

# Open Closed Principle

Daum Corp.  
백명석

# 발표목차

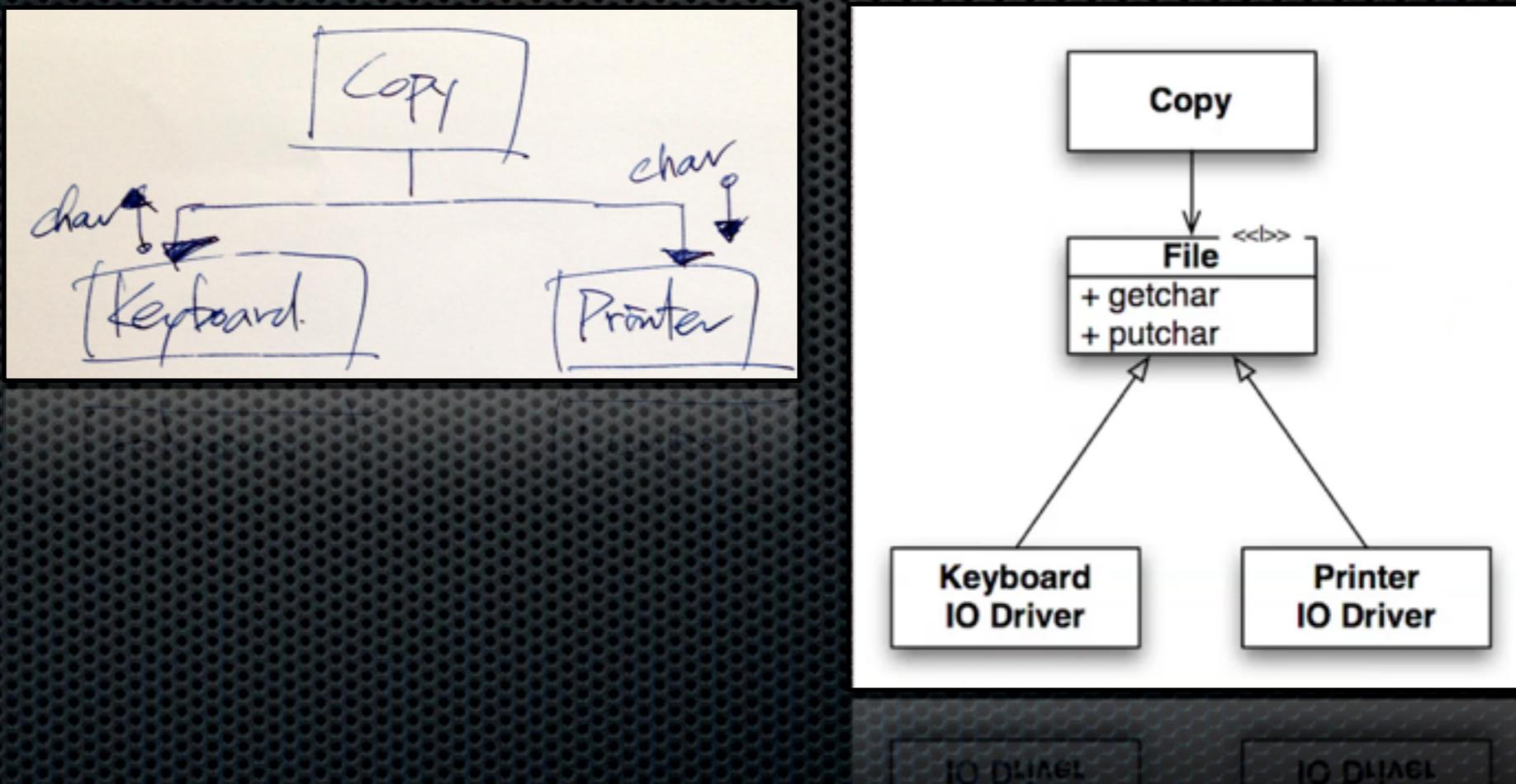
- Open and Closed
- Copy/POS Example
- OCP가 가능한 것인가 ?
- A Smelly Design
- Rigidity / Fragility / Immobility Problem
- De-Orderizing the Design - Git
- De-Orderizing the Design - 결과
- The Lie
- Two Solutions
- Agile Design Example

# Open and Closed

- Bertrand Meyer, “Object-oriented Software Construction”
- Open for extension
  - Add a new feature by adding a new type
- But Closed for modification
  - High Level Policy shouldn't be modified
- Easy to change the behavior of the module
- without having to change the source code of that module

# Copy Example

- Copy Module을 컴파일도 안하고 Low Level Details 를 변경할 수 있다(예. 장치 추가)
- Abstraction and Inversion
  - insert abstract interface between copy and device
  - cause the inverted dependencies



# POS Example

```
10 void checkOut(Receipt receipt) {  
11     Money total = Money.zero;  
12     for(Item item : items) {  
13         total += item.getPrice();  
14         receipt.addItem(item);  
15     }  
16     Payment p = acceptCash(total);  
17     receipt.addPayment(p);  
18 }
```

- 현찰을 받는 경우는 잘 동작.
- 신용카드를 받고자 할때는 확장을 해야함.

```
20     Payment p;  
21     if(credit)  
22         p = acceptCredit(total);  
23     else  
24         p = acceptCash(total);  
25     receipt.addPayment(p);
```

53

receipt.addPayment(p);

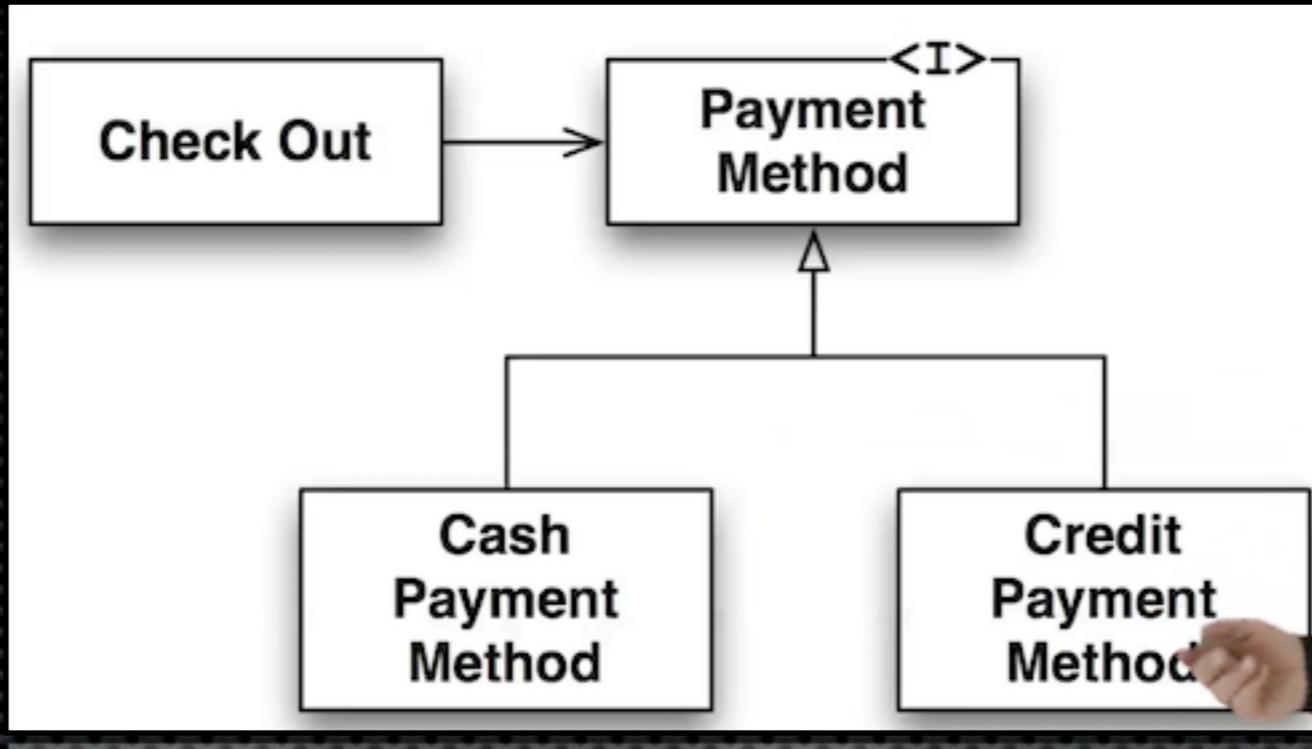
# POS Example

- OCP 위반
  - 확장을 위해 소스를 수정했다.
- 해결책
  - 확장이 필요한 행위를 Abstraction

```
10 void checkOut(Receipt receipt, PaymentMethod pm){  
11     Money total = Money.zero;  
12     for(Item item : items) {  
13         total += item.getPrice();  
14         receipt.addItem(item);  
15     }  
16     Payment p = pm.acceptPayment(total);  
17     receipt.addPayment(p);  
18 }
```

[8]

# POS Example



- 의존성 변화에 주목
  - CheckOut 알고리즘은 구현체에 의존하지 않음
  - PaymentMethod의 구현체는 Abstraction(PaymentMethod)에 의존
- CheckOut 모듈 수정 없이, PaymentMethod를 확장 할 수 있다.

# Is This Possible ?

- OCP를 준수하면 Modification을 완벽하게 제거할 수 있나?
  - 이론적으로는 OK
  - But 비실용적
- 2 Problems
  - main partition
  - Crystal ball problem

# A Smelly Design - 비용 출력

```
public void printReport(ReportPrinter printer) {  
    int total = 0;  
    int mealExpenses = 0;  
  
    printer.print("Expenses " + getDate() + "\n");  
  
    for (Expense expense : expenses) {  
        if (expense.type == BREAKFAST || expense.type == DINNER)  
            mealExpenses += expense.amount;  
  
        String name = "TILT";  
        switch (expense.type) {  
            case DINNER:  
                name = "Dinner";  
                break;  
            case BREAKFAST:  
                name = "Breakfast";  
                break;  
            case CAR_RENTAL:  
                name = "Car Rental";  
                break;  
        }  
        printer.print(String.format("%s\t%s\t$%.02f\n",  
            ((expense.type == DINNER && expense.amount > 5000)  
                || (expense.type == BREAKFAST && expense.amount > 1000)) ? "X" : " ",  
            name, expense.amount / 100.0));  
  
        total += expense.amount;  
    }  
  
    printer.print(String.format("\nMeal expenses $%.02f", mealExpenses / 100.0));  
    printer.print(String.format("\nTotal $%.02f", total / 100.0));  
}
```

```
@Test  
public void printEmpty() {  
    report.printReport(printer);  
  
    assertEquals(  
        "Expenses 9/12/2002\n" +  
        "\n" +  
        "Meal expenses $0.00\n" +  
        "Total $0.00",  
        printer.getText());  
}
```

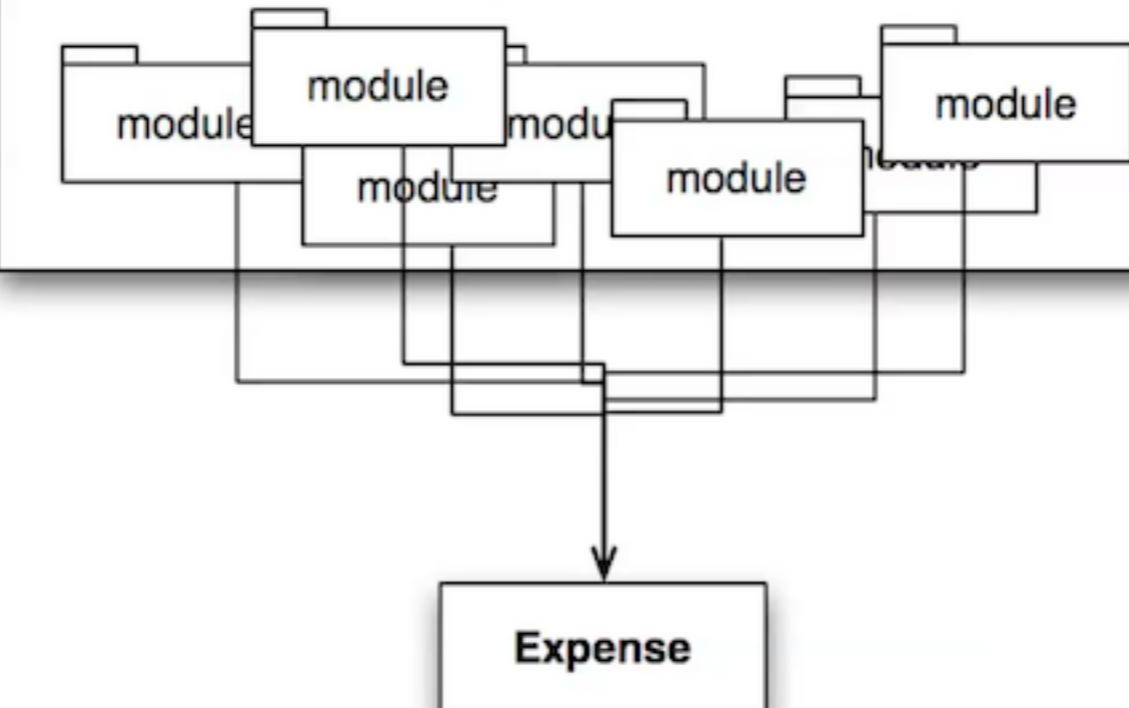
- 꽤 복잡. 이해를 해야 함. 다행히 유닛 테스트 존재

# 문제점

- SRP 위반
  - 비즈니스 규칙 + 메시지 생성 + 포맷팅
- OCP 위반
  - 비즈니스 규칙을 확장하려면 Modify가 필요
  - 메시지 생성/포맷팅을 확장하려면 Modify가 필요

# 문제점 - meal type 추가시 - Rigidity

## Huge interconnected Corporate Accounting System



Expense

Expense

# Fan-out Problem - Fragility

```
if (expense.type == BREAKFAST || expense.type == DINNER)
    mealExpenses += expense.amount;

String name = "TILT";
switch (expense.type) {
    case DINNER:
        name = "Dinner";
        break;
    case BREAKFAST:
        name = "Breakfast";
        break;
    case CAR_RENTAL:
        name = "Car Rental";
        break;
}
printer.print(String.format("%s\t%s\t%.02f\n",
    ((expense.type == DINNER && expense.amount > 5000)
     || (expense.type == BREAKFAST && expense.amount > 1000)) ? "X" : " ",
    name, expense.amount / 100.0));
```

ANSWER

```
ANSWER: 50000000.000000 X
          [ ] 50000000.000000 == BREAKFAST 50000000.000000 == DINNER ) 5 . X
          50000000.000000 == DINNER 50000000.000000 == BREAKFAST > 1000
```

- 모든 Design Smell중 Fragility가 가장 먼저 제거해야 할 대상이다.

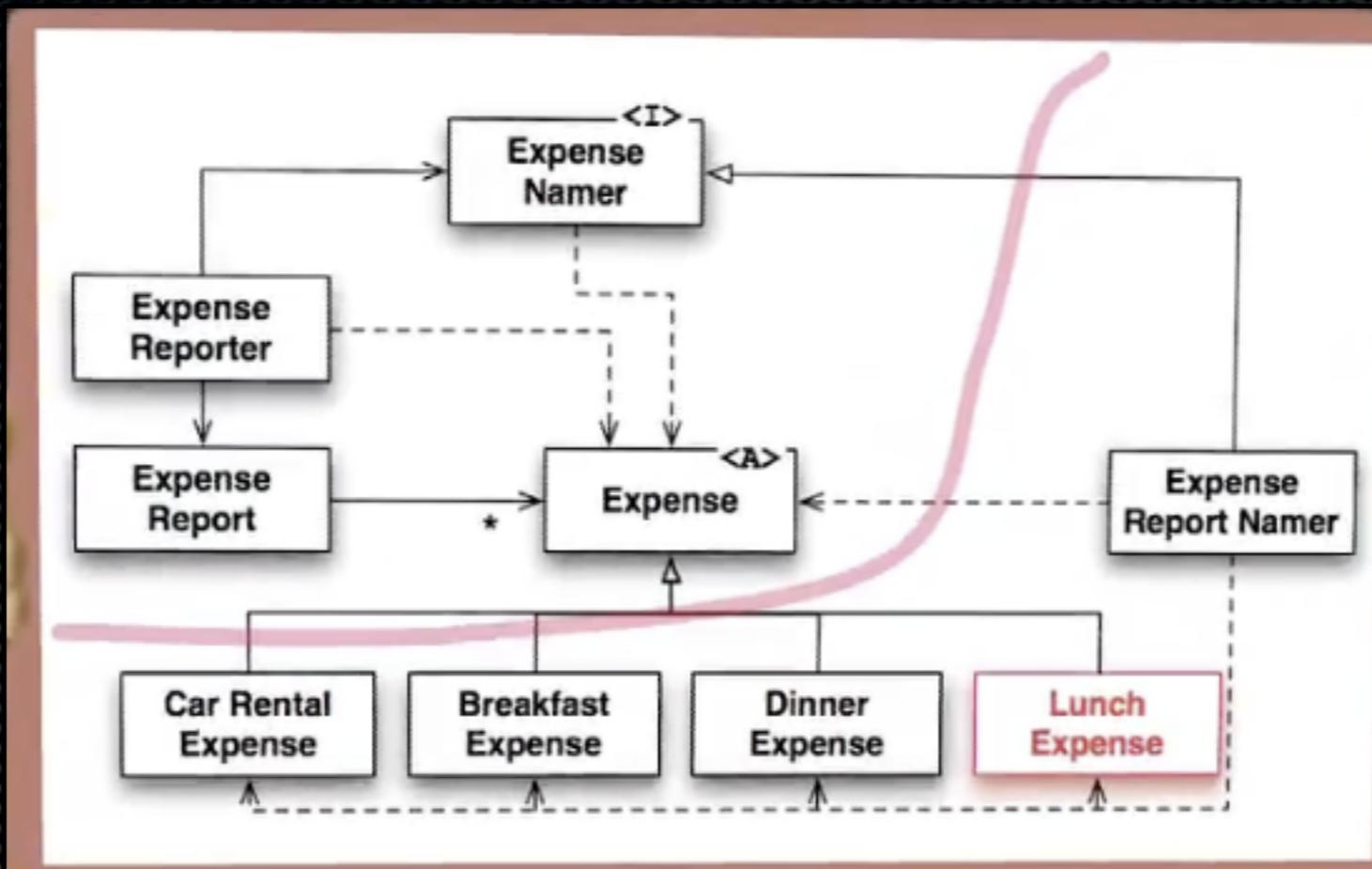
# 근본적인 비즈니스 변경 가정 - Immobility

- 고객에게 회계 시스템을 제공하기 위한 모든 기능을 처음부터 끝까지 제공하는 기업용 솔루션
- 새로운 타겟 마켓
  - 시스템의 대부분의 기능을 필요로 하지 않는 작은 비즈니스 영역
- 시스템을 컴포넌트로 분화하기로
- 작은 플러그인(식대 계산, 특별 점심 식대, 자동차 렌트 등)들에 대해 비용을 청구하기로
- 진짜로 원하는 것은 jar를 **별도로 판매/배포**하는 것.
  - switch/if 문장들에 의존하고 있어서 분리 불가.

# De-Orderizing the Design

- <https://github.com/unclebob/Episode-10-ExpenseReport.git>
- <https://github.com/msbaek/expense>

# De-Orderizing the Design



- 모든 의존성은 Application을 향함

# The Lie

- 고객이 주말(날짜 개념), Transportation 관련된 새로운 기능을 요구하면 대책이 없다.
- 아무도 이런 얘길 해주지 않아서다. 만일 미리 얘기해줬다면 이런 요구사항을 수용할 수 있도록 Abstraction을 적용했을 것이다.
- 내가 알았더라면 OCP를 준수하도록 했을텐데. 새로운 기능을 수정없이 확장할 수 있도록 설계했을 것이라 말이다. 내가 미리 알았다면 말이다.
- 그럼 OCP는 앞으로 어떤 확장이 필요할 지 알아야만 제대로 할 수 있다는 말인가 ?

# The Lie

- 당신이 아무리 잘 찾고, 잘 예측해도 고객은 반드시 당신이 준비하지 못한 것에 대한 가능 추가/변경을 요구 한다. - Unknown Unknowns
- 미래의 변경으로부터 보호 받도록 Abstraction을 적용하여 설계하는 것은 쉽다. 만일 미래에 어떤 변경이 있을지 알 수 있다면 말이다. 하지만 우린 그런 미래를 알 수 있는 Crystal ball이 없다.
- 고객은 우리가 변경으로 보호할 수 있도록 구현하는 것을 잊은 부분에 대해서 변경을 요구하는 능력을 가지고 있다.

# The Lie

- 이게 사람들이 말하기를 꺼리는 OCP, OOD에 대한 하나의 더러운 비밀이다.
  - OCP, OOD는 당신이 미래를 예측할 수 있을 때만 해당 기능을 보호할 수 있다.

# Two Solutions

- 그럼 어떻게 해야 하나 ? 미래를 예측하지 못하는데…
- 완벽한 선견력이 필요하다면 객체지향의 장점은 무엇인가 ?
- 지난 30년간 SW 산업은 이 문제와 투쟁해 왔다.
- 이러한 노력의 일환으로 Crystal Ball의 필요성을 제거하기 위한 2가지 주요한 접근법을 식별했다.

# Big Design Up Front(BDUF)

- 조심스럽게 고객과 문제 영역을 고찰한다
- 고객의 요구사항을 예측하여 도메인 모델을 만든다
- OCP가 가능하도록 도메인 모델에 추상화를 적용한다
- 변경된 가능성이 있는 모든 것들에 대한 청사진을 얻을 때까지 헛된 짓을 계속한다.
- 문제
  - 대부분의 경우 필요치 않는 추상화로 도배된 매우 크고, 무겁고 복잡한 쓰레기 설계를 만든다.
  - 추상화는 유용하고 강력한 만큼 비용도 크다.

# Agile Design

- 실용적이고, 반응을 하는 방법
- 가장 좋은 예시법은 은유법(메타포)이다

# Agile Design - 은유법

- 일련의 병사들이 적군의 사격에 포위됐다
- 총탄이 난발하고 있는 가운데 참호에 숨어있다.
- 적을 향해 집중 사격할 수 있으면 전투에서 승리한다.
- 문제는 어디에 적이 있는지 모른다는 것이다.
- 어떤 방향에서 적들이 총을 쏘는지 모른다. 만일 일어나서 방향을 살펴보려고 한다면 적을 찾고 겨냥하기 전에 총에 맞을 것이다.

# Agile Design

- 그래서 상사가 실행 가능한 결정을 내린다.
- “존슨 일어나”. 탕탕탕. 이제 총알이 날아온 방향을 안다.
- Agile Design은 이런 것이다.
- 최대한 빨리 고객의 요구사항을 끌어낼 수 있는 가장 단순한 일을 한다.
- 그럼 고객은 그 결과물에 대해 요구사항 변경(사격)을 시작한다.
- 그럼 어떤 변경이 요구되는지 알게된다.

# Agile Design

- 변화에 대한 가장 좋은 예측은 변화를 경험하는 것
- 발생할 것 같은 변화를 발견한다면 향후 해당 변화와 같은 종류의 변화로부터 코드를 보호할 수 있다.
- 고객이 요구할 모든 종류의 변경을 완벽하게 예측하고 이에 대한 변경에 대응하기 위해 Abstraction을 적용하는 대신,
- 고객이 변경을 요구할 때까지 기다리고 Abstraction을 만들어서 향후 추가적으로 재발하는 변화로부터 보호될 수 있도록 하라.

# Agile Design

- Agile Designer는 주단위 정도로 간단한 원가를 Deliver한다.
- 고객이 변경을 요구하면 Agile Designer는 코드를 리팩토링해서 그런 종류의 변경을 쉽게 할 수 있도록 Abstraction을 추가한다. OCP를 준수하도록

# Agile Design in Practice

- 물론 우리는 실제로 BDUF과 Agile 두 극단 사이에 살고 있다.
  - BDUF를 피해야 하지만 No DUF도 피해야 한다.
- 시스템에 대해서 사고하고 Decoupled 모델을 사전 설계하는 것은 가치있는 일이다.
- 하지만 간단하고 적은 면에 있다.
- 우리의 목적은 시스템의 기본 모양을 수립하는 것이지 모든 작은 상세까지 수립하는 것은 아니다.
- 문제에 대해서 과하게 생각하면 유지보수 비용이 높은 많은 불필요한 추상화를 만들게 된다.

# Agile Design in Practice

- 빨리 자주 Deliver하고, 고객의 요구사항 변화에 기반하여 리팩토링하는 것은 매우 가치 있다.
- 이럴때 OCP가 진가를 발휘한다.
- 하지만 간단한 도메인 모델없이 이렇게 진행하면 방향성 없는 혼란한 구조를 유발한다.

# Example

- 팀에 10명의 개발자가 있다고 가정하자.
- 향후 12주 동안 개발할 신규 프로젝트 주어졌다고 가정하자.
- 어떤 Design Process가 OCP에 가장 적합할까 ?
- 1주를 초기 요구사항의 범위를 한정하는데 보내서 간단한 도메인 모델의 아키텍처를 얻을 것이다.
- 요구사항은 정확하지 않을 것이고, 도메인 모델은 구체적이지 않을 것이다.
- 이때도 팀원들은 코딩을 할수 있다. 하지만 이때는 구현할 가능이 아니라 초기 요구사항과 아키텍처에 주안할 것이다.

# Example

- 어떤 팀들을 이 시기를 Iteration Zero라고 부른다.
- 유저스토리와 이 유저스토리를 둘러싼 아키텍쳐는 iteration 0에서 정확하지 않다.
- Iteration Zero에서 필요한 것은 앞으로의 진행 방향을 잡는 것이기에 초기의 정확성은 불가능하다. 이후 보정된다.
- 그런 후에 팀은 1-2주 반복을 수행한다. 각 반복의 목적은 고객이나 대리인 앞에서 수행 가능한 원가를 얻는 것이다.

# Example

- 사용자가 뭔가 동작하는 것을 보면 그들은 생각하기 시작하고 변경한다. 그런 변경이 Abstraction의 기반이 된다. 이 OCP를 준수하기 위해 팀은 Abstraction을 사용한다.
- 각 반복은 간단한 디자인 세션으로 시작한다. 팀원들이 이 요구된 변경을 살펴보고 향후 유사한 변경에 대해서 코드를 보호하기 위해 어떻게 OCP를 적용할 수 있는지 생각한다.
- 그럼 개발자들은 코드를 리팩토링하고 아키텍쳐 변경을 염두해 두고 새로운 기능을 추가한다. 개발자들은 TDD를 따라야 한다. 그리고 그들의 코드를 깨끗하고 변경하기 쉽도록 유지해야 한다.

# Example

- Iteration이 진행됨에 따라 OCP 적용율이 증가되어야 한다.
- 각 Iteration은 아키텍처를 기술하는 경계를 명시하고 강조해야 한다.
- 그리고 경계를 교차하는 의존성을 관리해야 한다.

# Reprise

- 마술이 아니라 공학이다.
- OCP을 완벽하게 준수하는 것은 불가능하다. 모든 것을 생각해 낼 수는 없다.
- 아무리 철저히 규칙을 준수하고 조심해도 결국 고객은 시스템 전반에 걸친 대대적인 수정이 필요한 변경을 생각해 낼 것이다.

# Reprise

- 당신의 목적은 변경의 고통을 완전히 제거하는 것이 아니다. 이것은 불가능하다. 당신의 목적은 변경을 최소화하는 것이다. 이게 OCP를 준수하는 디자인이 당신에게 주는 잇점이다.
- OCP는 시스템 아키텍처의 핵심이다.
- 완벽한 보호(변경으로부터의)를 얻을 수는 없지만, 얻기위해 투쟁할 필요는 있다.

# Conclusion

- Bertland meyer, “module should be open for extension but closed for modification”
  - 기존 코드를 수정하지 말고 새로운 코드를 추가함으로써 새로운 기능을 추가하도록 시스템을 만들 수 있다는 것을 의미
- OCP에 완벽하게 순응하기 위해서는 미래를 완벽하게 예측할 수 있어야한다.
- 그렇다고 희망이 없는 것은 아니다. 피드백과 리팩토링에 기반한 Iteration 프로세스를 사용함으로써 OCP에 충분히 잘 순응하는 시스템을 만들 수 있다.