해, 객체 지향 설계 관점에서 발사체 개체 생성의 특징을 살펴보면 다음과 같다.

- (1) 동일한 수많은 발사체 개체가 독립적으로 생성
- (2) 발사체는 발사장치에 따라 다른 특성을 가질 수 있음
- (3) 발사체 개체의 생성 제약은 해당 소프트웨어가 동작하는 시스템에 따라 달라짐

과 같다.

객체지향 설계방법 중 위의 특징들을 개선할 수 있도록 본 연구에서 도출한 전략은 다음과 같다.

- (1) 디자인 패턴을 활용하여 독립적으로 생성되도록 하자.
- (2) 발사체 개체는 생성 시 여러 발사체 형태가 가능하도록 팩토리 클래스가 필요하다.
- (3) 특정 상태값들을 가지지 않는 단순한 발사체 객체 표현을 위한 발사체 개체 생성이 가능하도록 Flyweight 패턴을 적용한다.(그림7)

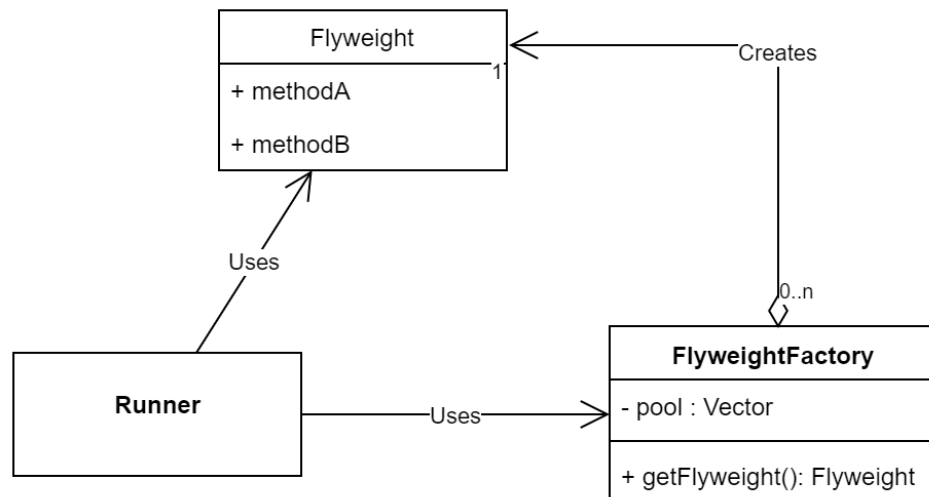


그림7. Flyweight 디자인 패턴

4. 개선 전략 적용 설계

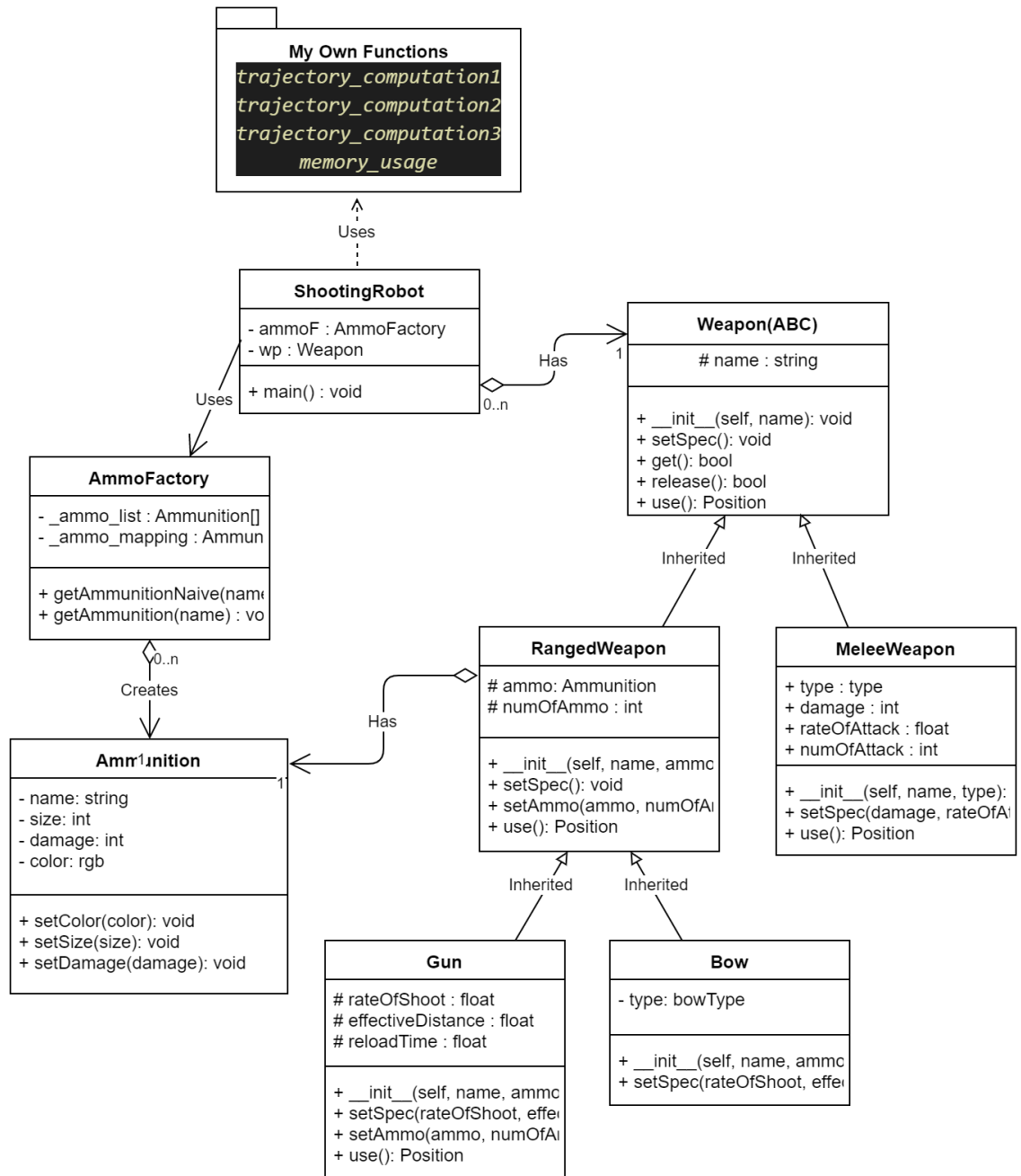


그림8. 설계한 슈팅로봇 관련 클래스 다이어그램

그림8은 본 연구에서 설계한 전체 프로그램에 대한 클래스 다이어그램이다.

슈팅로봇이 사용하는 무기의 최상위 추상 클래스는 **Weapon**이고, 원거리무기를 다루는 **RangedWeapon** 클래스와 근거리 무기를 다루는 **MeleeWeapon** 클래스를 자식으로 가진다. **RangedWeapon** 클래스는 본 연구에서 활용하는 총을 다루는 **Gun** 클래스와 활을 다루는 **Bow** 클래스를 자식으로 가진다.

본 연구에서 슈팅로봇이 활용하는 **Gun**에 장착하는 발사체를 다루는 **Ammunition** 클래스와 원하는 **Ammunition**을 만들어내는 **AmmoFactory** 클래스를 설계하였다. **Ammunition**을 생성 시, 디자인 패턴 중 **Flyweight** 패턴을 적용하였다.

또한, 궤도 연산을 위한 함수와 메모리 사용량을 측정하는 함수를 설계하여 본 연구의 검증을 위해 활용한다.

IV. 연구결과(분석결과/가설검정, 발사체 개체 설계) - 모의 테스트(실험)

1. 전체 시스템 구조

그림은 본 연구의 구현 환경과 프로그램 구조를 나타낸 것이다. 구현 환경은 Windows 10 환경에서 Visual Studio Code IDE를 활용하고 가장 대중적인 python을 프로그래밍 언어로 선택하였다. 실제 사용한 Python 버전은 v.3.11.2를 사용하였으며, 라이브러리는 삼각함수를 위해 `numpy`, 루트연산을 위해 `math`, 메모리 관련함수를 위해 `psutil`, 시간함수를 위해 `timeit` 등을 사용하였다.

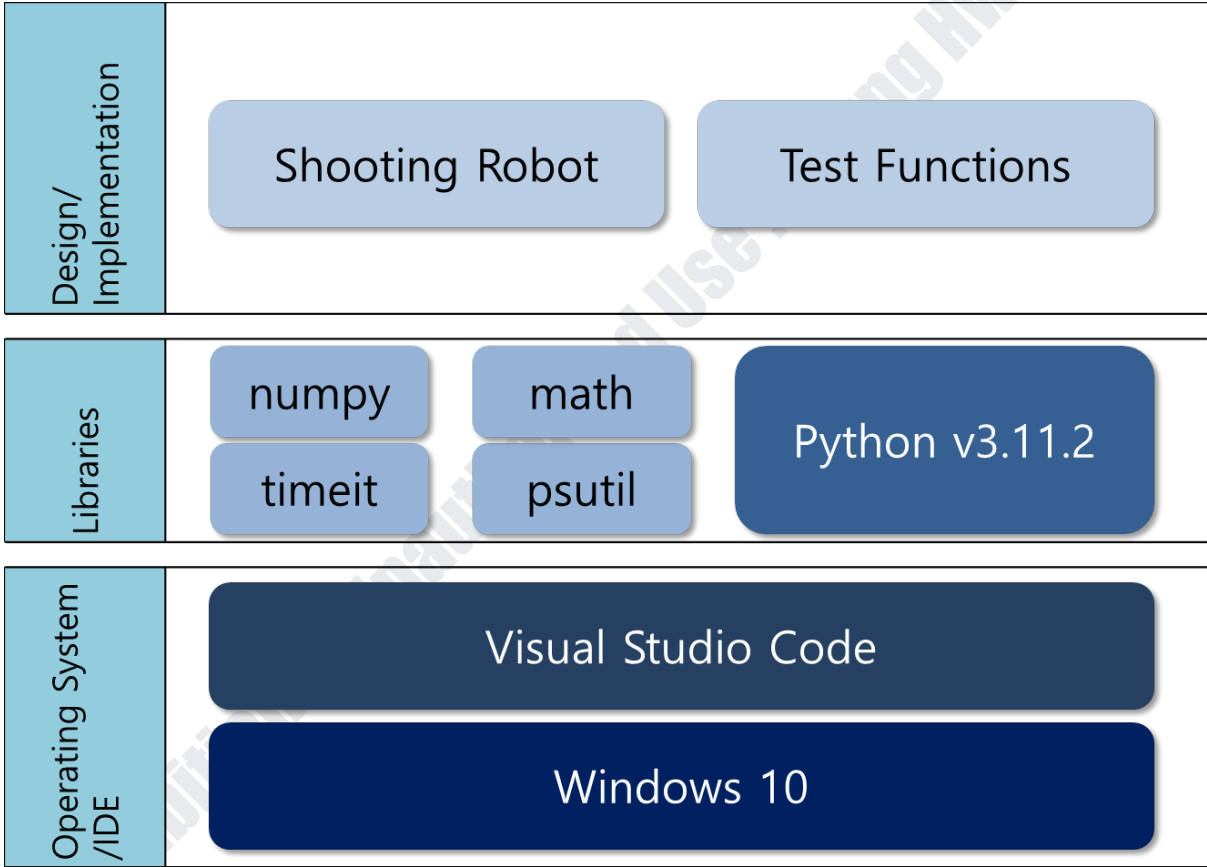


그림9. 대상 환경

2. 패턴 기반 발사체 생성 구현

일반적으로 발사체에 대한 객체는 필요할 때마다 필요한 수만큼 생성하여 사용한다. 본 연구에서 설계한 개체 생성 기법을 적용한 구현과 일반적인 개체 생성 방법을 구현에 대해 각 메모리의 사용량을 비교해본다. 실제 메모리 사용량이 적은 쪽이 소프트웨어 품질 특성 중 효율성이 더 높은 방법으로 볼 수 있다.

METHOD getAmmunitionNaively BEGIN Ammunition : Object that are shot from Weapon AmmoFactory : Factory that generate Specific Ammunition Ammunition List: List Data Structure for managing Ammunitions numOfAmmo : Number of Ammunition LOOP numOfAmmo Generate a Ammunition from AmmoFactory Add the Ammunition to Ammunition List END	METHOD getAmmunition BEGIN Ammunition : Object that are shot from Weapon AmmoFactory : Factory that generate Specific Ammunition Ammunition Map : Map Data Structure for managing Ammunitions numOfAmmo : Number of Ammunition LOOP numOfAmmo Get a Ammunition from Ammunition Map if ammunition is none Generate a Ammunition from AmmoFactory Add the Ammunition to Ammunition List END
---	--

그림10. 일반적인 개체 생성방법과 본 연구에서 설계한 생성기법 알고리즘

그림10은 일반적인 개체 생성방법(이하 개체 생성방법#1)과 본 연구에서 설계한 생성기법(이하 개체 생성방법#2)에 대한 기본 알고리즘을 나타낸다. 두 방식의 차이점은

3. 발사체 궤적 연산 기법 비교

앞서 도출한 발사체 궤적을 연산하기 위한 세 가지 계산식을 구현하여 실제 연산량을 비교해본다. GUI를 구현하지 않아서 실제 연산량에 가깝게 계산되지만, GUI를 구현시에는 그래픽에 대한 오버헤드가 더 커지게 된다.

세 가지 계산식을 간단히 요약하면 다음 표2와 같다.

[Trajectory Computation#1]	발사체의 예상 도달거리와 초기속도를 입력받아 시간에 관한 발사체의 x,y좌표를 계산하는 함수
[Trajectory Computation#2]	발사체의 예상 도달거리와 예상 도달시각을 입력받아 시간에 관한 발사체의 x,y좌표를 계산하는 함수
[Trajectory Computation#3]	발사체의 예상 도달거리와 발사각을 입력받아 시간에 관한 발사체의 x,y좌표를 계산하는 함수

표2. 본 연구에서 도출한 발사체 궤적 연산을 수행하는 계산식

```
def trajectory_computation1(R, v0) : # 궤도 연산 함수1 :
r(km), v0(km/s)
    theta = np.arcsin(R*0.0098/v0)/2
    T = v0*np.sin(theta)*2/0.0098
    for t in range(T):
        x = v0*np.cos(theta)*t
        y = v0*np.sin(theta)*t - 0.0049*t**2
    return x, y, T
```

```
def trajectory_computation2(R, T) : # 궤도 연산 함수2 :
r(km), t(s)
    v0 = math.sqrt((R/T)**2 + (0.0049*T)**2)
    theta = np.arcsin(R*0.0098/v0)/2
    for t in range(T):
        x = v0*np.cos(theta)*t
        y = v0*np.sin(theta)*t - 0.0049*t**2
    return x, y, T
```

```
def trajectory_computation3(R, theta) : # 궤도 연산 함수3:
r(km), theta(radian)
    v0 = math.sqrt(R*0.0098/np.sin(2*theta))
    T = v0*np.sin(theta)*2/0.0098
    for t in range(T):
        x = v0*np.cos(theta)*t
        y = v0*np.sin(theta)*t - 0.0049*t**2
    return x, y, T
```

위 코드들은 계산식들을 여러 라이브러리를 활용하여 Python 언어로 실제 구현한 함수들이다.

4. 실험 방법 - 개선전략 적용 구현

구현 실험을 위해 테스트 발사체 객체는 각 100개, 1000개, 10000개를 생성하여 테스트하였다. 본 연구를 위해 구현한 슈팅 로봇은 텍스트 기반으로 동작하고 발사체에 대한 설정은 이름, 색, 크기, 데미지, 개수 등을 프로그램 코드 내 **static** 형태로 저장하여 사용한다. 또한, 다양한 발사체를 생성하여 사용하기 위해서 슈팅로봇이 가진 발사장치는 **ComplexGun**이라 가정하였다. (아래 코드 참조)

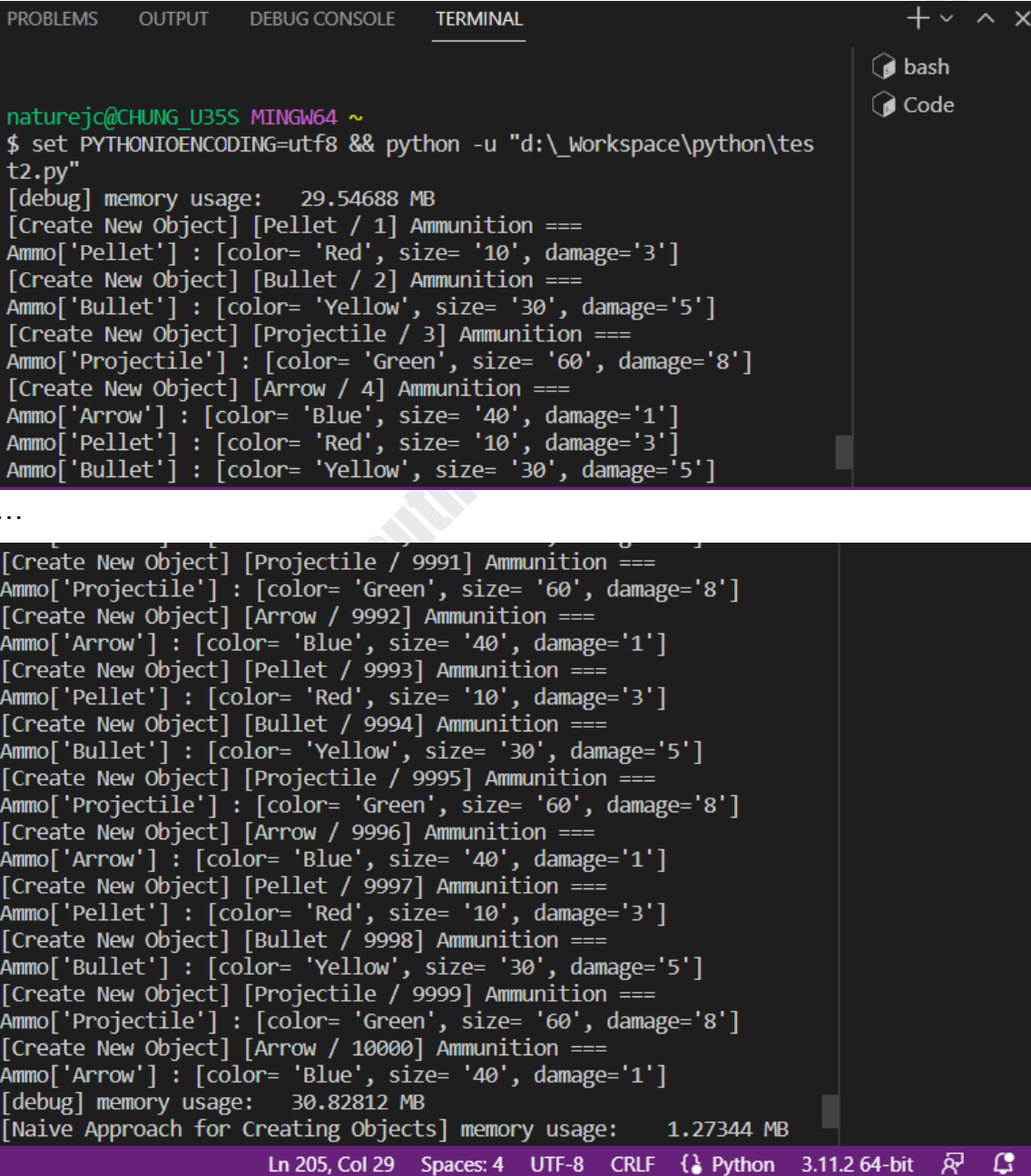
```
names = ["Pellet", "Bullet", "Projectile", "Arrow"]
colors = ["Red", "Yellow", "Green", "Blue"]
sizes = [10, 30, 60, 40]
damages = [3, 5, 8, 1]
nums = [9, 5, 3, 10]
```

```
gun = Gun("ComplexGun")
numOfShoot = NUM_OF_SHOOT # 1000 | 10000 | 100000
```

V. 결과 고찰 - 평가

1. 개체 생성 메모리 사용량 비교

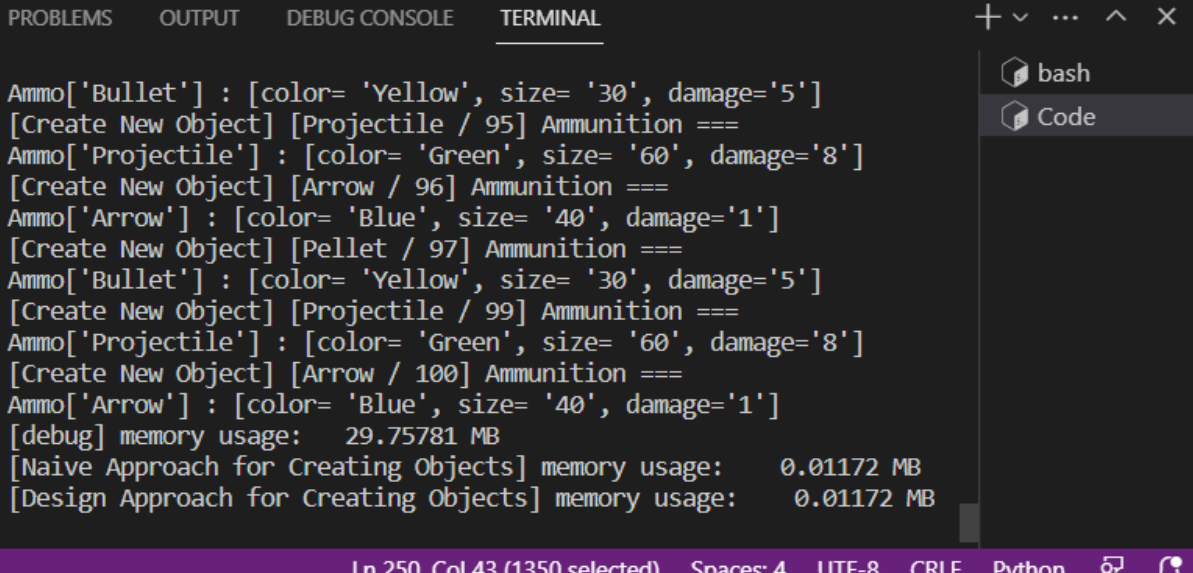
구현 상 객체를 생성하고 설정하는 테스트 동작들은 아래 그림 11과 같이 나타난다. 생성하기 전과 생성한 후의 전체 메모리 차이를 분석하여 실제 메모리 사용량을 계산한다.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
naturejrc@CHUNG_U35S MINGW64 ~
$ set PYTHONIOENCODING=utf8 && python -u "d:\_workspace\python\tes
t2.py"
[debug] memory usage: 29.54688 MB
[Create New Object] [Pellet / 1] Ammunition ===
Ammo['Pellet'] : [color= 'Red', size= '10', damage='3']
[Create New Object] [Bullet / 2] Ammunition ===
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 3] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 4] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
Ammo['Pellet'] : [color= 'Red', size= '10', damage='3']
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
...
[Create New Object] [Projectile / 9991] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 9992] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[Create New Object] [Pellet / 9993] Ammunition ===
Ammo['Pellet'] : [color= 'Red', size= '10', damage='3']
[Create New Object] [Bullet / 9994] Ammunition ===
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 9995] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 9996] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[Create New Object] [Pellet / 9997] Ammunition ===
Ammo['Pellet'] : [color= 'Red', size= '10', damage='3']
[Create New Object] [Bullet / 9998] Ammunition ===
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 9999] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 10000] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[debug] memory usage: 30.82812 MB
[Naive Approach for Creating Objects] memory usage: 1.27344 MB
Ln 205, Col 29 Spaces: 4 UTF-8 CRLF Python 3.11.2 64-bit
```

그림 11. 구현 테스트 동작

테스트 결과 화면의 [Naive Approach for Creating Object] memory usage는 일반적인 객체를 생성할 때마다 메모리를 할당해야 할 때의 메모리 사용량이고, [Design Approach for Creating Object] memory usage는 앞서 설계한 패턴 기반 생성 기법을 사용하여 다수의 객체들을 생성할 때의 메모리 사용량이다.

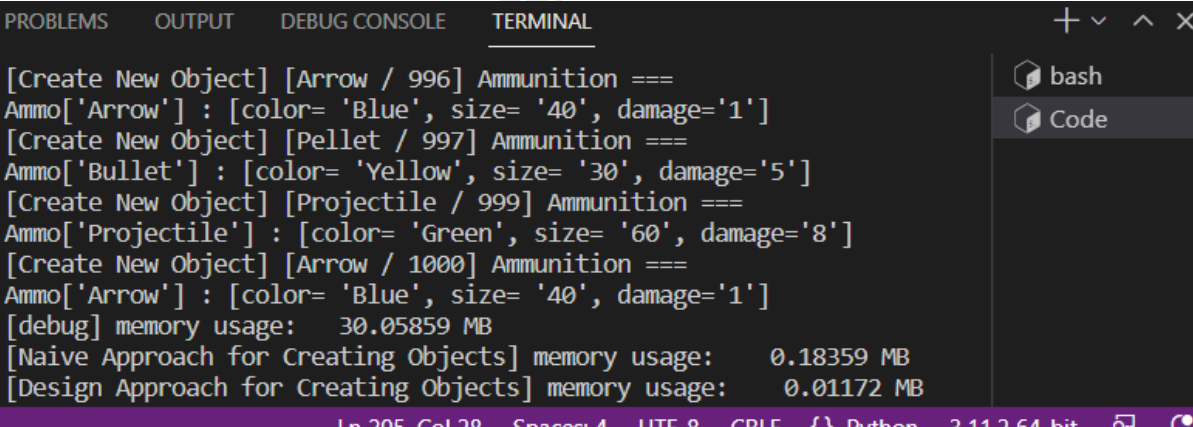


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 95] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 96] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[Create New Object] [Pellet / 97] Ammunition ===
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 99] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 100] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[debug] memory usage: 29.75781 MB
[Naive Approach for Creating Objects] memory usage: 0.01172 MB
[Design Approach for Creating Objects] memory usage: 0.01172 MB
```

Ln 250, Col 43 (1350 selected) Spaces: 4 UTF-8 CRLF Python

그림 12. 발사체 객체 100개 생성 시 메모리 사용량

그림 12는 발사체 객체 100개에 대한 실험결과를 보여준다. 일반적인 생성방법과 본 연구에서 설계한 생성기법의 메모리 사용량이 거의 동일하다.

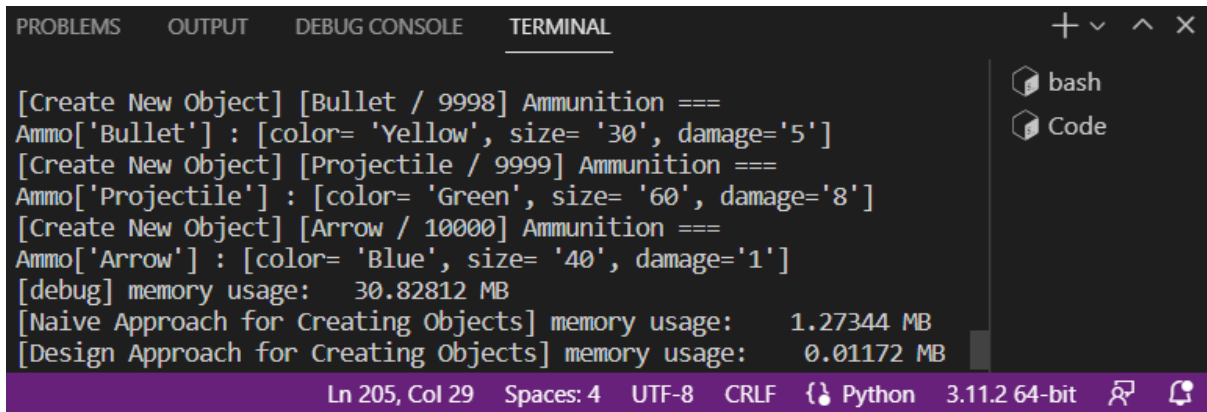


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[Create New Object] [Arrow / 996] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[Create New Object] [Pellet / 997] Ammunition ===
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 999] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 1000] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[debug] memory usage: 30.05859 MB
[Naive Approach for Creating Objects] memory usage: 0.18359 MB
[Design Approach for Creating Objects] memory usage: 0.01172 MB
```

Ln 205, Col 28 Spaces: 4 UTF-8 CRLF Python 3.11.2 64-bit

그림 13. 발사체 객체 1000개 생성 시 메모리 사용량

그림 13은 발사체 객체 1000개에 대한 실험결과를 보여준다. 일반적인 생성방법보다 본 연구에서 설계한 생성기법의 메모리 사용량이 10배이상 적다.



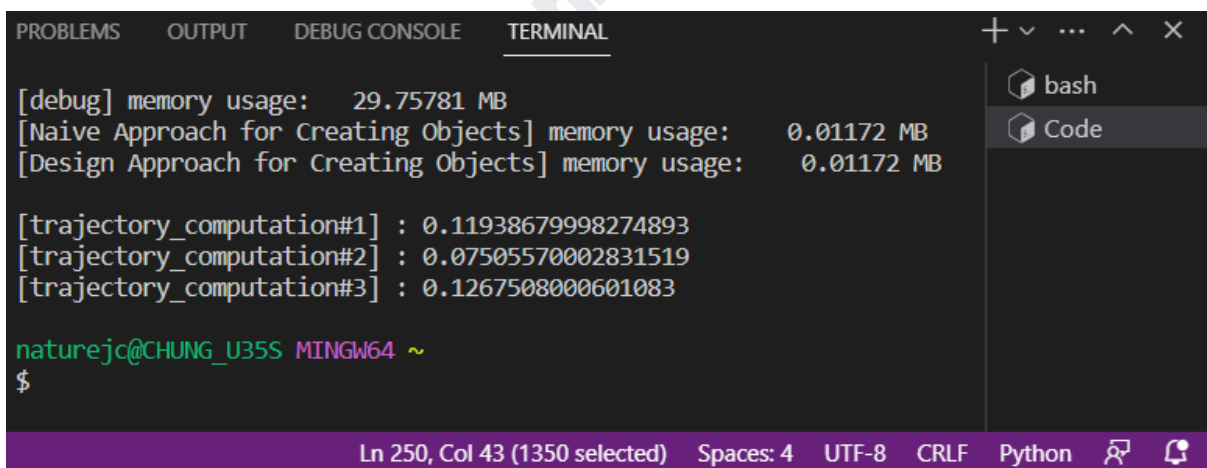
```
[Create New Object] [Bullet / 9998] Ammunition ===
Ammo['Bullet'] : [color= 'Yellow', size= '30', damage='5']
[Create New Object] [Projectile / 9999] Ammunition ===
Ammo['Projectile'] : [color= 'Green', size= '60', damage='8']
[Create New Object] [Arrow / 10000] Ammunition ===
Ammo['Arrow'] : [color= 'Blue', size= '40', damage='1']
[debug] memory usage: 30.82812 MB
[Naive Approach for Creating Objects] memory usage: 1.27344 MB
[Design Approach for Creating Objects] memory usage: 0.01172 MB
```

Ln 205, Col 29 Spaces: 4 UTF-8 CRLF Python 3.11.2 64-bit

그림 14. 발사체 객체 10000개 생성 시 메모리 사용량

그림 14는 발사체 객체 10000개에 대한 실험결과를 보여준다. 일반적인 생성방법보다 본 연구에서 설계한 생성기법의 메모리 사용량이 100배이상 적다. GUI, 다른 기능과 프로세스들이 적용되지 않은 단순 테스트프로그램 상의 데이터라 완벽하다고 할 수는 없지만 본 연구에서 설계한 생성기법의 사용 시에 의미있는 메모리 사용량 감소가 발생하는 것을 검증하였다

2. 발사체 궤적 연산을 위한 계산식 비교

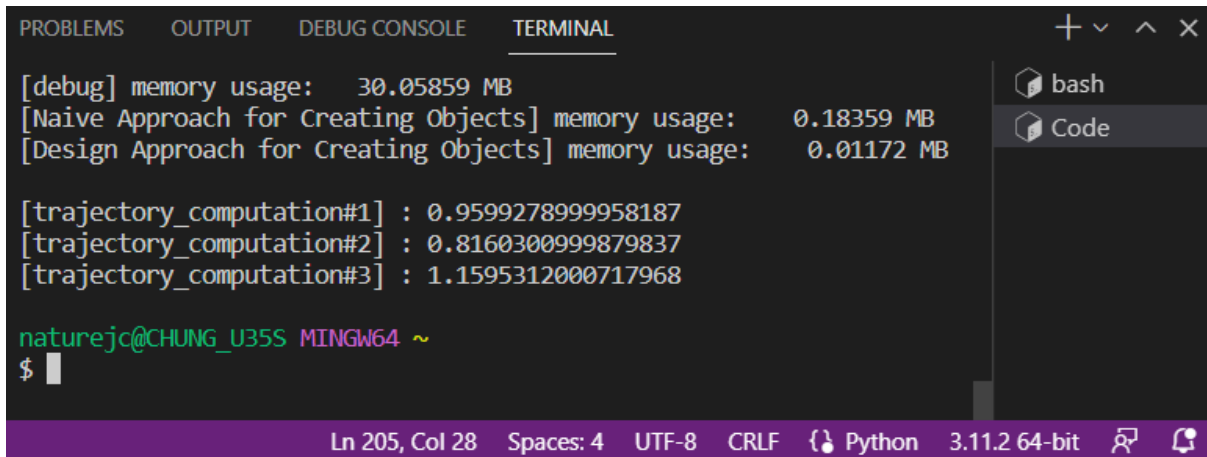


```
[debug] memory usage: 29.75781 MB
[Naive Approach for Creating Objects] memory usage: 0.01172 MB
[Design Approach for Creating Objects] memory usage: 0.01172 MB

[trajectory_computation#1] : 0.11938679998274893
[trajectory_computation#2] : 0.07505570002831519
[trajectory_computation#3] : 0.1267508000601083

naturejic@CHUNG_U35S MINGW64 ~
$
```

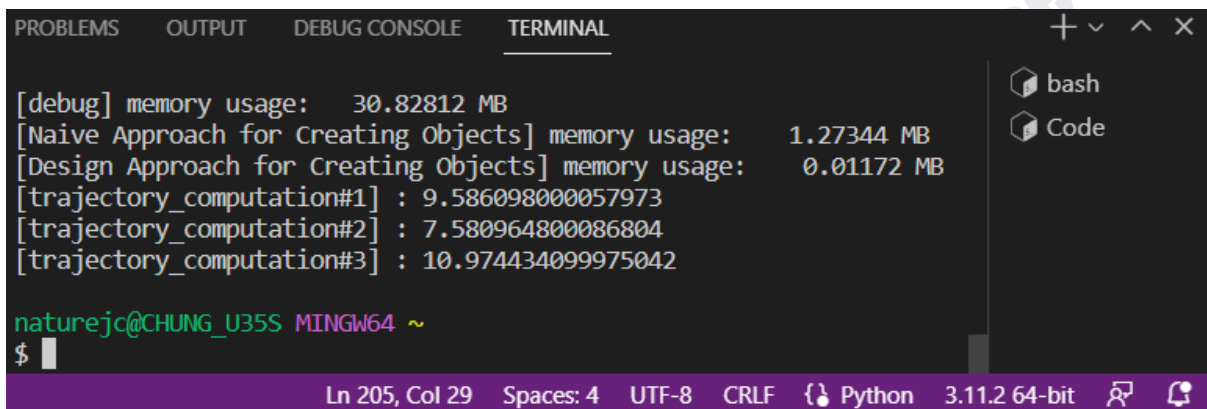
Ln 250, Col 43 (1350 selected) Spaces: 4 UTF-8 CRLF Python



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[debug] memory usage: 30.05859 MB
[Naive Approach for Creating Objects] memory usage: 0.18359 MB
[Design Approach for Creating Objects] memory usage: 0.01172 MB

[trajectory_computation#1] : 0.9599278999958187
[trajectory_computation#2] : 0.8160300999879837
[trajectory_computation#3] : 1.1595312000717968

naturejic@CHUNG_U35S MINGW64 ~
$
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[debug] memory usage: 30.82812 MB
[Naive Approach for Creating Objects] memory usage: 1.27344 MB
[Design Approach for Creating Objects] memory usage: 0.01172 MB

[trajectory_computation#1] : 9.586098000057973
[trajectory_computation#2] : 7.580964800086804
[trajectory_computation#3] : 10.974434099975042

naturejic@CHUNG_U35S MINGW64 ~
$
```

그림 15. 발사체 궤도에 대한 3가지 계산식의 연산시간

앞서 도출한 발사체 궤도에 대한 3가지 계산식에 대한 시간 측정을 통해 소프트웨어 품질 특성 중 효율성의 정도를 판단하였다. 3가지 계산식의 차이가 아주 크지는 않았지만, 결과적으로 구현 테스트 상 가장 효율성이 높은 계산식은 2번째 계산식으로 평가한다.

VI. 결론 및 향후 과제 (주장/시사점)

본 연구는 슈팅 로봇에서 다수의 발사체를 발사할 때 발생하는 오버헤드 문제를 대상으로 다루었다. 플랫폼 독립적인 소프트웨어 관점에서 다음과 같은 과정을 통해 제시한 문제를 개선하였다.

- (1) 다수 발사체 오버헤드 발생 문제를 객체지향 설계하여 python 프로그래밍 언어로 구현하였다.
- (2) 발사체 생성과 발사체 궤적 연산을 분석하여 메모리량과 시간 측정 함수를 구현하였다.
- (3) 오버헤드 문제를 개선하기 위한 Flyweight 패턴기반의 발사체 개체 생성 방안과 발사체 궤적 연산 비교 기법을 설계하였다.

(4) 각 설계별 메모리 사용량과 연산시간을 측정하여 비교하였다.

또한, 본 연구자가 진행했었던 관련연구[4]에서 수학 계산식의 오류를 본 연구를 진행하면서 발견하여 수정할 수 있었다.

본 연구에 대한 제약 사항은 소프트웨어 설계 관점에서 테스트가 이뤄졌기 때문에 실제 슈팅로봇이 동작하는 상황이 아닌 테스트환경에서 검증하였고, 슈팅로봇만 설계하여 검증하였기 때문에 전체 소프트웨어에 대한 설계/구현을 추후 진행할 수 있다.

VII. 참고문헌 / 연구 관련 용어집

[1] 게임 소프트웨어의 품질 평가 모델

: 정혜정한국인터넷정보학회인터넷정보학회논문지8(6)pp.115~1252007.12컴퓨터학

[2] 미적분의 쓸모 (*the use of calculus*)

: 한화택(*Hwataik Han*) 출판사-더퀘스트 ISBN-9791165219550

[3] 이렇게 흘러가는 세상 (*This is how the world goes*)

: 송현수(*Hyunsoo Song*) 출판사-MID ISBN-9791190116213

[4] 미적분 탄도미사일 구현 - 수학을 이용한 실생활 연구 보고서

: 정휘준 <https://hweejoon-chung.github.io/calculusTrajectory-assessmentMATH>

[5] 메모리가 제한적인 자바가상기계에서의 지역 재사용

: 김태인김성건한환수한국정보과학회정보과학회논문지 : 소프트웨어 및 응용34(6)pp.562~5712007.06컴퓨터학

[6] 효율적인 온라인 게임 서버를 위한 객체풀링 기법에 관한 연구

: 김혜영, 함대현, 김문성 한국게임학회논문지 2009, vol.9, no.6, 통권 31호 pp. 163-170 (8 pages)

[7] *Design Pattern (Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, Addison-Wesley)*

[8] *Observer*패턴을 적용한MMORPG파티 시스템 아이템 배분 방법

[9] 안드로이드 게임 프로그래밍을 위한 설계 패턴

:

김태석김신환김종수한국멀티미디어학회멀티미디어학회논문지10(8)pp.1060~10672007.08전자/정보통신공학