코딩의 기술

노트북: Coding
만든 날짜: 2020-08-18 오후 9:16 **수정한 날짜:** 2020-08-30 오전 12:37
작성자: yoonjeong_choi@tmax.co.kr
태그: Clean, Game

[1장 읽기 좋은 코드를 작성하는 기술]

- 읽기 좋은 코드
  - 코드의 보수성
    - 코드의 변경/추가/테스트가 얼마나 용이한지를 의미
  - 코드의 가독성은 코드의 보수성에 영향
    - 가동석 높고 보수하기 쉬운 코드를 작성
  - 다른 개발자에게 코드의 의도를 명확하게 전달하기 위해서 작성
  - 복잡한 코드 문제를 작게 나누고, 읽기 좋은 이름을 붙여 정리
    - i.e 복잡하게 커져 보린 코드의 문제를 단순화하고 작게 하는 기술
- 목차
  - 변수와 상수
  - 조건식과 계산식
  - assert문 활용
  - 제어문
    - 조건문(if)
    - 반복문(for)
    - 조건문(switch)
  - 함수
    - 기본 원칙
    - 함수화 패턴
    - 매개 변수 문제
    - 작은 함수의 필요성
    - 함수를 사용할 때의 마음가짐
  - 클래스
    - 클래스화 요령
    - 클래스화 패턴
    - 작은 클래스

[1. 변수와 상수]

- 의미가 명확한 변수 이름 붙이기
  - 변수의 목적이 판단 가능한 이름 명명

- 사용 목적을 가능한 구체적으로 적어라
- 참조되는 범위가 넓은 변수, 수명이 긴 변수는 신중하게 명명

- 매직 넘버에 이름 붙이기(enum)
  - 매직 넘버 : 코드를 작성한 본인만 알수 있는, 코드 내부에 들어 있는 상수값
  - 열거형(enum)을 사용하여 매직넘버에 이름을 붙여라

```
enum GameState {
    STATE_WALK,
    STATE_ATTACK,
    STATE_JUMP
};
```

  - 계산식 or 조건식에서 사용하는 매직 넘버에는 이름을 반드시 붙여라

[2. 조건식과 계산식]

- Idea
  - 코드 내부의 변수/함수에 이름을 붙여 사용하면 의도가 명확하게 전달 가능
  - 설명 전용 변수 사용
  - 계산식 또는 조건식의 함수화
    - 계산식/조건식의 의도를 함수 이름을 통해 명확하게 전달 가능

- 조건식 작성 요령
  - 복잡한 조선식을 구성하는 경우 설명 전용 변수 사용
    - 지역 변수(bool type)를 사용하여 조건식의 구성 요소에 이름을 붙이는 방법
    - 설명 전용 변수는 변하지 않기 때문에 const를 사용
      - const를 사용함으로써 설명 전용 변수임을 명시 가능

```
const bool isJump = (y > 0.0f);
const bool isAttack = (state == GameState::STATE_ATTACK);
const bool isRun = (speed > 10.0f) && !isJump && !isAttack;
if (isRun) {

}
```

  - 조건식의 함수화
    - 조건식의 구성 요소를 함수를 통해 true/false 계산
    - 설명 전용 변수에 대응했던 함수를 그대로 함수화
    - 복잡한 복합 조건은 분할하여 작성(아래의 isDash()처럼)

```
bool isJump() {
    return y > 0.0f;
}
bool isAttack() {
    return state == GameState::STATE_ATTACK;
}
bool isDash() {
    if (isJump()) return false;
    if (isAttack()) return false;
    if (speed < 10.0f) return false;
    return true;
}
```

- 계산식 작성 요령
  - 설명 전용 변수 사용
    - 복잡한 계산식을 분해하여 각 계산값의 결과를 설명 전용 변수에 저장
    - if 조건문의 조건식에 복잡한 계산식을 쓰지 않고 설명 전용 변수를 사용하여 간단하게 만든다
  - 계산식의 함수화
    - 계산식의 계산이 무엇을 하는 코드인지 파악하여 해당 코드를 함수화
    - 하나의 의미를 나타내는 코드는 함수화해야 하는 후보

[3. assert 활용]

- <cassert> 헤더 파일을 통해 사용

- assert
  - 프로그램의 사전 조건과 사후 조건을 명시적으로 활용하기 위해 사용
  - 매개 변수의 조건식이 false인 경우 오류 메시지를 표시하고 프로그램을 중지시키는 매크로
  - 디버그를 지원하기 위한 기능

- assert로 정지되면, 디버거가 프로그램 전체를 브레이크
  - 정지 시점에서의 콜 스택의 이력 확인 가능
  - Debug 모드 전용 기능
  - Release 모드로 컴파일하면 코드 내부의 assert는 모두 무효화

[4-1. 제어문 - 조건문(if)]

- if 조건문과 for 반복문을 사용하는 알고리즘은 대부분 일반적인 알고리즘 함수로 변환 가능
  - 이때, C++은 STL(표준 라이브러리)를 사용

- 상한값, 하한값 체크 단순화
  - 하한값의 경우 if(x>k){x=k} 대신 x=std::min(x,k)을 사용
  - 상한값의 경우 if(x<k){x=k} 대신 x=std::max(x-1,k)을 사용
  - 상한값과 하한값을 보정하는 코드는 자주 사용되는 패턴
    - Clamp 함수를 미리 만들어 사용

```
template<typename T>
int Clamp(T x, T low, T high) {
    assert(low <= high);
    return std::min(std::max(x, low), high);
}
```

    - boost C++ 라이브러리에 있는 함수
  - 랩 어라운드(wrap around)
    - 상한값에 이르면 하한값을 돌려주고 다시 계산하는 함수
    - 하한값과 상한값 사이의 숫자를 반복하고 싶을 때 사용
    - 상한값/하한값 체크를 포함한 함수
    - 정수형 랩 어라운드 함수

```
int WrapAround(int x, int low, int high) {
    assert(low < high);
    const int dist = (x - low) % (high - low);
    return (dist >= 0) ? (low + dist) : (high + dist);
}
```

    - 실수형 랩 어라운드 함수(fmod 사용)

```
float WrapAround(float x, float low, float high) {
    assert(low < high);
    const float dist = std::fmod(x - low, high - low);
    return (dist >= 0) ? (low + dist) : (high + dist);
}
```

  - 하한값/상한값을 다루는 것은 버그의 원인이 되기 쉬운 부분
    - Clamp 및 WapArround를 사용하여 오류 줄이기

- 중첩된 if 조건문 줄이기 - 조기 리턴
  - 조기 리턴 활용
    - 함수의 입구에서 예외 조건을 확인하고 조기에 리턴하는 것
    - 예외 조건에 해당하는 중첩된 if 조건문들을 빼내어 함수화하여 사용
  - 1개의 조건식 결과를 리턴하는 경우 if 조건문을 사용하지 말자
    - 조건식을 bool로 리턴하는 경우 조건식 자체를 return

```
bool isDead() {
    return health <= 0;
}
```

        - 숫자로 리턴하는 경우 삼항 연산자를 return

```
int bonusTime(int time) {
    return (time < 10) ? 1000 : 0;
}
```

- 중첩된 if 조건문 줄이기 - 중복된 조건식 통합
    - 중첩된 조건문 내부에서 조건이 중복된다면, 중복이 일어나는 대상을 먼저 판정
        - Before : wait_timer > WAIT_TIME 조건 중복

```
void RepeatedCondition() {
    if (state == GameState::STATE_FALL) {
        if (wait_timer > WAIT_TIME) {
            fall();
        }
    }
    if (state == GameState::STATE_MOVE) {
        if (wait_timer > WAIT_TIME) {
            move();
        }
    }
}
```

        - After : wait_timer > WAIT_TIME 조건 먼저 판정

```
void RepeatedCondition_Improved() {
    if (wait_timer > WAIT_TIME) {
        if (state == GameState::STATE_FALL) {
            fall();
        }
        if (state == GameState::STATE_MOVE) {
            move();
        }
    }
}
```

    - 복합 조건이 중복되는 경우에도 조건식을 통합
        - Before : wait_timer > WAIT_TIME 조건 중복

```
void RepeatedCondition() {
    if (state == GameState::STATE_FALL && wait_timer > WAIT_TIME) {
        fall();
    }
    if (state == GameState::STATE_MOVE && wait_timer > WAIT_TIME) {
        move();
    }
}
```

        - 복합 조건은 if 조건문을 2개 작성한 것과 같기 때문에 사실상 조건문이 2개
        - 중복된 조건문이 있으면, 해당 조건문 변경시 여러 곳을 수정해야 하기 때문에 통합 필요

- 조건식이 직접 관계하는 부분만 따로 떼어내서 국소화
    - if 조건문의 크기가 작아지고, 코드의 중복을 피하는 효과
    - Before : if 조건문 내부에서 계산식 중복

```
void accelerate() {
    if (isDash())
        position += direction*10.0f;
    else
        position += direction*5.0f;
}
```

- After : if 조건문에서 직접 관계하는 부분만 처리

```
void accelerate_Improved() {
    float speed = 5.0f;
    if (isDash())
        speed = 10.0f;
    position += direction*speed;
}
```

- 배열을 활용한 조건문 제거
  - 특정 순서를 가지는 값을 변환하는 함수의 경우 input 값들에 대한 if 조건문이 여러개 필요
    - 변환 후의 데이터를 저장하는 변환 전용 배열을 사용하여 if 조건문을 줄일 수 있다
    - 이때, 변환 전용 배열은 static const를 사용하여 쓸데없는 반복 초기화 방지
  - Before : input으로 들어 올 수 있는 값의 개수만큼 조건문 존재

```
int IdToNum(int id) {
    if (id == 0) return 10;
    if (id == 1) return 15;
    if (id == 2) return 20;
    if (id == 3) return 35;
    assert(!"Invalid Id");
    return 0;
}
```

  - After : 변환 전용 배열 idTable 사용

```
int IdToNum_Improved(int id) {
    static const int idTable[] = { 10,15,20,35 };
    assert((0<=id<=3) && !"Invalid Id");
    return idTable[id];
}
```

  - 연관 배열
    - unordered_map 클래스로 해시 구조
    - 변환 이전의 수치를 key, 변환 이후의 수치를 value로 하여 사용
    - Example :

```
const std::unordered_map<int, int> table = {
    { 10, 0 },{ 15, 1 },{ 30, 2 },{ 50, 3 }
};
int NumToId(int num) {
    assert((table.find(num) != table.end()) && !"Invalid Number");
    return table.at(num);
}
```

  - 결론 : 자료 구조를 잘 사용하면 if 조건문을 줄일 수 있음
    - 자료 구조로는 배열, unordered_map 등 활용 가능
    - 변경 or 추가해야하는 상황에서도 해당 데이터(자료구조를 이용한)만 수정하면 된다

- 결정표를 사용한 조건문 제거

- 결정표 : 판정 조건의 조합과 그에 대응하는 결과를 정리한 표
- 조건의 조합을 모두 if 조건문으로 판정하면 많은 조건문이 필요
  - 모든 조건을 기술하지 않는 실수 발생
  - 보수(추가/수정)시 복잡도 증가
- Example : 가위바위보 판정

```
enum Hand {Rock=0, Scissor=1, Paper=2};
enum Result {Win, Lose, Draw};
const Result resultTable[3][3] = {
    {Draw, Win, Lose},  // Rock
    {Lose, Draw, Win},  // Scissor
    {Win, Lose, Draw}   // Paper
};

Result judgement(Hand my, Hand target) {
    return resultTable[my][target];
}
```

- null 객체 사용
  - null 객체 : null 포인터를 대신하는 더미 객체
  - 여러 개의 null 체크를 해야 하는 설계는 피해야 한다
    - null 체크를 피하기 위해 null 객체 사용
  - null 객체에 해당하는 클래스의 부모 클래스는 반드시 모든 멤버 함수가 가상 함수여야 한다
    - i.e 부모 클래스는 interface(abstract class)
    - 객체 전용 변수에 nullptr을 넣지말고, null 객체를 넣어준다
  - Example : null 객체

```
class Actor {
    virtual void move() = 0;
    virtual void draw() = 0;
};

class NullActor : public Actor {
    virtual void move() override {
        // 아무것도 하지 않음
    }
    virtual void draw() override {
        // 아무것도 하지 않음
    }
};
```

[4-2. 제어문 - 반복문(for)]

- if 조건문과 for 반복문을 사용하는 알고리즘은 대부분 일반적인 알고리즘 함수로 변환 가능
  - 이때, C++은 STL(표준 라이브러리)를 사용

- 반복문 기본 원칙 : 1개의 작업만 반복
  - 2개 이상의 작업을 반복문 내부에 넣지 않는 것이 원칙

- STL : for_each
  - 모든 요소에 대해 특정 처리를 하는 가장 간단한 반복 알고리즘 함수
  - for_each(container.begin(), container.end(), function)
  - Example :

```
void DrawActors() {
    std::for_each(actors.begin(), actors.end(),
        [](Actor* actor) {actor->draw();}
    );
}
```

- STL : find_if
  - 특정 조건에 일치하는 것을 검색하는 반복 알고리즘 함수
  - find_if(container.begin(), container.end(), bool conditionFunction)
  - Example :

```
void SearchPlayer(int playerId) {
    auto player = std::find_if(actors.begin(), actors.end(),
        [&playerId](Actor* actor) {return actor->id() == playerId; }
    );
}
```

- 기타 STL 반복 알고리즘
  - min_element(container.begin(), container.end(), 대소비교함수)
    - 대소비교함수를 사용하여 container의 최소값의 반복자 반환
  - max_element(container.begin(), container.end(), 대소비교함수)
    - 대소비교함수를 사용하여 container의 최대값의 반복자 반환
  - all_of(container.begin(), container.end(), 판별조건함수)
    - 판별조건함수를 이용하여 container의 모든 요소가 조건에 대해 true인지 판별하여 bool 값 반환
  - count_if(container.begin(), container.end(), 판별조건함수)
    - 판별조건함수를 이용하여 container의 요소 중 조건이 true인 요소 개수 반환
  - copy_if(container.begin(), container.end(), back_inserter(targetContainer), 판별조건함수)

        - 판별조건함수를 이용하여 container의 요소 중 조건이 true인 요소들만을 targetContainer에 복사하는 함수

- 독자적인 반복문을 작성하지말고 STL 반복 알고리즘을 사용
    - 코드 자체가 설명적이고 가독성이 높아진다
    - 버그가 없는 신뢰성 높은 STL 함수를 사용함으로써 코드의 품질이 좋아진다

- 검색 반복문 분리
    - 상황 : 특정 조건에 맞는 것을 검색하고 처리를 실행하는 반복문
        - for 반복문 안에 if 조건문이 존재
    - find_if를 사용하여 특정 조건에 맞는 요소를 검색하여 반복문을 분리
        - for 반복문과 if 조건문의 분리 가능
    - 검색 반복문(find_if의 반환값)을 분리하여 함수화하면, 코드가 단순해지고 가독성이 높아진다

- 반복문 내부의 불필요한 조건 분리
    - 반복문 밖에 있어야 하는 if 조건문이 반복문 내부에 있는 경우에는 if문을 밖으로 분리
    - 불필요한 처리의 반복 방지
    - 반복문 내부 처리는 최소화

- 반복문 분할
    - 하나의 for 반복문에 여러 개의 처리가 있는 경우 STL 알고리즘 함수로 치환하는 것이 불가능
    - 1개의 반복문에 여러 처리가 있으면 코드가 복잡해 진다
    - 단순한 for 반복문으로 분할하여 STL 알고리즘으로 치환

- 람다식을 활용한 반복문 일반화
    - 불가능한 상황
        - 2차원 배열 조작은 중첩 반복문을 사용해야 하며, STL 알고리즘으로 치환 불가능
        - 독자적인 자료 구조를 사용해야 하는 상황에서는 해당 자료 구조가 STL 컨테이너가 아니기 때문에 STL 알고리즘 사용 불가능
    - std::function
        - 람다신, 함수 포인터, 함수 객체를 대입할 수 있는 범용적인 함수 래퍼 클래스

        - 람다식 or 함수 포인터를 대입해 변수를 생성하는 기능
    - **다시 공부 필요**

[4-3. 제어문 - switch 조건문]

- 게임 프로그래밍에서는 상태 변화를 처리할 때 switch 조건문을 많이 사용
    - switch 조건문을 남용하면 코드가 복잡해 진다

- case 내부의 함수화
    - case 내부에는 복잡한 코드를 작성하지 않는다
    - 오직 분기에만 집중
    - case에 반드시 일치해야 하는 경우에는 default 부분에 assert문 사용
    - Example :

```
void update(float deltaTime) {
    switch (state) {
    case GameState::STATE_ATTACK:
        attack(deltaTime);
        break;
    case GameState::STATE_JUMP:
        jump(deltaTime);
        break;
    case GameState::STATE_FALL:
        fall(deltaTime);
        break;
    default:
        assert(!"Invalid GameState");
        break;
    }
}
```

- 다형성을 사용한 분기
    - 상태 변화를 switch 조건문으로 처리하는 경우에는 State 패턴으로 변경 가능
    - State 패턴
        - State 패턴에서는 상태에 대한 인터페이스를 상속받아 각 상태를 클래스화 한다
        - 다형성으로 각 상태 갱신 처리를 분기
        - Example :

```
class State abstract {
public:
    virtual ~State();
    virtual void update(float deltaTime) = 0;
    virtual void move(float deltaTime) = 0;
};

class RunState : public State {
public:
    ~RunState();
    virtual void update(float deltaTime) override;
    virtual void move(float deltaTime) override;
};
```

    - 다루는 상태의 수가 적거나, 각 상태의 처리가 단순하면 switch 조건문 사용하는 것이 유리할 수 있음

[5-1.함수 - 기본 원칙]

- 함수화 기술은 보수성 높은 코드를 작성하는 기본

- 기본 원칙 1 : 1개의 함수에는 1개의 역할
    - 함수는 1개의 역할만 수행해야 한다
    - 복잡한 함수 이름은 주의 대상
        - 여러 개의 역할을 가지고 있는 함수의 이름이 복잡하기 때문
        - 복잡한 이름은 의도 파악하기 어려움
    - 기능이 적은 함수일수록 재사용이 쉽고, 변경에 대해 영향을 적게 받는다
- 기본 원칙 2 : 함수를 두 종류로 구분
    - Type1 - 계산과 알고리즘을 실행하고 실제 작업을 수행하는 함수
    - Type2 - Type1 함수들을 조합해서 흐름을 만드는 함수
    - 큰 함수를 제대로 작성하는 것은 어렵지만, 작은 함수를 제대로 작성하는 것은 쉽다
        - 큰일을 하는 함수는 Type2로 만들고, 기능들을 분할하여 Type1 함수들로 분할

[5-2. 함수 - 함수화 패턴]

- 함수화의 중요성
    - 코드의 중복된 부분은 반드시 함수화
        - (BUT) 코드의 중복만이 함수화의 대상이 아니다
    - 함수화를 통해 해당 코드에 의미있는 이름 부여 가능

- 함수화 패턴

    - 조건식 함수화
    - 계산식 함수화
    - 조건 분기의 블록 내부 함수화
    - 반복문 함수화
    - 반복문의 블록 내부 함수화
    - 데이터 변환 함수화
    - 데이터 확인 함수화
    - 배열 접근 함수화
    - 주석 부분 함수화

- 조건식 함수화
    - if 조건문의 조건식을 함수화
    - 조건식에 의미있는 이름 부여 가능
    - 복잡한 복합 조건을 함수화하여 가독성 up

- 계산식 함수화
    - 계산식의 구성 요소를 함수화
    - 계산식에 의미있는 이름 부여 가능
    - 계산식을 함수화하면 계산식을 자연어를 읽는 것처럼 읽기 가능

- 조건 분기의 블록 내부 함수화
    - if 조건문 또는 switch 조건문의 블록 내부 함수화
    - 조건문 블록 내부는 가능한 한 단순화하여 분기에만 집중 가능

- 반복문 함수화
    - 1개의 반복문에 대해서 1개의 함수 작성
        - 이때 반복문은 1개의 기능만 수행해야 한다
    - STL 반복 알고리즘 함수를 사용할 수 없는 경우에는 반드시 반복문을 함수화

- 반복문의 블록 내부 함수화
    - for 반복문의 블록 내부를 함수화하고, 중첩된 조기 조건 리턴을 사용
    - 조기 리턴에 if 조건문이 많이 나열되어 있으면, if 조건문들을 함수화하여 이름 부여
    - 블록 내부 함수화가 힘든 경우에는, 반복문 내부에서 2가지 이상의 일을 하고 있는지 확인하여 분할

- 반복문 블록 내부를 함수화하여, STL 알고리즘 함수를 사용할 수 있는 상태까지 단순화

- 데이터 변환 함수화
  - 데이터 변환과 관련된 부분은 따로 빼서 함수화
    - 데이터 변환은 if 조건문이나 변환 전용 자료 구조를 사용
    - 이러한 부분을 함수화하면 코드가 단순화
  - 팩토리 패턴 - 특정 조건에 대해서 객체를 생성
    - 상황 : 특정 조건에 대해서 캐릭터를 생성
    - 단순한 데이터 변환은 아니지만 해당 객체 생성 방식은 함수화 by 팩토리 패턴

- 데이터 확인 함수화
  - 데이터 확인과 관련된 부분은 따로 빼서 함수화
    - 데이터 변환은 if 조건문이나 변환 전용 자료 구조를 사용
    - 이러한 부분을 함수화하면 코드가 단순화
    - Before :

```
bool isEnemy(ActorId id) {
    if (id == ActorId::Slime) return true;
    if (id == ActorId::Slime) return true;
    if (id == ActorId::Slime) return true;
    return false;
}
```

    - After : 자료 구조로 데이터 확인

```
const std::unordered_set<ActorId> EnemyId = {
    ActorId::Dragon,
    ActorId::Goblin,
    ActorId::Slime
};
bool isEnemy_Improved(ActorId id) {
    return EnemyId.find(id) != EnemyId.end();
}
```

  - 데이터가 범위 내부에 있는지 확인하는 부분도 함수화

```
bool isInside(int x, int y) {
    bool isInsideHorizontal = (0 <= x && x < MAP_WIDTH);
    bool isInsideVertical = (0 <= y && y < MAP_HEIGHT);

    return isInsideHorizontal && isInsideVertical;
}
```

- 배열 접근 함수화
  - n차원 배열에 랜덤 접근하는 경우에는 배열에 접근하는 함수를 따로 구현
  - 배열 범위 외부를 참조하지 않도록 assert를 사용하여 인덱스가 범위 내부에 있는지 확인하는 처리 추가
  - 배열 접근 함수화가 어려운 경우에는 STL의 vector/array 컨테이너를 사용

        - STL 컨테이너는 Debug 모드일 때에 인덱스의 범위 체크를 자동으로 수행
    - C++의 기본 자료형을 사용한 배열은 버그가 일어나기 쉬운 부분
        - 특별한 이유가 없다면 사용을 피해야 한다

- 주석 부분 함수화
    - 코드가 길어지거나 특별한 기술을 사용하게 되면, 주석을 쓰고 싶은 충동 발생
        - 그 순간 함수화를 해야할 상황
    - 주석을 쓰기 전에 해당 부분의 함수화
    - 함수화하고 적절한 이름을 붙여주면 코드 블록에 주석을 쓸 필요 X

[5-3. 함수 - 매개 변수 문제]

- 함수의 매개 변수가 너무 많은 문제
    - 함수의 매개 변수가 많아지는 것은 설계 문제
    - 함수의 매개 변수를 줄이는 것은 설계를 개선하는 작업

- 함수의 매개 변수가 많은 원인
    - 욕심쟁이 함수
    - 부적절한 클래스화

- 욕심쟁이 함수
    - 1개의 함수가 너무 많은 일을 하려 하면 매개 변수의 수가 증가
    - 욕심쟁이 함수는 함수를 분할
        - 분할 단위가 작아질수록 각 함수의 매개 변수도 작아짐
    - 리턴값이 2개 이상 있는 함수는 되도록 구현 X
        - 리턴값이 2개 이상인 함수는 2개 이상의 기능이 있을 가능성이 높음

- 부적절한 클래스화
    - 클래스화가 부적절하면 매개 변수의 수 증가
        - 매개 변수를 모아 클래스화하면 매개 변수 감소
    - 클래스 내부의 멤버 변수는 매개 변수로 전달하지 않아도 참조 가능(call by reference)
        - 클래스화를 적절히 하면 함수의 매개 변수를 줄이는 효과
        - 연관성이 높은 매개 변수끼리 클래스화

    - 매개 변수가 너무 많다면 클래스화하라는 신호

[5-4. 함수 - 작은 함수의 필요성]

- 작은 함수의 필요성
    - 자기 설명적인 코드
        - 작은 함수의 이름을 통해 해당 코드에 의미있는 이름 부여 가능
        - 의미있는 이름을 통해 주석없이 코드의 가독성 up
    - 개별 테스트 기능
        - 개별적인 단위 테스트 가능
        - 테스트의 정밀도 상승
        - 테스트에서 정상적으로 작동하는 함수를 조합해서 함수를 만들면, 복잡해서 단위 테스트가 불가능한 함수도 신뢰성 상승

- 작은 함수에 의한 실행 속도 저하
    - Question : 함수 호출 오버헤드로 실행 속도 저하가 발생
    - Debug 모드에서는 컴파일러가 코드를 최적화 X
        - 함수화한 경우와 함수화하지 않은 경우에 실행 속도 차이 존재
    - Release 모드에서는 작은 함수가 인라인으로 전개
        - 인라인 전개 : 함수를 호출하는 대신 컴파일러가 해당 위치에 함수 내부의 처리르 복사해서 전개하는 기능
        - 인라인 전개에 의해 함수 호출의 오버헤드 X
    - 비주얼 스튜디오의 LTCG(링크 때 코드 생성) 기능
        - 링크 때에 함수를 인라인 전개하는 기능
        - 프로그램 전체인 함수가 최적화 대상이 되어 다른 파일에 있는 함수까지 인라인화
    - 컴파일러의 최적화 기능을 잘 활용하면 함수 호출로 인한 오버헤드 문제 발생 방지
        - 작고 단순한 함수일수록 컴파일러의 최적화 혜택
    - 실행 속도를 올리려면 알고리즘/자료구조 차원의 개선 생각
        - 세세한 코드 레벨의 최적화는 컴파일러에게 맡기는 것이 무난

- 재사용되지 않는 부분의 함수화
    - 오해 : 재사용만이 함수화의 유일한 장점이자 이유라고 착각
    - 복잡해지기 쉬운 부분을 국소화 가능
    - 복잡한 부분을 함수 내에 은폐하여 코드의 추상화

[5-5. 함수 - 함수를 사용할 때의 마음가짐]

- 함수를 사용할 때의 마음가짐
  - 일단 작성해보기
  - 처음 작성했던 함수는 밑그림
  - 문제의 본질 이해
  - 읽는 사람의 관점에서 확인

- 일단 작성해보기
  - 처음부터 잘 분할된 완벽한 함수 구현 불가능
  - 지저분하더라도 원하는 기능을 실행하는 함수 구현
  - 구현을 통해서 해당 문제에 대한 이해도가 높아진다

- 처음 작성했던 함수는 밑그림
  - 처음 작성한(일단 작성한) 함수는 밑그림
  - 구현한 함수는 계속적으로 개선 필요

- 문제의 본질 이해
  - 함수를 분할하기 위해서는 문제의 본질을 제대로 파악해야 한다
    i.e 함수를 제대로 분할하는 것은 문제의 본질을 제대로 이해하는 것과 직결
  - 함수를 조금씩 분할하다 보면 점점 문제의 본질에 접근

- 읽는 사람의 관점에서 확인
  - 완성된 함수는 다시 한번 읽기
  - 적절한 이름, 복잡한 부분 존재 유무 등을 객관적으로 판단
  - 미래에 변경/추가가 있을때 쉽게 대응 가능한지 판단

[6-1. 클래스 - 클래스화 요령]

- 클래스화도 공통 부분을 한꺼번에 정리한다는 목적에서 함수화와 비슷
  - 프로그램 설계의 요령은 문제를 작게 나누는 것

- 클래스화 요령
  - 큰 함수를 분할
    - 거대한 함수 내부에는 여러 개의 클래스 후보가 숨어 있음

        - 클래스화의 사전 준비로 큰 함수는 작은 함수로 분할
        - 클래스는 함수보다 높은 추상화 단계
        - 숨겨진 클래스를 찾기 위한 첫번째 단계
    - 함수 이름을 기반으로 클래스 이름 정하기
        - 멤버 함수 이름에서 동사 뒤에 붙어있는 명사가 클래스후보
        - Example : updatePlayer, drawPlayer 등의 멤버 함수 이름이 존재하면, Player에 대한 클래스화 필요
            - Player 클래스의 멤버 함수로 update, draw 처리

[6-2.클래스 - 클래스화 패턴]

- 클래스화 패턴 유형
    - 중복 부분 클래스화
    - 기본 자료형으로 구성된 멤버 변수 클래스화
    - 함수의 매개 변수 클래스화
    - 컨테이너 클래스화

- 중복 부분 클래스화
    - 여러 개의 클래스에서 코드가 중복되는 부분을 클래스화
    - 클래스들의 중복 부분은 상속/이양으로 해결
    - 상속 : 중복 부분을 한꺼번에 정리해서 부모 클래스로 생성
        - 상속을 사용한 재사용 방법을 "범화"라 부른다
    - 이양 : 중복 부분만 클래스화하고 나머지는 외부로 뺀다
        - 자신의 책임이었던 것을 다른 클래스에게 위임
        - 다른 클래스에 중복 부분만 클래스화한 클래스를 멤버 변수로 구성
    - 이양을 우선
        - 상속은 부모 자식 관계가 아니면 재사용 불가능
        - 상속은 부모 클래스가 변경될 때 자식 클래스에도 영향
        - 이양은 언제나 독립적으로 재사용 가능

- 기본 자료형으로 구성된 멤버 변수 클래스화
    - int, float과 같은 기본 자료형은 모두 클래스 후보
    - 기본 자료형은 어떤 역할을 보유
        - 해당 역할을 수행하는 클래스로 기본 자료형을 클래스화
    - 멤버 변수가 너무 많은 경우에는 관련성있는 멤버 변수로 모아 클래스화
    - 기본 자료형을 최소한으로 하는 것이 좋은 클래스를 만드는 요령

- 함수의 매개 변수 클래스화
  - "함수 - 작은 함수의 필요성"에서 설명
  - 클래스화하면 좋은 변수들을 따로 사용하면 함수의 매개 변수가 많아지는 문제 발생

- 컨테이너 클래스화
  - 컨테이너 조작은 반복문을 수반하므로 복잡해지기 쉬운 부분
    - 전용 클래스로 컨테이너를 감싸여 부적절한 조작 방지
  - Example : Game 클래스 내부에서 Particle 배열을 반복문을 통해 제어하는 상황
    - Game 클래스에서 Particle 배열과 관련된 부분을 모두 ParticleManager로 이동
    - ParticleManager의 각 메소드에서 Particle 배열을 조작
      - 각 반복문 처리를 STL 알고리즘 함수로 변경
      - 중복된 for_each 함수는 각 요소를 순회하는 사용자 정의 each 함수를 작성
  - 복잡한 컨테이너 제어는 해당 컨테이너 관리 클래스로 캡슐화
    - i.e 컨테이너 조작을 위한 전용 클래스 사용

[6-3.클래스 - 작은 클래스]

- 작은 클래스의 필요성
  - 프로그램의 말단에서 사용하는 작은 클래스
  - 작은 클래스가 없으면, 같은 조건식/계산식이 중복되어 등장
  - 작은 클래스를 사용한 추상화 레벨 up
  - 객체 지향 프로그래밍 : 작은 클래스를 조합하여 프로그램을 만드는 것

- 게임 프로그래밍에서 자주 사용되는 작은 클래스
  - Timer 클래스
  - Score 클래스
  - 범위 클래스
  - 2D 벡터 클래스
  - 정수형 2D 벡터 클래스
  - 사각형 클래스
  - 수학 관련 클래스
  - typedef 활용

- Timer 클래스
  - 일정 시간마다 특정 처리를 실행하는 코드는 게임 프로그램에서 자주 사용

    - 시간을 관리하는 Timer 클래스
        - 현재 시간
        - 특정 상태(타임 아웃)
        - 시간 업데이트 및 리셋 메소드
    - 시간이라는 float 변수를 클래스화해서 해당 변수가 시간이라는 의도를 명확하게 전달 가능

- Score 클래스
    - 게임 스코어를 관리하는 클래스
        - 게임 스코어의 범위 제한
        - 게임 스코어의 +/- 연산 메소드
    - 게임 스코어라는 int/float 변수를 클래스화해서 해당 변수가 게임 스코어라는 의도 명확하게 전달 가능

- 범위 클래스
    - 상한값과 하한값을 처리하는 클래스
        - 생성자를 통해 상한값, 하한값 관리
        - 조건식에서 다룬 Clamp 및 WrapAround 메소드
    - 범위 클래스의 멤버 함수는 모두 const 멤버 함수
        - 범위 객체 변수에 const를 붙이면 const 멤버 함수만 호출 가능

- 2D 벡터 클래스
    - 2D 벡터 클래스는 대부분의 게임 개발 전용 라이브러리에서 지원
    - 캐릭터의 좌표 또는 이동량 계산 등에 사용
    - 중요 메소드
        - 내적, 외적 계산
        - 길이, 거리 계산
        - 정규화
        - 연산자 오버로딩을 통한 벡터 간 덧셈/뺄셈, 스칼라와 곱셈/나눗셈 계산
        - 벡터 간 대소 관계
        - 벡터 간 각도 계산
        - 회전 계산
    - 중요 벡터 형태는 static const 멤버 변수로 저장
        - zero, 상하좌우 벡터

- 정수형 2D 벡터 클래스
    - 타일 기반 게임(체스/퍼즐)에서는 정수 자료형의 2D 벡터 클래스가 자주 사용

    - 정수 자료형 2D 벡터 클래스는 대부분의 게임 라이브러이에 없어 따로 구현 필요

- 사각형 클래스
    - 2D 게임의 충돌 판정 및 영역 내부 판정에 사주 사용
    - 멤버 변수
        - 사각형의 좌상단, 우하단 좌표
    - 중요 메소드
        - 점이 사각형 내부에 있는지 판정
        - 사각형 간 중첩 여부 판정
        - 평행 이동
        - 크기 변경

- 수학 관련 클래스
    - 3차원 벡터 클래스, 행렬 클래스, 평면 클래스, 사원 수 클래스
    - 라이브러리에서 제공하는 경우 적절히 사용 가능
        - 라이브러리에 따라 최소한의 계산만 지원하는 경우 스스로 함수를 추가
        - 계산 전용 라이브러리가 C로 작성된 경우 C++ 연산자 오버로딩 구현 X
            - C++에서는 C언어 구조체에 연산자 오버로드 추가 가능

- typedef 활용
    - 작은 클래스를 만드는 것이 어려운 경우 typedef를 사용하여 사용자 정의 자료형 작성
    - 작은 클래스화가 코드를 복잡하게 만든다면 typedef를 사용하는 정도로만 끝낼 수 있음
    - typedef는 자료형 이름을 변환하는 것에 지나지 않음
        - 클래스처럼 변수에 행위를 부여하거나 제한을 추가하는 것은 불가능
        - 클래스화가 불가능한 경우 사용하는 최후의 수단

[2장 간단한 설계를 위한 원칙과 패턴]

- 객체 지향 설계의 기본적인 사고 방식
  - 원칙과 패턴을 이용하면 더 좋은 설계에 가까워질 것
  - 객체 지향의 사고 방식과 원칙을 설계 방침으로 삼아 클래스 설계
- 목차
  - 객체 지향의 설계 기본
    - 캡슐화, 응집도, 결합도
    - 상속과 이양의 관계
    - 객체 지향 설계 원칙
    - 디자인 패턴
  - 클래스 설계 요령
    - 클래스 설계의 기본
    - 클래스의 역할
    - 클래스의 책임
    - 클래스의 추상도
    - 클래스 결합
    - 추상 인터페이스 사용 방법
    - 그 밖의 주의점 또는 테크닉
  - 클래스 설계
    - 초급편
    - 중급편
    - 고급편

[1-1. 객체 지향의 설계 기본 - 캡슐화,응집도,결합도]

- 캡슐화, 응집도, 결합도
  - 보수성이 높은 클래스를 설계할 때 사용하는 기본 요소
  - 객체 지향 이외의 범위에서도 중요한 개념

- 캡슐화
  - 객체 내부의 변수 또는 구현 상세 내용을 사용자로부터 은폐하는 것
  - 캡슐화의 목적
    - 객체 내부의 상태 보호
    - 객체의 구현 상세 은폐
    - 즉, 객체 내부 주고를 신경 쓰지 않아도 쉽게 사용할 수 있는 클래스 설계가 중요
  - 멤버 변수를 private으로 하는 것만이 캡슐화가 아님
  - 클래스 사용자 입장에서는 해당 클래스가 블랙박스처럼 사용되도록 해야한다
  - 멤버 변수에 대한 getter/setter를 구현하는 것은 피해야 한다
    - 클래스의 내부 구조가 외부에 노출됨

        - setter는 사용자가 클래스 내부의 변수를 직접 조작할 수 있는 함수로 구현시 주의 필요
        - "직접 하지말고 명령하라"라는 객체 지향 설계 원칙을 따른다
    - 상속 관계일 경우 protect는 피해야 한다
        - 부모 자식 관계에서도 객체 내부 정보는 은폐해야 한다
        - private으로 가능한 부분은 모두 private으로 만들어라
    - 추상 인터페이스를 사용한 캡슐화
        - 추상 인터페이스
            - 순수 가상 함수로만으로 구성된 추상 클래스
            - 함수의 구현 또는 멤버 변수가 없음
        - 추상 인터페이스 사용자에게 구현하는 클래스의 상세 정보를 완전히 은폐 가능

- 응집도
    - 응집도의 정의
        - 클래스가 하나의 역할에 얼마나 집중하는지를 나타내는 척도
        - 크고 복잡하며 여러 역할을 수행하는 클래스는 응집도가 낮다
    - 클래스의 응집도를 높이기 위해서는 각 역할을 서로 다른 클래스에게 이양
        - 이양된 클래스들로 구성된 클래스는 "다른 클래스 제어 역할"이라는 하나의 역할에 집중
        - i.e 클래스를 작게 나누고 역할을 분담
    - 일반적으로 멤버 변수의 수가 많은 클래스는 여러 역할을 수행해야 하므로 응집도가 낮아짐
        - 클래스의 멤버 변수를 최소한으로 만들자

- 결합도
    - 결합도의 정의
        - 다른 클래스와의 연관 정도를 나타내는 척도
        - i.e 클래스끼리 얼마나 영향을 주는지 나타내는 척도
        - 다른 클래스와 완전히 독립적인 클래스는 결합도가 낮다
    - 소결합
        - 결합도가 낮은 상태
        - 소결합 클래스는 외부 변경에 영향을 받지 않아 재사용 및 테스트 간편
    - 결합도의 좋고 나쁨 판단
        - 해당 클래스의 단위 테스트가 얼마나 간단할지 판단
        - 외부에 의존하는 클래스는 단위 테스트가 어려움
    - 테스트 주도 개발(TDD)
        - 단위 테스트를 먼저 작성하고 클래스를 구현하는 방법

        - 테스트 기반 개발이기 때문에 테스트하기 쉬운 소결합 클래스 구현 가능

[1-2. 객체 지향의 설계 기본 - 상속과 이양의 관계]

- 클래스의 관계
    - 상속
        - 부모의 힘을 사용하는 관계
        - 부모 클래스를 상속받아 구현
    - 이양
        - 다른 사람의 힘을 사용하는 관계
        - 다른 클래스를 멤버 변수로 구성하여 구현

- 상속보다는 이양을 사용
    - 상속 관계는 부모 클래스와 자식 클래스가 밀접하게 결합되어 있기 때문에 유연성이 떨어짐
        - 부모 클래스의 변경이 자식 클래스에게 영향
    - 추상 인터페이스를 사용한 이양 - 스트래티지 패턴
        - 상속 관계에서 오버라이드하는 부분을 따로 빼내어 추상 인터페이스로 만든다
        - 오버라이드 부분은 추상 인터페이스를 상속받은 클래스에서 구현
    - 이양의 유연성과 보수성
        - 상속 관계는 컴파일 할 때 결합하기 때문에, 실행 중에는 클래스 변경이 불가능
        - 이양 관계(스트래티지 패턴)에서는 인터페이스에 의한 다형성에 의해 실행 중에도, 해당 객체를 변경 가능

- 기능 상속과 추상 인터페이스 구현
    - 상속은 부모 클래스로부터의 "기능 상속"
        - Java 및 C#의 exteds 키워드
        - C++은 다중 상속이 가능하지만 Java 및 C#은 단일 상속만 가능
    - 추상 인터페이스는 "역할 구현"
        - Java 및 C#의 implements 키워드
        - C++은 상속/구현 구분이 없음
        - Java 및 C#은 implements를 통해 다중 상속과 비슷한 역할 수행

[1-3. 객체 지향의 설계 기본 - 객체 지향 설계 원칙]

- SOLID 원칙

    - 단일 책임 원칙(SRP)
    - 개방 폐쇄 원칙(OCP)
    - 리스코프 치환 원칙(LSP)
    - 인터페이스 분리 원칙(ISP)
    - 의존 관계 역전 원칙(DIP)
    - (+α) 데메테르 법칙

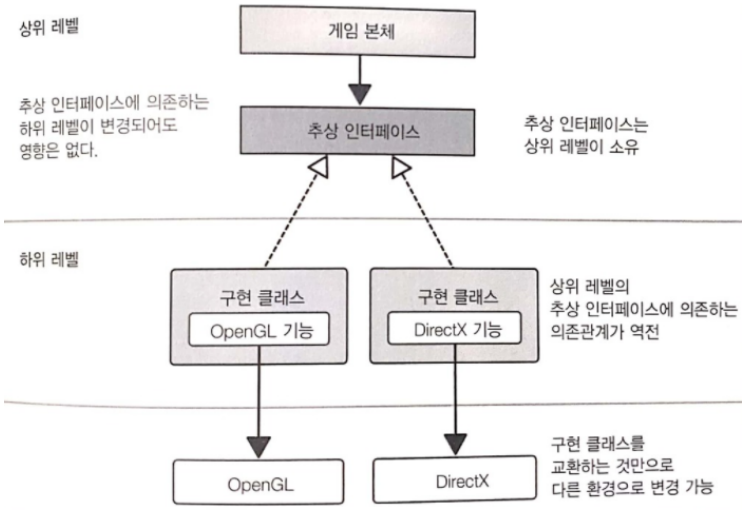
- 단일 책임 원칙(SRP, Single Responsibility Principle)
    - 클래스를 변경해야 하는 이유는 한 가지여야 한다
      i.e 하나의 클래스는 하나의 책임만 가져야 한다
    - 이 원칙은 클래스 응집도와 관련

- 개방 폐쇄 원칙(OCP, Open-Closed Principle)
    - 소프트웨어의 구성 요소는 확장에 관해서는 열려있어야 하고, 변경에 대해서는 닫혀있어야 한다
    - 변화하지 않는 부분(닫힌 부분-변경)과 변화하는 부분(열린 부분-확장)을 분리해야 한다
    - Example : 클래스 상속
        - 부모 클래스는 닫힌 부분이고, 자식 클래스는 열린 부분
        - 부모 클래스를 변경하지 않아도, 자식 클래스의 행동을 변화시키면서(override) 확장 가능
    - Example : 추상 인터페이스
        - 추상 인터페이스 클래스는 닫힌 부분이고, 구현하는 클래스는 열린 부분
        - 추상 인터페이스 구현 부분만 교환하면, 사용자 측 클래스의 행동을 간접적 변경 가능

- 리스코프 치환 원칙(LSP, Lisokv Substitution Principle)
    - 파생 자료형은 기본 자료형과 치환할 수 있어야 한다
      i.e 부모 클래스로 치환한 상태에서도 정상 작동해야 한다
    - 정확한 상속 관계로 구현되어 있다면 원칙이 위배되지 않는다
        - 자식 클래스로 과도한 다운캐스트해야 하는 경우 위배
        - 클래스는 항상 부모 클래스 그대로의 상태에서도 작동해야 한다
    - 상속 관계가 복잡해지면 LSP를 위반하는 클래스가 만들어질 가능성이 높음
        - 다중 상속 또는 깊은 상속 관계를 만드는 일은 되도록 피해야 한다
        - 간단하면서 근복적인 해결 방법은 상속에 의존한 설계를 하지 않는 것

- 인터페이스 분리 원칙(ISP, Interface Segregation Principle)
  - 클라이언트가 사용하지 않는 멤버 함수의 의존을 클라이언트에 강요하면 안된다
    i.e 클래스 사용자에게 불필요한 인터페이스는 공개하지 마라
  - 멤버 함수가 많은 큰 클래스일수록 ISP를 위반
    - 멤버 함수가 많으면 그만큼 복잡한 역할을 수행할 가능성 높음
    - 클래스는 최소한의 멤버 함수로만 구성
  - Example : C++ STL의 파일 스트림 클래스
    - 읽기 전용 인터페이스를 가지는 ifstream 클래스
    - 쓰기 전용 인터페이스를 가지는 ofstream 클래스
  - 다양한 애플리케이션에서 범용적으로 사용할 수 있게 설계된 클래스는 다수의 멤버 함수를 보유
    - 이러한 클래스를 사용하는 경우 필요한 멤버 함수만으로 구성된 전용 클래스를 따로 구현하여 간접적으로 사용하는 것이 좋음

- 의존 관계 역전 원칙(DIP, Dependency Inversion Principle)
  - 상위 모듈은 하위 모듈에 의존하지 않고, 두 모듈 모두 별도의 추상화된 것에 의존한다
    - 상위 모듈 : 사용하는 측 모듈
    - 하위 모듈 : 사용되는 측 모듈
  - 보통 상위 레벨 모듈은 하위 레벨 모듈을 사용하여 만든다
    - 상위 레벨은 하위 레벨에 직접 의존하므로, 하위 레벨이 변경되면 상위 레벨도 변경된다
      i.e 상위 모듈과 하위 모듈은 결합도가 높다
    - 상위 모둘과 하위 모듈의 관계를 역전시켜 결합도를 낮춰야 한다
  - 의존 관계 역전 방법
    - 사용자 측에 있는 상위 레벨 모듈의 요구에 맞춰 추상 인터페이스 작성
    - 작성된 추상 인터페이스를 통해 하위 모듈을 사용
    - 이때 추상 인터페이스는 반드시 상위 모듈이 소유
    - 추상 인터페이스를 거쳐 하위 모듈을 사용하면 하위 모듈과 직접적인 결합 피하기 가능
      => 하위 레벨 모듈의 변경 및 교환이 쉬어진다
  - Example : 하위 모듈인 OpenGL과 DirectX에 대한 추상 인터페이스



    - 상위 레벨이 되는 사용자 측 요구에 맞게 추상 인터페이스를 작성하면, 최소한의 멤버 함수를 가진 이상적인 추상 인터페이스 설계 가능
        - 인터페이스 분리 원칙을 위반하지 않는다
    - 의존 관계 역전 원칙을 사용하면 상위 레벨과 하위 레벨이 완전히 분리
    - 프로그램 전체를 계층화하고 분리할 때 의존 관계 역전 원칙 사용
        - 추상 인터페이스를 만들 때는 사용자라는 상위 관점에서 작성
    - 구체적인 것이 아닌 추상적인 것에 의존

- 데메테르 법칙 (최소 지식의 원칙)
    - 직접적인 클래스(직접적인 친구)와만 관련
        - 자기 자신
        - 자신이 가지는 클래스
        - 매개 변수로 전달한 클래스
        - 멤버 함수 내부에서 실체화한 클래스
    - 친구의 친구 또는 Singleton 패턴을 사용한 외부 클래스와 결합하면 데메테르 법칙 위반
        - 불충분한 클래스 책임 분할이 데메테르 법칙을 위반하는 원인
        - "직접하지 말고 명령하라"라는 개념을 의식하고 설계
    - 데메테르 법칙에 딱 맞게 설계하려다 보면 오히려 복잡해지는 경우도 있음
        - 참고를 위한 방침

[1-4. 객체 지향의 설계 기본 - 디자인 패턴]

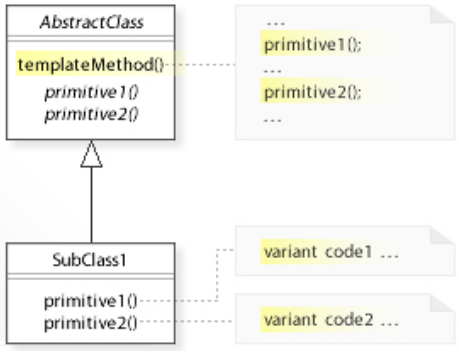
- 디자인 패턴

- 과거에 작성한 객체 지향 프로그램에서 반복해 나오는 형태와 그 해결 방법에 이름을 붙여 패턴화 한 것
- Reference : GOF의 디자인 패턴

구분	생성 패턴	구조 패턴	행위 패턴
클래스	Factory	Adapter(class)	Interpreter Template Method
객체	Prototype Builder Abstract Factory Singleton	Adapter(object) Bridge Composite Decorator Façade Flyweight Proxy	Command Mediator Memento Iterator Observer State Strategy Visitor

- 장래의 확장성 또는 재사용성을 높이기 위한 패턴이 많음

- 게임 엔진 및 클래스 라이브러리에서 자주 사용되는 패턴
  - Template Method 패턴
  - Strategy 패턴
  - State 패턴
  - Composite 패턴
  - Iterator 패턴
  - Observer 패턴
  - Singleton 패턴

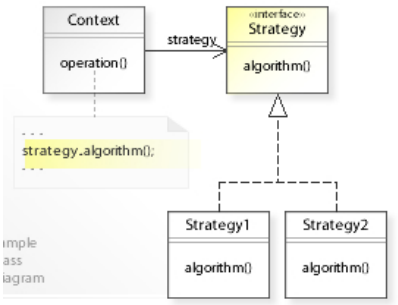
- Template Method 패턴
  - 부모 클래스에서 정형화한 처리 과정을 정의하고, 처리가 다른 부분을 자식 클래스로 구현하는 패턴
    - 처리 과정의 순서는 부모 클래스에서 정한다
    - 처리 과정를 구성 하는 함수들은 자식 클래스에서 정의
    - 

  - 상속을 사용한 재사용의 고전적인 패턴
  - Example
    - 부모 클래스

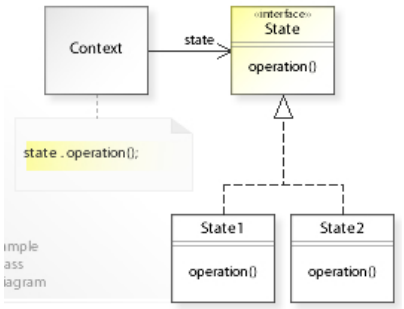
```cpp
// 템플릿 메소드 패턴의 제일 기본적인 구조
class TemplateMethodPattern {
    // 템플릿 메소드를 가지고 있는 제일 기본적인 클래스
public:
    virtual void templateMethod() final {
        // 템플릿 메소드 : 알고리즘을 위한 템플릿
        primitiveOperation();
        concreteOperation();
    }

    virtual void hook() {
        // 아무것도 안하는 메소드
        return;
    }

protected:
    virtual void primitiveOperation() = 0; // 상황에 따라 바뀌는 단계 : 구현 필수
    virtual void concreteOperation() {
        // 모든 알고리즘안에 있는 공통된 단계
        std::cout << "concreteOperation()" << std::endl;
    }

};
```

- 자식 클래스

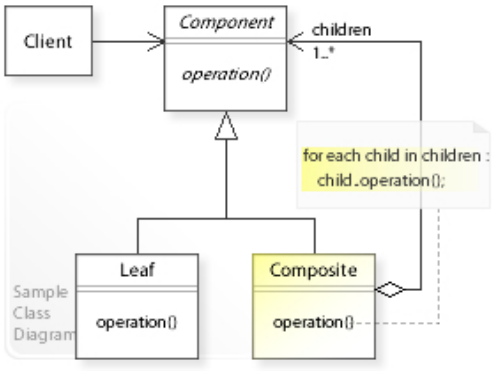
```cpp
// 특정 알고리즘을 위한 구현
class myAlgorithm : public TemplateMethodPattern {
protected:
    void primitiveOperation() override {
        std::cout << "myAlogorithm : primitiveOperation()" << std::endl;
    }
};
```

- Strategy 패턴
  - 알고리즘이 변화하는 부분을 클래스화해서 교환할 수 있게 하는 패턴
    - 교환할 수 있는 알고리즘을 구현한 클래스가 "Strategy(전략)"에 해당
    - 

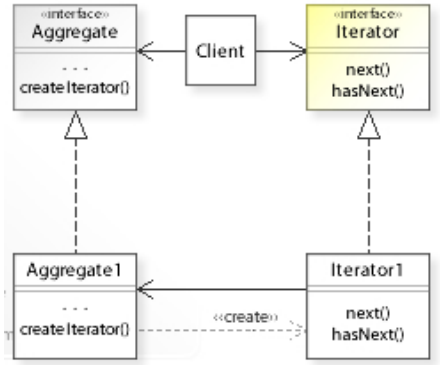
  - Example : 난이도에 따라 AI 플레이어의 행동이 다르다
    - Strategy 패턴을 적용하지 않은 경우에는 switch/if 조건문을 사용하여 해당 난이도에 대한 AI 플레이어 행동에 대한 코드를 만들어야 한다
      => 난이도가 많아질수록 분기에 대한 처리가 많아짐
    - Strategy 패턴에서는 AI 플레이어에 대한 인터페이스를 만들고, 난이도 별 AI 플레이어 구현 클래스를 만든다
  - switch 조건문이 나온다면 Strategy 패턴으로 해결할 수 없는지 확인
  - 추상 인터페이스를 사용해서 구현 부분을 교환하는 고전적인 패턴

- State 패턴
  - 객체가 가지는 여러 가지 상태를 클래스화하는 패턴
    - 어떤 행동(method)가 상태 별로 다를 때 해당 행동을 멤버 변수로 갖는 클래스를 만드는 패턴
    - 

  - Example : 캐릭터의 상태(대기,이동,공격)에 대해서 특정 행동이 다름
    - State 패턴을 적용하지 않은 경우에는 switch/if 조건문을 사용하여 해당 상태에 대한 행동에 대한 코드를 만들어야 한다
      => 상태가 많아지거나 변경되는 경우 분기에 대한 처리가 많아짐
    - State 패턴에서는 State에 대한 인터페이스를 만들고, 각 상태에 대한 구현 클래스를 만든다
  - State 패턴 vs Strategy 패턴
    - 구현 방법 자체는 비슷
    - State 패턴은 상태에 따라 행위가 달라지고, Strategy 패턴은 동일한 결과에 과정을 다르게 교체하여 도출한다
    - 즉, Strategy 패턴은 한가지 작업을 사용하는데 사용할 수 있는 알고리즘이 여러개인 경우 사용하는 패턴

- Composite 패턴
  - 트리 구조로 재귀적인 데이터를 만든 객체를 관리하는 패턴
    - 재귀적인 데이터는 Node 클래스를 바탕으로 상속해서 생성
    - 이때 Node 클래스는 여러 자식 노드를 관리 가능
    - 

  - Composite 패턴은 객체 관리 외에도 씬 그래프라 불리는 그래프 객체 처리에 사용

- Iterator 패턴
  - 컨테이너 클래스 내부 요소에 순서대로 접근하는 방법을 제공
    - 컨테이너 클래스 내부의 자료 구조를 은폐한 채 쉽게 접근하는 방법 제공
    - STL 컨테이너인 vector,list,map 등의 클래스도 Iterator 패턴 사용
      => 자료구조(컨테이너)에 관계없이 STL 알고리즘 함수 적용 가능
    - 

  - Example :
    - Iterator 인터페이스

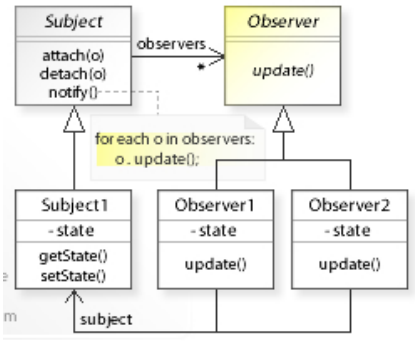
```
template <typename T>
class Iterator {
public:
    virtual bool hasNext( ) = 0;
    virtual T next( ) = 0;
};
```

    - 컨테이너 인터페이스

```
// 메뉴 안에 들어 있는 항목들
class MenuItem {
protected:
    std::string name;
    std::string description;
    bool vegetarian;
    double price;

public:
    MenuItem(std::string name="NULL", std::string description= "NULL",
             bool vegetarian=0, double price=0.)
        : name(name), description(description), vegetarian(vegetarian), price(price)
        {   }

    std::string getName( ) {
        return name;
    }

    std::string getDescription( ) {
        return description;
    }

    bool isVegeterian( ) {
        return vegetarian;
    }

    double getPrice( ) {
        return price;
    }
};
```

    - 컨테이너 객체

```
class DinerMenuIterator : public Iterator<MenuItem> {
protected:
    MenuItem* items;
    int curpos;

public:
    DinerMenuIterator(MenuItem* items) :curpos(0) {
        this->items = items;
    }

    bool hasNext() {

    }
};
```

- Observer 패턴
  - 어떤 객체의 상태 변화를 다른 객체에 통지해주는 패턴
    - 이벤트 통지를 받는 대상이 Observer(관측자, eventListener)
    - 이벤트를 통지하는 클래스에서는 관측자 인터페이스 구현체를 멤버 변수로 가지고 있으며, 특정 이벤트에서 해당 관측자 객체의 함수를 호출
    - 

  - STL의 function 클래스를 이용한다면, 관측자 클래스가 필요하지 않음
    - 이벤트 통지를 받는 처리를 람다식 내부에 직접 작성 가능
  - 언어의 일부 기능으로 Observer 패턴 지원
  - GUI를 다루는 클래스 라이브러리에서 필수로 쓰이는 패턴

- Singleton 패턴
  - 객체 인스턴스화를 1개로 제한하고, 전역에서 접근할 수 있게 하는 패턴
  - 간단한 구현 방법
    - 생성자 및 대입 연산자를 private으로 지정하여 외부에서의 인스턴스화 방지
    - static 멤버 함수를 이용하여 인스턴스 추출
      - 이 함수에서 클래스의 인스턴스화를 한 번만 수행하도록 한다
      - 인스턴스 객체를 static 변수로 설정함으로써 한 번만 생성
  - Singleton 패턴 남용 금지

        - Singleton 클래스와 결합하는 클래스는 데메테르 법칙 위반
        - 클래스 독립성이 떨어져서 단위 테스트 어려움
        - 멀티스레딩에서는 mutex 클래스 등을 사용한 배타 제어 필요
        - 전역 변수와 같은 단점이 있음을 인식
- 디자인 패턴의 여러 가지 구현 방법
    - 하나의 디자인 패턴도 구현 방법은 여러가지
        - 구현 방법이 다르다 해도 패턴 자체의 본질은 달라지지 않는다
    - STL의 function 클래스를 추상 인터페이스 대신 사용하는 방법
        - function 클래스에는 람다식, 함수 포인터, 함수 객체를 넣을 수 있음
        - 추상 인터페이스를 사용할 때보다 유연성이 높음

- 디자인 패턴 활용
    - 프로그램의 규모가 커지고 복잡해질수록 디자인 패턴을 활용하게 됨
    - 기존 엔진이나 라이브러리 내부에서 디자인 패턴 찾아보기
        - 디자인 패턴을 효과적으로 활용할 수 있는 힌트
        - 패턴의 구조에 대한 이해 가능
        - 디자인 패턴이 어떤 곳에서 어떠한 형태로 사용되는지 아는 것이 중요

- 디자인 패턴 남용
    - 소규모 프로그램에서의 남용 주의
        - 디자인 패턴으로 인한 재사용성과 확정성은 높아지지만, 클래스 개수가 많아지고 구조가 복잡해지는 경향이 있음
        - 단순한 소규모 프로그램이 복잡해지는 주객전도 현상 발생
    - 무리한 디자인 패턴 사용을 조심하고, 객체 지향 설계 원칙을 중심으로 클래스 설계

[2-1. 클래스 설계 요령 - 클래스 설계의 기본]

- 클래스 설계는 현실 세계의 업부 분담과 비슷
    - 각 클래스의 역할을 정하고 업무 책임을 분담하는 작업

- 클래스의 이름은 담당하는 업무의 역할
    - 1개의 클래스가 담당하는 업무가 많아지면 역할도 불분명
    - 역할이 불분명하면 클래스의 이름이 길어지고 어려워짐

- 클래스는 하나의 역할에 대해 하나의 책임을 갖게 설계
    - 클래스 이름이 간단해지고 의도가 명확

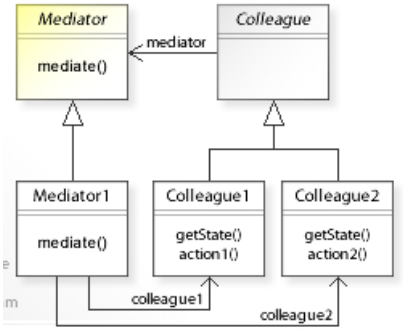
- 클래스의 역할과 책임에 집중
    - 자료 구조와 알고리즘은 객체를 작동시키는 수단

[2-2. 클래스 설계 요령 - 클래스의 역할]

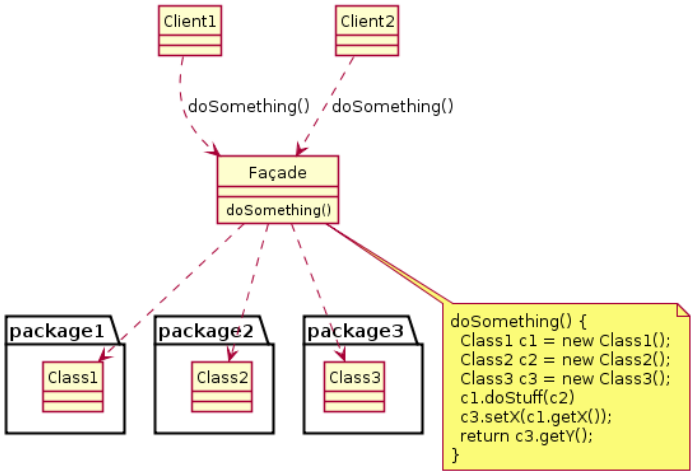
- 역할에 따른 클래스 분류
    - 작업 역할 클래스와 관리 역할 클래스
    - 중재 역할 클래스
    - 창구 역할 클래스
    - 생성 역할 클래스
    - 전용 클래스와 범용 클래스

- 작업 역할 클래스와 관리 역할 클래스
    - 작업 역할 클래스
        - 구체적인 구현을 하는 말단 클래스
        - 1개의 역할에 대한 책임을 가지는 클래스
    - 관리 역할 클래스
        - 작업 역할 클래스를 제어하는 클래스
        - 여러 개의 작업 역할 클래스를 엮는 책임을 가지는 클래스
        - 실질적인 작업은 수행하지 않고, 작업자들을 조정하는 역할
    - 큰 클래스를 분할 할때 작업 역할 클래스를 만들어 책임을 이양

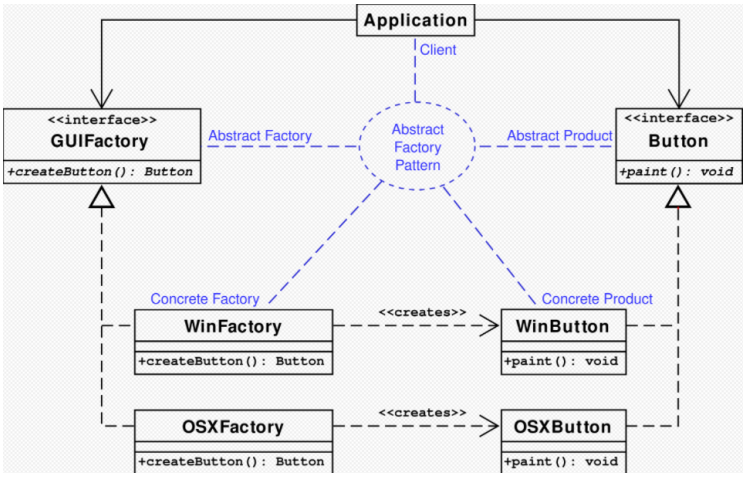
- 중재 역할 클래스
    - 객체들의 조정을 전문으로 담당하는 클래스
        - 작업 중재 자체가 해당 클래스의 역할
        - 중재자를 사용한 의존 관계 단순화 가능
    - 중재 역할 클래스를 사용하여 클래스의 결합 수 감소 효과
        - 중재 클래스가 중재하는 클래스들을 소결합 상태로 만들 수 있음
        - Mediator 패턴에 해당



- 창구 역할 클래스
  - 작업을 의뢰받고, 실질적인 작업을 다른 클래스에 전달하는 담당하는 클래스
    - 하위 레벨 클래스들과 상위 레벨 클래스들 사이의 창구 역할
    - 중재 역할 클래스는 클래스 간 역할을 조정하고, 창구 역할 클래스는 모듈(클래스 모음) 간 역할을 조정한다
    - 중재 역할과 마찬가지로 클래스의 결합 수를 줄이는 효과
    - 내부 조직 구조를 은폐하는 효과
    - i.e 외부와 내부 사이의 인터페이스 역할
  - Pacade 패턴
    - 창구 역할 클래스는 Pacade 패턴에 해당
    - 서브시스템의 일련의 인터페이스에 대해 통합된 인터페이스 제공
    - 고수준 인터페이스를 정의하여 서브시스템을 유연하게 사용



- 생성 역할 클래스
  - 객체 생성을 전문으로 수행하는 클래스
    - 생성할 객체의 변경 및 수정이 간단
    - 인스턴스화 할 객체들 간 연관 감소 효과
    - 객체 생성이 복잡한 과정이 필요한 경우 생성 역할 클래스를 사용하는 것이 좋음

- Abstract Factory 패턴
  - 객체 생성을 전문으로 담당하는 클래스를 생성하는 디자인 패턴
  - 생성 역할 클래스(공장)를 추상화하는 패턴
  - 추상화를 통해 생성 역할 클래스(공장) 자체의 교환 가능
  - 

- 전용 클래스와 범용 클래스
  - 특정 도메인에서만 사용하는 "전용 클래스"와 재사용을 목적으로 하는 "범용 클래스"를 구별하여 설계
  - 전용/범용 클래스 구분
    - 특정 애플리케이션 전용
    - 특정 애플리케이션 범용
    - 게임 애플리케이션 범용
    - 다양한 애플리케이션 범용
  - 코드의 의도를 파악할 수 있게 적절히 조절하여 클래스 설계
    - 어중간한 범용 클래스를 만들거나 범용 클래스만 사용하여 코드를 작성하면 의도 파악 어려움
  - 전용/범용 클래스 설계 방법
    - 처음에는 애플리케이션 전용 클래스를 생성하여 코드의 의도 전달
    - 전용 클래스를 생성할 때는 범용 클래스를 사용
    - What 부분은 전용 클래스, How 부분은 범용 클래스
  - 클래스 라이브러리로 제공되는 클래스는 범용 클래스
    - 필요한 인터페이스만 추출한 전용 클래스를 이용한 간접적인 라이브러리 사용
      i.e 인터페이스 분리 원칙
  - 어쩔 수 없이 설계된 문제가 있는 클래스의 경우에도 전용 클래스를 통해 간접적으로 사용하면 설계가 안전

[2-3. 클래스 설계 요령 - 클래스의 책임]

- 클래스에 명확한 역할과 책임을 갖게 하는 것이 중요
  - 클래스의 역할을 제대로 찾아도, 책임을 제대로 이양하지 못하면 안된다
  - 사용자 측의 코드가 간단해지도록 담당 클래스에 책임을 제대로 이양해야 한다

[2-4. 클래스 설계 요령 - 클래스의 추상도]

- 클래스 추상도
  - 추상도가 높을수록 코드가 단순해져서 수정 및 변경에 유연하게 대응 가능
  - 추상도를 높이려면 "문제의 본질 파악"이 필수
  - 클래스 또는 멤버 함수의 추상도를 높이면 구현 상세가 캡슐화되고 코드가 단순해 진다

- 클래스 사용자 측 시점에서 이상적인 인터페이스를 고민하는게 중요
  - 클래스 사용자 측의 시점부터 탑다운 방식으로 생각하는 것이 추상도 높은 클래스를 설계하기 쉽다

[2-5. 클래스 설계 요령 - 클래스 결합]

- 클래스 결합의 좋고 나쁨을 결합 수로만 판단 불가능
  - 결합 수가 1이어도 나쁜 결합일 수 있음
  - 서로 전혀 연관 없는 클래스가 결합된 경우 데메테르 법칙 위반

- 클래스의 나쁜 결합을 해결하기 위해서 추상 인터페이스 사용
  - 추상 인터페이스를 통한 간접적인 결합
  - 추상 인터페이스에 의해 구현 수단을 제공하는 클래스와의 관계를 소결합으로 만들고 구현 수단의 교환 가능
  - 추상 인터페이스를 통해 구체적인 구현 수단과의 의존 관계를 클래스 외부로 빼낼 수 있음
  - 구현 수단이 변화하는 부분은 추상 인터페이스화 대상 후보

- 멤버 함수를 하나만 가지는 추상 인터페이스는 STL의 `function` 클래스 사용 가능

[2-6. 클래스 설계 요령 - 추상 인터페이스 사용 방법]

- 추상 인터페이스 활용 방법
  - 클래스 공통화
  - 구현 수단 교환
  - 변화하기 쉬운 부분 분리
  - 부적절한 결합 관계 분리
  - 패키지 경계 분리

- 클래스 공통화
  - 정적 언어에서의 다른 형식의 클래스들의 일괄 관리 가능
  - 공통 추상 인터페이스 형식을 바탕으로 구현된 서로 다른 클래스들을 일괄적으로 관리

- 구현 수단 교환
  - Strategy 패턴
  - 구현 수단을 교환하고 싶은 경우 추상 인터페이스 사용
  - 사용자 측과 구현 측의 중간에 추상 인터페이스를 거치면서, 사용자 측 코드를 변경하지 않아도 구현 수단 변경 가능

- 변화하기 쉬운 부분 분리
  - 개방-폐쇄 법칙
  - 변화하기 쉬운 부분을 추상 인터페이스화하여 클래스 외부로 빼낼 수 있음

- 부적절한 결합 관계 분리
  - 직접 결합되어서는 안 되는 클래스를 분리할 때 사용
  - 중재 역할 또는 창구 역할을 담당하는 추상 인터페이스를 작성하여 클래스 간 직접적 결합 분리

- 패키지 경계 분리
  - 패키지
    - 프로그램 규모가 커지면 관련 있는 클래스를 그룹화해서 패키지(이름 공간, 모듈)로 정리
    - 다른 패키지에 있는 클래스와 직접 결합하면 변경에 영향을 받을 가능성 존재
  - 패키지를 독립시키고 외부 변경의 영향을 피하고 싶을 때 추상 인터페이스 사용
    - 의존 관계 역전의 원칙
    - 패키지 내부의 클래스는 추상 인터페이스를 거쳐 외부 클래스를 사용

- 추상 인터페이스는 사용자 측 배치
  - 사용자 측의 내부 클래스로 만들 수 있으면 내부 클래스로 정의
  - 내부 클래스를 만들지 못하는 상황에서는, 사용자의 클래스가 있는 패키지 안에서 정의

- 추상 인터페이스는 분리와 교환을 위한 도구
  - 주로 "결합 관계 분리" 또는 "구현 교환"을 할 때 사용하는 도구
  - 추상 인터페이스의 과도한 남용은 피해야 한다
    - 추상 인터페이스로 인해 클래스 수가 증가하고, 전체 프로그램이 복잡해진다

[2-7. 클래스 설계 요령 - 그 밖의 주의점 또는 테크닉]

- C++ 클래스의 특성을 사용한 테크닉
  - 생성자로 완전한 상태 만들기
  - 멤버 함수 호출 순서와 관련된 대처 방법
  - 초기화는 생성자, 뒤처리는 소멸자로 시행
  - 구현 상세를 은폐하는 이름 붙이기

- 생성자로 완전한 상태 만들기
  - 객체는 생성자가 호출된 시점에 완전한 상태를 생성해야 한다
    - 별도의 초기화 함수 호술이나 setter를 통한 멤버 변수 설정 방식은 지양
  - 호출 순서에 제한이 있는 클래스는 사용자에게 부담이 되고 찾기 어려운 버그의 원인
  - 클래스 생성자에서 모든 초기화 작업을 수행하고, 소멸자에서 완전한 종료 처리

- 멤버 함수 호출 순서와 관련된 대처 방법
  - 멤버 함수의 호출 순서에 제약이 있다면, 순서대로 호출할 것을 설계적으로 보장해야 한다
    - 호출 순서를 보장하려면 호출 순서를 제어하는 클래스를 만들어야 한다
    - 제어 역할 클래스를 통해 작동하도록 클래스를 다시 설계
  - Example : 호출 순서에 제약이 있는 Game 클래스
    - GameRunner 클래스가 Game 클래스의 각 멤버 함수의 호출 순서를 지킨다
    - GameRunner 클래스는 Game 객체를 멤버 변수로 가지고 있는다

        - GameRunner 클래스에서 게임 자체에 대한 역할을 Game 클래스에 이양하는 방식
    - 상속을 통한 제어 역할 클래스는 Template Method 패턴, 이양을 통한 제어 역할 클래스는 Stratedy 패턴
        - Template Method 패턴 대부분은 Strategy 패턴을 사용한 이양 스타일로 변경 가능

- 초기화는 생성자, 뒤처리는 소멸자로 시행
    - 초기화 처리 및 종료 처리가 필요한 API 또는 클래스는 반드시 생성자와 소멸자를 통해 해당 처리를 확실하게 해야 한다
    - Example : C언어의 파일 출력 함수의 클래스화

```
class TextFileWriter {
public:
    TextFileWriter(const std::string& fileName) : file_(NULL) {
        file_ = fopen(fileName);
    }

    ~TextFileWriter() {
        fclose(file_);
    }

    void write(const std::string& text) {
        fputs(text.c_str(), file_);
    }

private:
    FILE* file_;
};
```

        - 실제로 STL의 fstream 계열 클래스가 생성자 및 소멸자에서 수행
    - 파일 또는 텍스처(OpenGL) 등의 리소스 관리는 클래스의 생성자와 소멸자에서 구현
      i.e 생성자와 소멸자로 리소스 관리(RAII, Resource Acquisition is Initialization)

- 구현 상세를 은폐하는 이름 붙이기
    - 클래스의 멤버 함수 이름은 구현 방법의 상세를 은폐하는 이름으로 설정
        - 구현 방법이 달라졌을때, 함수의 이름과 구현의 불일치가 발생
        - 함수의 목적으로 이름을 사용
    - Exmample : 사망 판정 함수
        - Before : 플래그 변수로 사망 판정을 수행

```
bool getDeadFlag() const {
    return isDeadFlag;
}
```

        - 문제점 : 플래스 변수가 아닌 다른 방식으로 사망 판정을 수행할 때 함수명과 구현 방법에 불일치

        - After : 함수의 목적 - 사망 판정을 그대로 함수 이름으로 사용

```
bool isDead() const {
    return isDeadFlag;
}
```

[3-1. 클래스 설계 - 초급편]

- 대략적인 클래스 조사
    - 대략적으로 클래스 리스트업
        - 대상의 문제가 어떤 구성 요소로 이루어 졌는지 떠오르는 대로 리스트업
    - 클래스 리스트업 한 후 클래스 이름만으로 간단한 클래스 다이어그램 작성
        - 클래스 간 연계 구상
    - 대략적인 설계 후 코드를 작성
        - 코드를 작성하면서 재설계
        - 초기 단계의 설계에 집착 X

- 사용자 관점에서 멤버 함수의 사양 결정
    - 초기 단계부터 public 멤버 함수의 사양 고려
        - 해당 함수의 역할과 책임이 중요
    - 사용자의 관점에서 구현 상세는 보이지 않고, 직관적인 이름을 사용

- 멤버 함수 구현
    - 멤버 함수의 사양이 결정되면 구현 시작
        - 처음에는 큰 함수가 만들어져도 괜찮다
        - 중요한 것은 일단 제대로 작동하는 코드 작성
    - 제대로 작동하게 되면 함수에 대한 리팩토링 진행
        - 큰 멤버 함수를 작은 private 멤버 함수로 만들어 분할
        - 각 멤버 변수의 행동 하나하나 작은 함수로 분할하는 것이 요령
    - 멤버 함수를 분할하면서 클래스의 책임 관계가 명확
        - 책임이 여러 개일수록 멤버 함수의 수 증가
        - 멤버 함수가 많은 경우 클래스 분할 고려

- 역할과 책임을 생각하면서 클래스 분할
    - 멤버 함수 분할 기반으로 클래스 분할
        - 멤버 변수 하나하나가 클래스의 후보
        - 멤버 함수를 분할하다 보면 클래스 후보가 보인다

    - "단일 책임 원칙"을 바탕으로 외부로 뺄 수 있는 책임이 있다면 다른 클래스로 이양
    - 작업 클래스와 관리 클래스를 파악하면 책임 나누기가 쉬어짐

- 결합을 생각
    - 부적절한 클래스와의 결합이 없는 지 확인
        - 결합의 좋고 나쁨은 해당 클래스의 단위 테스트가 얼마나 간단할지 생각해보자
    - 부적절한 결합은 추상 인터페이스를 통한 간접적인 결합으로 변경

- 캡슐화, 응집도, 결합도 확인
    - 클래스의 응집도가 특히 중요
        - 응집도 높은 클래스는 역할과 책임이 명확하기 때문

[3-2. 클래스 설계 - 중급편]

- 문제 영역과 구현 영역 분리
    - 문제 영역(What)
        - 해결해야 할 문제의 핵심 부분
        - 특정 환경에 의존하지 않게 설계
    - 구현 영역(How)
        - OS, 그래픽, 사운드 등의 API를 사용해서 문제 영역의 구현 방법을 제공하는 부분
        - 작동 환경에 의존 e.g)운영체제
    - 문제 영역과 구현 영역 사이에 추상 인터페이스를 두어 다른 환경에 이식 가능

- 문제 영역을 중심으로 설계
    - 중간 규모 이상의 프로그램을 설계할 때는 문제 영역 중심으로 설계
    - 문제 영역은 애플리케이션 본체가 "무엇인지"를 명시
    - 문제 영역에는 애플리케이션의 개요 설명을 작성하고, 구현 영역에는 애플리케이션의 상세 설명을 작성

- 추상 인터페이스로 분리
    - 문제 영역과 구현 영역을 분리하는 방법으로 객체 지향에서는 추상 인터페이스를 사용
        - 문제 영역이 요구하는 기능을 추상 인터페이스화하는 것

        - 구현 영역만 교환하면 문제 영역은 다른 환경에서도 이식 가능
    - 추상 인터페이스는 문제 영역 내부에 구현 by 의존 관계 역전 원칙
        - 구현 영역이 문제 영역에 맞춰야만 의존 관계 역전이 가능
        - 문제 영역에 기반을 두는 탑다운 관점으로 설계

- 복잡한 것은 여러 개의 계층으로 분리
    - 대규모의 복잡한 애플리케이션은 여러 개의 계층을 만들어 단계적으로 구현
    - 계층을 단계적으로 구현하고, 계층 사이에 추상 인터페이스를 둔다
    - 각 계층 클래스의 응집도가 높아지고, 하나 하나의 클래스를 축소하는 효과

[3-3. 클래스 설계 - 고급편]

- 주의 사항
    - 클래스 설계 이전의 단계에 대한 내용
        - 초기 단계의 설계 방침은 이후의 모든 설계에 영향
    - 경험이 없는 경우 이해 하기 힘듦

- 컨셉과 컨텍스트(?)
    - 좋은 설계를 하기 위해서는 컨셉(기본 개념)과 컨텍스트(문맥)을 고려해야 한다
    - 설계 의도를 명확하게 하려면 컨셉 필요
        - 컨셉 예시 : 보수성을 우선한다, 실행 속도를 우선한다
        - 클래스/함수 단위의 설계에도 컨셉 필요
    - 문제 해결에는 여러 가지 방법 존재
        - 컨셉이 명확하면 최대한 컨셉에 가까운 방법을 선택 가능

- 규모에 맞게 설계
    - 애플리케이션의 규모에 맞게 설계해야 한다
    - 충분한 설계를 하지 않고 대규모화되면 코드가 부너지고 유지보수가 어려워 진다

- 코드의 수명
    - 재사용할 가치가 있는 수명이 긴 코드는 시간을 투자하여 설계

        - 코드를 재사용할 수 있게 설계하는 것은 상당한 시간과 노력이 필요
    - 수명이 짧은 코드에 많은 시간을 투자하는 것도 문제

- 작업의 편의성
    - 코드의 단순함보다는 작업의 편의성에 우선
    - 컴퓨터 입장이 아닌 사람의 입장에서 작업

- 하나의 방법에 대한 집착 지양
    - Example : 객체 지향을 사용하는 것이 항상 가장 좋은 선택은 아니다
    - 애플리케이션 특징에 맞는 프로그래밍 언어와 개발 환경을 선택

- 최소한의 코드만 작성
    - 불필요한 코드 지양
        - 코드의 양에 비례해여 보수하는 시간 및 노력이 커진다
        - "어떻게 코드를 작성할지" < "어떻게 하면 코드를 작성하지 않을지"
    - 함수화 및 클래스화를 통한 중복 제거와 코드의 재사용
    - 한꺼번에 불필요한 코드를 제거해버릴 수 없는지 생각하는 것도 중요
    - 이미 있는 것을 다시 만들 필요가 없다
        - 기존 라이브러리나 미들웨어 검토
        - 개발자의 목적은 코드를 작성하는 것이 아니라 애플리케이션을 작성하는 것

[3장 소스 코드 품질 측정]

- 목차
    - 매트릭스 측정
        - 파일 또는 함수 단위 매트릭스
        - 객체 지향 매트릭스
    - 코드 클론 검출
    - 정적 코드 분석
    - 매트릭스 활용 방법
    - 툴 소개

[1. 매트릭스 측정]

- 매트릭스 측정을 하면 소스 코드의 품질 정령화 가능
  - 코드의 유지성 or 가독성을 일정한 기준에 따라 수치적 평가 가능

[1-1. 매트릭스 측정 - 파일 또는 함수 단위]

- 파일 단위 또는 함수의 단위로 측정할 수 있는 기본적인 매트릭스
  - 코드 줄 수
  - 주석 줄 수
  - 문장 수
  - 최대 중첩 수
  - 사이크로매틱 복잡도
  - Halstead 복잡도
  - 보수성 지표

- 코드 줄 수
  - 소스 코드의 줄 수를 의미
    - 일반적으로 주석이나 빈 줄은 포함하지 않는다
  - 소스 코드의 본질적인 크기를 측정할 때는 문장 수 매트릭스를 사용

- 주석 줄 수
  - 소스 코드의 주석의 줄 수를 의미
  - 주석없이 읽을 수 있는 코드가 제일 이상적이지만 실제로는 어려움
    - 주석 줄 수 가 매우 적은 소스 파일은 유지 보수가 어려움

- 문장 수
  - 문장의 개수를 모두 더한 것을 의미
    - "문장"은 프로그램 하나의 작동을 나타내는 단위
    - C/C++의 경우 세미콜론으로 구분하여 한 문장으로 계산
    - if 조건문 또는 for 반복문 같은 제어문의 키워드와 세미 콜론의 개수를 모두 더한 것이 C/C++에서의 문장 수
  - 문장 수에 주석, 빈줄, {} 등은 포함하지 않는다

- 최대 중첩 수

    - if 조건문 또는 for 반복문과 같은 제어문에서 사용되는 중첩의 최대 수 의미
        - 매트릭스 측정 툴에 따라서 클래스 또는 함수 정의에 사용된 중첩을 포함하는 경우도 존재
    - 중첩이 적을수록 가독성 증가
        - 중첩 수가 3이상일 때, 가독성이 매우 낮아진다
        - if 조건문의 판정 부분을 함수화하거나 조기 리턴을 활용하여 중첩 수를 줄일 수 있음

- 사이크로매틱 복잡도
    - 제어 흐름의 복잡성을 수치화한 것
        - if 조건문 또는 for 반복문 등의 제어문 수가 많을수록 수치가 커진다
    - 각 제어문에 대한 사이크로매틱 복잡도 계산
        - 복합 조건으로 사용하는 && 또는 || 등읜 연산자를 더한다
        - switch 조건문의 경우 switch 조건문 1개를 복잡도 1로 정의하거나, case 식의 개수를 모두 복잡도에 더하는 2가지 방식
    - 함수 단위의 사이크로매틱 복잡도 기준
        - 0 ~ 5 : 단순한 구조
        - 6 ~ 10 : 좋은 구조
        - 30 ~ 49 : 의문을 가져야하는 구조
        - 50 ~ 74 : 테스트 또는 디버그가 어려운 구조
        - 75 ~   : 수정할 때 문제가 발생하는 원인이 될 수 있는 구조
    - 사이크로매틱 복잡도 낮을수록 코드가 단순
        - 복잡도를 낮추는 것이 보수성을 높이는 기본
        - 이 책에서 소개한 테크닉을 사용하면 복잡도가 10을 넘길 일은 없음
        - 함수 단위에서는 복잡도가 5를 넘지 않도록 목표

- Halstead 복잡도
    - 코드 내부에서 사용되는 연산자와 피연산자의 수를 이용하여 계산한 복잡도
        - 코드 내부의 어휘 수를 바탕으로 복잡도를 구하는 방법
        - 한번에 많은 변수를 다루거나, 복잡하고 긴 계산식을 사용하는 경우 복잡도 증가
    - Hastead 복잡도의 정의
        - N * (log(n))^2
        - N = N1 + N2 : 프로그램의 길이
            - N1 : 연산자의 수
            - N2 : 피연산자의 수
        - n = n1 + n2 : 포로그램의 어휘 수

        - n1 : 연산자 종류
        - n2 : 피연산자 종류

- 보수성 지표(MI, Maintainability Index)
    - 코드의 줄 수, 사이크로매틱 복잡도, Halstead 복잡도 3개 요소를 사용하여 코드의 보수성을 수치화한 것
        - 수치가 높을수록 보수성이 좋다는 의미
    - 보수성 지표의 정의
        - MI = 171 - 5.2*log(H) - 0.23*C - 16.2*log(LOC)
        - H : Halstead 지표
        - C : 사이크로매틱 복잡도
        - LOC : 코드 줄 수
        - 중요도 : LOC > H > C
    - 코드 배부의 주석 비율을 더해서 계산하는 경우도 있음
        - MI = 171 - 5.2*log(H) - 0.23*C - 16.2*log(LOC) + 50*sin(sqrt(2.4*CM))
        - H : Halstead 지표
        - C : 사이크로매틱 복잡도
        - LOC : 코드 줄 수
        - CM : 주석 비율
    - Visual Studio에서 계산하는 보수성 지표
        - MI = MAX(0, (171 - 5.2*log(H) - 0.23*C - 16.2*log(S))*100/171)
        - H : Halstead 지표
        - C : 사이크로매틱 복잡도
        - S : 문장 수
        - 해당 식을 사용하면 함수 단위의 보수성을 100점 만점으로 평가 가능

[1-2. 매트릭스 측정 - 객체 지향 매트릭스]

- 객체 지향 언어 고유의 매트릭스
    - 응집도
    - 결합도
    - 상속의 깊이와 자식 클래스 수

- 응집도(LCOM, Lack of Cohesion in Methods)
    - 클래스가 하나의 역할에 얼마나 집중하는지를 나타내는 척도
        - 클래스의 벰버 변수를 몇 개의 멤버 함수가 참조하는지를 바탕으로 계산
        - 응집도가 높을수록 0에 가까워지고, 낮을수록 1에 가까워진다

    - 응집도 정의
        - LCOM = (m - sum(mA)/a) / (m-1)
        - a : 클래스의 멤버 변수 개수
        - m : 클래스의 멤버 함수 개수
        - mA : 특정 멤버 변수에 접근하는 멤버 함수의 수
            - sum(mA) : 각 변수들에 대한 mA의 합계
        - 각 멤버 변수가 모든 멤버 함수에서 참조되면, LCOM은 0으로 응집도가 높다고 말할 수 있음
    - 어떤 멤버 변수를 참조하지 않고 있는 멤버 함수가 있으면 LCOM은 non-zero
        - 해당 멤버 함수로 인해 클래스의 역할이 2개 이상일 가능성 존재
    - LCOM이 0.5보다 높으면 책임이 많다고 판단
        - 클래스 분할을 검토
        - 응집도를 최대한 0에 가깝게 하는 것이 좋음

- 결합도(CBO, Coupling Between Object Class)
    - 관련 있는 외부 클래스의 개수를 의미
    - 멤버 변수, 함수의 매개 변수, 상속 관계 등에서 외부 클래스와 관련 있는 부분을 조사한 수
    - 결합한 외부 클래스의 수가 많은 경우 변경에 영향을 받기 쉬우므로 주의

- 상속의 깊이와 자식 클래스 수
    - 상속의 깊이(DIT, Depth of Inheritance)와 자식 클래스 수(NOC, Number of Childer)는 상속으로 인한 결합 수를 의미
    - 상속이 객체 지향 프로그래밍의 특성 중 하나지만, 최근에는 상속보다 이양을 우선하는 설계를 좋게 평가

- 클래스의 메서드 수(WMC, Weighted Methods per Class)
    - 클래스의 멤버 함수 수를 의미
        - 간단하게 멤버 함수의 수를 모두 더하여 계산
        - 사이크로매틱 복잡도 또는 문장 수를 바탕으로 가중치를 구해 계산
    - 일반적으로 멤버 함수가 많을수록 복잡한 클래스

[2. 코드 클론 검출]

- 코드 클론
    - 코드의 중복을 의미
    - 코드의 중복은 보수성을 현저하게 낮추는 원인

        - 중복된 코드를 수정하려면 해당 부분을 모두 찾아 수정해야 하기 때문
    - 코드 클론 검출 툴을 사용하여 중복된 코드 블록 검출 가능
    - 의미 없는 복붙은 지양

[3. 정적 코드 분석]

- 정적 코드 분석
    - 소스 코드를 해석해서 오류를 자동으로 검출하는 기술
        - 버퍼 오버플로 또는 메모리 릭 등의 오류를 자동 검출
        - 초기화 되지 않는 변수 또는 결과가 같은 조건식 등의 단순 실수 발견
    - 정적 코드 분석 툴을 활용하면 코드의 품질을 높일 수 있음
    - 오류 지적이 너무 많으면 오류의 원인을 수정하는 일에 너무 많은 시간이 투자
        - 오류 지적 수를 줄이기 위해서는 코딩이 끝날 때마다 코드 분석 수행