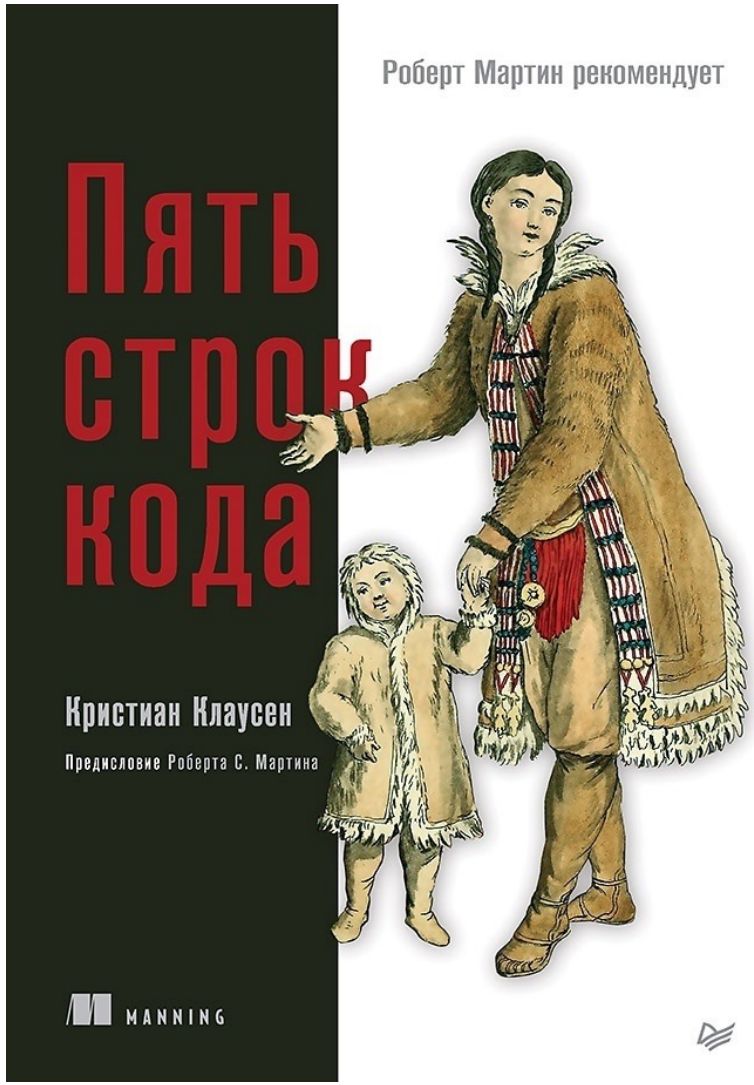
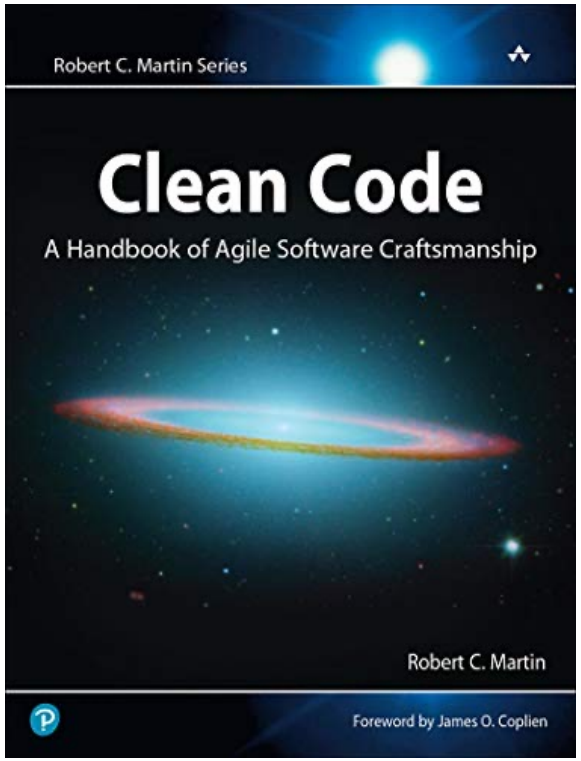
리팩터링 패턴 실전 적용



POSTECH CSE 21 하태혁



- 전문적인 개발, 코드 리팩터링 경험 X
- Clean Code 읽어 본 적 X

리팩토링과 코드 품질을 처음 공부해본 컴퓨터공학과 학생의 시선에서
이 책을 리뷰하고 실용적으로 적용할 수 있었던 부분

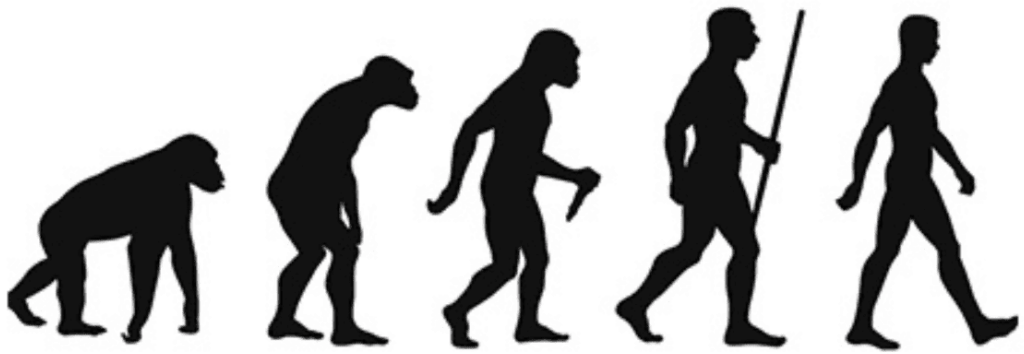
Five Lines of Code 감상평

1. 실용적이다

- Typescript 게임 예제 코드를 직접 리팩터링하면서 패턴을 익힐 수 있다
- 해당 패턴을 사용하는 이유에 대해서도 이해하기 쉽게 잘 설명되어 있다
- 리팩터링 패턴 뿐만 아니라, 변수와 함수의 이름, 개발자가 개발을 하면서 명심해야 할 마인드(시간 관리, 코드 관리, 컴파일러를 사용하는 방법) 등도 함께 소개되어 있다

2. 쉽다

직접 다룰 수 있는 코드 예제 뿐만 아니라 비슷한 유형의 다른 예제 코드도 책에 담겨있다. 패턴을 더 쉽게 익힐 수 있으며, 입문자의 입장에서 반복적으로 패턴을 적용해 보면서 학습 효과를 높일 수 있다



Refactoring

Improving the Design of Existing Code

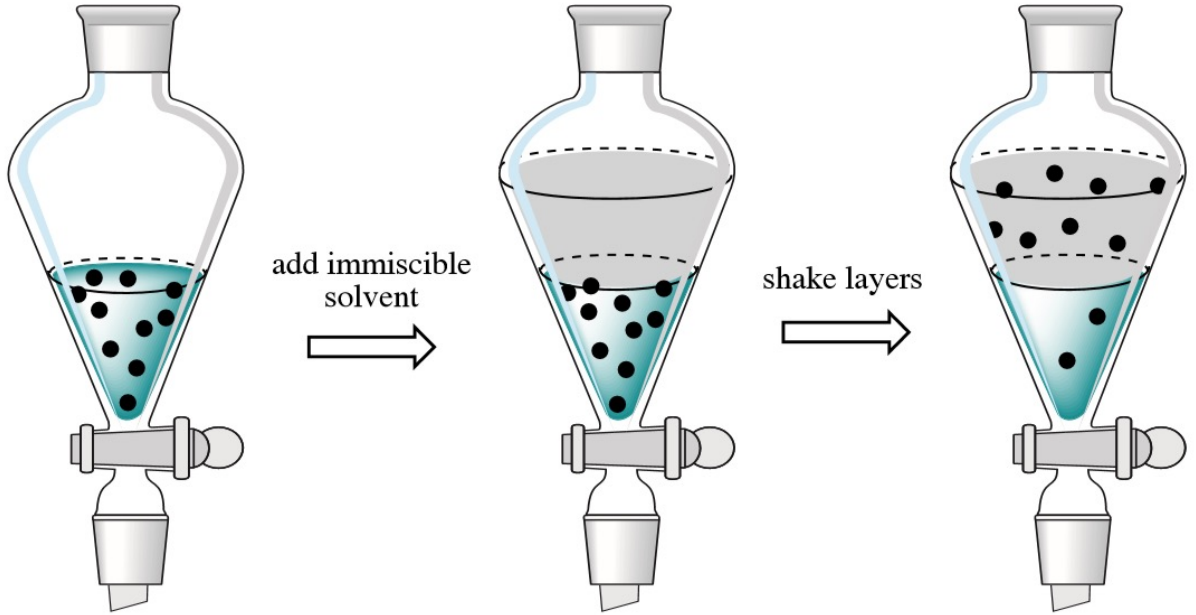
1. 실용적이고 적용하기 좋았던 패턴들

2. 오히려 불편했던 부분들(이해하기 어려웠던 부분들)

3. 같이 이야기해보고 싶은 부분

1. 실용적이고 적용하기 좋았던 패턴들

: 사실 패턴은 모두 실용적이고 적용하기 좋았으나 그 중에서도 특히 도움되었던 패턴이 있다



메서드 추출

- 한 메서드의 일부를 가져와 고유한 메서드로 추출한다
- 어떤 기능을 하는 코드를 크게 두 가지 영역으로 분리할 수 있다면 그 두 가지 부분을 각각의 메서드로 분리하는 것이다

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

const int MAX_NODES = 1001;
using namespace std;

int N, M, V;
vector<int> graph[MAX_NODES];
bool visited_dfs[MAX_NODES] = {false};

```

```cpp
void DFS(int startNode)
{
    visited_dfs[startNode] = true;
    cout << startNode << " ";
    for (const int adjNode : graph[startNode])
    {
        if (!visited_dfs[adjNode])
        {
            DFS(adjNode);
        }
    }
}

```

```cpp
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    cin >> N >> M >> V;
    for (int i = 0; i < M; i++)
    {
        int n1, n2;
        cin >> n1 >> n2;
        graph[n1].push_back(n2);
        graph[n2].push_back(n1);
    }
    for (int i = 1; i <= N; i++)
    {
        sort(graph[i].begin(), graph[i].end());
    }

    DFS(V);

    // BFS
    cout << "\n";
    bool visited[MAX_NODES] = {false};
    queue<int> q;
    q.push(V);
    visited[V] = true;

    while (!q.empty())
    {
        int front = q.front();
        q.pop();
        cout << front << " ";

        for (const int adjNode : graph[front])
        {
            if (!visited[adjNode])
            {
                q.push(adjNode);
                visited[adjNode] = true;
            }
        }
    }

    return 0;
}

```

```cpp
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    cin >> N >> M >> V;
    for (int i = 0; i < M; i++)
    {
        int n1, n2;
        cin >> n1 >> n2;
        graph[n1].push_back(n2);
        graph[n2].push_back(n1);
    }
    for (int i = 1; i <= N; i++)
    {
        sort(graph[i].begin(), graph[i].end());
    }

    DFS(V);

    // BFS
    cout << "\n";
    bool visited[MAX_NODES] = {false};
    queue<int> q;
    q.push(V);
    visited[V] = true;

    while (!q.empty())
    {
        int front = q.front();
        q.pop();
        cout << front << " ";

        for (const int adjNode : graph[front])
        {
            if (!visited[adjNode])
            {
                q.push(adjNode);
                visited[adjNode] = true;
            }
        }
    }

    return 0;
}
```

=>

```cpp
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    getInput();
    DFS(V);
    cout << "\n";
    BFS(V);

    return 0;
}
```

```cpp
void BFS(int startNode)
{
    bool visited[MAX_NODES] = {false};
    queue<int> q;
    q.push(startNode);
    visited[startNode] = true;

    while (!q.empty())
    {
        int front = q.front();
        q.pop();
        cout << front << " ";

        for (const int adjNode : graph[front])
        {
            if (!visited[adjNode])
            {
                q.push(adjNode);
                visited[adjNode] = true;
            }
        }
    }
}
```

```cpp
void sortGraph()
{
    for (int i = 1; i <= N; i++)
    {
        sort(graph[i].begin(), graph[i].end());
    }
}

void getInput()
{
    cin >> N >> M >> V;
    for (int i = 0; i < M; i++)
    {
        int n1, n2;
        cin >> n1 >> n2;
        graph[n1].push_back(n2);
        graph[n2].push_back(n1);
    }
    sortGraph();
}
```

```cpp
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    getInput();
    DFS(V);
    cout << "\n";
    BFS(V);

    return 0;
}
```

클래스로의 코드 이관

: 기능을 클래스로 옮기기 때문에 클래스에 사용되었던 코드들이 자연스럽게 클래스 안으로 옮겨지게 된다

데이터 캡슐화

변수와 관련된 불변속성을 지역화하고 응집력을 더 명확히 한다

→ 메서드를 클래스 안으로 이동시키는 것

→ 불변속성의 변수를 클래스 안으로 이동시키는 것

```cpp
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    Graph graph;
    graph.readInput();
    graph.DFS(graph.getStartNode());
    cout << "\n";
    graph.BFS(graph.getStartNode());

    return 0;
}
```

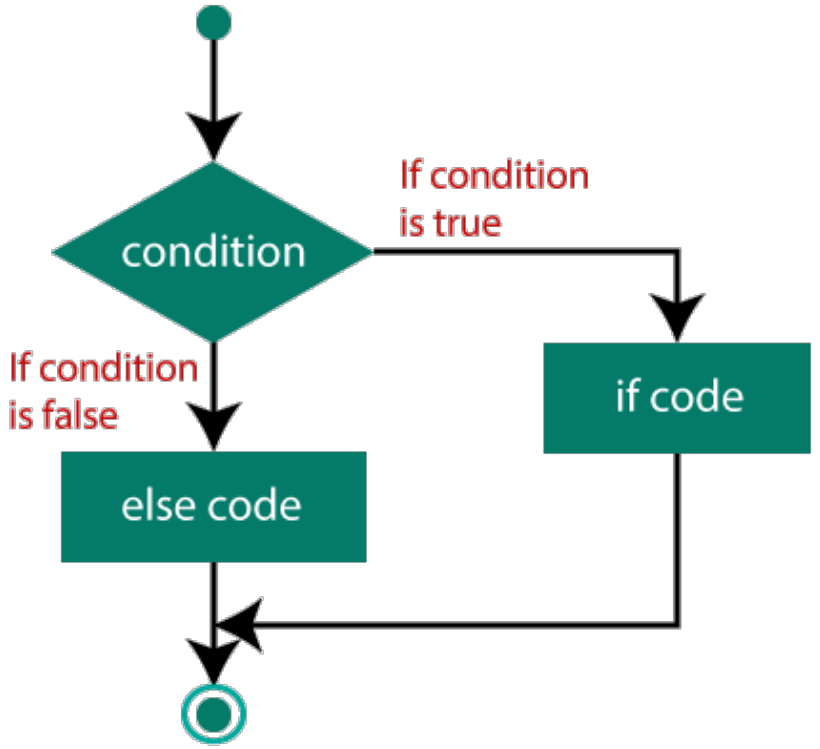
- 그래프로 설정해서 객체에서 함수를 불러오는 방식으로 처리 → 내부적으로 처리
- _DFS, sortNodes와 같은 함수는 private으로 설정하여 외부에서 쉽게 인식할 수 없도록 처리
- 기타 네이밍과 관련된 내용들도 도움이 되었다

2. 오히려 불편했던 부분들 (이해하기 어려웠던 부분들)

리팩터링 패턴에서는 좋은 내용들이 많았지만, 규칙에서는 이해하기 어려운 규칙이 몇개 존재했다

Branch와 관련된 코드들(if, else, switch)

- if 문은 함수의 시작에만 배치한다
- if 문에서 else를 사용하지 말 것
- switct를 사용하지 말 것



학생의 기준에서는 알고리즘 문제를 풀거나 프로그램을 짜야할 때

잘 동작하는 걸 만드는게 우선이기 때문에, else를 사용하지 않는 건 불가능에 가까웠다

2. 오히려 불편했던 부분들 (이해하기 어려웠던 부분들)

리팩터링 패턴에서는 좋은 내용들이 많았지만, 규칙에서는 이해하기 어려운 규칙이 몇개 존재했다

호출 또는 전달 한 가지만 할 것(추상화 수준을 맞추는 것) & 메서드 추출을 너무 많이 했을 때

- 이것 저것 다 메서드로 추출하다보니 한눈에 묶어서 기능을 파악하기 어려웠다

```
// 적당히 메서드를 추출했을 때
- DFS
    - DFS1
    - DFS2
    - DFS3
- BFS
    - BFS1
    - BFS2
    - BFS3
```

```
128  // 그러나 내부에 있는 기능을 모두 바깥으로 빼다보면
129  - DFS
130  - DFS1
131  - DFS2
132  - DFS3
133  - BFS
134  - BFS1
135  - BFS2
136  - BFS3
```

- 메서드로 전부 추출한 이후 클래스로의 코드 이관을 거치지 않으면 함수 쓰임의 분류가 명확하지 않아서 오히려 코드가 복잡해졌다

3. 같이 이야기 해보고 싶은 부분

코드 리팩터링을 마쳤지만 여전히 의문이 남았던 부분

```
let rawMap: number[][] = [
  [2, 2, 2, 2, 2, 2, 2, 2],
  [2, 3, 0, 1, 1, 2, 0, 2],
  [2, 4, 2, 6, 1, 2, 0, 2],
  [2, 8, 4, 1, 1, 2, 0, 2],
  [2, 4, 1, 1, 1, 9, 0, 2],
  [2, 2, 2, 2, 2, 2, 2, 2],
];

// 7, 5 => blue key, lock

let map = new Map();
```

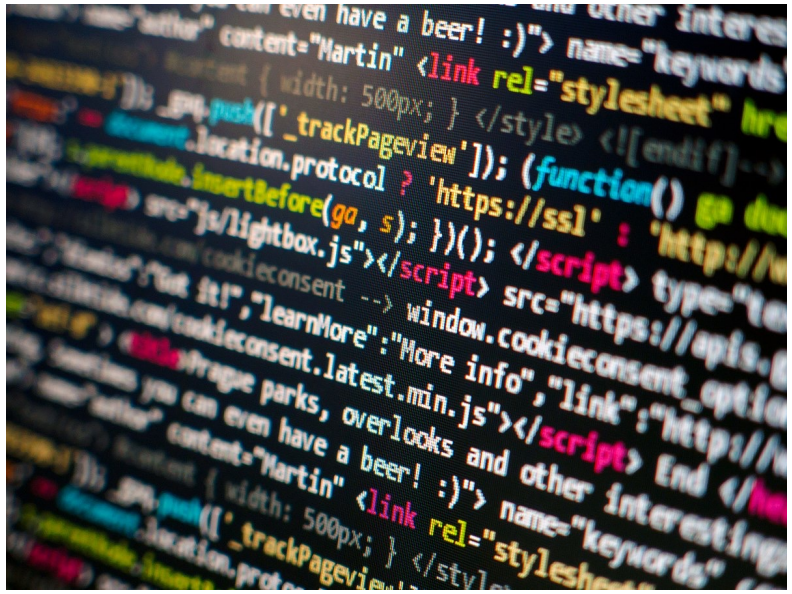
전역 변수

- map이라는 건 전역변수이다

- 어떤 메소드에서 호출하더라도 인자로 넘겨줄 필요가 없다

- 그러나 전역변수로 선언하면, 특정 부분에서는 함부로 호출해서

값을 바꿀 위험성이 있지 않을까?



Q & A