chapter 3: Functions

Small

Listing 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

Listing 3-1 (continued)
HtmlUtil.java (FitNesse 20070619)

```
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Listing 3-2
HtmlUtil.java (refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

왼쪽 코드와 동일한 역할을 한다.

함수의 첫번째 규칙: 작아야 한다

함수의 두번째 규칙: 더 작아야 한다

- 수백줄의 함수는 불필요하다. 가능한 짧게!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Do One Thing

> 함수는 한가지 동작을 해야 하고, 잘 해야 하고, 그것만 해야 한다.

- 많은것을 하게 될수록 복잡해질 뿐이다
- 함수를 줄인다는 것
  = 함수 내에서 특정 부분을 다른 함수로 빼낼 수 있다는 것

Structured Programming

- Dijkstra's rules of structured programming
  1. Every function have One entry and One exit
  2. One `return` statement in a function
  3. no `break`, `continue` in a loop
  4. NEVER EVER any `goto`

Abstraction

One Level of Abstraction per Function

- 함수 내에서의 추상화 단계는 동일해야 한다.
  - ex. `getHtml()`, `parser.render(path)`, `.append("\n")`
    이 셋이 같이 있어서는 안된다.

Stepdown Rule

A를 하려면 B1, B2가 필요하다

B1을 하려면 C1, C2를 해야한다

B2를 하려면 D1, D2, D3을 해야한다

C1을 하려면 ...

이와 같이 할 일을 쪼개면서 같은 레벨에 있는 기능끼리를 함수로 묶자

Switch Case

스위치문은 사실 하나의 함수가 여러 작업을 하게 되는 결정적인 요인이다.

Listing 3-4
Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

문제점

- 함수가 바뀔 수 있는 원인이 여러가지다
  → Single Responsibility principle 위배
- 새로운 type이 생길때마다 커진다 → Open Closed Princile 위배
- calculatePay가 아니라 deliverPay, isPayday같이 비슷한 구조의 다른 함수가 있으면 복붙을 해야 한다

해결방법

Listing 3-5
Employee and Factory

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

해결된 점

- 새로운 인스턴스를 만드는 일 한가지만 한다
- 새로운 타입이 생기면 저 정의에 추가만 해주면 된다
- 비슷한 함수가 있어도 switch문을 복붙할 필요는 없고, 각 하위 클래스에서 구현만 다르게 하면 된다 (다형성)

switch 사용 방법

1. 단 하나만 존재
2. polymorphic object 생성에만 사용
3. 상속으로 인해 **가려져야 함!**

Function Arguments

- 이상적인 함수의 인자 수는 **0개**이다

  `includeSetupPage()` VS `includeSetupPageInto(newPageContent)`

  - 테스트를 작성할 때 더욱 문제가 심각해진다.
    모든 인자의 조합을 테스트해야하므로...
- input, output은 각각 인자, 반환값에 들어가야 한다
  - output이 인자로 들어가면 안된다

    `void transform(StringBuffer out)`
  - 위 코드보다 아래 형태가 훨씬 명확하다

    `StringBuffer transform(StringBuffer in)`
- Flag식 인자
  - boolean을 인자로 받는다는 것
    → if else로,
    두가지 이상의 일을 한다는 것! 최악
  - 두개로 나눠서 따로 호출을 하던가 해라.
- 인자 두개 (dyadic)

- 좌표계처럼 `point (0,0)` 같은건 아무 상관 없다
- `assertEqual(expected, actual)` 같이 선언해놓으면
  나중에 값 넣을때 뭐가 먼저인지 헷갈리지 않을까? 피하도록 하자.

Argument Object

인자 여러개를 하나로 묶어서 개수를 줄이자!

```
makeCircle(double x, double y, double r)
->
makeCircle(Point center, double r)
```

Argument List

```
String format(String format, Object... args)
```

- `Object...` 자체가 하나의 인자이다
- 따라서 두개의 인자를 받지만 실제 사용할 때는 dynamic하게 사용

Side Effect

함수는 한 가지 동작만을 해야 한다
그러나 다른 의도치 않은 일을 한다 → 부작용 side effect

```
public boolean checkPassword(String userName, String password) {
  User user = UserGateway.findByName(userName);
  if (user != User.NULL) {
    String codedPhrase = user.getPhraseEncodedByPassword();
    String phrase = cryptographer.decrypt(codedPhrase, password);
    if ("Valid Password".equals(phrase)) {
      Session.initialize();
      return true;
    }
  }
  return false;
}
```

- 함수 이름은 checkPassword이고, bool을 반환한다
  - 따라서 비밀번호가 맞는지 확인하는 것만 해야 한다.

- 그러나! `Session.initialize();` 부분이 side effect이다.
  - 만약 세션 초기화를 하면 안되는 상황에 저 함수가 호출된다면?
    세션이 날아갈 수도 있다. ㅠㅠ

Command Query Separation

함수는 1) 객체의 상태를 바꾸거나 2) 값을 반환 **둘중 하나만** 해야 한다.

예를 들어

```
public boolean set(String attribute, String value);
```

- 위 함수는 성공하면 true 실패하면 false를 반환한다.
  따라서 아래와 같은 상황이 펼쳐진다.

```
if(set("user", "value")){ ... }
```

- 조건문 안에 set이 들어가..?
  user값이 value가 된건지 이미 value인건지 헷갈린다!
- 따라서 boolean으로 반환할거면 체크만 하지 어떠한 동작을 하지 마라
- error code를 반환하는 경우 이 rule을 어기게 된다.
  - 또한 반환값에 따라서 즉시 어떠한 조치를 취해야 한다는 단점이 있다.
  - 따라서 exception을 쓰고, try catch를 사용하자
    - try에서 하는 일도 함수로 빼낼 수 있다면 빼자
    - 왜냐하면 에러 핸들링도 “one thing”에 포함되기 때문이다.

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```