테스트 주도 개발로 배우는 객체 지향 설계와 실천

Growing object-oriented software, guided by tests

서문

서문

매번 반복되는 의문과 혼란

- 어떻게 하면 TDD를 잘 적용할 수 있을까?
- 어디에서 시작해야 할까?
- 왜 단위 테스트와 전 구간(E2E) 테스트를 모두 작성해야 할까?
- 테스트가 개발을 '주도한다'는 것은 무슨 뜻일까?
- 복잡한 기능은 어떻게 테스트할 수 있을까?

저자는 이 책에서 프로젝트마다 반복적으로 생기는 이런 의문에 초점을 맞췄다고 합니다.

서문

성장하는 객체 지향 소프트웨어

- 책의 원제는 **Growing** object-oriented software, guided by tests
  - 점진적으로 개발해 나간다
  - 뭔가를 항상 동작하게 만들어 둔다
  - 훌륭한 software에서 볼 수 있는 생명체 같은 특성
  - "객체란 서로 메시지를 주고받는 생물학적 세포와 비슷해야 한다" - 엘런 케이

책의 원래 제목은 '테스트의 안내를 받아 성장하는 object-oriented software'입니다.
Growing이란 단어를 쓴 이유는,
- 생물학적 세포와 비슷하게
- 항상 동작하게 만들어 두면서
- 점진적으로 개발해 나간다

는 의미

서문

성장하는 객체 지향 소프트웨어

- 책의 원제는 Growing object-oriented software, **guided by tests**
  - 의도가 더 분명해 진다
    - 작업 전 코드가 무엇을 해야 하는지 명확히 기술해야 한다
  - 회귀(regression) 테스트라는 안전망이 돼 준다
  - 시스템 품질을 높이려면 Test로부터 피드백을 받는 데 집중해야 한다

회귀테스트(Regression Test) :
- 수정(Modification)이 기대하지 않은 결과를 발생시키지 않는다는 것을 증명하기 위한 시스템이나 컴포넌트에 대한 선택적 재테스트

'테스트의 안내를 받는다'의 의미는, TDD로 진행하면서
- 의도가 더 분명해지고
- 작업이 끝나면, 회귀 테스트라는 안전망이 되어 주고
- 시스템 품질을 높일 수 있는 피드백을 받을 수 있기 때문

입니다

서문

대상 독자

지식이 풍부한 독자

1장 테스트 주도 개발의 핵심은 무엇인가?

테스트 주도 개발의 핵심

소프트웨어 개발은 학습의 과정

- 흥미로운 프로젝트(= 가장 큰 이익을 줄 만한 프로젝트)에는 **예상치 못한 요소**가 상당히 많다
- 갖가지 중요한 구성 요소가 조합된 시스템은 **너무나 복잡**해서 개인이 해당 시스템의 모든 가능성을 **이해하기는 어렵다**
- 프로젝트에 관련된 **모두는** 프로젝트가 진행되면서 **배우는 것**이 있어야 한다
  - 프로젝트에서 **달성할 바**가 무엇인지
  - **잘못 이해**하고 있는 바를 식별
  - **해결**하고자 **협업**해야 한다

저자는 소프트웨어 개발이 학습의 과정이라고 생각하고 있습니다.

- 프로젝트에는 매번 예상치 못한 요소가 발견되고
- 너무 복잡해서 개인이 모든 가능성을 이해하기 어렵기 때문에
- 프로젝트의 관련된 모두는 프로젝트가 진행되면서 배우는 것이 있어야 한다고 강조합니다.

테스트 주도 개발의 핵심

피드백은 가장 기본적인 도구다

팀에는 반복적인 활동 주기가 필요하다

- 각 주기마다 **새로운 기능을 추가**하고 완료한 작업에 관한 **피드백**을 받는다
- 각 주기마다 특정 환경에 **배포**
- 짧게는 초 단위에서 길게는 월 단위에 이르는 **중첩된 고리형** 시스템
- **점진**적이고 **반복**적인 개발

이런 복잡한 시스템으로부터 학습하고 개선하기 위해선 피드백을 활용해야 한다고 주장합니다.
그것도 다양한 반복적인 활동 주기가 필요하다고 해요.
각 주기마다 피드백을 받고, 피드백으로부터 학습하고, 다시 시스템에 적용하는,
그런 점진적이고 반복적인 개발 방식이 필요하다고 합니다.

테스트 주도 개발의 핵심

피드백은 가장 기본적인 도구다

중첩된 고리형 시스템(system of nested loops)

- 짝 프로그래밍, 단위 테스트, 인수 테스트, 일별 회의, 반복 주기, 출시
- 중첩된 **피드백** 고리는 서로를 강화한다
  - 안쪽에서 잡지 못한 오류는 바깥쪽 고리에서 포착
- **안쪽 고리는 기술적 세부 사항**에 좀 더 집중한다
  - 단위 코드의 역할과 시스템 나머지 부분과의 통합 여부
- **바깥쪽 고리는 조직과 팀**에 좀 더 집중한다
  - 사용자의 요구를 충족 하는지, 팀이 효과적으로 운용되고 있는지

책에는 이런 다양하고 반복적인 활동 주기를 '중접된 고리형 시스템'이라고 부르고 있습니다. 여기에는,
즉각적으로 피드백을 받을 수 있는 짝 프로그래밍부터 (단위테스트와 인수테스트 같은) 자동화 테스트,
(스크럼처럼) 일별 회의, (스프린트 같은) 반복 주기, 그리고 언제든 배포할 수 있는 자동화된 빌드 시스템까지... 다양합니다.
피드백을 받을 수 있는 건 모두 동원 하려는 것 같아요. 왜냐면, 앞에서 말한 것처럼,
시스템은 너무 복잡하고, 미래를 예측하기 점점 더 어려워지기 때문이죠.
책에는 없지만, XP나 스크럼 같은 애자일 개발 방법론에 영향을 받은 것 같아요.

"어떤 피드백이든 일찍 받을 수록 좋다"

테스트 주도 개발의 핵심

개발자의 코딩 비용은 매우 비싸다

- 프로그래머는 코드 읽기와 디버깅에 대부분의 시간을 보낸다
- 코딩 보다는 디버깅을 쉽게 만드는 것이 더 개선 효율이 좋다
- 피드백은 잘못된 것을 빨리 발견하고 불필요한 작업을 하지 않을 기회

프로그래머는 코드 읽기와 디버깅에 대부분의 시간을 보낸다는 연구 결과가 있습니다.
그 말은 코드를 쓰면 쓸 수록, 읽거나 디버깅할 것이 늘어난다는 말이죠.
그래서 저는 중첩된 피드백 고리가

- 잘못된 것을 빨리 발견하고
- 불필요한 작업을 하지 않을 기회를 준다고 생각합니다.

테스트 주도 개발의 핵심

피드백은 가장 기본적인 도구다

점진적이고 반복적인 개발 방식

- 모든 계층과 구성 요소를 구축한 다음 통합 (X)
- 시스템은 **언제나 통합**되어 있고 **배포할 준비**가 돼 있는 상태
- **계속**해서 충분한 상태에 이를 때까지 **피드백에 응답**해 기능 구현을 다듬기

앞에서 언급했던 '점진적이고 반복적인 개발 방식'에서는 구성 요소를 다 만든 뒤에 통합하는 방식이 아닙니다.
항상 동작하는 시스템을 만들어 둔다고 했었죠?
언제나 통합되어 있고, 배포할 준비가 돼 있어야 한다고 합니다.

지속적 통합(Continuous Integration)
지속적 서비스 제공 (Continuous Delivery)

책에서는 직접적으로 언급은 하지 않았지만, CI나 CD의 개념을 설명하고 있어요.

테스트 주도 개발의 핵심

CI vs CD

지속적인 통합(Continuous Integration)

- 자동화된 통합 시스템
- 동작하는 코드인지 지속적으로 피드백을 받는다

참고

- 지속적 통합이란 무엇입니까?
- CI (Continuous Integration) 이란?

CI라 불리는 지속적인 통합은,

- 자동화된 통합 시스템을 의미하고
- CI의 가치는 동작하는 코드인지 지속적으로 피드백을 받는 것입니다

테스트 주도 개발의 핵심

CI vs CD

지속적 서비스 제공 (Continuous Delivery)

- 변경 사항이 자동으로(혹은 운영팀에 의해) 프로덕션 환경으로 배포될 수 있는 것
- 개발팀과 비즈니스팀과의 가시성 해결
- 프로덕트에 관해 고객/비즈니스팀에게 피드백 받는다

참고

- 지속적 전달이란 무엇입니까?

CD라 불리는 지속적인 서비스 제공은,

- 변경 사항이 자동으로 특정 환경으로 배포될 수 있는 것을 의미합니다
- 비즈니스 팀은 항상 동작하는 시스템을 확인할 수 있게 됩니다
- CD의 가치는 프로덕트에 관해 고객이나 비즈니스팀에게 피드백을 받는 것입니다

이 CI, CD는 각각 뒤에서 언급할 내부 품질과 외부 품질에 영향을 줍니다

테스트 주도 개발의 핵심

시스템 규모를 믿을 수 있는 방식으로 키우기

늘 일어나는 **예상치 못한 변화에 대처**하기 위해선

- 회귀 오류를 잡아줄 꾸준한 테스트가 필요하다
  - 기존 기능을 망가뜨리지 않고 새 기능을 추가
  - (시스템 규모와 상관없이) 수동 테스트를 자주하는 것은 비실용적
  - 자동화! **자동화!**

또한 저자는 시스템 규모를 신뢰할 만한 방식으로 키우기 위해서
회귀 오류를 잡아줄 꾸준한 테스트가 필요하다고 주장합니다.
새 기능을 추가할 때, 기존 기능을 망가뜨리지 않았다는 걸 확인해야 하는데,
수동 테스트를 자주하는 것은 효율적이지 않으므로, 자동화를 매우 강조하고 있습니다.
게다가 직접 테스트하길 좋아하는 개발자는 없다고도 했었죠.

테스트 주도 개발의 핵심

시스템 규모를 믿을 수 있는 방식으로 키우기

늘 일어나는 **예상치 못한 변화에 대처**하기 위해선

- 코드를 가능한 한 단순하게 유지해야 한다
- 꾸준히 리팩토링해야 한다
  - 설계를 개선하고 단순화하고
  - 중복을 제거하며
  - 코드가 명확하게 자신의 역할을 표현하도록

그리고 코드를 가능한 한 단순하게 유지해야 한다고 합니다
- 시스템은 한 사람이 다 이해하기엔 너무나 복잡하니까요

이를 위해 꾸준히 리팩토링을 해야한다고 주장합니다

테스트 주도 개발

테스트 주도 개발

TDD를 통해 얻을 수 있는 것

시스템 구현 - 시스템이 동작하는가?

- 완전한 회귀 스위트가 늘어난다
- 콘텍스트를 선명하게 인지하는 동안 오류를 탐지한다

TDD를 통해 얻을 수 있는 것은,
시스템 구현 측면에선 '시스템이 동작하는가'를 알 수 있습니다.

테스트 주도 개발

TDD를 통해 얻을 수 있는 것

설계의 품질 - 시스템이 잘 구조화돼 있는가?

- 다음 작업에 대한 인수 조건이 명확해진다
- 느슨하게 결합된 구성 요소를 작성할 수 있게 된다
- 과도한 최적화를 하거나 불필요한 기능을 더하지 않게 된다

그리고 설계의 품질 면에서는,
'시스템이 잘 구조화 돼 있는가'를 알 수 있습니다.

- 다음에 할 작업에 대해 명확히 인지하는 것
- 느슨하게 결합된 구성 요소를 작성할 수 있고요
- 과도한 최적화나 불필요한 기능을 더하지 않게 됩니다.

"버그 말곤 잃을 게 없어요"

TDD를 통해 얻을 수 있는 것

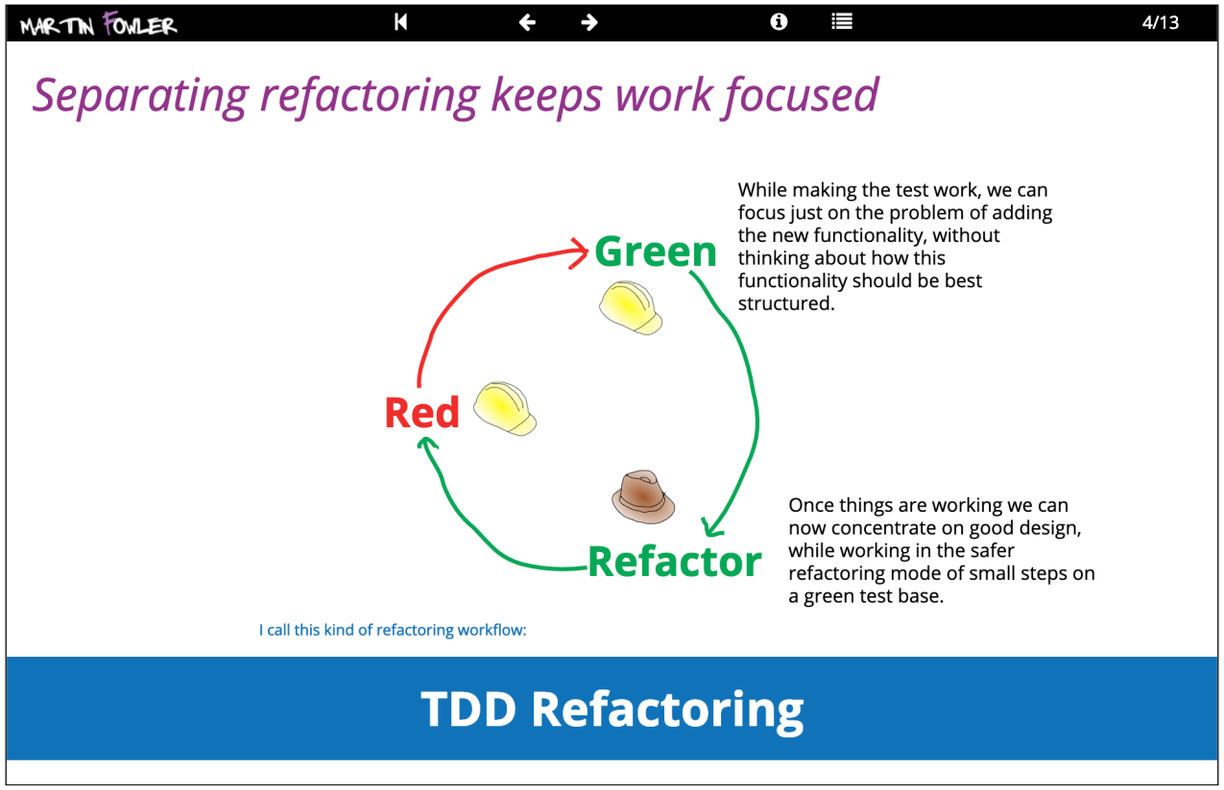
켄트 벡 - RIP TDD

- 번역본 링크 : https://justhackem.wordpress.com/2018/06/24/what-tdd-solves/

TDD가 해결해주는 것

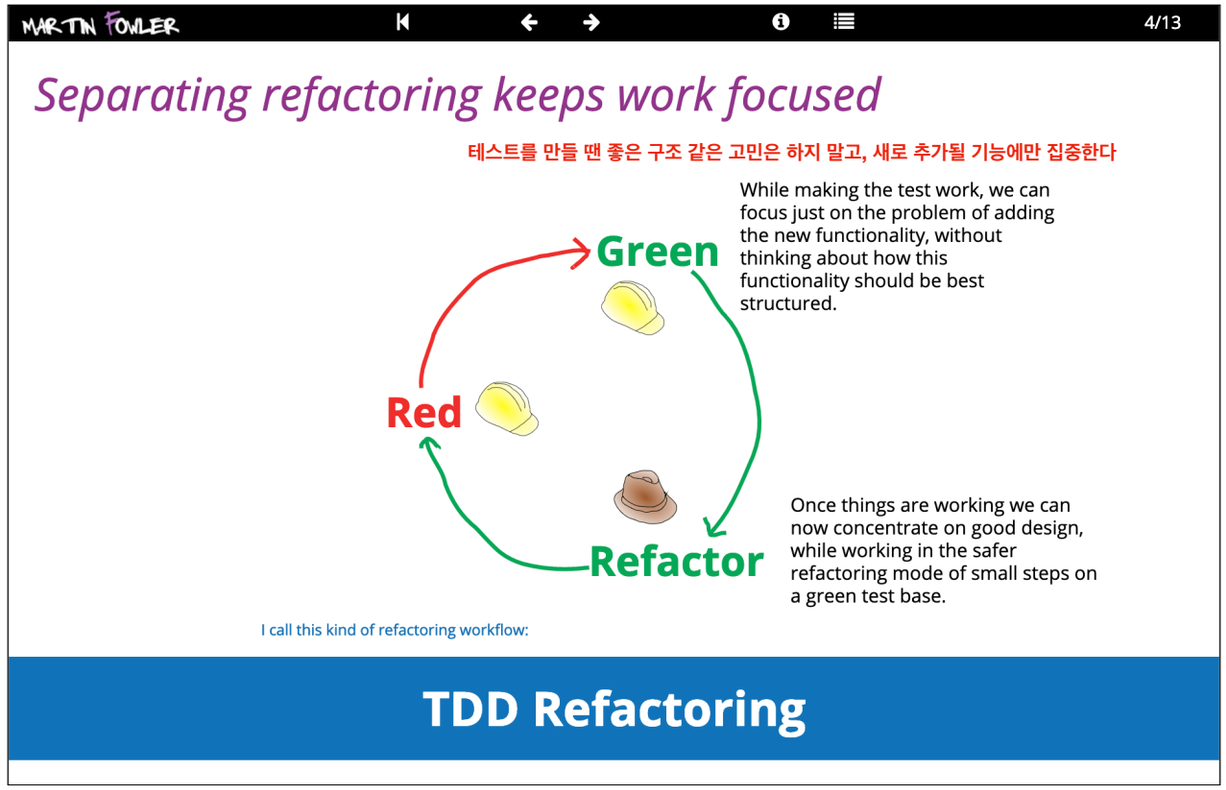
- 오버 엔지니어링
- API 피드백
- 논리 오류
- 문서화
- 막막함
- 구현 사고와 인터페이스의 분리
- 합의
- 걱정

켄트 벡은 RIP TDD란 글에서 TDD가 해결해주는 것을 아래 8가지로 이야기 했습니다. 나중에 한번 읽어보시기 바랍니다.



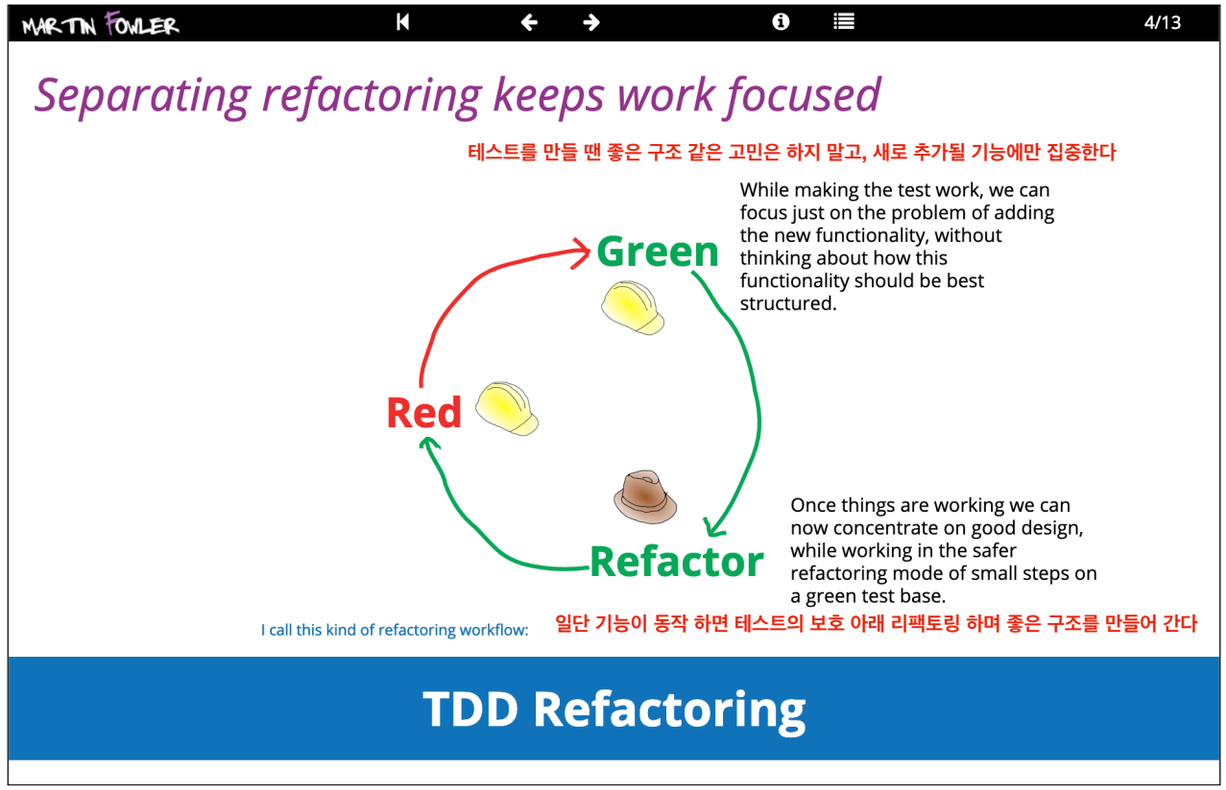
https://martinfowler.com/articles/workflowsOfRefactoring/

얘는 마틴 파울러가 설명하는 리팩토링 워크플로우입니다.



https://martinfowler.com/articles/workflowsOfRefactoring/

Red 단계에서 테스트를 만들고
Green 단계로 작업할 때는 새로 추가될 기능에만 집중합니다.
좋은 구조 같은 건 이 단계에서 고민하지 않습니다.



https://martinfowler.com/articles/workflowsOfRefactoring/

일단 기능이 동작하면,
아까 만든 테스트 코드가 회귀 스위트의 역할을 하고,
테스트의 보호 아래서 작은 단계로 안전하게 리팩토링을 한다고 합니다.

테스트 주도 개발

리팩토링

- 기존 코드의 **작동 방식을 바꾸지 않고** 표현을 개선한다
- 각 리팩토링은 **이해하기 쉽고 안전**할 정도로 규모가 작아야 한다
- 각 리팩토링 단계를 거친 후에도 **여전히 동작**하는지 확인해야 한다
- 작은 규모의 개선 사항을 찾아내는 식으로 진행되는 미시적 기법
  - 여러 작은 단계를 **엄격하고 지속**적으로 적용해야만 커다란 구조적 개선으로 이어진다

그래서 리팩토링이란 것은 작동 방식이 아닌 표현을 개선하는 과정입니다.
그리고 각 단계는 이해하기 쉽고 안전해야 합니다.
리팩토링 이후에도 여전히 동작해야 합니다.
이렇게 작은 단위로 안전하게 코드 개선을 진행하는 동안, 일일이 수동테스트로 확인한다면 매우 지루하고 귀찮을 겁니다. 그래서 저자는 자동화를 강조하는 것입니다.

여기까진 익숙한 **TDD**였는데요

조금 더 큰 그림을 그려봅시다

테스트 주도 개발

단위 테스트 만으로 만족하십니까?

- TDD 프로세스가 주는 아주 중요한 혜택을 놓치는 셈
- 그 코드, 사용하는 곳이 있나요?
- 시스템의 다른 부분과 통합될 수 있나요?
- 아니면 다시 작성하세요. 실패하는 테스트부터.

저자는 테스트 주도 개발이 이렇게 좋긴 한데, 단위 테스트 만으로 만족할 수 있냐는 질문을 하고 있습니다. TDD 프로세스가 주는 아주 중요한 혜택을 놓친다고 했는데, 중첩된 피드백 고리를 구성하면 다양한 피드백을 받고, 잘못된 시스템을 고칠 기회를 얻을 수 있는데도, 단위 테스트만 작성할 거냐고 묻는 것이죠.
단위 테스트 만으로는 그 코드가 실제 사용되는지, 시스템에 통합될 수 있는지 알 수 없으니까요.

"우리는 인수테스트를 작성하는 것으로 시작합니다"

인수테스트

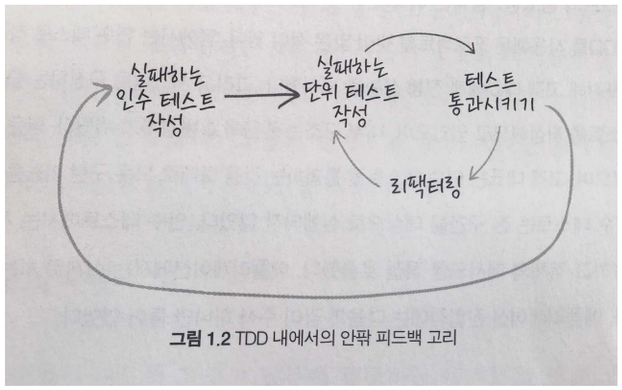


그림 1.2 TDD 내에서의 안팎 피드백 고리

- 만들고자 하는 기능을 시험하는 테스트
- 실패하는 동안 아직 그 기능이 구현하지 않았음을 보여준다

이건 중첩된 피드백 고리를 보여주는 그림입니다.
만들고자 하는 기능을 먼저 인수테스트로 작성하고요. 이걸 구현하는 과정에서 TDD 사이클이 돌아가겠죠? 인수테스트는 그 과정에서 계속 실패할 것이고, 아직 그 기능이 구현되지 않았음을 명확하게 보여줍니다.

인수테스트

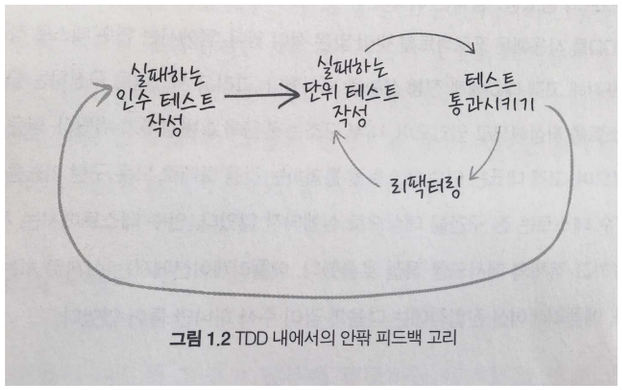


그림 1.2 TDD 내에서의 안팎 피드백 고리

- 바깥쪽 테스트 고리는 보여줄 수 있는 진척도를 측정하는 수단
    - 테스트 스위트가 증가하면 시스템을 변경할 때 회귀 실패에서 보호받을 수 있다
- 안쪽 고리는 개발자들에게 도움이 된다
    - 단위테스트는 코드 품질을 유지하는데 도움이 되고, 작성한 후에는 바로 통과해야 한다.
    - 실패하는 단위 테스트는 소스 저장소에 절대 커밋해서는 안 된다

그래서 바깥쪽 테스트 고리는 진척도를 측정하는 수단이 되고요.
안쪽 고리는 개발자들에게 좋은 피드백을 주겠죠.

인수테스트

전 구간(End to End) 테스트

- 시스템 내부 코드를 가능한 한 직접 호출하지 않는다
- 시스템 외부 환경과의 상호작용을 포함해야 한다
- '경계 간 테스트' 뿐 아니라 해당 시스템을 구축하고 배포하는 프로세스까지
- 자동화된 빌드
  - 소스 체크인 > 코드 컴파일 > 단위 테스트 실행 > 시스템에 통합하고 패키지화 > 운영 환경 수준으로 배포 > 외부 접근 지점을 통해 테스트

인수테스트를 구현하는 방법은 다양한데요. 저자는 전 구간 테스트를 한다고 합니다.
단순히 UI나 외부 시스템을 활용하는 '경계 간 테스트'가 아니고,
코드 수정부터 시스템 통합과 배포까지를 아우르는 개념으로 이야기 하고 있습니다.

인수테스트

(중첩된 피드백 고리에서) 테스트의 계층 구조와 목적

- 인수 테스트 : 전체 시스템이 동작하는가?
- 통합 테스트 : 변경할 수 없는 코드를 대상으로 코드가 동작하는가?
- 단위 테스트 : 객체가 제대로 동작하는가? 객체를 이용하기가 편리한가?

외부 품질과 내부 품질

외부 품질

- 시스템이 **고객과 사용자의 요구**를 얼마나 잘 충족하는가
  - 기능, 신뢰성, 가용성, 응답성 등
- 계약의 일부

내부 품질

- 시스템이 **개발자와 관리자의 요구**를 얼마나 잘 충족하는가
  - 이해하기 쉬운가, 변경하기 쉬운가
- 시스템 동작 방식을 안전하고 예상 가능한 상태로 바꿀 수 있게 만드는 것

외부 품질과 내부 품질

전구간 테스트로는

- 시스템의 외부 품질을 알 수 있다
- 우리 팀이 도메인을 얼마나 잘 이해하는지 알 수 있다

But

- 코드를 얼마나 잘 작성 했는지는 알 수 없다

외부 품질과 내부 품질

단위 테스트로는

- 내부 품질을 개선하는 데 도움이 된다
  - 테스트 픽스처에서 해당 단위를 시스템 바깥에서 실행할 수 있게 구조화해야 하기 때문

But

- 시스템이 전체적으로 동작하는지 충분히 확신할 수 없다

외부 품질과 내부 품질

통합 테스트는

- 두 테스트 사이의 어딘가...

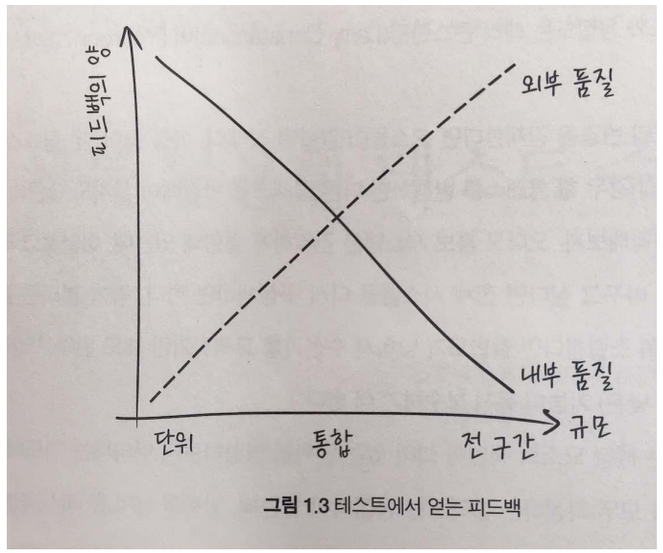


그림 1.3 테스트에서 얻는 피드백

그래서 단위테스트 만으로 너희가 할 일을 다 한 게 맞냐고 지적하는 것으로 보여요.

결합도와 응집도

클래스를 단위 테스트하려면

- 클래스가 대체할 수 있는 **명시적인 의존성**과
- 호출하고 검증할 수 있는 **명확한 책**임을 지녀야 한다

즉,

- 느슨하게 결합되고
- 응집력이 높아야 한다

결합도와 응집도도 언급을 했는데요.
단위테스트를 제대로 하기 위해선,
클래스가 명시적인 의존성과 명확한 책임을 지녀야 한다고 했습니다.
느슨하게 결합되고 응집력이 높아야 한다는 의미입니다.

결합도와 응집도

결합도

- **한 변경이 다른 것의 변경을 강제**한다면 요소들이 결합된 상태
- 느슨하게 결합된 기능이 유지 보수하기 더 쉽다

응집도

- 해당 요소의 **책임이 의미 있는 단위를 형성**하는지 나타낸다
- 해당 클래스가 전체 개념을 나타내지 않는다면 응집력이 있을 확률이 낮다
  - 예 : URL에서 일부만 파싱하는 클래스

2장 객체를 활용한 테스트 주도 개발

"중요한 것은 **'메시지 전달'**이며, **...** 위대하고 성장 가능한 시스템을 만들 때의 핵심은 **모듈 간의 의사소통**에 있지, 모듈의 내부 특성이나 작동 방식에 있지 않다."

- 앨런 케이 *

* 컴퓨터는 쉽고 단순해야 한다, GUI의 선구자 앨런 케이

엘런 케이란 사람의 말이 종종 소개 되는데요. 이 사람은 객체 지향 개발 방식의 장점을 알리기 위해서 스몰토크라는 언어를 개발했습니다. 거의 최초의 객체지향 언어라고 꼽히고 있어요. 아직 살아계십니다.
이 분은 객체 지향 시스템에서 중요한 것은 모듈간의 의사소통, 즉 메시지 전달이라고 했습니다.

객체를 활용한 테스트 주도 개발

값(value) vs 객체(object)

값(value)

- 변하지 않는 양이나 크기
- 두 값을 식별자로 비교하는 것은 적절하지 않다

같은 클래스로 구현되어 있다고 해도, 모두 동일한 객체로 보면 안된다는 내용이 있습니다.
값은 변하지 않는 양이나 크기를 나타내고 식별자로 비교할 수 없습니다.

객체를 활용한 테스트 주도 개발

값(value) vs 객체(object)

객체(object)

- 시간이 지남에 따라 상태가 변할 수도 있다
- 식별자(identity)가 있는 계산 절차(computational process)
- 변경 가능한 상태를 이용한 객체의 행위
- 이 책에선 **식별자와 상태, 처리 과정**을 지닌 인스턴스를 가리킨다

이 책에서 객체는 식별자와 상태와 처리과정을 지닌 인스턴스를 의미합니다.

객체를 활용한 테스트 주도 개발

객체망

객체 지향 시스템은 **협업하는 객체의 망**으로 구성되어 있다

- 객체는 **메시지**로 의사소통한다
- 자신이 이해할 수 있는 메시지를 처리하는 **메소드**가 있다
- 이 과정에서 사용하는 내부 상태를 **캡슐화**한다

객체 지향 시스템은 협업하는 객체의 망으로 구성되어 있다고 이야기 합니다.

- 메시지로 의사소통하고
- 메시지를 처리하는 메소드가 있고
- 이 과정에서 각 객체는 내부 상태를 캡슐화하는 것이죠

객체를 활용한 테스트 주도 개발

객체망

객체 지향 시스템은 **협업하는 객체의 망**으로 구성되어 있다

- **객체를 생성**해 서로 메시지를 주고받을 수 있게 **조립**하는 과정을 거친다
- 시스템의 행위는 **객체의 조합**(객체의 선택과 연결 방식)을 통해 나타나는 특징
- 객체의 구성을 변경해 시스템 작동 방식을 바꿀 수 있다
- 이런 **선언적(declarative)** 접근법으로 방법이 아니라 **목적에 집중**할 수 있다

객체망에서 시스템의 행위란 객체의 조합을 통해 나타난다고 합니다.
그래서 객체의 구성을 변경해서 시스템의 작동 방식을 바꿀 수 있다고 합니다.
이런 걸 선언적 접근법이라고 부르는데, 내부 구현이나 방법 보다는 목적 자체에 집중할 수 있습니다.

객체를 활용한 테스트 주도 개발

메시지를 따르라

선언적 접근법에서는 일반적인 의사소통 패턴을 따른다

- 객체의 역할
- 객체에서 전달 가능한 메시지
- (구상) 클래스 대신 (추상) 인터페이스로 객체의 역할을 파악한다

선언적 접근법에서는 일반적인 의사소통 패턴을 따른다고 합니다.
객체의 역할을 정의하고, 전달 가능한 메시지를 선언하고, 구상 클래스 대신 추상 인터페이스로 객체의 역할을 파악한다는 의미입니다.
객체는 일반적으로 이런 식으로 의사소통 방식을 정의한다고 보면 되겠습니다.

객체를 활용한 테스트 주도 개발

메시지를 따르라

의사소통 패턴은 **객체 간**에 있을 법한 **관계**에 의미를 부여한다

- 정적인 분류를 너머 **동적인 의사소통**의 구조로 바라본다
- 이는 프로그래밍 언어로는 명확히 표현되지 않는다
- 도메인 모델이 명확히 드러나지 않는다

이런 의사소통 패턴은 객체 간의 관계에 의미를 부여하는 것인데,
객체의 정적인 분류 보다는 동적인 의사소통의 구조로 바라보는 것입니다.
프로그래밍 언어로는 이런 관계가 명확히 표현 되지는 않는다고 해요.
다르게 표현하면, 도메인 모델이 명확히 드러나지 않는다는 의미입니다.

역할, 책임, 협력자의 관점으로 바라보기

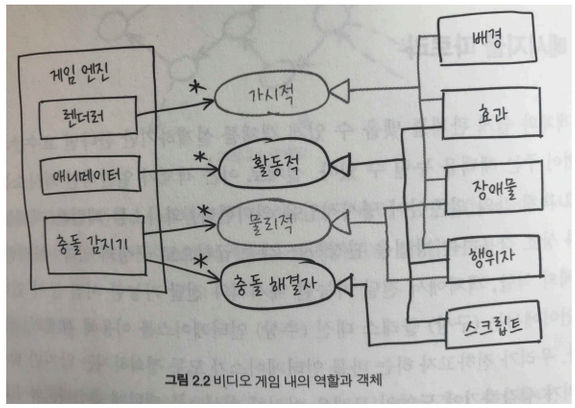


그림 2.2 비디오 게임 내의 역할과 객체

- 게임엔진에서 바라보는 관점과 특정 역할을 담당하는 객체를 구현하는 관점은 다르다
- 정적인 분류와 동적인 의사소통 간의 불일치

책의 예제에서는,
- 게임엔진에서 바라보는 관점
- 특정 역할을 하도록 객체를 구현하는 관점
은 서로 다를 수 밖에 없고, 깔끔하게 설계하기 어렵다고 이야기 합니다.
이를 정적인 분류와 동적인 의사소통 간의 불일치라고 표현하기도 했죠.

역할, 책임, 협력자의 관점으로 바라보기

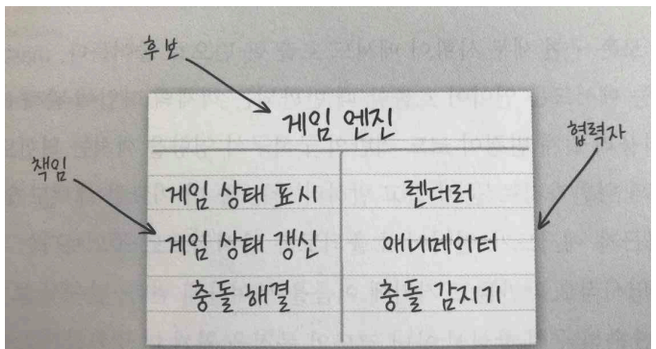


그림 2.3 비디오 게임에 대한 CRC 카드

- CRC(Class, Responsibility, Collaborator) 카드
  - [설계] CRC카드 , 클래스다이어그램
- 객체는 역할을 하나 이상 구현한 것
- 후보 : 역할. 관련된 **책임의 집합**
- 책임 : 어떤 과업을 수행하거나 정보를 알아야 할 **의무**
  - 클래스가 다른 클래스에 제공하는 서비스. 오퍼레이션
- 협력자 : 클래스가 **책임을 완료하기 위해** 사용하는 클래스

이런 의사소통 패턴을 정리하는 방법으로 CRC 카드를 소개하기도 했습니다.
CRC는 Class, Responsibility, Collaborator를 의미하고요. 객체를 역할, 책임, 협력자의 관점으로 바라보고 있습니다.
역할은 관련된 책임의 집합, 책임은 어떤 과업을 수행하거나 정보를 알아야 할 의무, 협력자는 책임을 완료하기 위해 사용하는 다른 클래스를 의미합니다.

객체를 활용한 테스트 주도 개발

묻지 말고 말하라(Tell, Don't ask)

- 디미터의 법칙(Law of Demeter)
- 객체의 의사소통이란
  - 호출할 객체가 무엇을 원하는지 기술
  - 호출된 객체가 그것을 어떻게 실현할지 결정
- 다른 객체를 탐색해 뭔가를 일어나게 해서는 안된다
- 호출자는 객체의 내부 구조나 역할 인터페이스 너머에 존재하는 구조를 알 필요가 없다

객체 간의 의사소통은 어떻게 해야 하는가라는 질문에는 '묻지 말고 말하라'라고 합니다. Tell, Don't ask. 혹은 디미터의 법칙이라고도 합니다.
객체의 의사소통에선

- 호출할 객체가 무엇을 원하는지 기술해야하고
- 그것을 어떻게 실현할 지는 호출된 객체가 결정해야 합니다

객체를 활용한 테스트 주도 개발

묻지 말고 말하라(Tell, Don't ask)

```
((EditSaveCustomizer)) master.getModelisable()
    .getDockablePanel()
        .getCustomizer())
            .getSaveItem().setEnabled(Boolean.FALSE.booleanValue());
```

- 호출할 객체의 내부 구조에 의존하고 있어서 유연하지 않다

책의 예제는 master의 내부 구조에 의존하고 있어서, 이 코드는 master 클래스가 변경될 때마다 함께 변경되어야 합니다. 유연하지 않죠.

객체를 활용한 테스트 주도 개발

묻지 말고 말하라(Tell, Don't ask)

```
((EditSaveCustomizer)) master.getModelisable()
    .getDockablePanel()
        .getCustomizer())
            .getSaveItem().setEnabled(Boolean.FALSE.booleanValue());
```

- 호출할 객체의 내부 구조에 의존하고 있어서 유연하지 않다

master**.allowSavingOfCustomisation();**

master에 꼬치꼬치 물어보지 말고, 진짜로 원하는 게 뭔지 이야기 하라는 의미입니다.
원하는 것만 얻을 수 있으면, 내부적으로 어떻게 구현할지 관심을 두지 않습니다.

객체를 활용한 테스트 주도 개발

그래도 가끔은 물어라

- 항상 말하기만 하진 않는다
  - 값과 컬렉션으로부터 정보를 가져올 때
  - Factory를 이용해 새 객체를 생성할 때
- 스스로 답을 결정하기 위한 정보를 묻지 말고 **질의 메소드**를 추가하자
- 대상 객체의 상태 정보를 질의하기 보다 **의도**를 서술해야 한다
  - 적절한 객체에 행위가 자리 잡도록

하지만 항상 말하기만 하진 않는다고 하죠. 꼭 물어봐야 하는 경우도 생깁니다.
이럴 때는 물어본 뒤 스스로 답을 결정하지 말고, 해당 클래스에 질의 메소드를 추가하는 것이 좋다고 합니다. 의도를 서술하는 것이죠.

객체를 활용한 테스트 주도 개발

그래도 가끔은 물어라

```
public class Train {
    private final List<Carriage> carriages [...]
    private int percentReservedBarrier = 70;

    public void reserveSeats (ReservationRequest request) {
        for (Carriage carriage : carriages) {
            if (carriage.getSeats().getPercentReserved() < percentReservedBarrier) {
                request.reserveSeatsIn(carriage);
                return;
            }
        }
        request.cannotFindSeats();
    }
}
```

역시 책에 나온 예제입니다.
캐리지가 충분한지 판별하는 것이 이 코드의 의도입니다.
지금은 판별 로직이 바뀌면 이와 비슷한 코드는 모두 함께 변경이 되겠죠. Train Class는 Carrige 클래스와 결합도가 매우 높습니다.

객체를 활용한 테스트 주도 개발

그래도 가끔은 물어라

```
public class Train {
    private final List<Carriage> carriages [...]
    private int percentReservedBarrier = 70;

    public void reserveSeats (ReservationRequest request) {
        for (Carriage carriage : carriages) {
            if (carriage.getSeats().getPercentReserved() < percentReservedBarrier) {
                request.reserveSeatsIn(carriage);
                return;
            }
        }
        request.cannotFindSeats();
    }
}
```

carriage.**hasSeatsAvailableWithin(percentReservedBarrier)**

이렇게 코드를 수정하면 Train class가 원하는 의도를 명확하게 알 수 있고, 나중에 정책이 변경되거나 구현이 바뀐다 해도 Train class는 변경되지 않겠죠.

객체를 활용한 테스트 주도 개발

협력 객체의 단위 테스트

모두가 묻지 말고 말하고만 있으면 단정문(assertion)은 어떻게 쓸 수 있을까?

- 테스트 대상 객체의 이웃을 mock 객체로 대체
- 대상 객체가 가짜 이웃과 어떻게 상호작용할지(expectation) 지정
- 테스트는 객체에 필요한 보조 역할을 파악하는 데 도움이 된다
  - 인터페이스 발견(interface discovery)
- 협력 객체의 역할을 Interface로만 인식한다면 실제 구현까지 꼭 필요할까?

모두가 묻지 말고 말하고만 있으면 내부 구현은 드러나지 않는다는 말인데,
그럼 단정문은 어떻게 쓸 수 있을까요?
여러가지 방법이 있겠지만, 저자는 목 객체로 대체하면 된다고 합니다.
그럼 대상 객체가 가짜 이웃과 어떻게 상호작용할지 예상 구문을 지정할 수 있고요,
객체가 필요로 하는 보조 역할을 무엇인지 파악할 수도 있습니다. 책임을 구현하는데 있어서 협력자가 누구인지, 어떤 역할을 하는지.
게다가 협력 객체의 역할을 인터페이스로 인식하고 있다면,
테스트에서 실제 구현체까지 필요하지 않을 수 있다고 이야기 합니다.

객체를 활용한 테스트 주도 개발

협력 객체의 단위 테스트

모조 객체(mockery)

- 테스트의 콘텍스트를 담고 있고
- 목 객체를 생성하고
- 테스트에 대한 예상 구문과 스텁 행위를 관리

목으로 테스트 하기 위해 모조 객체라는 것도 소개합니다.
테스트의 콘텍스트를 담고, 목객체를 생성하고, 예상 구문과 스텁의 행위를 관리하는 걸 모조 객체라고 부른다고 했습니다.

객체를 활용한 테스트 주도 개발

Mock을 활용한 테스트의 핵심 구조

- (arrange) 필요한 목 객체 생성
- (arrange) 대상 객체를 포함한 실제 객체 생성
- (arrange) 대상 객체에서 목 객체가 어떻게 호출될지 예상 구문 기술
- (act) 대상 객체에서 유발(trigger) 메소드 호출
- (assert) 결과 값이 유효한지, 예상되는 메소드 호출이 모두 일어났는지 확인

TALK