# TDD를 위한 개선된 테스트 프레임워크 설계

손병길° 류호연 박재흥
경상대학교 컴퓨터과학과 소프트웨어공학연구실
wd@windystudio.net°, nollai@magicn.com, pjh@nongae.gsnu.ac.kr

# A Design of Improved Test-Framework for TDD

Byeong-kil Sohn° Ho-yeon Ryu Jae-Heung Park
Dept. of Computer Science, Gyeongsang National Univ., Korea

요 약

TDD는 테스트를 기반으로 하는 점진적인 소프트웨어 개발 방법으로, 리팩토링 과정을 통해 정제된 디자인을 얻을 수 있다. TDD에서는 개발에 사용되는 프로그램 언어를 지원하는 테스트 프레임워크를 사용하며 리팩토링 브라우저, 테스트 테스터, 테스트 커버리지 등의 도구들이 개발에 사용된다. 본 논문에서 제안하는 pytodo는 TDD의 to-do list를 트리구조인 to-do tree로 작성하고, 이를 테스트 코드 관리, 테스트 코드에 대한 테스트에 활용하는 것을 목적으로 한다.

## 1. 서 론

TDD(test driven development: 테스트 주도 개발)는 점진적인 개발 접근 방법으로써 Kent Beck에 의해 소개되었다[1]. TDD에서는 새로운 코드를 추가하기 전에 먼저 테스트를 작성하고, 이러한 테스트를 바탕으로 프로그램을 작성하게 된다. 또한 과감한 리팩토링(refactoring)[2] 과정을 통해 지속적으로 디자인을 정제하게 된다.

TDD에서는 개발자들이 사용 가능한 테스트 프레임워크(test-framework)를 가지고 있다고 전제한다. jUnit, cppUnit, pyUnit과 같은 해당 프로그램 언어를 지원하는 테스트 프레임워크 없이 TDD를 진행하는 것은 불가능하다[3]. TDD가 확산됨에 따라 테스트 프레임워크 외에도 TDD의 특징을 살린 도구들이 개발, 발표 되고 있다. 대표적인 도구의 분류로는 리팩토링 브라우저, 테스트 테스터, 테스트 커버리지(test coverage) 도구 등이 있다.

본 논문에서 제안하는 도구는 TDD 과정에서 작성하는 to-do list를 트리구조로 표현하여 to-do tree로 작성하고, 테스트와 함께 보관할 수 있도록 해주며, to-do tree 의 항목과 관련된 테스트를 검색하고 테스트를 위해 작성된 작업 코드(working code)의 테스트 커버리지를 CFG(control flow graph; 제어 흐름 그래프)로 표시하는 것을 목적으로 한다.

트리로 표현된 to-do list 와 테스트 코드(test code) 그리고 CFG 의 연관을 보여주는 본 도구는 기본적인 TDD의 개발과정을 지원함과 동시에 이후의 문서화 작업이나 코드의 평가 등에 있어 유용하게 활용할 수 있을 것이다.

## 2. 관련연구

### 2.1 TDD

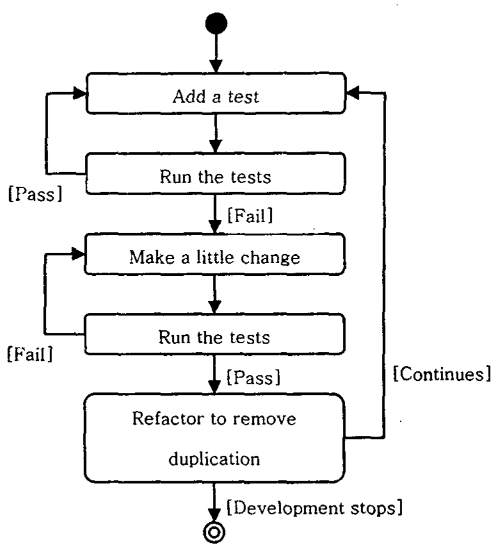
TDD는 test-first programming 혹은 test-first development 로도 알려져 있으며, 먼저 실패하는 테스트를 작성하고 테스트를 통과하는 작업 코드를 후에 작성하는 점진적인 개발 방법이다. 따라서 작성한 테스트는 검증(validation) 보다는 명세(specification)의 역할을 하게 된다[3]. 즉 작업 코드에 대한 디자인을 테스트를 통해 생각할 수 있게 되는 것이다. 또한 TDD는 프로그래밍 기술이다. TDD는 점진적인 디자인의 성장과 함께 점진적인 코드의 완성을 동시에 이끌어 낸다. 부수적으로 TDD는 이러한 개발 과정을 통해 최적화된 단위 테스트(unit-test)를 얻게 된다. 본 논문에서는 TDD 개발 과정에서 작성하는 테스트(test) 집합을 테스트 코드(test code), 본래의 요구사항을 위해 작성하는 코드를 작업 코드(working code)로 명칭 한다. 물론 TDD에서의 작업 코드는 실패하는 테스트가 있을 때에만 작성된다.

### 2.2 TDD의 과정

TDD는 (그림 1)과 같이 다섯 단계의 과정으로 이루어 진다. TDD에서는 개발 과정을 시작하기에 앞서 완성된 코드가 수행할 항목(item)들을 리스트로 작성하게 되는데, 이를 일반적으로 to-do list라 부른다. TDD의 과정은 모든 to-do list의 항목이 만족될 때까지 여러 번 반복된다.

### 2.3 TDD와 to-do list

to-do list는 TDD 개발 과정에서 해야 할 일을 알려주며, 집중을 유지할 수 있게 해준다. 또한 개발이 언제 완료 되는지도 알려 준다[1]. 한번의 TDD 개발 과정은 먼저 to-do list에서 하나의 항목을 선택하는 것으로 시작된다. 그리고 한번의 TDD과정이 끝났을 때 새로운 필요에 의하여, 또는 선택한 항목이 한번의 TDD과정으로 해결하기 힘든 경우 to-do list는 추가로 작성 된다. 이때 원래의 항목이 세부 항목으로 분리되거나 새로운 항목이 to-do list에 추가 된다. to-do list의 이러한 특징으로 인해 본 논문에서는 to-do list를 트리구조로 표현함으로써 보다 많은 유용성을 얻을 수 있을 것으로 가정하게 되었다.




(그림 1) TDD의 과정

## 3. pytodo

### 3.1 pytodo 개요

본 논문에서 제안하는 *pytodo*는 TDD의 to-do list를 트리구조로 표현하는 to-do tree를 작성, 활용하는 것을 기본으로 하여 테스트 코드의 검색과 각 테스트에 대한 작업 코드의 CFG 표시를 목적으로 한다. pytodo는 프로그램 언어로 python을 선택하고 있으며 테스트 코드와 작업 코드 역시 python을 지원한다. 테스트 코드를 위한 기본 테스트 프레임워크로는 pyUnit을 이용한다.

### 3.2 TDD 개발 과정에서 pytodo의 역할

1. to-do tree의 편집
2. to-do tree의 항목으로 테스트 코드 검색
3. 선택한 테스트에 의한 작업 코드의 CFG 표시

### 3.3 pytodo에서의 *to-do list*

TDD에서 기존의 to-do list가 하는 일은 다음 세가지로 압축할 수 있다.

1. 해야 할 일의 기록
2. 한번의 TDD과정에서 선택한 항목에 대한 집중
3. 작업 완료에 대한 지표

pytodo에서의 to-do list는 위의 세 가지에 더하여 다음의 역할을 수행한다.

1. 트리구조의 표현(to-do tree)
2. 테스트 코드를 검색 할 수 있는 index
3. 테스트 코드와 함께 저장, 문서로 활용

### 3.4 문서로서의 to-do tree

TDD에서 테스트 코드는 작업 코드와 함께 반드시 보관되지만 to-do list의 보관은 필수적이지 않다. 이미 테스트 코드 자체가 기본적인 기술 문서로서 충분하다고 생각하는 것이다. 물론 테스트 코드를 통해서 to-do list를 역으로 구성하는 것은 그리 어려운 일은 아니다. 하지만 상대적으로 높은 가독성에도 불구 하고 테스트 코드는 거의 작업 코드와 1:1 비율에 이를 만큼 분량면에서 만만치 않다. 뿐만 아니라 테스트 코드를 해석하는 일은 to-do list의 항목을 읽는 것과는 분명 다르다.

TDD로 개발이 진행되는 중이나 또는 개발이 완료된 후에 계속적인 개발이나 유지보수 혹은 단순한 보고를 위해서라도 반드시 문서화는 요구된다. 테스트 코드가 기본적인 기술 문서로서 좋은 가치를 지니지만 그것만으로는 충분하지 않은 것이다. 이때 to-do tree는 테스트 코드에 대한 보다 추상적인 문서로써 테스트 코드를 훌륭히 설명할 수 있다. pytodo는 작성한 to-do tree를 테스트 코드와 함께 보관 할 수 있게 해준다.

### 3.5 pytodo의 활용

pytodo는 TDD 과정에서 다음과 같은 활용이 예상된다. 테스트 코드의 변경 시 기존의 경우 테스트에 의해 영향 받는 부분을 찾고 변경 방법을 검토 하는데, pytodo는 이러한 과정에서 CFG를 활용할 수 있다. 또한 기존의 TDD에서는 작업 코드의 신뢰성을 테스트 코드에만 의존하였으나 pytodo를 활용할 경우 테스트에 의한 CFG를 추가로 활용하여 작업 코드를 평가 할 수 있다. 또한 리팩토링을 진행 할 때에도 코드의 중복뿐만 아니라 CFG상에서의 흐름을 검토 하여 더 좋은 디자인을 이끌어 낼 수 있을 것이다. 또한 테스트 코드의 평가나 문서화에서도 좋은 자료로 활용 할 수 있을 것으로 예상된다.

### 3.6 pytodo 시스템

*pytodo*는 프로그램 언어를 위한 기본적인 편집기에 대하여 보조적인 역할을 수행한다. pytodo에서 작성하는 to-do tree는 테스트 코드의 주석으로 포함되며 pytodo를 개발 과정에서 제외 하더라도 원래의 개발 과정 진행에 아무런 영향을 주지 않는다. pytodo에 의해 테스트 코드가 불려졌을 때 to-do tree와 테스트 코드는 분리되어 처리되며 새로운 항목을 작성하거나 to-do tree를 통해 테스트를 검색하는 일이 가능하다. 또한 테스트를 선택하여 작업 코드의 CFG를 표시할 수 있다.

## 4. 예

다음은 TDD에 관한 Kent Beck의 유명한 저서인 'test-driven development by example'에 실린 'the money example'에 대한 실행 모듈을 TDD로 개발하며 작성한 to-do list(그림 4)와 to-do tree(그림 5), 그리고 하나의 테스트를 선택하여 그것에 대한 CFG를 구한 것을 보여준다.

- 5$ + 5$ = 10USD
- "5$ + 5$"의 문자열 표현 ->
  [Money().dollar(5), "+" , Money().dollar(5)]
- 입력된 "5$"에 대하여 dollar(5) 객체 생성
- Unknown Expression (예, "12345")
- 5$ + 10CHF = 10USD
- Unknown Expression (예, "aa$ + bCHF")

(그림 4) to-do list

(그림 4)의 to-do list는 (그림 5)의 to-do tree와 비교하기 위한 것이며 실제 작성된 트리는 (그림 5)와 같다.

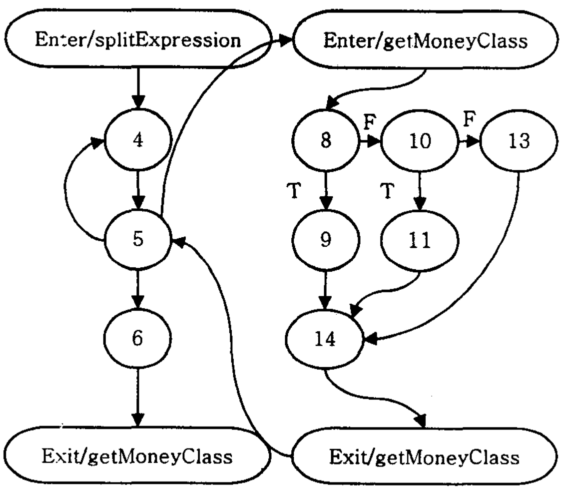
1) 5$ + 5$ = 10USD
1.1) "5$ + 5$"의 문자열 표현 ->
[Money().dollar(5), "+", Money().dollar(5)]
1.1.1) 입력된 "5$"에 대하여 dollar(5) 객체 생성
1.2) Unknown Expression (예, "12345")
1.2.1) Unknown Expression (예, "aa$ + bCHF")
2) 5$ + 10CHF = 10USD

(그림 5) to-do tree

(그림 5)의 to-do tree에서 항목 1.1)을 선택 했을 때의 테스트는 (그림 6)의 테스트 코드에서 굵은 글씨체로 표시된 부분이다.

```
import unittest
class TestClass(unittest.TestCase):
(중간 생략)
  def testSplitExpression(self):
    result = Calculate().splitExpression("5$ + 5$")
    self.assertEquals([Money().dollar(5),
                    "+",       Money().dollar(5)],
result)
```

(그림 6) 테스트 코드




(그림 8) 그림 7의 작업 코드에 대한 CFG

선택한 테스트와 관련된 작업 코드의 부분(그림 7)과 그에 대한 CFG(그림 8).

```
1 class Calculate:
2 def splitExpression(self, Expression):
3    result = []
4    for item in Expression.split():
5      result.append(self.getMoneyClass(item))
6    return result

7 def getMoneyClass(self, Token):
8    if Token[-1] == "$" and Token[:-1].isdigit():
9      result = Money().dollar(int(Token[:-1]))
10   elif len(Token) > 3 and Token[-3:] == "CHF"
                and Token[-3:].isdigit():
11     result = Money().franc(int(Token[:-3]))
12   else:
13     result = Token
14   return result
```

(그림 7) 선택한 테스트와 관련된 작업 코드의 부분

## 5. 결론

TDD는 디자인 단계에서 해야 하는 결정을 점진적인 프로그래밍 과정을 통해 결정해 나갈 수 있어 최적화된 디자인을 얻을 수 있다. 또한 자동화된 테스트를 포함하고 있으며 일반적인 프로그래밍 방식에 비해 적은 오류를 가진 코드를 얻을 수 있다[5]. 위와 같은 장점으로 TDD는 향후 중요한 개발 방법으로 자리할 것으로 예상된다. 본 논문에서는 TDD로 개발을 진행하는데 도움이 될 수 있는 pytodo라는 도구를 제안하였다. 향후 연구과제로는 pytodo 설계를 구현하는 것이며, pytodo를 직접 프로그램 프로젝트에 활용하여 더 많은 자료를 수집하고 유용성을 검증하는 것이다.

## [참고문헌]

[1] Kent Beck, "Test-Driven Development By Example", 2003
[2] Martin Fowler, "Refactoring", 1999
[3] Scott W. Ambler, "Agile Database Techniques", 2003
[4] 강원임, 정보 흐름 그래프를 이용한 정적 프로그램 슬라이싱에 관한 연구, 2000
[5] Randy A. Ynchausti, "Integrating Unit Testing into a Software Development Team's Process", 2001