

프로그래밍 역량 강화 전문기관, 민코딩

---

# Refactoring 개요

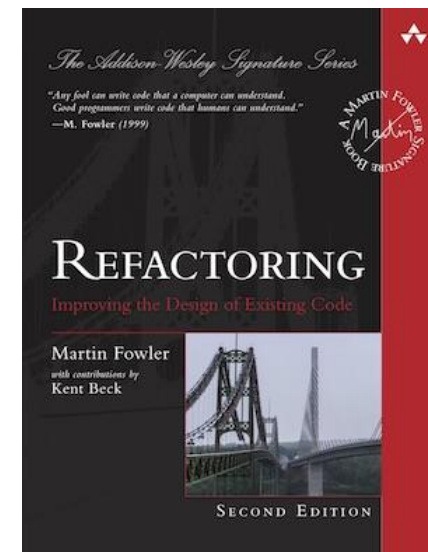


# 목차

1. Refactoring 개념
2. 리팩토링 Quiz
3. 리팩토링 Example
4. Code Readability
5. 리팩토링시 고려할 점

# 효율적인 코드 최적화를 위한 리팩토링

# Refactoring 개념



# 리팩토링이란?

새로운 소스코드를 만들어 내는 것이 아니다.

외부 동작은 변경하지 않고,  
내부 구조를 변경하는 작업이다.

- Clean한 소스코드로 수정하는 방법이 리팩토링이다.

# 리팩토링의 시작

## 1. 켄트 벡

- XP 개념을 저서에서 발표 (1999)
- Refactoring 개념 등장

## 2. 마틴 파울러 + 켄트벡

- 마틴 파울러는 켄트벡의 리팩토링을 보고, 반했음
- Refactoring 의 개념을 정리하여, 책을 냈음 (2000)

마틴파울러의 리팩토링에 대한 Idea Quiz.

## 리팩토링 Quiz

## Q1. 리팩토링 Simple Quiz

**버그 수정은 리팩토링 작업에 포함된다.**  
**(O, X)**

# A1. 버그 수정은 리팩토링에 포함 안함

답은 X

버그를 수정하면, 고쳐지는 것이므로 출력결과가 바뀌게 된다.  
따라서, 버그 수정은 리팩토링 작업에 포함되지 않는다.

리팩토링은

“출력결과를 유지한채, 내부 구조를 재구성하는 것”을 뜻한다.



## Q2. 리팩토링 Simple Quiz

**기능 추가 구현과 동시에 리팩토링을 하는 것은  
시간을 절약할 수 있는 좋은 방법이라고 소개한다.**

**(O, X)**

- 리팩토링 책 저자, 마틴파울러 기준

## A2. 리팩토링할 때 기능 추가 권장 안함

답은 **X**

마틴 파울러는 리팩토링과 기능 추가를 동시에 하는 것은 권장하지 않는다.

1. 먼저, 기능추가와 리팩토링은 별개 작업이다.  
기능 추가는 출력결과가 바뀌기 때문
2. 기능을 추가하기 전, 리팩토링을 먼저 하는 것을 추천한다.  
리팩토링을 하면 가독성이 좋고, 유지보수 하기 좋아지기 때문에  
보다 기능추가하기 수월해진다.

### Q3. 리팩토링 Simple Quiz

**프로젝트 초기  
아키텍트의 설계를 완벽히 하면, 리팩토링이 필요 없다.  
(O, X)**

A3. 기능추가가 있는 한, 리팩토링은 필요하다.

답은 **X**

초기 상황에서 완벽한 설계라고 해도,  
기능 추가와 버그 수정들이 있으면 **설계는 점차 무너진다.**

따라서 리팩토링은 지속적으로 필요하다.

## Q4. 리팩토링 Simple Quiz

**리팩토링은 개발자마다 Clean한 코드의 수준이 다르다.  
따라서, 리팩토링의 결과는 개발자 마다 다를 수 있다.**

**(O, X)**

## Q4. 리팩토링은 주관적이다.

답은 O

개발자마다 주어진 환경, 고객사 요구조건 예상치,  
개발자의 관점에 따라 주관적이다.

- 클린한 코드로 판별하는 기준 : 객관적 < **주관적**
- “무조건 이렇게 코딩해야한다” 는 없다.

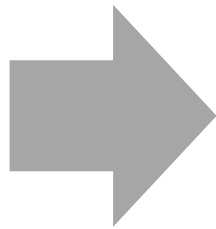
자신의 논리적 판단을 근거로 코드를 수정하는 작업

# 깔끔한 방 정리

심한 경우는

누가 봐도, 정리가 된 코드와 정리가 안된 코드가 구분이 된다.

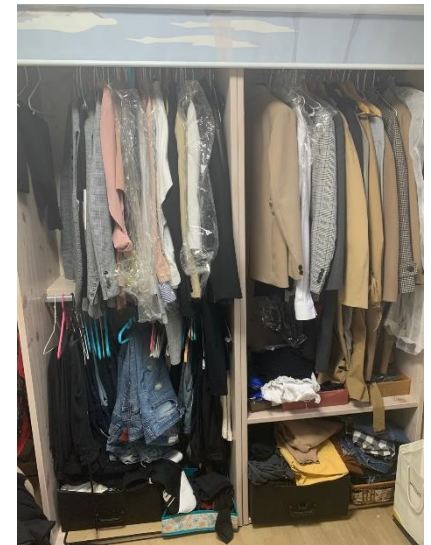
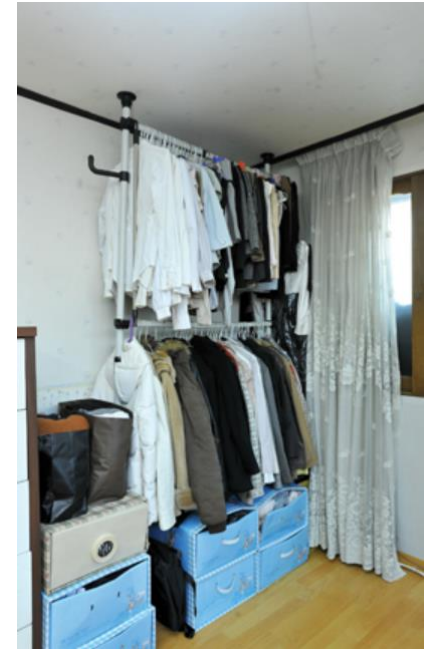
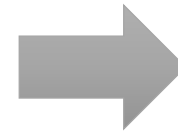
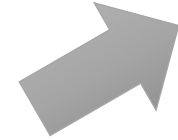
소스코드도 마찬가지이다.



# 선택적인 방 정리 방법

어떻게 정리할 것인지에 대해서는, 방법의 차이가 크다.

- 미래 예측 : 차후 옷을 더 많이 구매가 될지 / 버릴지
- 사용성 판단 : 매일 꺼내는 옷인지, 가끔 꺼내는 옷인지

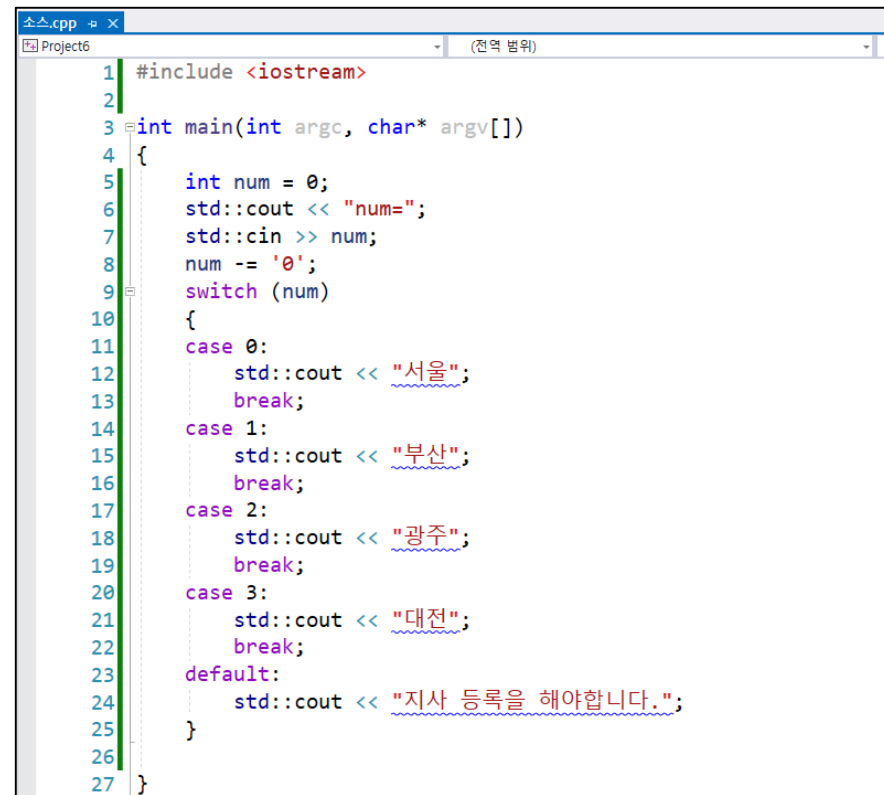




# 리팩토링은 주관적이다.

중복적이고, 유닛 테스트가 어렵고, 누가봐도 가독성이 떨어지는 코드  
→누구나 공감하는 리팩토링 필요성

상대방이 한 리팩토링이 마음에 들수도, 아닐수도 있음  
→주관적 판단



```
1 #include <iostream>
2
3 int main(int argc, char* argv[])
4 {
5     int num = 0;
6     std::cout << "num=";
7     std::cin >> num;
8     num -= '0';
9     switch (num)
10    {
11        case 0:
12            std::cout << "서울";
13            break;
14        case 1:
15            std::cout << "부산";
16            break;
17        case 2:
18            std::cout << "광주";
19            break;
20        case 3:
21            std::cout << "대전";
22            break;
23        default:
24            std::cout << "지사 등록을 해야합니다.";
25    }
26
27 }
```

# [정리] 리팩토링 특징과 장점

## 특징

1. 리팩토링은 참신한 새로운 주장이 아니다.
2. 출력 결과에 변화가 없어야 한다.
3. 개발자들마다 결과가 다를 수 있다.

중요한 것은,  
클린한 코드로 만들려는 노력이다.

# 리팩토링 Example

# SplitAndSum 소스코드 분석하기

해당 소스코드를 다운로드 후 빌드,  
소스코드를 이해해보자.

```
int splitAndSum(string text)
{
    int result = 0;

    if (text.empty())
    {
        result = 0;
    }
    else {
        vector<string> values;
        text += "-";

        // split
        int a = 0, b = 0;
        while (1) {
            b = text.find("-", a);
            if (b == -1) break;

            string temp = text.substr(a, b - a);
            values.push_back(temp);

            a = b + 1;
        }

        // sum
        for (int i = 0; i < values.size(); i++) {
            int temp = stoi(values[i]);
            result += temp;
        }
    }
    return result;
}
```

Text가 비었으면 리턴하고,  
Text를 Split 후 Int로 바꾸고 Sum을 구한다.

<https://gist.github.com/mincoding1/019ab5cfe20ad67033335313327a33b3>

# Code Readability

리팩토링으로 코드 가독성을 더 높이자.

- 코드 가독성 : 소스코드를 논리적으로 이해하기 위한 정도를 나타냄

```
int splitAndSum(string text)
{
    int result = 0;

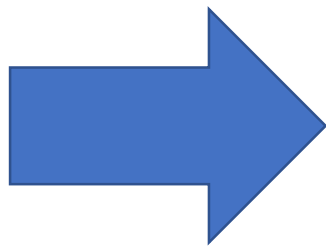
    if (text.empty())
    {
        result = 0;
    }
    else {
        vector<string> values;
        text += "-";

        // split
        int a = 0, b = 0;
        while (1) {
            b = text.find("-", a);
            if (b == -1) break;

            string temp = text.substr(a, b - a);
            values.push_back(temp);

            a = b + 1;
        }

        // sum
        for (int i = 0; i < values.size(); i++) {
            int temp = stoi(values[i]);
            result += temp;
        }
    }
    return result;
}
```



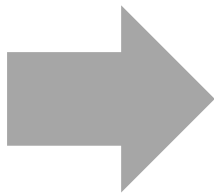
```
int splitAndSum(string text)
{
    if (text.empty()) return 0;
    return getSum(toInts(split(text)));
}
```

Text가 비었으면 리턴하고,  
Text를 Split 후 Int로 바꾸고 Sum을 구한다.

# 한 단계의 들여쓰기를 한다.

✓함수로 빼서, 들여쓰기를 없앤다.

```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
    }
    else {
        vector<string> values;
        text += "-";
        // split
        int a = 0, b = 0;
        while (1) {
            b = text.find("-", a);
            if (b == -1) break;
            string temp = text.substr(a, b - a);
            values.push_back(temp);
            a = b + 1;
        }
        // sum
        for (int i = 0; i < values.size(); i++) {
            int temp = stoi(values[i]);
            result += temp;
        }
    }
    return result;
}
```



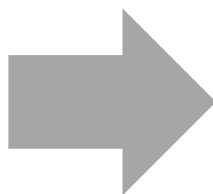
```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
    }
    else {
        vector<string> values;
        split(&text, &values);
        getSum(&result, values);
    }
    return result;
}
```

# else를 없앤다.

return 을 사용하여, 필요없는 else를 지운다.

- 코드의 흐름을 더 단순화 하기 위해, Guard 절이 참이면 즉시 실행을 종료하는 코드로 리팩토링
- 이를 **Replace Nested Conditional with Guard Clauses** 라고 한다.

```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
    }
    else {
        vector<string> values;
        split(&text, &values);
        getSum(&result, values);
    }
    return result;
}
```



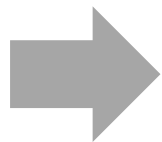
```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
        return 0;
    }
    vector<string> values;
    split(text, values);
    getSum(result, values);
}
```

# return 값은 첫 번째 인자로

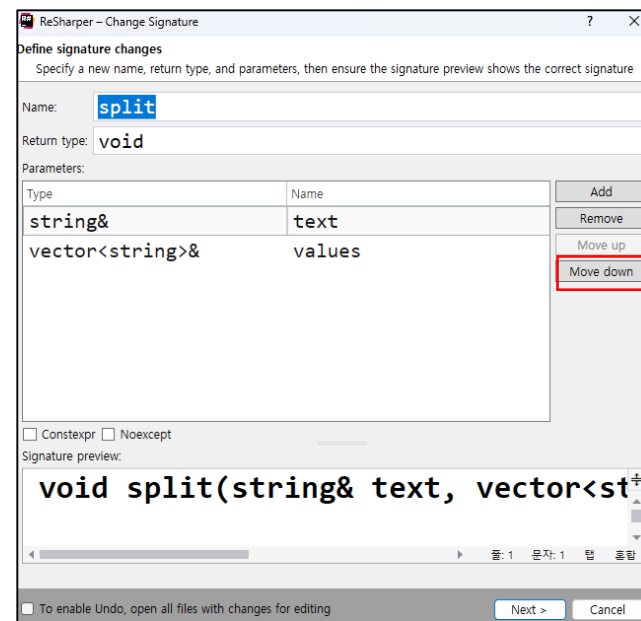
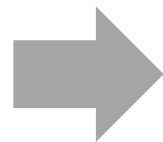
결과를 저장할 변수 값의 위치를 통일시킨다.

- 결과를 받을 변수를 가장 앞쪽 파라미터로 배치한다.

```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
        return 0;
    }
    vector<string> values;
    split(text, values);
    getSum(result, values);
}
```



```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
        return 0;
    }
    vector<string> values;
    split(text, values);
    getSum(values, result);
}
```



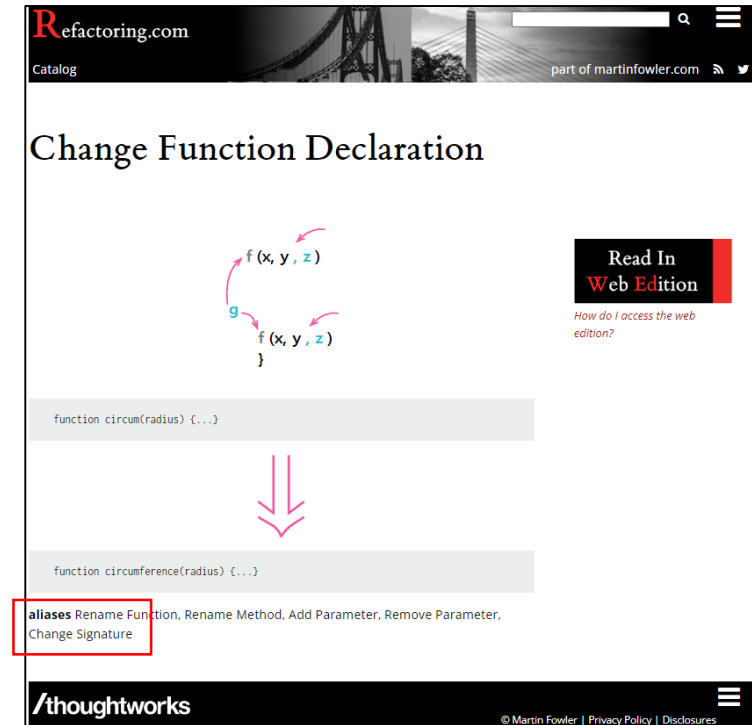
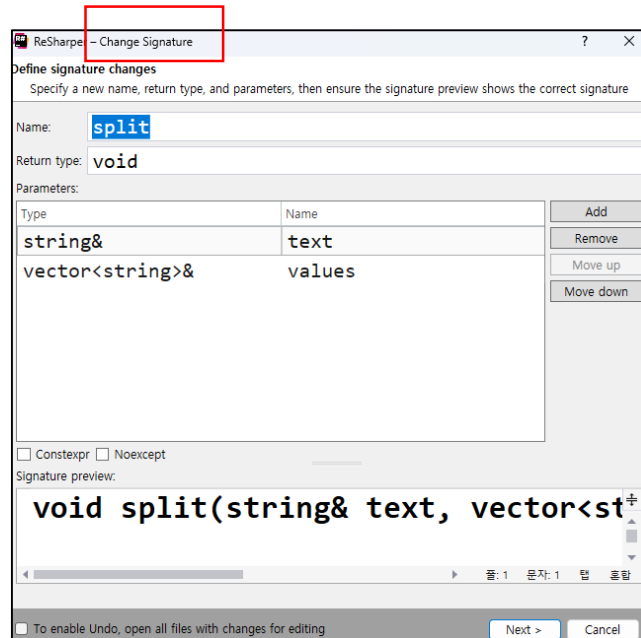
순서를 바꾼다.

split 후 결과를 values에 담는다.  
getSum 후 결과를 result에 담는다.



# [참고] Change Signature

- ✓마틴파울러 : Change Function Declaration
  - Visual Studio 에서는 Change Signature 라고 부른다.

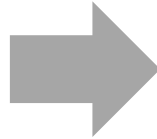


# pass by reference 대신 return value

pass by reference 보다는  
결괏값을 return value로 하는 것이 해석하기 편리하다.

```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
        return 0;
    }
    vector<string> values;
    split(values, text);
    getSum(result, values);
}
```

수동으로 편집해야한다.



```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
        return 0;
    }
    vector<string> values;
    values = split(text);
    result = getSum(values);
    return result;
}
```

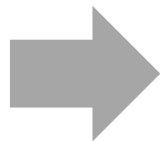
split과 getSum 내부 코드도  
정상동작하도록 수정하자.

# 필요 없는 코드 제거하기

✓필요 없는 코드를 지운다.

- Remove Dead Code

```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        result = 0;
        return 0;
    }
    vector<string> values;
    values = split(text);
    result = getSum(values);
    return result;
}
```



```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        return 0;
    }
    vector<string> values;
    values = split(text);
    result = getSum(values);
    return result;
}
```

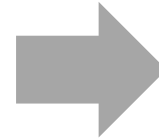
# range-for 문으로 변경

✓ 조금 더 읽기 편한 코드로 변경 (getSum 메서드 내부)

- <https://en.cppreference.com/w/cpp/language/range-for>

```
8 int getSum(vector<string> values)
9 {
10     int result = 0;
11     for (int i = 0; i < values.size(); i++) {
12         temp = stoi(values[i]);
13         result += temp;
14     }
15     return result;
16 }
```

Apply clang-tidy fix for [modernize-loop-convert]  
Inspection: 'modernize-loop-convert clang-tidy check'

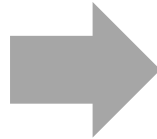
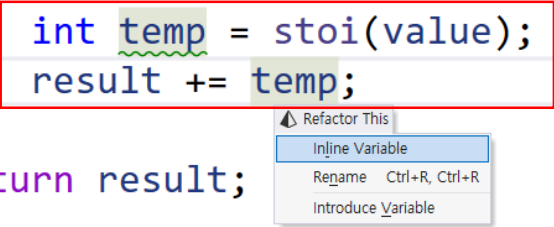


```
int getSum(vector<string> values)
{
    int result = 0;
    for (auto& value : values)
    {
        int temp = stoi(value);
        result += temp;
    }
    return result;
}
```

# Inline Variable

✓임시변수를 제거한다.

```
int getSum(vector<string> values)
{
    int result = 0;
    for (auto& value : values)
    {
        int temp = stoi(value);
        result += temp;
    }
    return result;
}
```



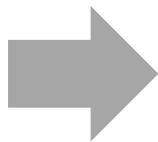
```
int getSum(vector<string> values)
{
    int result = 0;
    for (auto& value : values)
    {
        result += stoi(value);
    }
    return result;
}
```

# getSum 메서드 역할을 2 개로 분리한다. - 1

✓ getSum의 두 가지 역할

1. 문자열을 수로 변경한다.
2. 수들의 sum을 구한다.

```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        return 0;
    }
    vector<string> values;
    values = split(text);
    result = getSum(values);
    return result;
}
```



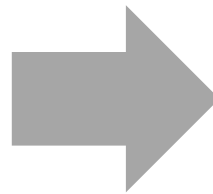
```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        return 0;
    }
    vector<string> values;
    values = split(text);
    vector<int> nums = toInts(values);
    result = getSum(nums);
    return result;
}
```

# getSum 메서드 역할을 2 개로 분리한다. - 2

✓getSum을 두 개의 함수로 분리 : Split Loop

- toInts : 문자열 배열을 수 배열로 변환
- getSum : 수 배열의 합을 리턴

```
int getSum(vector<string> values)
{
    int result = 0;
    for (auto& value : values)
    {
        result += stoi(value);
    }
    return result;
}
```



```
int getSum(vector<int> nums)
{
    int result = 0;
    for (auto& num : nums)
    {
        result += num;
    }
    return result;
}
```

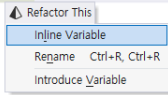
```
vector<int> toInts(vector<string> numString)
{
    vector<int> result;
    for (auto& str : numString)
    {
        result.push_back(stoi(str));
    }
    return result;
}
```

# 임시변수 제거하기

의미 파악에 도움이 되지 않는 변수 제거

- Inline Variable

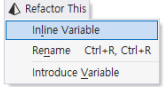
```
int splitAndSum2(string text)
{
    int result = 0;
    if (text.empty())
    {
        return 0;
    }
    vector<string> values;
    values = split(text);
    vector<int> nums = toInts(values);
    result = getSum(nums);
    return result;
}
```



A refactor menu is shown for the variable 'result'. The menu includes options: 'Refactor This', 'Inline Variable', 'Rename Ctrl+R, Ctrl+R', and 'Introduce Variable'. 'Inline Variable' is highlighted.



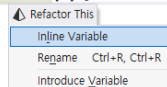
```
int splitAndSum2(string text)
{
    if (text.empty())
    {
        return 0;
    }
    vector<string> values;
    values = split(text);
    vector<int> nums = toInts(values);
    return getSum(nums);
}
```



A refactor menu is shown for the variable 'nums'. The menu includes options: 'Refactor This', 'Inline Variable', 'Rename Ctrl+R, Ctrl+R', and 'Introduce Variable'. 'Inline Variable' is highlighted.



```
int splitAndSum2(string text)
{
    if (text.empty())
    {
        return 0;
    }
    vector<string> values;
    values = split(text);
    return getSum(toInts(values));
}
```



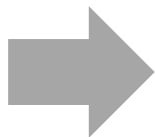
A refactor menu is shown for the variable 'values'. The menu includes options: 'Refactor This', 'Inline Variable', 'Rename Ctrl+R, Ctrl+R', and 'Introduce Variable'. 'Inline Variable' is highlighted.



# 추상화 Level을 맞추어준다.

더 이해하기 쉽게 변경

```
int splitAndSum2(string text)
{
    if (text.empty())
    {
        return 0;
    }
    return getSum(toInts(split(text)));
}
```



```
int splitAndSum2(string text)
{
    if (isEmpty(text)) {
        return 0;
    }
    return getSum(toInts(split(text)));
}
```

# 클래스로 추출하기

## ✓Extract Class

- 특정 코드를 클래스를 만들어 이동시킨다.

```
int main()
{
    cout << SplitAndSum().splitAndSum2("100-10-20");
    return 0;
}
```

```
class SplitAndSum
{
public:
    int splitAndSum2(string text)
    {
        if (isEmpty(text)) {
            return 0;
        }
        return getSum(toInts(split(text)))
    }

    int getSum(vector<int> nums)
    {
```

# Class 메서드 순서

- ✓ public / private 구분
- ✓ 메서드 순서를 읽기 좋게 배치

가장 중요한 메서드

```
class SplitAndSum
{
public:
    int splitAndSum2(string text)
    {
        if (isEmpty(text)) {
            return 0;
        }
        return getSum(nums:toInts(numString:split(&text)));
    }

private:
    bool isEmpty(string text)
    {
        return text.empty();
    }

    vector<string> split(string& text)
    {
        vector<string> result;
        text += "-";
        // split
        int a = 0, b = 0;
        while (1) {
            b = text.find(_Ptr: "-", _Off:a);
            if (b == -1) break;
            string temp = text.substr(_Off:a, _Count:b - a);
            result.push_back(_Val:temp);
            a = b + 1;
        }
        return result;
    }

    vector<int> toInts(vector<string> numString)
    {
        vector<int> result;
        for (auto& std::string & str : numString)
        {
            result.push_back(_Val:stoi(_Str:str));
        }
        return result;
    }

    int getSum(vector<int> nums)
    {
        int result = 0;
        for (auto& int & num : nums)
        {
            result += num;
        }
        return result;
    }
};

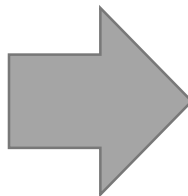
int main()
{
    cout << SplitAndSum().splitAndSum2(text:"100-10-20");
    return 0;
}
```

# [참고] 들여쓰기 단계

## ✓ 4 levels of indentation vs 2 levels

- indent 레벨이 높으면  
이해하기 어렵기에, 메서드 추출로 이해하기 쉽게 한다.

```
// Remove vertices from mesh.
for each vertex in vertices.to_remove:
{
    for each edge in vertex.edges:
    {
        for each face in edge.faces:
        {
            face.remove_edge(edge);
            if (face.size() < 3)
                face.remove();
        }
        edge.remove();
    }
    vertex.remove();
}
```



```
for each vertex in vertices.to_remove:
{
    for each edge in vertex.edges:
        edges_to_remove.insert(edge);
}

for each edge in edges_to_remove:
{
    for each face in edge.faces:
        faces_to_rebuild.insert(face, edge);
}

for each face, edge in faces_to_rebuild:
{
    face.remove_edge(edge);
    if (face.size() < 3)
        face.remove();
}

for each edge in edges_to_remove:
    edge.remove();

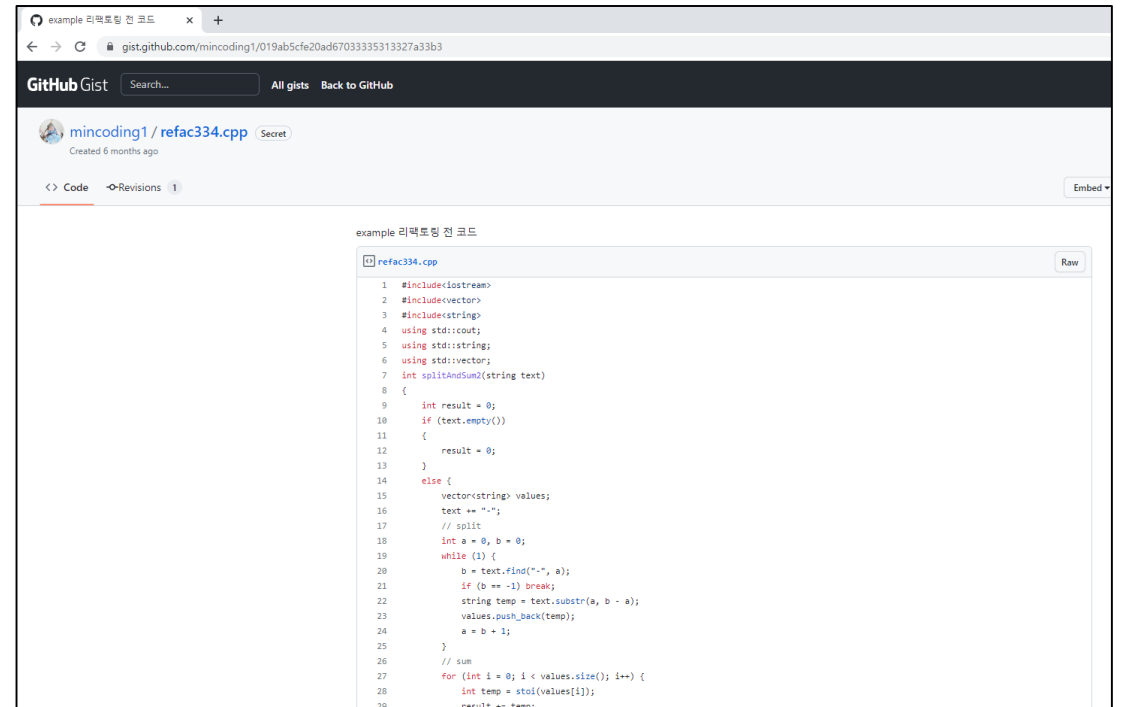
for each vertex in vertices_to_remove:
    vertex.remove();
```

C++ 11 이전에 나왔던,  
for each, in 문법이지만 사용을 권장하지 않는다.  
[참조] <https://learn.microsoft.com/ko-kr/cpp/dotnet/for-each-in?view=msvc-170>

# [도전] 직접 리팩토링 해보기

## 미션

1. else 없애기
2. indentation level 줄이기
3. 메서드를 추가한다면,  
하나의 역할만 하는 메서드로 추가



```
1 #include<iostream>
2 #include<vector>
3 #include<string>
4 using std::cout;
5 using std::string;
6 using std::vector;
7 int splitAndSum2(string text)
8 {
9     int result = 0;
10    if (text.empty())
11    {
12        result = 0;
13    }
14    else {
15        vector<string> values;
16        text += "-";
17        // split
18        int a = 0, b = 0;
19        while (1) {
20            b = text.find("-", a);
21            if (b == -1) break;
22            string temp = text.substr(a, b - a);
23            values.push_back(temp);
24            a = b + 1;
25        }
26        // sum
27        for (int i = 0; i < values.size(); i++) {
28            int temp = stoi(values[i]);
29            result += temp;
```

<https://gist.github.com/mincoding1/019ab5cfe20ad67033335313327a33b3>

가독성 좋은 코드에 대해서

# Code Readability

# 코드 가독성

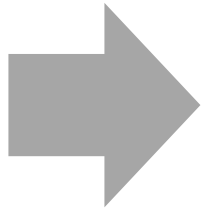
소스코드를 보고, 코드의 의도하는 동작, 알고리즘을 얼마나 쉽게 이해할 수 있는지를 나타낸다.

가독성을 높이는 몇 가지 예시를 살펴보자.

# Rename Variable

이름을 짓는데 공들여라.

```
int h = 10;  
int w = 20;  
int i = h * w;
```



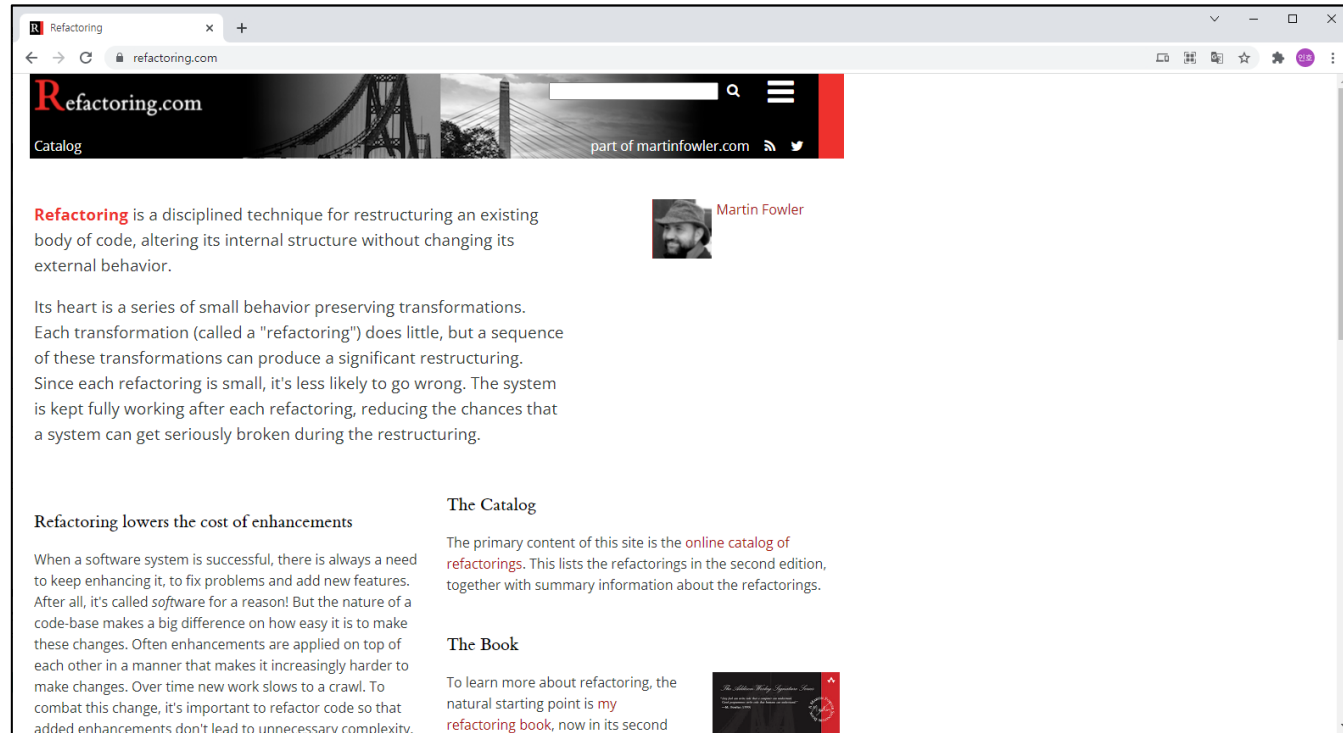
```
int height = 10;  
int width = 20;  
int area = height * width;
```



# [참고] 마틴파울러 refactoring 사이트

소개되는 빨간색 이름들은, 마틴파울러가 정한 이름

- <https://refactoring.com/catalog/>





# Replace Magic Literal

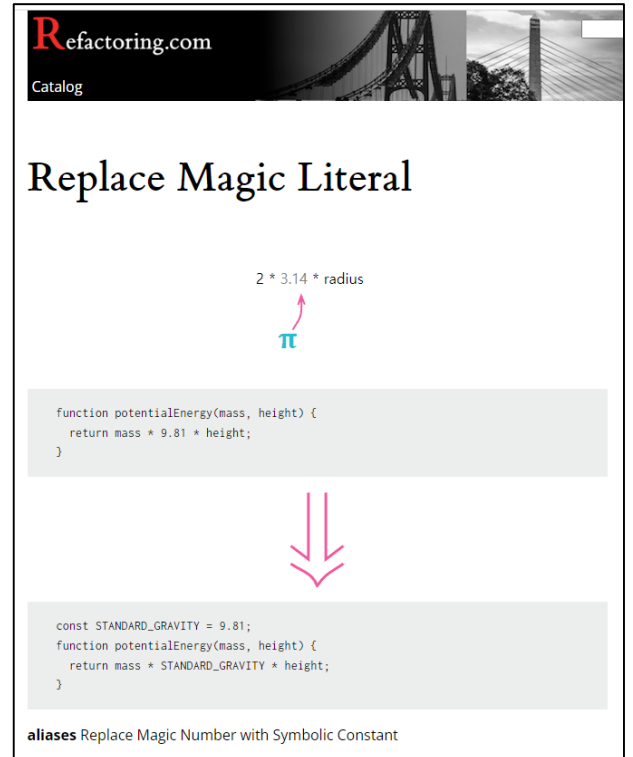
매직넘버 치환하기

→ 예측할 수 있는 코드

```
const int MAX_BOOTING_TICK_COUNT = 85;

while(MAX_BOOTING_TICK_COUNT <= getTickCount()) {
    ...
}
```

빨간색 글씨  
= 마틴파울러가 정한 이름



The screenshot shows the Refactoring.com website with the title 'Replace Magic Literal'. It illustrates a code transformation where a magic number is replaced by a symbolic constant. The top part shows a formula  $2 * 3.14 * radius$  with a red arrow pointing to the value 3.14, which is labeled with the Greek letter  $\pi$ . Below this, a code block shows a function `potentialEnergy` that uses the magic number 9.81. A large red arrow points down to a second code block where the magic number 9.81 has been replaced by a constant `STANDARD_GRAVITY`. At the bottom, the text 'aliases Replace Magic Number with Symbolic Constant' is visible.

Refactoring.com  
Catalog

## Replace Magic Literal

$2 * 3.14 * radius$   
 $\pi$

```
function potentialEnergy(mass, height) {  
    return mass * 9.81 * height;  
}
```

↓

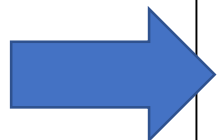
```
const STANDARD_GRAVITY = 9.81;  
function potentialEnergy(mass, height) {  
    return mass * STANDARD_GRAVITY * height;  
}
```

aliases Replace Magic Number with Symbolic Constant

# 상수 의존관계

상수끼리 관계가 있는 경우,  
관계를 표시하도록 해주는 것이 가독성에 더 좋다.

```
const int BOOT_AREA_KB = 200;  
const int HEAP_AREA_KB = 800;
```



```
const int BOOT_AREA_KB = 200;  
const int HEAP_AREA_KB = BOOT_AREA_KB * 4;
```

# Rename Variable

flag 보다는 found가 더 명확함

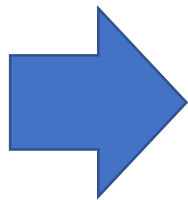
```
int main()
{
    vector<string> list = { "ABC", "BTS", "KFC" };

    string myFavorite = "KFC";

    int flag = 0;
    for (int i = 0; i < list.size(); i++) {
        if (list[i] == myFavorite) {
            flag = 1;
            break;
        }
    }

    if (flag) cout << "최애 존재";
    else cout << "없다";

    return 0;
}
```



```
int main()
{
    vector<string> list = { "ABC", "BTS", "KFC" };

    string myFavorite = "KFC";

    int found = 0;
    for (int i = 0; i < list.size(); i++) {
        if (list[i] == myFavorite) {
            found = 1;
            break;
        }
    }

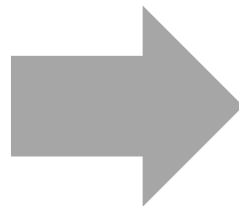
    if (found) cout << "최애 존재";
    else cout << "없다";

    return 0;
}
```

# Change Function Declaration

명확한 함수명 사용

```
class Cal {  
private:  
    int a;  
  
public:  
    int sum(int v)  
    {  
        return a + v;  
    }  
}
```



```
class Cal {  
private:  
    int totalCoin;  
  
public:  
    int accumulationCoin(int inputCoin)  
    {  
        return totalCoin + inputCoin;  
    }  
}
```

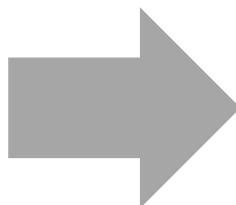
# Extract Function

```
void printAllData()
{
    // make data field
    vector<int> vect(10);

    // cal acc value
    int cnt = 0;
    for (int i = 0; i < 10; i++){
        vect[i] = cnt;
        cnt += i;
    }

    // check exist data 10
    int flag = 0;
    for (int i = 0; i < 10; i++){
        if (vect[i] == 10) {
            flag = 1;
            break;
        }
    }

    // print result
    if (flag == 1) {
        std::cout << "발견";
    }
    else {
        std::cout << "미발견";
    }
}
```



```
vector<int> makeDataField()
{
    return vector<int>(10);
}

int calAccValue(vector<int>& vect)
{
    int cnt = 0;
    for (int i = 0; i < 10; i++) {
        vect[i] = cnt;
        cnt += i;
    }
}

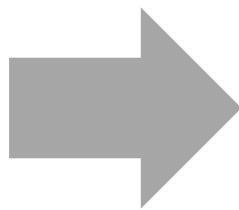
bool isExistData(vector<int> vect, int target)
{
    for (int i = 0 ; i < vect.size(); i ++){
        if (vect[i] == target) {
            return true;
        }
    }
    return false;
}

void printAllData()
{
    vector<int> vect = makeDataField();
    calAccValue(vect);
    if (isExistData(vect, 10)){
        std::cout << "발견";
    }
    else {
        std::cout << "미발견";
    }
}
```

# Inline Function

1회성 함수는 inline 시키기

```
int isCompare(int a, int b) {  
    if (a == b) return 1;  
    return 0;  
}  
  
bool checkExistData(vector<int> vect, int tarNum) {  
    for (int i = 0; i < 10; i++) {  
        if (isCompare(vect[i], 10) == 1) {  
            return true;  
        }  
    }  
    return false;  
}
```



```
bool checkExistData(vector<int> vect, int tarNum) {  
    for (int i = 0; i < 10; i++) {  
        if (a == b) {  
            return true;  
        }  
    }  
    return false;  
}
```

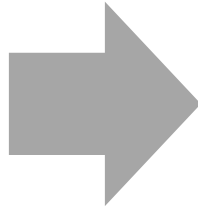


# Extract Variable 1

식을 이해하기 쉽게 하기 위해, 이름을 붙인다.

- **Introduce Explaining Variable** : 설명하기 위한 변수 도입하기

```
double getCircleSize(Pack pack) {  
    return (((pack.DAend - pack.DAstart) / 2) * 0.5 + (((pack.DBend - pack.DBstart) / 2) * 0.5));  
}
```

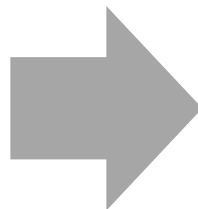


```
double getCircleSize(Pack pack) {  
    double DAmid = (pack.DAend - pack.DAstart) / 2;  
    double DBmid = (pack.DBend - pack.DBstart) / 2;  
  
    double halfDAmid = DAmid * 0.5;  
    double halfDBmid = DBmid * 0.5;  
  
    return halfDAmid + halfDBmid;  
}
```

# Extract Variable 2

의미를 이해하기 쉽다.

```
int getBestPrice(int dCode[3]) {  
    if (dCode[0] + dCode[1] > dCode[2]) return dCode[2];  
    return dCode[0] + dCode[1];  
}
```

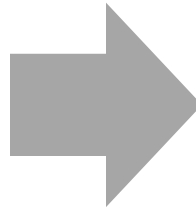


```
int getBestPrice(int dCode[3]) {  
    int salePrice = dCode[0] + dCode[1];  
    int specialPrice = dCode[2];  
    if (salePrice < specialPrice) return salePrice;  
    return specialPrice;  
}
```

# Inline Variable

변수는 표현식에 이름을 붙여 의미를 이해하기 쉽게 만들지만, 이미, 의미가 충분하여 필요없는 경우도 있다.

```
bool isBoxSizeMoreThan15(Pack box) {  
    int boxSize = box.size;  
    if (boxSize > 15) {  
        return true;  
    }  
    return false;  
}
```



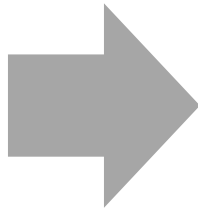
```
bool isBoxSizeMoreThan15(Pack box) {  
    if (box.size > 15) {  
        return true;  
    }  
    return false;  
}
```

# Encapsulate Variable

## 캡슐화를 할 것

- 최대한 객체의 필드는 private로 유지한다.
- 필드 값을 제어하는 통로를 줄인다.
- 유효값 검사 가능

```
Box* box = new Box();  
box->size = 10;  
box->color = "BLUE";
```



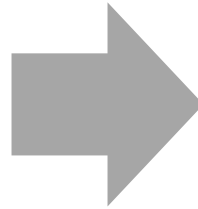
```
Box* box = new Box();  
box->setSize(10);  
box->setColor("BLUE");
```

# Introduce Parameter Object

파라미터 객체를 만들어 사용하기

- Introduce 라는 용어는 도입, 추출의 의미로 쓰인다. (inline의 반대 용어)

```
class Main {  
    void drawPixel(int x, int y) {  
        //..  
    }  
    void drawCircle(int x, int y) {  
        //..  
    }  
    void drawTree(int x, int y) {  
        //..  
    }  
};
```



```
struct Axis {  
    int x, y;  
};  
  
class Main {  
    void drawPixel(Axis tar) {  
        //..  
    }  
    void drawCircle(Axis tar) {  
        //..  
    }  
    void drawTree(Axis tar) {  
        //..  
    }  
};
```

# Replace Control Flag with Break

## Control Flag 제거하기 (Remove Control Flag)

- break / return 을 하는 방법으로 리팩토링 한다.

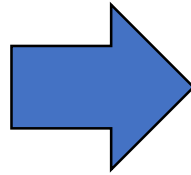
```
int main()
{
    vector<string> list = { "ABC", "BTS", "KFC" };

    string myFavorite = "KFC";

    int found = 0;
    for (int i = 0; i < list.size(); i++)
    {
        if (list[i] == myFavorite)
        {
            found = 1;
        }
    }

    if (found) cout << "최애 존재";
    else cout << "없다";

    return 0;
}
```



```
int main()
{
    vector<string> list = { "ABC", "BTS", "KFC" };

    string myFavorite = "KFC";

    for (int i = 0; i < list.size(); i++)
    {
        if (list[i] == myFavorite)
        {
            cout << "최애 존재";
            return 0;
        }
    }

    cout << "없다";

    return 0;
}
```

# [도전] 리팩토링

본인의 스타일대로 리팩토링 시작!

**가독성이 좋은 코드로 리팩토링**

## [리팩토링 훈련 규칙]

- else 최소화
- 아규먼트 최소화
- 들여쓰기 최소화
- 전역변수 사용 금지
- 작은 수정 마다 정상적으로 값 나오는지 테스트

```
flag재거_java.class
1  import java.util.ArrayList;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          //25+61=100
8          //1 ~ 5자리수 덧셈 수식이 맞는지 확인하는 프로그램
9          //띄어쓰기 없음
10
11         String str = "25+61=86"; //PASS
12         //String str = "12345+12345=24690"; //PASS
13         //String str = "5++5=10"; //ERROR
14         //String str = "10000+1=10002"; //FAIL
15
16         int flag1 = 0;
17         int cnt1 = 0;
18         int cnt2 = 0;
19         int p1 = 0, p2 = 0;
20         //+=와 = 개수 확인
21         for (int i = 0; i < str.length(); i++) {
22             if (str.charAt(i) == '+') {
23                 cnt1++;
24                 p1 = i;
25             }
26             else if (str.charAt(i) == '=') {
27                 cnt2++;
28                 p2 = i;
29             }
30         }
31     }
32 }
```

<https://gist.github.com/mincoding1/81a3372b22699697bd8c0f111d0a0a67>

# 리팩토링의 첫 번째 목적

리팩토링을 통해,

**가독성이 더 좋게 만들기**



# ReSharper C++ Reference

# Reference 모음

## ✓ReShaper C++ 공식 사이트

- <https://www.jetbrains.com/resharper-cpp/>
- [https://www.jetbrains.com/help/resharper/Languages\\_CPP.html](https://www.jetbrains.com/help/resharper/Languages_CPP.html)

## ✓Visual Assist와 비교 자료

- [https://www.jetbrains.com/resharper-cpp/documentation/resharper\\_cpp\\_vs\\_visual\\_assist.html](https://www.jetbrains.com/resharper-cpp/documentation/resharper_cpp_vs_visual_assist.html)
- <https://www.wholetomato.com/>

## ✓ReShaper 공식 Quick Tips

- <https://www.youtube.com/playlist?list=PLQ176FUIyIUbPQ61fNCrX16mPmO63dWo2>

팀원들을 고려한 리팩토링

리팩토링시 고려할 점

# 리팩토링을 할 때 고려할 점

리팩토링을 적절히 수행하면 코드의 품질이 올라간다.

- 가독성이 증가
  - 유지보수가 편리
  - 개발속도 증가

무조건적인 리팩토링을 하기 전,  
고려해야 할 점들이 존재한다.

# 리팩토링 시 고려할 점 1

**자신이 리팩토링을 하는 이유를 정확히 구분하자. (메타인지)**

- 오버엔지니어링 (본인의 만족)  
정말로 팀에 필요한 리팩토링인지

확장성 / 생산성을 과하게 고려하여  
코드 복잡성 / 개발시간을 높인다.

# 리팩토링 시 고려할 점 2

## 팀원들의 감정

- 누군가 허락없이 내가 짠 코드를 수정하면 감정 문제로 퍼진다.
- 트러블 / 작은 감정문제는 생산성 하락을 초래한다.

# 리팩토링 시 고려할 점 3

레거시 코드에는 의도가 있다.

→ 코드만 보고, 소스코드의 모든 의도를 파악하는 것은 어렵다.

→ 의도가 불분명한 코드는 코드 개발자와 대화 필요

# 리팩토링 시 고려할 점 4

**버전관리도구, Test 환경**이 충분히 갖춰져야 한다.

- 리팩토링 수행 전 후 결과에 대한 Unit Test / Black Box Test 등 환경이 갖춰져야 한다.
- 개발중이 아닌, 개발이 완료되어 동작되는 모듈에 대해 리팩토링을 한다.
- 버그가 거의 없어야 한다.  
→ 버그가 너무 많은 프로그램은 리팩토링 작업이 힘들다.