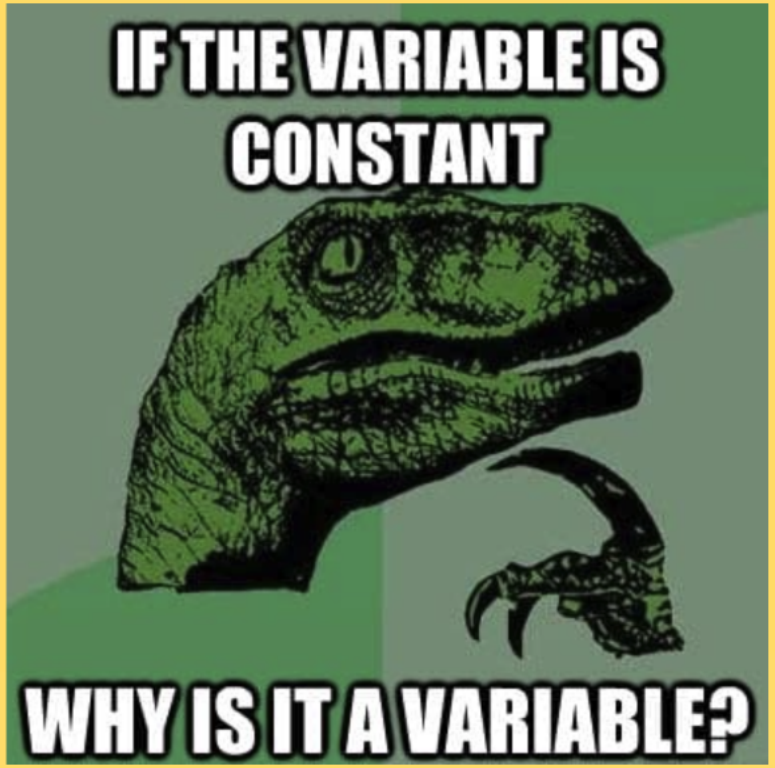

DIVERGENT CHANGE
PRIMITIVE OBSESSION
SWITCH
MUTUAL DATA
LAZY ELEMENT
안 좋은 코드란? - 2편
챌린지반 특강 4강
- 김하연 튜터

6. 가변 데이터? (Mutual Data)

- 소프트웨어 개발에 있어서 중요한 '데이터 불변성'의 개념
- 데이터가 생성된 후에는 데이터가 변경되는 것에는 신중을 가해야한다.

- 변수가 변경안되게 막아볼라고 애쓰는 리팩토링의 일부 예시들
  - 변수 캡슐화하기
  - 변경 로직을 별도의 메소드로 분리
  - 조회 함수와 변경 함수는 항상 분리
  - 필요없다면 setter 함수는 제거하기



```
// 안 좋은 예시
6 references
class User {
    2 references
    public string Name { get; set; }

    1 reference
    public void UpdateName(string newName) {
        Name = newName;
    }
}

var user = new User();
user.UpdateName("Alice");
```

```
// 좋은 예시
6 references
class User {
    // Name 속성을 private setter를 사용하여 불변성을 유지함.
    2 references
    public string Name { get; private set; }

    3 references
    public User(string name) {
        Name = name;
    }

    // 이름을 굳이 변경하겠다면, 이 메소드를 통해 새로운 user 인스턴스를 생성해야함.
    // 이렇게 하면 데이터의 불변성을 유지하는 원칙을 지키면서도, 필요에 따라 변경할 수 있는 유연성을 주는 것.
    1 reference
    public User WithUpdatedName(string newName) {
        return new User(newName);
    }
}

var user = new User("Bob");
var updatedUser = user.WithUpdatedName("Alice");
```


DIVERGENT CHANGE
PRIMITIVE OBSESSION
SWITCH
MUTUAL DATA
LAZY ELEMENT
안 좋은 코드란? - 2편
챌린지반 특강 4강
- 김하연 튜터

7. 뒤엉킨 변경 (Divergent Change)

- 객체 지향 프로그래밍의 중요한 원칙: 단일 책임 원칙 (Single Responsibility Principle, SRP)
- 하나의 클래스는 하나의 기능만 한다.
- 한 클래스에 여러 기능이 집중되어 있어서 하나의 변경이 여러 부분에 영향을 미칠 때 발생함.
- 샷건 수술 (Shotgun Surgery): 변경이 필요할 때 여러 다른 클래스를 수정해야 하는 상황

```
// 안 좋은 코드 예시
// 이 클래스는 금융상품 관리, 데이터 베이스 연결, 데이터 가져오기 혼자 다 해 먹음
1 reference
class FinancialProductManager {
    0 references
    public void AddFinancialProduct() { /*...*/ }
    0 references
    public void RemoveFinancialProduct() { /*...*/ }
    0 references
    public void ConnectToDatabase() { /*...*/ }
    0 references
    public void FetchDataFromDatabase() { /*...*/ }
}
```

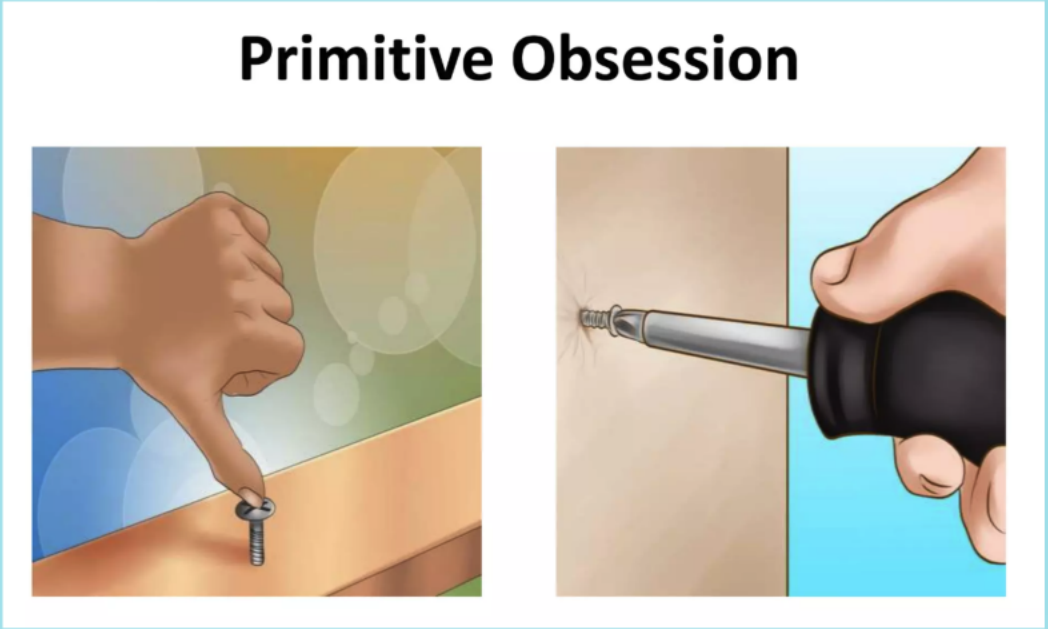
```
// 좋은 코드 예시
// 금융 상품 관리와 데이터베이스 관리를 분리함.
// 각 클래스는 자신의 책임에만 집중한다.
1 reference
class FinancialProductManager {
    0 references
    public void AddFinancialProduct() { /*...*/ }
    0 references
    public void RemoveFinancialProduct() { /*...*/ }
}

0 references
class DatabaseManager {
    0 references
    public void Connect() { /*...*/ }
    0 references
    public void FetchData() { /*...*/ }
}
```


DIVERGENT CHANGE
PRIMITIVE OBSESSION
SWITCH
MUTUAL DATA
LAZY ELEMENT
안 좋은 코드란? - 2편
챌린지반 특강 4강
- 김하연 튜터

8. 기본형 집착 (Primitive Obsession)

- 복잡한 데이터를 단순한 기본형 (int, string)에 과도하게 의존하는 경향
- **복잡한 개념은 기본형 대신 클래스나, 구조체를 사용해서 차라리 해결하자.**
- 예: 전화번호나 화폐 단위는 사실 단순 문자열이나 숫자에서 끝나지 않고, 여러 규칙과 로직을 포함할 수 있음.



```
// 안 좋은 예시
// 화폐단위랑, 전화번호를 그냥 변수로만 처리하고 끝냄
1 reference
class Order {
    0 references
    public string Currency; // 예: "USD", "EUR"
    0 references
    public string PhoneNumber; // 예: "+1-123-456-7890"

    // 다른 메소드들...
}
```

```
// 좋은 예시
// 화폐랑 전화번호를 별도의 클래스로 만들고, 클래스 내에서 각자 로직과 유효성 검사를 추가함.
2 references
class Currency {
    1 reference
    private string Code; // 예: "USD", "EUR"
    0 references
    public Currency(string code) {
        // 유효성 검사...
        Code = code;
    }

    // 관련 메소드들...
}

2 references
class PhoneNumber {
    1 reference
    private string Number; // 예: "+1-123-456-7890"
    0 references
    public PhoneNumber(string number) {
        // 유효성 검사...
        Number = number;
    }

    // 관련 메소드들...
}

1 reference
class Order {
    0 references
    public Currency Currency;
    0 references
    public PhoneNumber PhoneNumber;

    // 다른 메소드들...
}
```


DIVERGENT CHANGE
PRIMITIVE OBSESSION
SWITCH
MUTUAL DATA
LAZY ELEMENT
안 좋은 코드란? - 2편
챌린지반 특강 4강
- 김하연 튜터

9. 반복되는 Switch문 (Switch, If)

- switch, if문이 코드에 중복되는 문제
- switch문은 사실 협업에서 보기 힘듬 (코드 리뷰시 백퍼센트 까이는 구조)
- 객체 지향 프로그래밍의 다형성을 활용하도록 하자.
- 각 조건에 따른 행동을 별도의 클래스로 정의하고, 이걸 인터페이스나 추상 클래스로 다루는게 멋진 코드

```
// 안 좋은 코드
4 references
public enum AnimalType {
    1 reference
    Dog,
    1 reference
    Cat,
    1 reference
    Bird
}

0 references
public class Animal {
    1 reference
    public AnimalType Type { get; set; }

    0 references
    public string MakeSound() {
        switch (Type) {
            case AnimalType.Dog:
                return "Bark";
            case AnimalType.Cat:
                return "Meow";
            case AnimalType.Bird:
                return "Tweet";
            default:
                throw new NotImplementedException();
        }
    }
}
```

```
// 좋은 코드
3 references
public interface IAnimal {
    3 references
    string MakeSound();
}

0 references
public class Dog : IAnimal {
    1 reference
    public string MakeSound() {
        return "Bark";
    }
}

0 references
public class Cat : IAnimal {
    1 reference
    public string MakeSound() {
        return "Meow";
    }
}

0 references
public class Bird : IAnimal {
    1 reference
    public string MakeSound() {
        return "Tweet";
    }
}
```


DIVERGENT CHANGE
PRIMITIVE OBSESSION
SWITCH
MUTUAL DATA
LAZY ELEMENT
안 좋은 코드란? - 2편
챌린지반 특강 4강
- 김하연 튜터

10. 성의 없는 요소 (Lazy Element)

- 너무 오바하지 말자.
- 필요하지 않는 클래스, 메소드, 인터페이스는 그때 그때 삭제해주거나, 인라인화를 하자.

```
// 안 좋은 코드
2 references
class UnnecessaryClass {
    1 reference
    public int Add(int a, int b) {
        return a + b;
    }
}

1 reference
class Calculator {
    1 reference
    private UnnecessaryClass helper = new UnnecessaryClass();

    0 references
    public int CalculateSum(int x, int y) {
        return helper.Add(x, y);
    }
}
```

```
// 좋은 코드
1 reference
class Calculator {
    0 references
    public int CalculateSum(int x, int y) {
        return x + y;
    }
}
```


DIVERGENT CHANGE
PRIMITIVE OBSESSION
SWITCH
MUTUAL DATA
LAZY ELEMENT
안 좋은 코드란? - 2편
챌린지반 특강 4강
- 김하연 튜터