경험적 리팩터링

2014.08

차민창

목차

취지

- **1일차**
  - 이론적
  - 체계적
- **2일차(오늘 강의)**
  - **경험적**
  - **휴리스틱**

1. 경험적 리팩터링

리팩터링 왜 해야 하는가?

- **리팩터링의 기대효과는 무엇인가?**

- **자주 하는 질문**
  - 우리 팀은 비즈니스적으로 어떤 상황인가?
  - 대체적인 개발자 수준은 어떤가?
  - 사용할 수 있는 시간은 어느정도인가?
  - 우리가 선택할 수 있는 리팩터링 중 이게 가장 효과적인가?

- **결국 맥락(Context)에 근거한 개발자의 합리적 판단이 매우 중요하다고 봄**

리팩터링 해부

- **지난 몇년 간 해온 리팩터링을 곰곰이 생각해보았을 때...**

- **접근방식에 따른 분류**
  - 구조 리팩터링 프레임워크 변경, 구조 패턴 적용
  - 코드 리팩터링 Rename, Extract Method, 단순화 시키기

- **리팩터링 지점에 따른 분류**
  - 오래되거나 제어가 어려운 소프트웨어 혹은 코드 레거시 코드
  - 현재 개발 중인 코드 개발 중 지속적으로 구고적/소규모 리팩터링

리팩터링 해부(Cont`)

오늘 주로 얘기할 주제

- **지난 몇년 간 해온 리팩터링을 곰곰이 생각해보았을 때...**

- **접근방식에 따른 분류**
  - 구조 리팩터링 프레임워크 변경, 구조 패턴 적용
  - **코드 리팩터링 Rename, Extract Method, 단순화 시키기**

- **리팩터링 지점에 따른 분류**
  - **오래되거나 제어가 어려운 소프트웨어 혹은 코드 레거시 코드**
  - 현재 개발 중인 코드 개발 중 지속적으로 구고적/소규모 리팩터링

2. 코드 리팩터링 사례

코드 리팩터링의 타당성

긍정적/부정적 리듬

- **코드를 읽어도 이해하기 어려움**
  - 변수 이름이 s
  - s는 무엇일까?
- **이해하기 어려우니, 수정하기 어려움**
  - s의 목적과 의도를 역추적하는 탐정이 되어야 함
- **어려움을 안고 수정하다 보면 위험함**
  - A,B 수정 후 배포를 함
  - 하지만 배포 후 A,B,C를 '함께 ' 수정했어야 한다는 것을 알게 됨
- **장애가 나서 복구하는 데 많은 시간을 소비함**

코드 리팩터링의 타당성(Cont`)

긍정적/부정적 리듬

1. 이해가 어려움
2. 수정이 어려움
3. 장애
4. 위축/소극적
5. 다음에 그곳을 수정하는 사람도 건드리지 않고 1번 부터 계속 반복 됨

해결책은?

코드 리팩터링의 타당성(Cont`)

긍정적/부정적 리듬

1. 이해가 쉬움
2. 수정이 쉬움
3. 문제 없이 배포
4. 성공경험으로 인해 더 적극적 개선

코드 리팩터링의 타당성(Cont`)

첫 번째 교훈

- **부정적 리듬을 긍정적 리듬으로 전환**
  - 위험을 무릅쓰더라도 한번은 **고리를 끊고** 이해하기 쉽게 만들어야 함

코드 리팩터링의 타당성(Cont`)

이해하기 어려운 코드의 영향

2011년 어느 밤…

‘String callList(String service, String userid, String starttime, String endtime, String start, String end)’ 메서드를 개선하고 있다.

- **개발자A) service하고 userid는 알겠는데 starttime, endtime, start, end는 어떤 의미야?**
- **개발자B) 음, 뭐더라. 아마 starttime~endtime은 리스트 가져올 때 검색시간범위을 말하는 거야. (잠시 뭔가를 확한다.) 그리고 start, end는 음… 우선 소스 한번 보자. (소스를 잠시 본다.) 아 아마도 start, end는 페이지를 말하는 거야. 예를 들어 1페이지에서 20페이지면 1,20 이렇게 넣어주면 된다.**
- **개발자A) 아~ 그런데 네가 방금 설명해준 내용을 코드를 통해 드러낼 수는 없을까?**
- **개발자B) 아…**

결국 해당 메서드를 한번 실행해보기 까지 약 10분 정도가 걸렸다.

코드 리팩터링의 타당성(Cont`)

이해하기 어려운 코드의 영향

2011년 어느 날 근무시간에...

WHERE type = ? 이 포함 된 SQL을 수정하고 있었다. type이 뭐지?

1. **데이터베이스의 스키마와 저장된 데이터를 보았다. 그런데 관련 필드가 int 형이고 1,2,3이 저장되어 있는 것 말고는 더 이상의 정보가 없었다.**
2. **관련 모델 클래스를 보았다. type이 1이냐 2냐에 따라 분기하는 조건을 보았지만 분기별 코드만을 가지고는 1,2,3이 무엇인지 정확히 알 수 없었다.**
3. **옆 동료에게 물어봤다. 뭔가 예상 가는 게 있지만 확실히 모르겠다고 한다.**
4. **마지막으로 JSP에서 관련 코드를 찾아보고 있는 중 결국 옆 동료가 다른 동료에게 물어봐 알려주었다. type은 뉴스기사유형이었고 1은 텍스트, 2는 포토, 3은 Vod 유형의 기사였다.**

이 사실을 알기 까지 약 15분 정도가 걸렸다.

코드 리팩터링의 타당성(Cont`)

두 번째 교훈

- **읽기 어려운 코드는 생각보다 많은 시간을 허비하게 만듬**
- **1명이 한번 쓴 코드가 N명에게 N차례 읽힘**
- **그렇다면 이해하기 쉬운 코드를 작성하려면 어떻게 해야 하나?**
  - 배려
  - 인지
  - 표현기술

Magic Number를 사용하지 말자

```
if (count > 33) {
  return;
}
```

Magic Number를 사용하지 말자(Cont')

```
private static final int maximumTemplateCount = 33;

if (count > maximumTemplateCount) {
  return;
}
```

Magic Number를 사용하지 말자(Cont')

```
if (migrationStatus == null) {
  …
}
```

- **만약 설계자의 의도가 아래 둘 중 하나라면?**
  - 마이그레이션이 안 된 상태를 나타내는 데 null을 사용
  - 마이그레이션이 완료 된 상태는 나타내는 데 null을 사용

Magic Number를 사용하지 말자(Cont')

1. Extract Method

```
private boolean notMigrated(String status) {
  return status == null;
}
if (notMigrated(migrationStatus)) {
  …
}
```

2. Enum

```
if (migrationStatus == MigrationStatus.NOT_MIGRATED) {
  …
}
```

의도를 나타내는 이름을 사용하자

```
public String doJob(String str) {
  …
}
```

의도를 나타내는 이름을 사용하자(Cont')

```
public String parse(String html) {
  …
}
```

의도를 나타내는 이름을 사용하자(Cont')

```
public String removeTagsFrom(String html) {
  …
}
```

의도를 나타내는 이름을 사용하자(Cont')

```
public String toPlainTextFrom(String html) {
  …
}
```

의도를 나타내는 이름을 사용하자(Cont')

```
public String removeTagsFrom(String html) {
  …
}

public String toPlainTextFrom(String html) {
  return removeTagsFrom(html);
}
```

의도를 나타내는 이름을 사용하자(Cont')

Context를 활용하자

```
public String make() {
  …
}
```

의도를 나타내는 이름을 사용하자(Cont')

Context를 활용하자

```
public class PlainTextMaker {
  public String make() {
    …
  }
}
```

의도를 나타내는 이름을 사용하자(Cont')

Context를 활용하자

```
public class FacebookRemoteAPI {
  public String makeBody() {
    …
  }
}
```

의도를 나타내는 이름을 사용하자(Cont')

Context를 활용하자

```
public class FacebookRemoteAPI {
  public String makeContents() {
    …
  }
}
```

의도를 나타내는 이름을 사용하자(Cont')

```
public class HttpPhotoUploaderStep2 {
  …
}
```

읽기 쉽게 요약하자

```
public List<File> findAll() {
  …
  ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
  Enumeration<URL> resources = classLoader.getResources(".");

  List<File> dirs = new ArrayList<File>();
  while (resources.hasMoreElements()) {
    URL resource = resources.nextElement();
    Dirs.add(new File(resource.getFile()));
  }
  …
}
```

읽기 쉽게 요약하자(Cont')

```
public List<File> findAll() {
  …
  List<File> childFiles = childFilesUnderRoot();
  …
}
```

최대한 빨리 매듭짓자

```
public void processRequestOf(Server server) {
  if (server != null) {
    Client client = server.getClient();
    if (client != null) {
      Request current = client.getRequest();
      if (current != null) {
        process(current);
      }
    }
  }
}
```

최대한 빨리 매듭짓자(Cont')

```
public void processRequestOf(Server server) {
  if (server == null) {
    return;
  }

  Client client = server.getClinet();
  if (client == null) {
    return;
  }

  Request current = client.getRequest();
  if (current == null) {
    return;
  }

  process(current);
}
```

가급적 대칭을 지키자

district.setDong(“정자”)
district.setGu(“분당”);

다음으로 어떤 메서드명이 나올 것 같은가?

distirct.setMetropolitanOrSi(“성남”);

예상했던 메서드명이 맞는가?

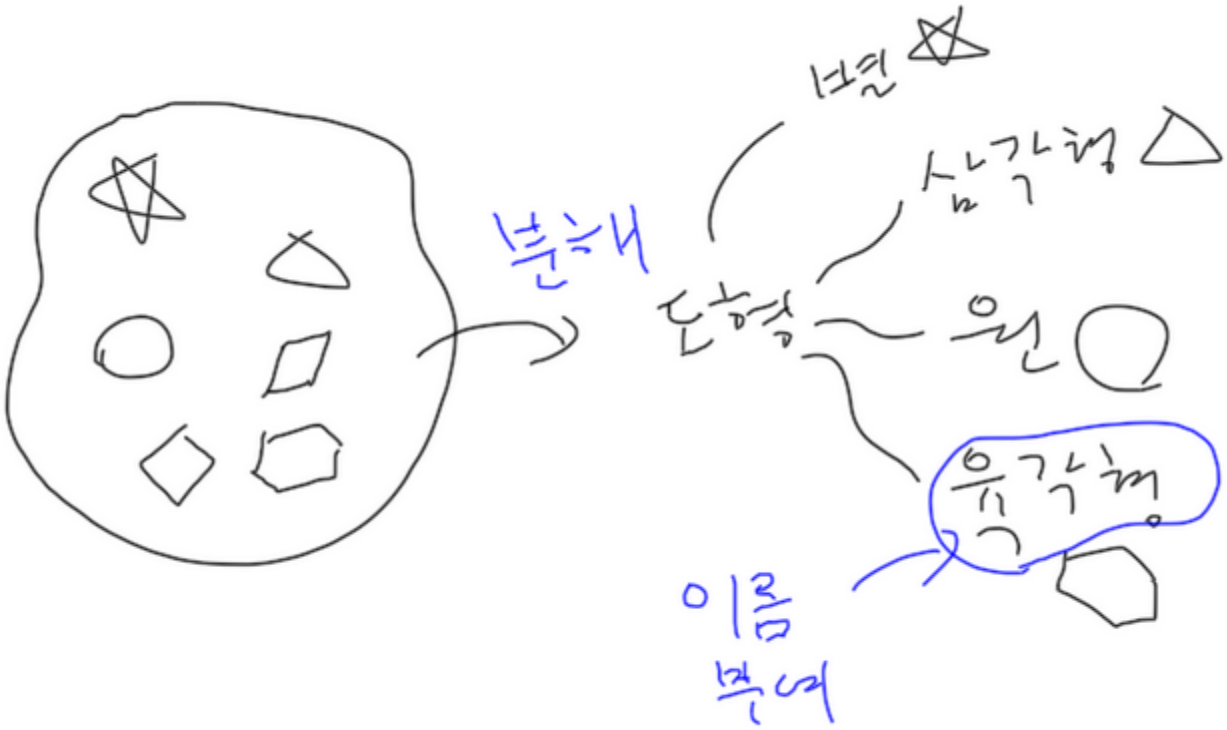
가급적 대칭을 지키자

```
district.setDong(“정자”)
district.setGu(“분당”);
distirct.setSi(“성남”);
```

복잡한 조건식을 피하자

```
public String convertIntoPlainTextFrom(Magazine... magazines) {
        StringBuffer result = new StringBuffer();

        for (Magazine m : magazines) {
                if ("애완견".equals(m.category().toUpperCase())) {
                        if (m.ranking() <= 20) {
                                result.append("[인기있는 애완견 잡지]");
                        } else if (m.ranking() <= 40) {
                                result.append("[볼만한 애완견 잡지]");
                        }
                        result.append(m.name()).append(":").append(m.category());
                        if (m.ranking() == 1) {
                                result.append("☆The Best Pets☆");
                        }
                } else if ("예술".equals(m.category().toUpperCase())) {
                        if (m.ranking() <= 250) {
                                result.append("[인기있는 예술 잡지]");
                        } else if (m.ranking() <= 500) {
                                result.append("[볼만한 예술 잡지]");
                        }
                        result.append(m.name()).append(",").append(m.category());
                        if (m.ranking() == 1) {
                                result.append("☆The Best Art☆");
                        }
                } else if ("자동차".equals(m.category().toUpperCase())) {
                        result.append("[볼만한 자동차 잡지]");
                }

                if (m.name().contains("쿠페") || m.name().contains("Coupe")) {
                        result.append("[스포츠카전문]");
                        result.append(m.name()).append("-").append(m.category());
                } else {
                        result.append(m.name()).append("-").append(m.category());
                }

                if (m.ranking() == 1) {
                        result.append("☆The Best Auto☆");
                }
        } else if ("책".equals(m.category().toUpperCase())) {
                if (m.ranking() <= 5) {
                        result.append("[인기있는 책 잡지]");
                } else if (m.ranking() <= 10) {
                        result.append("[볼만한 책 잡지]");
                }
                result.append(m.name()).append(",").append(m.category());
                if (m.ranking() == 1) {
                        result.append("☆The Best Books☆");
                }
        } else if ("비즈니스".equals(m.category().toUpperCase())) {
                if (!m.name().contains("탈세")) {
                        if (m.ranking() <= 8) {
                                result.append("[인기있는 비즈니스 잡지]");
                        } else if (m.ranking() <= 16) {
                                result.append("[볼만한 비즈니스 잡지]");
                        }

                        result.append(m.name()).append(",").append(m.category());
                }

        } else if ("교육".equals(m.category().toUpperCase())) {
                if (m.ranking() <= 30) {
                        result.append("[인기있는 교육 잡지]");
                } else if (m.ranking() <= 60) {
                        result.append("[볼만한 교육 잡지]");
```

복잡한 조건식을 피하자(Cont')



복잡한 조건식을 피하자(Cont')

```
public String convertIntoPlainTextFrom(Magazine... magazines) {
  StringBuffer result = new StringBuffer();

  for (Magazine magazine : magazines) {
    result.append(MagazineType.plainTextOf(each));
    result.append(LINE_SEPARATOR);
  }

  return result.toString();
}
```

전략패턴(GOF)를 활용하면 위와 같이 바꿀 수 있다.

사용하면 안 되는 클래스/메서드가 있을 때

```
/**
 * @deprecated
 */
public class ~~Legacy~~ {
}
```

위와 같이 deprecated를 해놓으면 이클립스 등에서 코딩할 때 아래처럼 취소선이 나와, 개발자의 주의를 환기시킨다.

```
~~CodeRefactoringExercise~~ sut = new CodeRefactoringExercise(new CStop(), new RStop(), new Notifier());
```

사용하면 안 되는 클래스/메서드가 있을 때(Cont')

```
/**
 * @deprecated
 */
public class ~~Legacy~~ {
}
```

사용하면 안 되는 클래스/메서드가 있을 때(Cont')

```
/**
 * @deprecated 이러저러해서 {@link ToBe}를 사용하기 바람
 */
public class Legacy {
}
```

코드로 표현하기 어려운 의도가 있을 때

```
/**
 * 스킨이 수정될 때 함께 수정되어야 하는 각 <code>CustomizableComponentType</code>
 * 의 몇몇 세부 속성들을 표현한다.
 *
 * Skin 클래스에 함께 추가되어도 되지만 Skin에 함께 있어야 하는 부분인지
 * 확신할 수 없어 별도로 분리해놓았다.
 *
 * @see DesignComponentType
 * @see Skin
 * @author 차민창
 */
public class WidgetDependingOnSkin {
```

뻔한 중복주석은 피하자

```
// 이름
private String name;

/**
  * 이름을 반환한다.
  */
public String getName() {
  return name;
}
```

테스트 코드도 이해하기 좋게하자

```
@Test
public void testStop1() {
  …
}

@Test
public void testStop2() {
  …
}
```

테스트 코드도 이해하기 좋게하자(Cont')

```
@Test
// serviceShouldBeStoppedCompletely
public void 서비스를_완전히_중단한다() {...
}

@Test
// serviceShouldBeStopped_ButCanReadContents
public void 서비스를_중단하지만_읽기는_가능하다() {
  ...
}
```

만약...

```
String callList(service, userid, searchCriteria);

public class SearchCriteria {
  private long startTime;
  private long endTime;
  private int startPage;
  private int endPage;
  ...
}
```

만약...(Cont`)

```
if (newsType == NewsType.TEXT) {
  ...
}

if (newsType == NewsType.PHOTO) {
  ...
}

if (newsType == NewsType.VOD) {
  ...
}
```

짝 프로그래밍으로 실습

지금까지 소개했던 방법과 본인만의 방법을 활용하여,
짝 프로그래밍으로 읽기 어려운 레거시 코드를 개선해보자.

CodeRefactoringExercise.java

짝 프로그래밍으로 실습(Cont`)

짝 프로그래밍

- **모든 실습은 짝 프로그래밍을 진행**
- **짝 프로그래밍으로 실습을 진행하는 이유**
  - 코드 리팩터링 사례는 옆 동료하고 깊은 관계가 있음
  - 문제를 둘이 함께 고민해보는 경험

3. 안전하게 리팩터링 하기

검증의 필요성

- 방금 실습하며 수정한 코드가 예전처럼 잘 동작한다는 보장이 있는가?
- 리팩터링을 했을 때 그동안 어떻게 해왔는가?

검증의 필요성(Cont`)

- 읽기 어려운 코드가 있음
- 리팩터링 하고 배포함
- 장애가 발생함
- 움츠려 들고 용기를 잃게 됨
- 결과로 부정적 리듬을 유지하게 됨

검증의 필요성(Cont`)

- **따라서 리팩터링을 할 때는 검증에 대해 항상 함께 고민 해야 함**

- **그렇다면 검증은 어떻게 하는 게 좋을까?**
  - 우선 배포하고 반응을 살펴보기?
  - QA(Quality Assurance)에게 위임하기?
  - 개발자가 APP 등 직접 조작해보며 확인하기?

개발자 테스트의 필요성

문제 발견 시점	관련 자원
개발	담당 개발자
QA	QA 버그 관리 시스템 개발 리더 담당 개발자
배포 후	고객 고객 담당자 장애 관리 시스템 기획자 버그 관리 시스템 유지보수 개발 리더 담당 개발자 QA 배포

언제 검증하는게 합리적?

두 가지 필수 고려 요소

- **[영향력] 수정에 따른 영향력은 얼마만큼인가?**
  - public? private? (Encapsulation)
  - 메서드 혹은 클래스를 사용하는 곳이 몇 군데? (Coupling)

- **[검증방법] 영향 받는 부분의 검증을 어떻게 해야 하나?**
  - 웹브라우저에서 기능 확인?
  - 자동화 된 단위/통합 테스트

자동화 테스트

개요

- 자동화 테스트는 말 그대로 자동으로 수행되는 테스트
- 한번 테스트를 작성해놓으면, 리팩터링 등으로 인해 무엇인가 변경될 때마다 쉽게 다시 실행해볼 수 있음
- 자동으로 실행되기 때문에 사람의 노력이 거의 들어가지 않음

자동화 테스트(Cont`)

API 연동 개발 사례

- **난 뉴스 개발자다. 이번에 블로그 API와 연동하는 부분을 개발해야 한다.**
- **기획에서 전달한 요구사항은 아래와 같다.**
  - 사용자가 덧글을 입력하면 덧글을 화면에 보여줄 때 사용자의 블로그 프로필 이미지를 함께 표시
- **내가 블로그 API 서버 다운이나 점검을 할 때는 어떻게 하냐고 물어보았더니 아래와 같이 답변했다.**
  - 여하간에 이유로 이미지를 못 가져오면 UX에서 미리 정해준 기본 프로필 이미지를 표시
- **난 개발을 시작했다.**

자동화 테스트(Cont`)

API 연동 개발 사례

- **TDD에는 익숙하지 않아 먼저 코드를 작성했다.**

```
public class BlogAPI {
  private HttpClient client = new HttpClient();
  public String profileImageUrlOf(String userId) {
    BlogProfile profile;
    try {
      profile = (BlogProfile)client.get("..." + userId);
    } catch (FailedBlogAPICallException e) {
      LOG.error(e.toString(), e);
      final String defaultImageUrl = "...";
      return defaultImageUrl;
    }
    return profile.profileImageUrl();
  }
}
```

자동화 테스트(Cont`)

API 연동 개발 사례

- **다음으로 테스트 코드를 작성했다.**
- **블로그에 테스트용 아이디 tester가 이미 있다고 하여 해당 아이디를 활용했다.**

```
@Test
public void 블로그_프로필_이미지_URL를_가져온다() {
  BlogAPI sut = new BlogAPI();
  String result = sut.profileImageUrlOf("tester");
  assertEquals("http://image.blog.naver.com/tester.jpg",result);
}
```

- **처음에 API서버의 ACL 때문에 잠시 어려움이 있었지만 ACL을 신청하고 몇 시간 뒤 테스트가 성공했다.**
- **마지막으로 선배에게 코드리뷰만 받으면 된다.**

자동화 테스트(Cont`)

API 연동 개발 사례

- **선배가 테스트 코드를 보더니 묻는다.**
  - "블로그 API 서버가 점검 중일 때 정말 기본 이미지가 반환되는지도 테스트 해야 하는 거 아냐?"
- **의견대로 테스트 코드를 작성하려 했더니 갑작스레 의문이 든다.**
  - 블로그 API 서버를 점검 중으로 만들려면 어떻게 해야 하지?
- **어떻게 테스트 해야 할지 방법을 모르겠다.**

자동화 테스트(Cont`)

왜 테스트가 어려울까?

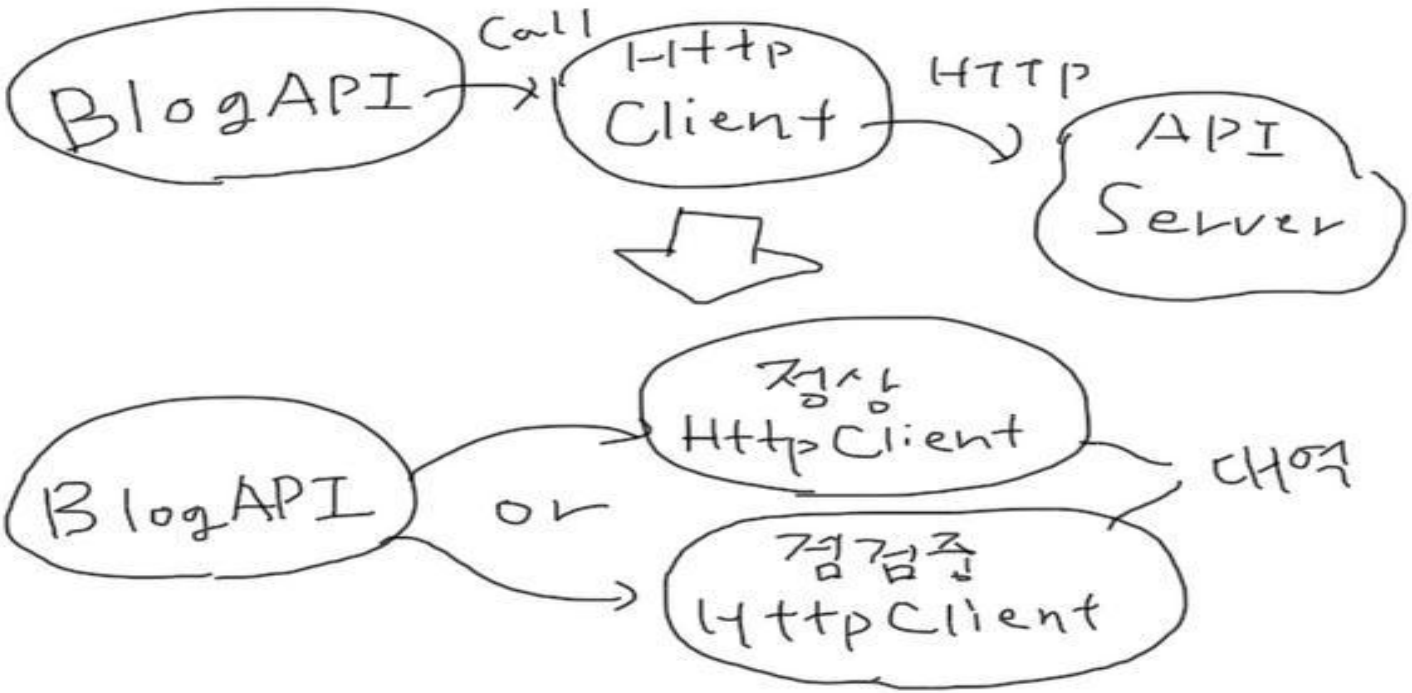
- 지금 작성한 테스트로는 블로그 API 서버의 다양한 상황을 만들어보기 어렵기 때문이라 생각한다.

- 통제할 수 있는 방법이 있을까?

자동화 테스트(Cont`)

Dependency Breaking

- 점검 상황을 만들려면 Dependency Breaking을 해야 한다.
- Dependency Breaking은 테스트 대상의 의존성을 끊어버리고 대역으로 바꿔 치기 하는 것이다.



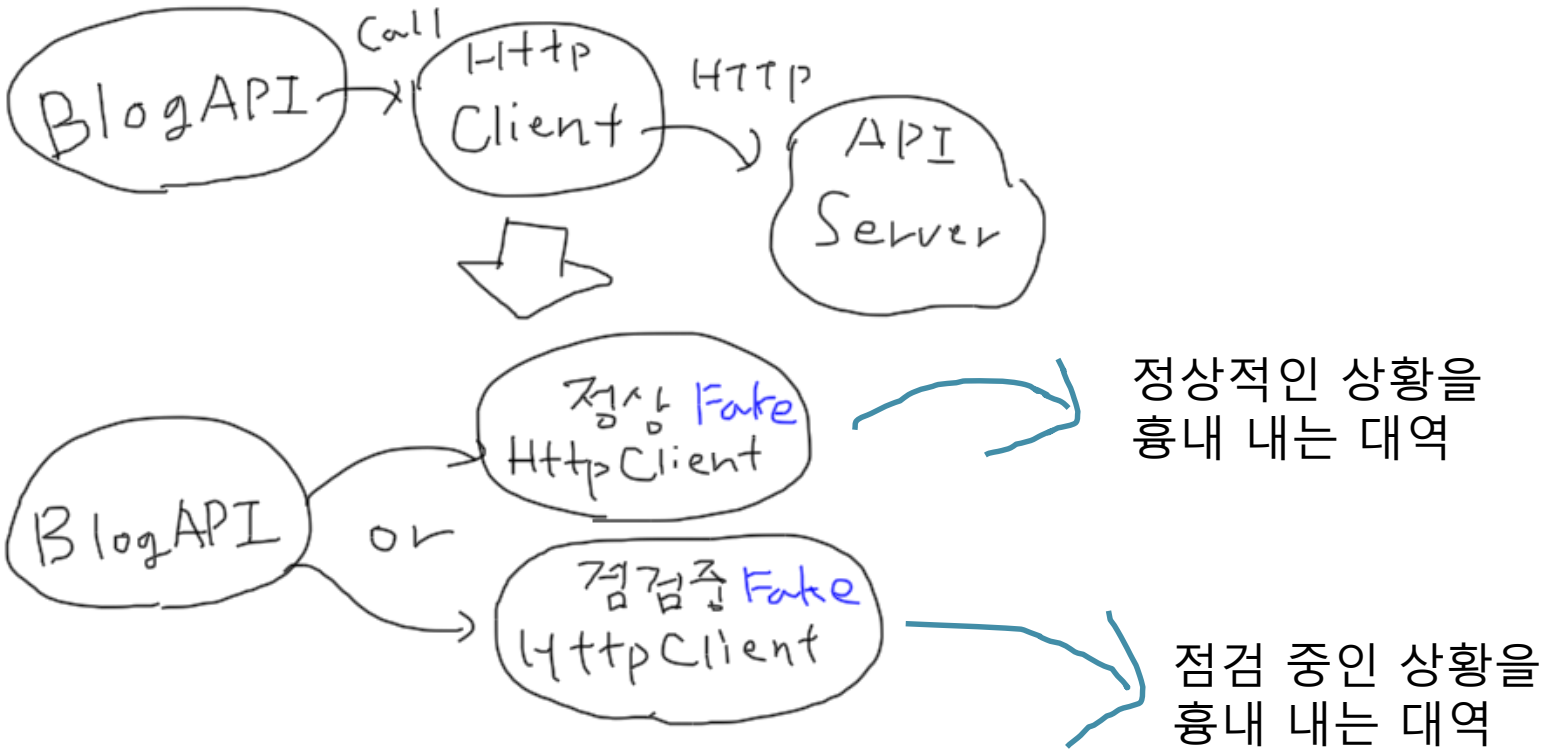
자동화 테스트(Cont`)

대역(Double)이란?

- **테스트가 의존하는 클래스 혹은 메서드를 대신함**
- **미리 정해진 행동을 함**
  - 어떤 행동을 할지 지정하는 것을 Stubbing이라고 부름
- **테스트 실행 후 대역의 어떤 메서드와 인터랙션이 있었는지 검사하기도 함**
  - 이를 보통 Verifying라고 부름
- **대역은 상속 등으로 직접 구현할 수도 있고, Mock Library를 이용할 수도 있다.**

자동화 테스트(Cont`)

대역(Double)이란?



자동화 테스트(Cont`)

대역과 테스트용이성

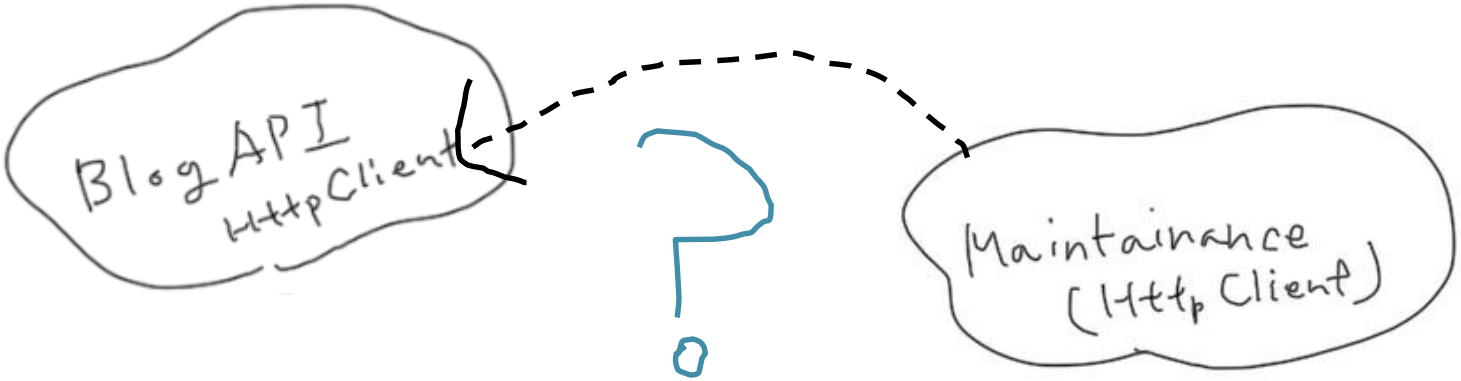
- **Dependency Breaking와 대역의 개념을 숙지했으니 대역을 한번 만들어보자.**

```
private class Success extends HttpClient {
  @Override
  public BlogProfile get(String userId) throws FailedBlogAPICallException {
    BlogProfile profile = new BlogProfile();
    profile.setProfileImageUrl("...");
    return profile;
  }
}
private class Maintainance extends HttpClient {
  @Override
  public BlogProfile get(String userId) throws FailedBlogAPICallException {
    throw new FailedBlogAPICallException("블로그 API 서버 점검중입니다");
  }
}
```

자동화 테스트(Cont`)

대역과 테스트용이성

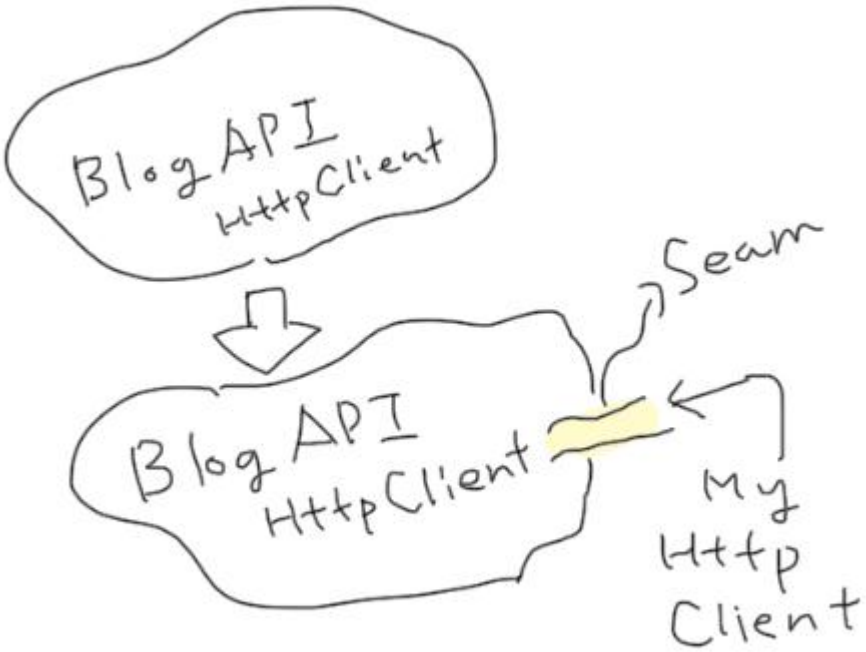
- 이제 대역을 테스트에 적용해봐야겠다. 앗! 그런데 대역을 테스트에 어떻게 적용하지?
- 선배에게 물어봐야겠다. 진짜 HttpClient를 내가 미리 정한대로 동작하는 대역으로 바꿔 치기 하면 된다고 하네.
- 그런데 소스를 살펴보니 바꿔 치기 할 방법이 없는데 어떻게 하지?



자동화 테스트(Cont`)

테스트용이성와 Seam

- 대역을 만들고 테스트를 통제하려 할 때 대역으로 바꿔 치기 할 수단이 없을 때도 많다.
- 이럴 때 BlogAPI는 테스트용이성이 떨어진다고 말한다.
- 테스트용이성을 확보하려면 대역이 비집고 들어갈 수 있는 공간이라 할 수 있는 Seam을 만들어줘야 한다.



자동화 테스트(Cont`)

Seam을 이용한 의존성 교체

- 아래와 같이 Seam을 만들었다.

```
public class BlogAPI {
  private HttpClient client = new HttpClient();

   protected void setHttpClientDouble(HttpClient double) {
     this.client = dobule;
   }
   ...
}
```

자동화 테스트(Cont`)

Seam을 이용한 의존성 교체

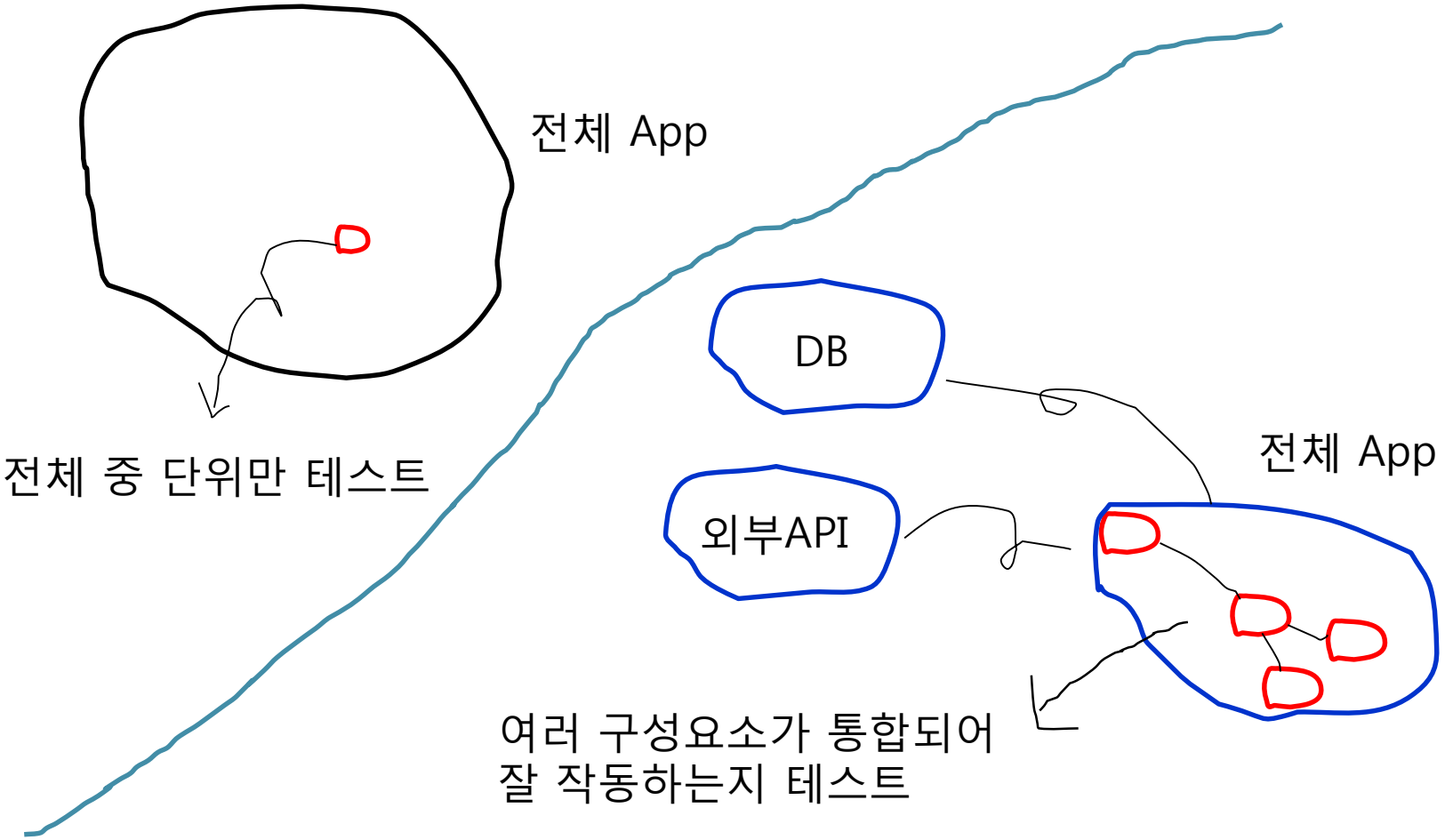
- Seam을 만들고 나니 드디어 선배가 말한 점검중인 상태를 테스트 할 수 있게 되었다.

```
@Test
public void 블로그API서버가_점검중이면_기본_프로필_이미지_URL를_가져온다() {
  BlogAPI sut = new BlogAPI();
  sut.setHttpClientDobule(new Maintainance());
  String result = sut.profileImageUrlOf("anyId");
  assertEquals(DEFAULT_PROFILE_IMAGE_URL, result);
}
```

- 근데 사람들이 단위 테스트, 통합 테스트를 얘기하던데 내가 이번에 만든 것 무슨 테스트지?

자동화 테스트(Cont`)

단위 테스트와 통합 테스트



자동화 테스트(Cont`)
단위 테스트와 통합 테스트

	단위 테스트	통합 테스트
초점	특정 단위를 정밀하게 검사	관련한 여러 구성요소의 유기적 결합
범위	OOP에서 보통 하나의 메서드	여러 클래스, 여러 시스템 등
특징	1. 매우 빠르게 실행 2. 외부에 독립적(대역 사용)	1. 비교적 느리게 실행 2. 외부에 의존성 (예. Blog API 서버 등)
예시	방금 이야기 중 개발자가 마지막에 만든 테스트 외부로부터 완벽하게 고립된 상태에서 BlogAPI가 테스트 됨	방금 이야기 중 개발자가 처음 만든 테스트 BlogAPI, HttpClient, Blog 실제 서버가 연계되어 테스트가 실행 됨

자동화 테스트(Cont`)

단위 테스트와 통합 테스트

- 둘은 상호보완적 관계
- 단위 테스트만 만들면 런타임 시 NPE 등으로 제대로 동작하지 않는 때가 허다
- 통합 테스트만 만들면 세밀한 부분에서 오작동하는 때가 허다하고, 블로그 API 점검과 같이 통제하지 못하는 상황도 생김

자동화 테스트(Cont`)

상태 검증과 행위 검증

- 검증방법으로는 크게 두 가지가 있음
  - 상태 검증
  - 행위 검증

자동화 테스트(Cont`)

상태 검증과 행위 검증

- 상태 검증 예시

```
privte String id;
public boolean isOwn(String userId) {
  return this.id.equals(userId);
}

@Test
public void 사용자가_본인인지_알수있다() {
  User a = new User("userA");

  assertTrue(a.isOwn("userA"));
  assertFalse(a.isOwn("userB"));
}
```

자동화 테스트(Cont`)

상태 검증과 행위 검증

- **행위 검증 예시**

PostService.java

```
private Facebook facebook;
private PostRepository postRepository;

public PostService(Facebook facebook,
PostRepository postRepository) {
  this.facebook = facebook;
  this.postRepository = postRepository;
}

public void write(Post post) {
  if (post.isFacebookPublishable()) {
    facebook.publish(post);
  }
  postRepository.save(post);
}
```

```
@Test
public void 페이스북에_발행할수있다() {
  Post post = new Post("테스트 포스트");
  post.markFacebookPublishable();

  postService.write(post);

  // Facebook에 발행했는지 어떻게 검증해야 하는가?
}
```

자동화 테스트(Cont`)

상태 검증과 행위 검증

- **해결방법#1 - 상속받아서 테스트 가능하게 메서드 조작하기**

```
@Test
public void 페이스북에_발행할수있다() {
  Facebook fakeFacebook = new FakeFacebook();
  PostService postService = new PostService(fakeFacebook, new FakePostRepository());
  Post post = new Post("테스트 포스트");
  post.markFacebookPublishable();

  postService.write(post);

  assertTrue(fakeFacebook.published);
}
private class FakePostRepository extends PostRepository { @Override public void save(Post) {} }
private class FakeFacebook extends Facebook {
  boolean published = false;
  @Override
  public void publish(Post post) {
    this.published = true;
  }
}
```

자동화 테스트(Cont`)

상태 검증과 행위 검증

- **해결방법#2 – Mock 라이브러리 이용하기**

```
@RunWith(MockitoJUnitRunner.class)
public class FacebookTest {
  @Mock private Facebook facebook;
  @Mock private PostRepository postRepository;

  @Test
  public void 페이스북에_발행할수있다() {
    PostService postService = new PostService(facebook, postRepository;
    Post post = new Post("테스트 포스트");
    post.markFacebookPublishable();

    postService.write(post);

    Mockito.verify(facebook).publish(); //publish가 실제로 호출되었나?
  }
}
```

자동화 테스트(Cont`)

상태 검증과 행위 검증

- **실제 행위 검증 사례와 When**

```
@RunWith(MockitoJUnitRunner.class)
public class InitialValuePullerTest {
  private InitialValuePuller sut;
  private final String key = "key";
  @Mock private CoRepository coRepository;
  @Mock private OriginalRepository originalRepository;
  @Mock private LRUKeyUpdateTime keyUpdateTime;
  @Test
  public void initialValueShouldBePulledToCoRepository() {
    Item initialValue = new Item(key, "anyValue");
    Mockito.when(coRepository.lock(key)).thenReturn(true);
    Mockito.when(originalRepository.read(key)).thenReturn(initialValue);
    sut = new InitialValuePuller(coRepository, originalRepository, keyUpdateTime);

    sut.ensurePulled(key);

    Mockito.verify(coRepository).insert(initialValue);
  }
}
```

자동화 테스트(Cont`)

Mock 라이브러리의 기능

- **Mock 라이브러리는 대역(Double) 역할을 해주며 크게 두 가지의 기능을 제공**

- **Stubbing**
  - 테스트 대상의 의존하고 있는 객체의 행위를 조작해줌
    - Mockito.when(coRepository.lock(key)).thenReturn(true);
    - Mockito.when(coRepository.lock(key)).thenReturn(false);

- **Verifying**
  - 특정 시점 후 Mock 상에 발생한 인터랙션이 있었는지 확인해줌
    - Mockito.verify(corepository).insert(initialValue);
    - Mockito.verify(facebook).publish();

Cover & Modify

개요

- 산업표준인 Edit & Pray의 반대 개념
- 수정하기 전 테스트로 안전망(Safety Net)을 만들고 수정하는 것
- 테스트는 수정 시 기존 동작변경이 없음을 보장할 수 있어야 함

Cover & Modify(Cont')

작업절차

1. 리팩터링 할 범위를 커버할 수 있는 테스트 작성

2. 모든 테스트를 통과시킴

3. 필요한 리팩터링 중 일부분을 수행
   (뭔가 실수하면 테스트가 마지막으로 성공한 시점으로 복원하기 위함임)

4. 테스트를 수행해보고 실패한 테스트가 없는지 확인

5. 수정할 부분이 남아있다면 3번으로 돌아가 반복

실습

아래 파일을 Cover & Modify를 활용해 개선 해보자.
또한 각각의 예제가 의미하는 바를 논의해보자.

CoverAndModifyExercise1.java

CoverAndModifyExercise2.java

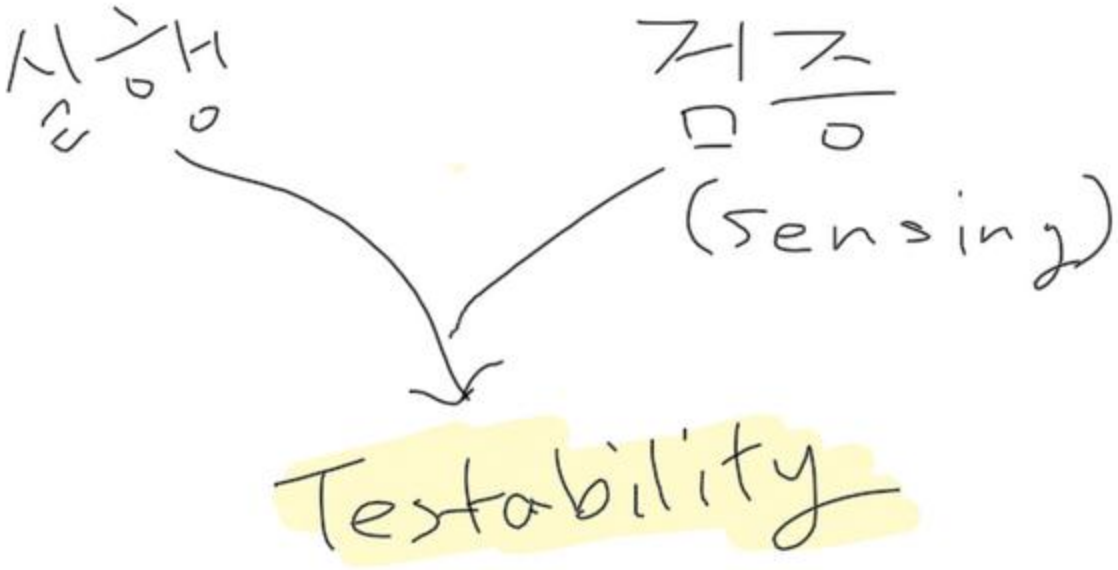
CoverAndModifyExercise3.java

4. 이미 작성 된 코드 테스트 하기

테스트용이성이란?

- 테스트를 할 때 테스트가 용의한지를 나타내는 단어
- 통합 테스트나 UI 테스트에서도 테스트용이성 문제가 있을 수 있지만,
  테스트용이성을 언급할 때는 보통 단위 테스트를 할 때임
- TDD로 개발되지 않은 코드는 대게 테스트용이성 문제가 있음

테스트용이성을 결정짓는 두 가지 요소



딜레마

- 테스트용이성을 확보하려고 하니 기존 코드를 수정해야 함
- 기존 코드를 수정하려니 아무 안전망도 없으니 손대기가 어려움

테스트용이성 확보 때문에 수정을 할 때는 기존 코드를 최대한 보전

방향성

- **지금부터 나오는 예제는 복잡해 보일 수 있음**
- **하지만 침착하게 보면 결국 두 가지를 해결하는 것(Dependency Breaking)**
  - Seam 만들어 내기
  - 대역(Double)으로 바꿔치기
- **결국 테스트용이성 문제는 Seam을 어떻게 만들고, 대역을 어떻게 만들어 넣을지에 대한 문제이다.**

생성이 안될 때

Context Dependence

```
public class Legacy{
  private static final Config config = ConfigFactory.getConfig();
  …
  public Legacy() {
    if (config.get(“name”).equals(“someName”) {
      ..
    }
  }
}
```

- 위에서 ConfigFactory는 다른 클래스를 통해 초기화가 되어야 정상적으로 동작한다고 가정
- 위 클래스를 어떻게 생성할 수 있을까?

생성이 안될 때(Cont')

Context Independence

```
public class Legacy {
  private static final Config config;
  ...
  pubic Leagcy() {
    this(ConfigFactory.getConfig()); // 기존 동작 보존
  }

  public Legacy(Config passedConfig) { // Seam
    this.config = passedConfig;
    if (config.get("name").equals("someName") {
      ...
    }
  }
}
```

생성이 안될 때(Cont')

Test Code

```
public class LegacyTest {
  private static final Config fakeConfig = new Config() {
    public void get(String key) {
      return “someName”;
    }
  }

  @Test
  public void Leagcy를_생성할수있다() {
    new Legacy(fakeConfig);
  }
}
```

실행제어가 안될 때

Singletone + Hidden Dependency

```
public class Legacy {
  public stop(){
    if (ServiceMaintenence.getInstance().isROS()) {
      ..
    } else {
      ..
    }
  }
}
```

- **isROS()의 상태를 쉽게 바꿀 수 없다고 가정**
  - 예) 초기 생성시 해당 값을 데이터베이스에서 가져온다.
- **단위 테스트 시 각 분기별로 실행하려면 어떻게 해야 할까?**

실행제어가 안될 때(Cont')

Static Setter

```
public class ServiceMaintanence {
  ~~private~~ protected ServiceMaintanence() {}

  protected static void setFakeInstance(ServiceMaintenance fakenstance) {
    instance = fakeInstance;
  }

  public ServiceMaintanence getInstance() {...}

  public boolean isROS () {return ros;}
}
```

** 주의할 점은 반드시 테스트가 끝난 후 원래 instance로 원상복귀 시켜야 한다. 그렇지 않으면 다른 테스트에 영향을 미친다.*

실행제어가 안될 때(Cont')

Test Code

```
public class LegacyTest {
  private class FakeServiceMaintenence extends ServiceMaintenence {
    public FakeServiceMaintenence(boolean fakeROS) {
      super.ros = fakeROS;
    }
  }

  @Test
  public void contentsShouldBeRead_WhenROSIsActive() {
    ServiceMaintenence.setFakeInstance(new FakeServiceMaintenence(true));
    sut.stop();
    ...
  }
}
```

결과 검증이 안될 때

```
public class Legacy {
  private Display display; //showLine만 정의되어 있는 Interface

  public Legacy(Display display) {
    this.display = display;
  }

  public void scan(String barcode) {
    ...
    String itemLine = item.name() + " " + item.price();
    display.showLine(itemLine);
    ...
  }
}
```

- 실행은 쉬울 것 같은데, Barcode에 따라 화면에 표시가 잘 되었는지 어떻게 검증할 수 있을까?

결과 검증이 안될 때(Cont')

```
public class LegacyTest {
  private class FakeDisplay implements Display {
    private String lastLine = "";
    @Override
    public void showLine(String line) {
      lastLine = line;
    }
    public String lastLine() {
      return lastLine;
    }
  }
  @Test
  pubic void itemLineShouldBeDisplayed()  {
    Legacy sut = new Legacy(new FakeDisplay());
    sut.scan(someBarcode);
    assertEquals(expectedLine, sut.lastLine());
  }
}
```

테스트를 시작할 수 없을 때

```
public class Legacy {
  private static String name = DataHandlerFactory.getDataHandler().get("name");

  public void execute() {
    doSomething(name);
  }
}
```

- 이 클래스는 참조되는 순간 ExceptionInInitializerError 오류가 발생한다.

테스트를 시작할 수 없을 때(Cont`)

```
public class Legacy {
  private static String name; //name이 이 클래스 참조시에 초기화되지 않게 필드 할당을 피한다.

  public void execute() {
    doSomething(getName()); //처음 호출될 때는 name을 할당해야 하기 때문에 간접 호출을 한다.
  }

  private String getName() {
    if (name == null) {
      name = DataHandlerFactory.getDataHandler().get("name");
    }
    return name;
  }

  public static void setName(String fakeName) {
    name = fakeName;
  }
}
```

테스트를 시작할 수 없을 때(Cont`)

Test Code

```
@Test
public void StaticSetter와_Getter를_이용하여_의존성을_교체한다() {
  Tobe.setName("fakeName");
  new Tobe().execute();
}
```

- 위와 같이 Static Setter를 이용하여 이름을 바꾼다. 내부적으로 getName이 호출될 때 이미 name이 할당되었기 때문에 초기화 되지 않은 DataHandlerFactory를 참조하지 않는다.

정리

- 실무에서는 더 다양한 사례가 있을 수 있지만, Dependency Breaking 시각에서 방금 전 사례들을 응용하면 대부분 해결이 가능할 것으로 예상
- 실무에서는 Fake객체보다는 Mock라이브러리를 이용하여 Mock객체를 활용함
- Mock라이브러리의 목적은 결국 소개한 예제의 Fake객체와 유사하며, 혹이라도 Mock라이브러리의 기능의 한계점이 있을 때는 Fake 혹은 Mock 객체를 직접 만들어서 사용해도 됨(단 언어의 특성에 따라 한계가 있을 수 있음)
- TDD를 하면 테스트용이성 문제가 거의 없어지기 때문에 TDD를 권장
- 위에서 Mock객체는 XUnit Pattern의 Mock의 정의와는 달리 Mock Library를 통해 제공되며 Stubbing, Verifying 기능을 모두 수행하는 객체를 지칭하는 데 사용되었음.

5. 리팩터링을 통해 배운 교훈

통합 테스트를 적극 활용하자

- **DAO(Repository) 레이어를 완전히 리팩토링**
  - 오래된 프레임워크와 신규 프레임워크의 혼재
  - DAO(Repository)의 상속구조를 보면 팀에서 별도로 구현한 것만 몇 개 이상

- **문제점**
  - 깊은 상속구조로 말미암아 어디서 어떤 동작이 일어나는지 파악하기 어렵고 수정하기 어려움(예. Sharding)
  - 테스트가 고려되지 않은 구조라 DAO(Repository) 테스트 불가

- **기대하는 바**
  - DAO(Repository)의 뼈대를 이루는 핵심적인 디자인과 의도가 잘 드러나게
  - DAO(Repository)가 테스트 가능하게

- **영향 범위**
  - 프로젝트 전체

통합 테스트를 적극 활용하자(Cont~)

- **검증방법**
  - DAO(Repository)를 관통하는 통합 테스트 작성
  - 예시

```
someItems = ... //ItemA, ItemB, ItemC
itemDAO.insert(someItems);
List<Item> items = itemDAO.selectBy(recentItemTen);
assertEquals(itemA, items.get(0));
assertEquals(itemA, items.get(1));
```

- **결과**
  - 내부적으로 거의 모든 기반 클래스를 새로 작성(TDD)
  - 위 DAO(Repository) 테스트가 있었기 때문에 안전하게 마무리

큰 규모 리팩터링도 점진적으로 해나가자

- **스킨 개선 프로젝트**
  - 스킨의 디자인 추가 등 변경이 있을 때마다 점검 및 DB 마이그레이션 필요
- **문제점**
  - 사용자들이 매우 민감하게 반응하는 스킨을 변경하기 어렵고, 이로인해 비즈니스 속도에 따라하기 어려움
- **기대하는 바**
  - 점검 및 DB 마이그레이션 없이 스킨 변경이 가능하게
- **영향 범위**
  - 프로젝트 전체

큰 규모 리팩터링도 점진적으로 해나가자(Cont`)

- **리팩터링 수행하려 할 때 문제점**
  - 스킨은 코드 수준에서도 크고 잦게 자주 변경되는 부분
  - 리팩터링 기간은 약 3개월
  - 이 때 리팩터링 브랜치와, 서비스 브랜치가 별도로 가면 머지 비용을 감당하기 어려울 수도 있음

- **지속적 통합**
  - 스킨 개선을 할 때 수정해야 할 부분이 10개라고 가정할 때, 1주일에 1개씩 적용해서 배포함
  - 미리 배포하기 어려운 부분은 최종 배포로 미루거나, 상황에 따라 해결해나감

- **검증방법**
  - 변경되는 범위에 가능한 Cover & Modify를 기반으로 개선
  - QA 인력에 의한 스킨 회귀 테스트

개선에 앞서 팀을 고려하자

- **함께 일하는 팀에 대한 배려가 없는 개선?**
  - 평이한 구조 VS 정교하고 우아한 구조
  - 쉽지만 표현에 아쉬움이 있는 영어 단어 VS 어렵지만 정확하고 뉘앙스까지 전달하는 영어 단어

- **기술적 흥미에 따른 패턴 적용?**
  - 예) 학습한 직후 어떻게든 GOF 디자인 패턴을 적용

훈련이 필요함을 인식하자

- **하루아침에 코드 리팩터링 사례를 작성할 수 있게 되긴 어려움**
  - 한 동료는 변수 명 때문에 2시간 이상 고민
- **때로 개발지연을 일으키기도 함**
  - 심한 몰입은 금지
- **그러나 지속적으로 추구해야 함**
  - 한 사람이 한번 작성한 코드는 여러 사람이 여러 번 읽음
  - 즉, 투자에 성공 할 확률이 매우 큰 게임

확신이 없을 때는 물어보자

- 내가 직관적이라고 생각한 코드가 동료에게는 아닐 수 있음
- 확신이 안 설 때는 동료에게 의견을 물어보는 것도 좋음
- 한,두 달 뒤에 내 코드를 다시 읽어보고 평가해보는 것도 좋음
- 짝 프로그래밍과 코드리뷰는 코드 리팩터링 사례를 훈련하는 가장 좋은 방법이라고 생각함

지속적 리팩터링을 하자

- **리팩터링만을 위한 일정을 할당해주기 어려울 때가 많다.**

- **레거시를 변경하기 위해 지속적 리팩터링을 하면 좋다. 의미는 아래와 같다.**
  - 리팩터링 : 외부 동작을 바꾸지 않으면서 내부 구조를 개선하는 방법으로, 소프트웨어 시스템을 변경하는 프로세스
  - 지속적 : 리팩터링을 별도의 작업이 아니라 개발의 한 부분으로 생각하는 것. 모든 개발자가 일정을 산출할 때 테스트 시간을 포함하는 것과 유사

- **코드 리팩터링 사례에 좋은 Rename, Extract Method 리팩터링**
  - IDE에서 제공하는 기능을 활용해 수행하면 Side Effect도 거의 없고 시간도 얼마 안 걸림
  - 개인적으로 느끼기에 그 효과는 매우 높음

리팩터링에도 All or Nothing을 적용하자

- **부분적인 개선? 레거시와 코드 리팩터링 사례의 공존?**
  - ‘type’이 프로젝트 전반적으로 사용되고 있는데 의미가 모호하다. 그래서 내가 작업하는 클래스만 ‘newsType’으로 변경?
  - 나아진 건가? 복잡도나 혼란이 없을까?

- **All or Nothing 원칙을 적용**
  - 네이밍을 변경하려면 전체 모두의 네이밍을 변경
  - 신규 클래스를 도입하려면 기존 클래스를 완전히 제거

Cover & Modify의 경제성을 고려하자

- **바쁜 현장에서 모든 것을 Cover & Modify하기는 어렵다.**
- **이론적으로 완벽한 테스트는 불가능하다.**
- **상황에 따라 합리적으로 Cover의 강도를 조정, 다시 말해 덜 중요한 변경은 순발력을 고려하여 Cover 강도를 조정한다.**
- **수정할 때 스스로 '자신감을 가질 정도'가 좋은 기준이 된다.**
- **다만, 중요한 변경에 대해서는 철저하게 적용하는 것이 좋다.**
  - 장애위험성이 높을 때
  - 영향력이 큰 공통 부분일 때(예. Common, Utility)

Cover & Modify의 경제성을 고려하자(Cont`)

```java
public class EconomicalLegacyTest {
    Legacy sut = new Legacy();

    @Test
    public void 서버에_들어온_요청을_정상적으로_처리한다() {
        Request request = new Request();
        Client client = new Client(request);
        Server server = new Server(client);

        sut.processRequestOf(server);

        assertThat(sut.processed, is(true));
    }
}
```

```java
public class Legacy {
    public Boolean processed = Boolean.FALSE;

    public void processRequestOf(Server server) {
        if (server == null) {
            return;
        }

        Client client = server.getClinet();
        if (client == null) {
            return;
        }

        Request current = client.getRequest();
        if (current == null) {
            return;
        }

        process(current);
    }

    private void process(Request currentRequest) {
        processed = Boolean.TRUE;
    }
}
```

예외적 실행경로
(Exceptional Flow)

주 흐름
(Main Flow)

99% 이상은
이 흐름으로 실행 될 것임

유지보수가 개발자에게 주는 선물

- 2008년 제가 했던 통계 프로젝트는 분산 컴퓨팅 기반으로 확장성, 안정성, 속도 등을 매우 신중하게 고려하여 성공적인 오픈
- DeView에서 확장성, 안정성, 속도가 어떻게 '잘' 고려되었는지 발표
- 하지만 유지보수를 시작하자마자 확장성, 안정성 등에 고려하지 못한 부분이 많고, 디자인 또한 문제가 많다라는 것을 인지
- 결국 2012년에 개선 프로젝트가 시도 됨
- 이렇듯 유지보수를 해봐야지만 배울 수 있는 부분이 있음
- 개인적으로 교류하는 개발자 중 뛰어난 분들은 유지보수를 해본 개발자를 높이 평가

구조 리팩터링과 코드 리팩터링

- **구조 리팩터링의 장단점**
  - 장점은 합리적 목표를 가지고 진행했을 때 효과가 극대화 된다는 점
  - 단점은 기간이 오래 걸릴 때가 잦고 위험성이 매우 크다는 점
  - IT 업계의 부정적 사례

- **구조 리팩터링의 예시**
  - 웹 개발 시 Repository 레이어가 있음
  - 이 레이어에 코드 리팩터링을 많이 수행함
    - SQL 가독성 높이기
    - 타입 안전성 높이기
  - 하지만 구조 리팩터링(Spring-Data라는 SQL 자동생성해주는 프레임워크 도입) 후에는 코드 리팩터링이 아예 필요없어짐

- **코드 리팩터링의 장단점**
  - 장점은 일상에서 가볍게 자주 수행할 수 있고, 생각보다 효과가 클 수 있다는 점
  - 단점은 구조에 문제가 있다면 코드 수준에서 아무리 리팩터링 해봐야 한계가 있음

최고의 리팩터링은?

- **최고의 리팩터링은 “제거”하는 것이라 생각**

- **코드 리팩터링 예시**
  - 안 쓰는 코드 제거
  - 중복 코드 제거
  - 불필요하게 복잡한 코드 단순화

- **구조 리팩터링 예시**
  - 안 쓰는 프로젝트 제거
  - 단순 반복해서 작성하던 코드 프레임워크에 위임

Quick And Dirty!

- **아름다운 코드를 썼다. 그런데 제품이 실패했다.**
- **단지 동작하는 지저분한 코드를 썼다. 그런데 제품이 성공했다.**
- **제품의 품질과 기술적 품질의 관계에 대한 생각**
  - 제품의 품질은 서비스가 성공하게 한다.
  - 기술적 품질은 서비스가 성공했을 때 경쟁력을 유지할 수 있게 한다.
- **Quick And Dirty**
  - 두 품질을 너무 완벽하게 추구하지 말고 Fast-Fail 전략
  - 성공한 후에 두 품질을 모두 높이는 것

Quotes(코드 리팩터링 사례 관련)

- Clean code that works.
  - Ron Jeffries

- Programs are read more often than they are written.
  - Implementation Pattern Chapter2

- Like any documentation that you write, you have to think about what will be important to the reader.
  - Working Effectively With 레거시 Code

- Managers may defend the schedule and requirements with passion; but that's their job. It's your job to defend the code with equal passion.
  - Clean Code Chapter1

6. 요약

요약

- 합리적으로 리팩터링
- 리팩터링 할 때는 Cover & Modify를 활용하여 안전하게 수행
- 레거시의 테스트용이성문제는 Dependency Breaking을 활용하여 해결

레거시를 통해 배운 것

- **개인적으로 레거시에 대해 고민하며 아래와 같은 부분을 배웠음**
  - OOP 여러 원칙의 중요성
  - 테스트 관련 지식

- **특별한 것 없이 반복되는 일상 속에서도 도전의식과 재미를 자주 느낄 수 있었음**

참고 서적

- **Working Effectively With 레거시 Code - 강력추천**
  - 현존하는 책 중에 레거시에 대한 통찰력이 가장 뛰어난 책이라고 생각함. 레거시개선에 관심이 있는 분은 일독을 강력하게 권하고 싶음. 이 자료의 많은 부분에서 위 책의 내용을 참고 했음.
- **구현패턴(Implementation Patterns) - 강력추천**
  - 다른 개발자의 중요성을 인지하고 배려하는 프로그래밍을 강조. 인지를 통해 프로그래밍의 여러 요소를 패턴화. 코드 리팩터링 사례에 관심이 있는 분은 일독을 강력하게 권하고 싶으며, 패턴 라이팅에 관심 있으신 분 또한 참고하면 좋을 것 같음.
- **리팩터링**
  - 리팩터링의 중요성을 소개하며, 각 리팩터링 별로 점진적으로 안전하게 리팩터링 하는 과정을 보여줌.
- **클린코드**
  - 단순히 동작하는 코드가 아닌 동작하는 깔끔한 코드를 작성하기 위한 넓은 범위의 조언을 해줌.

X Appendix

코드 리팩터링 사례 Reference

- Magic Number를 사용하지 말자
  - 리팩터링 8장 Replace Magic Number with Symbolic Constant
  - 클린코드 17장 G25
- 의도를 나타내는 이름을 사용하자
  - 클린코드 17장 G20
  - 구현패턴 3장 Communication, 8장 Intention Revealing Names
- 읽기 쉽게 요약하자
  - 구현패턴 3장 Symmetry, 8장 Composed Method
  - 리팩터링 6장 Extract Method
- 최대한 빨리 매듭짓자
  - 구현패턴 3장 Local Consequences, 7장 Guard Clause
  - 리팩터링 7장 Replace Nested Conditional with Guard Clauses
- 가급적 대칭을 지키자
  - 구현패턴 3장 Symmetry
- 복잡한 조건식을 피하자
  - 리팩터링 9장 Replace Conditional with Polymorphism

X Appendix(Cont`)

코드 리팩터링 사례 Reference

- 사용하면 안 되는 클래스/메서드가 있을 때
  - http://download.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html
- 코드로 표현하기 어려운 의도가 있을 때 , 뻔한 중복주석은 피하자
  - 클린코드 4장

X Appendix

이미 작성 된 코드 테스트 하기 Reference

- 테스트용이성을 결정짓는 두 가지 요소, 해결책은 Dependency Breaking
  - Working Effective With 레거시 Code 3장
- 생성이 안될 때
  - Working Effective With 레거시 Code 25장 - Parameterize Constructor
  - Growing Object-Oriented Software, Guided By Tests 6장 - Context Independence
- 실행제어가 안될 때
  - Working Effective With 레거시 Code 25장 - Introduce Static Setter
- 결과 검증이 안될 때
  - Working Effective With 레거시 Code 3장