C#으로 배우는 적응형 코드

디자인 패턴과 SOLID 원칙 기반의 애자일 코딩

Adaptive Code via C#:
Agile coding with design patterns and SOLID principles

Authorized translation from the English language edition, entitled ADAPTIVE CODE VIA C#: AGILE CODING WITH DESIGN PATTERNS AND SOLID PRINCIPLES, 1st Edition by HALL, GARY MCLEAN, published by Pearson Education, Inc, publishing as Microsoft Press, Copyright © 2015 by Gary McLean Hall. All rights reserved.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Korean language edition published by J-Pub Co. Copyright © 2015

이 책의 한국어판 저작권은 에이전시 원을 통해 저작권자와의 독점 계약으로 제이펍 출판사에 있습니다.
신저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

C#으로 배우는 적응형 코드

초판 1쇄 발행 2015년 11월 26일

지은이 게리 맥린 홀
옮긴이 장현희
펴낸이 장성두
펴낸곳 제이펍

출판신고 2009년 11월 10일 제406-2009-000087호
주소 경기도 파주시 문발로 141 뮤즈빌딩 403호
전화 070-8201-9010 / **팩스** 02-6280-0405
홈페이지 www.jpub.kr / **이메일** jeipub@gmail.com

편집부 이민숙, 이 슬, 이주원 / **소통·기획팀** 민지환, 현지환
용지 신승지류유통 / **인쇄** 해외정판사 / **제본** 광우제책사

ISBN 979-11-85890-37-1 (93000)
값 30,000원

※ 이 책은 저작권법에 따라 보호를 받는 저작물이므로 무단 전재와 무단 복제를 금지하며,
이 책 내용의 전부 또는 일부를 이용하려면 반드시 저작권자와 제이펍의 서면동의를 받아야 합니다.
※ 잘못된 책은 구입하신 서점에서 바꾸어 드립니다.

제이펍은 독자 여러분의 아이디어와 원고 투고를 기다리고 있습니다. 책으로 펴내고자 하는 아이디어나 원고가 있으신 분께서는 책의 간단한 개요와 차례, 구성과 저(역)자 약력 등을 메일로 보내주세요. **jeipub@gmail.com**

C#으로 배우는 적응형 코드

디자인 패턴과 SOLID 원칙 기반의 애자일 코딩

Adaptive Code via C#

Agile coding with
design patterns and SOLID principles

게리 맥린 홀 지음 / **장현희** 옮김

Jpub 제이펍

※ 드리는 말씀

- 이 책에 기재된 내용을 기반으로 한 운용 결과에 대해 저/역자, 소프트웨어 개발자 및 제공자, 제이펍 출판사는 일체의 책임을 지지 않으므로 양해 바랍니다.
- 이 책에 기재한 회사명 및 제품명은 각 회사의 상표 및 등록명입니다.
- 이 책에서는 ©, ®, ™ 등의 기호를 생략하고 있습니다.
- 이 책에서 사용하고 있는 실제 제품 버전은 독자의 학습 시점에 따라 책의 버전과 다를 수 있습니다.
- 일부 내용이나 URL은 우리나라의 실정에 맞게 역주 표기 없이 수정되었음을 밝힙니다.
- 이 책에 나오는 예제 코드와 부록 B는 다음의 사이트에서 다운로드할 수 있습니다.
  - https://github.com/Jpub/AdaptiveCodeWithCSharp
- 책의 내용과 관련된 문의사항은 옮긴이나 출판사로 연락주시기 바랍니다.
  - 옮긴이: aspnetmvp@gmail.com
  - 출판사: jeipub@gmail.com

차례

옮긴이

머리말

소프트웨어 개발에 있어서 애플리케이션과 소스 코드의 구조를 결정하고 모듈화하여 재사용이 가능하면서도 유지보수가 쉬운 코드를 만드는 방법은 계속해서 강조되어 왔습니다. 특히, 객체지향 프로그래밍 환경에서 디자인 패턴의 습득 및 활용이 보편화되면서 많은 개발자가 패턴을 학습하고 이를 현업에 적용하기 위해 노력해 왔습니다. 그러나 초급 개발자들에게는 패턴의 적용이 쉽지 않습니다. 역자는 그 이유를 대부분의 관련 도서들이 각각의 패턴이 '무엇'이며 '어떻게' 구현할 수 있는지에만 초점을 맞추었을 뿐, '어떤' 상황에 적용해야 하는지에 대해서는 적절한 예시를 들지 않았기 때문이라고 생각합니다.

물론, 경험이 풍부한 개발자들은 언제, 어떤 패턴을 적용할 것인지를 결정하는 게 그다지 어렵지 않을 수도 있습니다. 하지만 디자인 패턴에 아직 익숙하지 않은 개발자에게는 '어떤 개발 언어를 선택해서 공부해야 하는가'만큼이나 혼란스럽고 결정을 쉽게 내리기 힘든 게 바로 '어떤 상황에 어떤 패턴을 적용할 것인가'를 결정하는 일일 것입니다.

디자인 패턴과 관련된 도서들이 지니는 두 번째 문제점은 각각의 패턴이 어떤 문제를 해결할 수 있으며 어떻게 코드를 작성해야 하는지에 대해서는 설명하지만, 이를 위해 사용된 예제 코드는 정말 예제일 뿐 실제로 처한 상황과는 동떨어진 코드인 경우가 대부분이라는 점입니다. 그런 면에서 이 책은 앞서 이야기한 두 가지 문제를 매우 효과적으로 해결하였습니다. SOLID 원칙에 따라 적용 가능한 패턴을 구분 짓고, 이를 바탕으로 실제 동작하는 예제를 구현하면서 어떤 패턴을 어떤 상황에 적용할 수 있는지를 명쾌하게 설명하고 있습니다. 특히, 이제는 대부분의 기업이 채택하고 있는 스크럼 방법론과 짝을 이루어 소프트웨어 개발 주기 전체를 살펴볼 수 있도록 하고 있습니다.

특히, 이 책의 대부분을 차지하고 있는 SOLID 원칙은 지금껏 역자가 읽었던 그 어떤 책이나 블로그 포스트보다도 상세하면서도 명확하게 설명되어 있습니다. 더불어 이를 현실적인 예제

를 통해 보여주고 있어서 실감 나는 학습이 될 것입니다. 경험이 적은 개발자에게는 이 책의 내용이 앞으로의 성장에 매우 탄탄한 밑거름이 되어 줄 것이며, 경험이 풍부한 개발자는 이 책 곳곳에서 다양한 아이디어를 얻을 수 있을 것입니다.

개인적으로 이 책을 번역하면서 가장 안타까웠던 점은 국내 시장에서 큰 힘을 발휘하지 못하고 있는 마이크로소프트의 .NET 프레임워크와 C# 언어를 기초로 작성된 책이라는 점입니다. 하지만 스크럼이나 SOLID 원칙이 언어와는 무관한 개념이기에, 설령 C# 언어를 모른다 하더라도 다른 개발 언어에 대한 경험이 있다면 이 책을 읽기에는 무리가 없을 것입니다. 부디 이 책이 독자 여러분의 개발 능력을 한 단계 더 성장시킬 수 있는 계기가 될 수 있기를 바랍니다.

마지막으로, 좋은 책을 먼저 읽어 보게 하고 독자에게 소개할 기회를 주시는 제이펍의 장성두 실장님, 그리고 늘 한마음으로 응원해 주는 사랑하는 가족에게 감사의 말씀을 전합니다.

옮긴이 **장현희**

이 책에 대하여

이 책의 제목에 사용된 적응형 코드(adaptive code)는 이 책에서 소개하고자 하는 원칙들을 적용하여 산출한 결과물을 제대로 설명하는 단어라 할 수 있다. 적응형 코드를 한마디로 말하자면, 새로운 요구 사항이나 예상하지 못한 시나리오에 봉착했을 때 코드를 크게 수정하지 않고도 이를 적용해 나갈 수 있는 능력을 갖춘 코드이다. 이 책은 마이크로소프트 닷넷(Microsoft .NET) 프레임워크 기반의 C# 프로그래밍에서 활용되는 다양한 실용적 사례를 모두 모아 소개하는 것을 목표로 하고 있다. 책에서 소개하는 내용 중 일부는 이미 다른 책에서 언급되었을 수도 있지만, 그런 책들은 지나치게 이론에만 치중하거나 닷넷 개발 환경을 충분히 고려하지 않고 있다.

프로그래밍이라는 과정은 느리게 진행될 수 있다. 적응형 코드를 작성하면 변화를 수용하지 못하는 예전 방식의 코드를 다룰 때보다 빠르고 쉽게 코드를 수정할 수 있으며, 오류가 발생할 가능성도 줄어들게 된다. 개발자라면 누구나 알고 있겠지만, 요구 사항(requirements)은 어떤 변화의 시발점이다. 성공적인 소프트웨어 프로젝트와 그렇지 않은 프로젝트를 구분하는 핵심 가치는 변화를 관리하는 방법의 차이이다. 요구 사항에 따른 변화에 대해 개발자들은 다양한 방법으로 대응해 나갈 수 있지만, 그중에서도 지속적으로 이목을 끄는 두 가지 상반된 관점의 접근법이 존재한다.

첫 번째 접근법의 경우, 개발자들은 개발 과정부터 클래스 디자인에 이르기까지 매우 엄격한 관점을 채택하게 된다. 프로젝트는 마치 50년 전에 천공 카드를 이용해서 만들어진 것만큼이나 유연함과는 거리가 멀다. 폭포수(waterfall) 접근법은 소프트웨어는 자유롭게 변경될 수 없다는 생각에 바탕을 두고 있다. 요구 사항을 분석하고, 디자인하고, 구현하고, 테스트하는 모든 과정에서 발생하는 의사결정은 분명하면서도 단방향이어서 일단 구현이 시작된 이후에는 고객들이 요구 사항을 변경하기가 대단히 어렵게 (또는 변경할 수 있더라도 엄청난 비용이 발생하게)

만든다. 그래서 코드는 변화에 대응할 필요가 없으며, 프로세스가 전부일 뿐 변화는 수용되지 않는다.

두 번째 접근법인 애자일(agile) 방법론은 강경한 방법론의 또 다른 형태가 아니라 그런 방법론에 대응하기 위한 방법론이다. 애자일 프로세스의 목적은 변화를 고객과 개발자 사이의 계약 일부로 포용하는 것이다. 만일 고객이 제품 일부를 변경하고 그에 대한 보수를 지급할 의향이 있다면, 이때 일시적으로 발생하는 비용은 현재 개발이 어느 정도 진행되었는지가 아니라 어느 정도 변화가 필요하냐에 따라 정해져야 한다.

물리적 엔지니어링과 달리 소프트웨어 엔지니어링은 변경이 가능한 도구, 즉 소스 코드를 다루는 일이다. 집을 구성하는 건축물은 말 그대로 건축 과정에 따라 만들어지게 된다. 따라서 집의 디자인을 변경하는 데 따른 비용은 전체 건축 과정의 완료 여부와 밀접하게 관련 있다. 만일 프로젝트가 시작하지 않은 상황이라면 (즉, 아직 청사진을 그리는 단계라면) 변화의 비용은 상대적으로 저렴할 수 있다. 그러나 창틀이 완성되고, 전기가 공급되기 시작하고, 수도관이 연결된 상태에서 위층의 화장실을 아래층의 부엌 옆으로 옮기려 한다면 그 비용은 상상을 초월할 것이다. 반면에 소스 코드의 경우, 기능의 위치를 변경하고 그에 대한 내비게이션을 위해 사용자 인터페이스를 변경하는 일은 통상적으로 그다지 어려운 일은 아니다. 물론, 때에 따라서는 어려운 일이 될 수도 있다. 결과적으로는 변경을 위해 일시적으로 발생하는 비용이 변화 자체를 가로막는 경우가 종종 발생한다. 필자가 파악한 바로는 코드의 적응성이 낮을 때 이런 일들이 발생하곤 한다.

이 책은 현실적인 예제 및 실용적인 적응형 코드와 함께 애자일 방법론을 설명하고 그에 대한 예시를 보여 주고자 한다.

누구를 위한 책인가?

이 책은 이론과 실제의 간극을 좁히기 위해 쓰인 책이다. 이 책이 대상으로 하는 독자는 디자인 패턴, SOLID 원칙, 단위 테스트(unit test) 및 리팩토링(refactoring) 등에 대한 보다 실용적인 예제를 탐구하는 경험 있는 프로그래머들이다.

지식의 폭을 넓히고자 하거나 현실의 프로그래밍이 간단한 예제나 이론을 통해 해결되지 않아서 이 책이 소개하는 여러 이점을 활용하고자 하는 초보 개발자들에게도 유용할 것이다. 이제

많은 개발자가 SOLID 원칙에 대해 이해하고 있지만, 개방/폐쇄 원칙(open/closed principal, 제6장에서 다룬다)과 리스코프 치환 원칙(Liskov substitution principal, 제7장에서 다룬다)처럼 복잡한 원칙에 대해서는 아직 많이 어려워하고 있다. 설령, 경험이 풍부한 프로그래머라 하더라도 의존성 주입(dependency injection, 제9장에서 다룬다)의 장점을 완벽히 이해하지 못하는 경우도 있다. 마찬가지로, 인터페이스(interface, 제3장에서 다룬다)가 가져다주는 유연성(또는 적응성) 역시 종종 과대 포장되기도 한다.

이 책은 중급 개발자가 대중화된 패턴과 실제 사례들을 어떻게 바라보는 것이 유용하며, 또 어떤 경우에 장기적으로 해가 될 수 있는지를 배워나가는 데도 도움을 제공한다. 필자는 직원을 채용하기 위한 면접 과정에서 그들이 작성한 코드 예제에서 많은 공통점을 발견했다. 대체로 후보자들이 지닌 기술의 차이는 그다지 크지 않았다. 다만, 더 뛰어난 프로그래머가 되기 위해 올바른 방향으로 나아갈 수 있도록 조금씩 조정해 줄 필요가 있을 뿐이다. 예제 코드에서는 특히 엔토리지(Entourage) 안티패턴(제2장에서 다룬다)과 서비스 로케이터(Service Locator) 안티패턴(제9장에서 다룬다)을 상당수 발견할 수 있었다. 이에 대한 실용적인 대안과 그에 대한 근거는 앞으로 이 책을 통해 설명할 것이다.

이 책이 가정하고 있는 것들

이 책을 읽기에 이상적인 독자는 C#과 문법적으로 유사한 자바나 C++ 같은 프로그래밍 언어에 대한 약간의 실무 경험을 가지고 있는 독자들이다. 또한, 조건식 분기, 반복문, 그리고 표현식 등과 같은 절차형 프로그래밍(Procedural Programming)의 기본 개념들을 알고 있다면 더욱 좋다. 그 외에 클래스를 사용한 객체지향(Object Oriented) 프로그래밍의 개념과 인터페이스에 대한 기초 개념을 이해하고 있어야 한다.

이 책이 적합하지 않은 독자

프로그램 작성 방법을 이제 막 학습하기 시작한 독자라면 이 책은 적합한 책이 아니다. 이 책은 독자들이 기본적인 프로그래밍의 개념을 이해하고 있다는 가정하에 고급 주제들을 다루고 있다.

이 책의 구성

이 책은 세 개의 부로 구성되며, 각 부의 내용은 독립적으로 구성되었다. 즉, 순서와 관계없이 읽을 수 있다는 뜻이다. 각 장은 개별적인 주제를 다루고 있으며, 필요한 경우에는 다른 장의 내용을 참조하는 경우도 있다.

제1부: 애자일의 기본기 갖추기

1부는 적응성을 고려한 방법으로 소프트웨어를 개발하기 위한 기본기를 다룬다. 이를 위해 코드의 적응성을 요구하는 고수준의 애자일 프로세스인 스크럼(Scrum)에 관해 설명한다. 이 부를 구성하는 장들은 인터페이스, 디자인 패턴, 리팩토링 및 단위 테스트 등을 심도 있게 설명한다.

■ 제1장: 스크럼을 소개합니다

이 장에서는 이 책 전체의 근간을 이루며 애자일 프로젝트 관리 방법론인 스크럼 방법론을 설명한다. 스크럼 방법론을 통해 생산될 수 있는 산출물과 규칙, 매트릭스 및 스크럼 프로젝트의 단계들에 대해 깊이 있게 해설한다. 마지막으로, 개발자들이 애자일 환경에서 자신을 통제하는 법과 작성하는 코드를 관리하는 방법에 대해서 알아본다.

■ 제2장: 의존성과 계층화

이 장에서는 의존성과 구조적 계층에 대해 살펴본다. 솔루션이 올바르게 구성되어야 비로소 코드가 적응성을 가질 수 있게 되기 때문이다. 이를 위해 여러 종류의 의존성에 관해 설명한다. 퍼스트파티(first-party), 서드파티(third-party), 그리고 프레임워크(framework)가 바로 그것이다. 이 장에서는 (개발자가 피해야 할) 안티패턴부터 (개발자가 수용해야 할) 패턴에 이르기까지 개발자들이 의존성을 관리하고 정리하는 방법들을 소개한다. 또한, 관점지향 프로그래밍(AOP, Aspect-Oriented Programming)과 비대칭 계층화(asymmetric layering) 등의 고급 주제에 관해서도 소개한다.

■ 제3장: 인터페이스와 디자인 패턴

현대의 닷넷 개발 환경에서는 인터페이스가 널리 사용되고 있다. 그러나 간혹 잘못 적용하거나, 잘못 이해하고 있거나, 혹은 적절하지 못한 곳에 사용하고 있는 경우도 비일비재하다. 이

장에서는 보다 대중적이면서도 실용적인 디자인 패턴들을 살펴봄으로써 인터페이스의 개념을 설명한다. 구현된 클래스로부터 단순히 인터페이스를 추출하는 수준이 아니라, 문제를 해결하기 위해 인터페이스가 얼마나 정교하게 구현될 수 있는지를 구체적으로 설명한다. 믹스인(mixin), 덕-타이핑(duck-typing) 및 능동형 인터페이스(fluent interface) 등을 통해 프로그래머가 가진 가장 중요한 무기를 다채롭게 활용하는 방법을 익히게 된다.

■ 제4장: 단위 테스트와 리팩토링

단위 테스트와 리팩토링은 이제는 반드시 갖추어야 할 기본 소양이 되어 가고 있다. 이 둘은 매우 밀접하게 관련이 있으며, 이 둘의 조화를 통해 적응형 코드를 개발해 나가게 된다. 단위 테스트라는 안전그물이 없다면 리팩토링은 에러를 양산하게 될 뿐이다. 반대로, 리팩토링을 하지 않는다면 다루기 어렵고 보수적이며 이해하기 어려운 코드만 만들어질 뿐이다. 이 장은 아주 기본적인 단위 테스트 예제에서 시작하여 능동적 단정(fluent assertions), 테스트 주도 개발(TDD, Test Driven Development) 및 모의 객체(mocking) 등 실용적인 고급 패턴들을 적용하여 점차 확장해 나간다. 리팩토링에 대해서는 소스 코드의 가독성과 유지보수성을 향상시키기 위해 실세계에서 사용되고 있는 기법들을 예제를 통해 설명한다.

제2부: SOLID 원칙에 기반한 코드 작성하기

2부는 1부의 내용을 토대로 기초를 쌓아올리는 과정이다. 각 장은 SOLID의 개별 원칙들을 하나씩 설명한다. 이 장들이 강조하는 것은 각 원칙을 단지 이론적으로 설명하는 데 그치지 않고 그 원칙을 구현한 실질적인 구현 예제를 제공하는 것이다. 각 장의 예제를 실무에 적용해 보면서 SOLID 원칙을 어떻게 구체화할 수 있는지를 명확히 보여 준다.

■ 제5장: 단일 책임 원칙

이 장은 데코레이터(Decorator)와 어댑터(Adapter) 패턴을 이용해 단일 책임 원칙을 구현하는 방법을 소개한다. 이 원칙을 적용하면 클래스의 개수는 늘어나지만, 각 클래스를 구현하는 코드의 양은 현저히 줄어든다. 그런 후 모든 기능을 구현한 거대한 클래스와 큰 문제를 해결하기 위한 작은 과정 하나에 초점을 맞춘 작은 클래스들을 비교해 본다. 작은 클래스들을 조합하면 이 클래스들을 하나의 클래스로 구현하는 것 이상의 능력을 갖추게 됨을 보게 될 것이다.

■ 제6장: 개방/폐쇄 원칙

개방/폐쇄 원칙은 이름은 단순하지만 코드에 미치는 영향은 어마어마하다. 이 원칙은 SOLID 원칙을 준수하는 코드에는 코드를 추가할 수는 있어도 수정할 수는 없음을 명확히 하는 역할을 담당하는 원칙이다. 또한, OCP와 관련해 예측된 다양성의 개념을 살펴보고 이 개념을 통해 개발자들이 적응성을 갖출 수 있는 부분을 식별할 수 있는 능력을 기르는 데 도움을 준다.

■ 제7장: 리스코프 치환 원칙

이 장은 리스코프 치환 원칙을 코드에 적용했을 때, 특히 가이드라인을 통해 개방/폐쇄 원칙을 강제하고 의도되지 않은 변경을 억제했을 때 얻을 수 있는 결과물들의 긍정적인 효과에 관해 설명한다. 또한, 코드의 계약(contract, 사전, 사후 조건 및 데이터 무결성 등)을 처리하기 위한 코드 계약 도구의 활용법도 알아본다. 더불어 공변성(covariance)와 반공변성(contravariance) 및 불변성(invariance)의 규칙이 무너졌을 때의 부정적 영향에 관해 설명한다.

■ 제8장: 인터페이스 분리

되도록 볼륨을 작게 유지해야 하는 것은 비단 클래스뿐만이 아니다. 이 장에서는 인터페이스 역시 대부분 너무 큰 단위로 정의되고 있음을 보여 준다. 인터페이스 분리는 종종 과도하게 활용되는 경향이 있지만, 사실은 매우 간단한 개념이다. 이 장에서는 인터페이스를 최대한 작은 크기로 유지할 때의 이점에 관해 설명한다. 또한, 클라이언트 및 아키텍처 관점에서 인터페이스 격리가 필요한 여러 상황을 보여 준다.

■ 제9장: 의존성 주입

이 장은 이 책의 나머지 장들을 모두 활용하기 위한 내용을 다룬다. 의존성 주입을 활용하지 않는다면 많은 것들이 불가능해진다. 따라서 의존성 주입은 대단히 중요한 개념이다. 이 장에서는 의존성 주입의 개념을 설명하고 이를 구현하기 위한 여러 방법을 비교한다. 또한, 객체의 생명 주기(lifetime)에 대한 관리, 제어의 역행(Inversion of Control) 컨테이너를 다루는 방법, 서비스 위치(service location)와 관련된 일반적인 안티패턴, 그리고 컴포지션 루트(composition root)와 레졸루션 루트(resolution root)를 식별하는 방법에 대해서도 알아본다.

제3부: 적응형 예제

3부에서는 이 책에서 다루었던 내용을 모두 조합하여 예제 애플리케이션을 구현한다. 이 부를 구성하는 장들은 많은 양의 코드로 이루어져 있지만, 해당 코드에 대한 풍부한 해설도 제공한다. 이 책은 애자일 개발 환경을 다루고 있으므로 각 장은 스크럼의 스프린트와 연결되며, 모든 작업은 백로그(backlog) 아이템의 결과물이며, 클라이언트는 계속해서 요구 사항을 변경한다.

■ 제10장: 적응형 예제 – 소개

이 장에서는 앞으로 개발할 애플리케이션을 소개한다. 독자들은 ASP.NET MVC 5를 토대로 채팅 애플리케이션을 개발하게 될 것이다. 아키텍처를 계획하고 이를 구현하기 위한 가이드라인으로서 간단한 디자인을 보여 준다. 또한, 백로그의 기능들에 대해서도 설명한다.

■ 제11장: 적응형 예제 – 스프린트 1

테스트 주도 개발(TDD, Test Driven Development) 기법을 도입하여 애플리케이션의 채팅방을 생성하고 메시지를 주고받는 첫 번째 기능을 구현한다.

■ 제12장: 적응형 예제 – 스프린트 2

이 장에서는 고객이 요구 사항을 변경하고, 팀은 적응형 코드를 통해 이 변경 사항을 수렴하게 된다.

부록

부록에서는 몇 가지 참고 자료를 소개한다. 특히, Git(깃) 소스 관리 도구를 활용하는 방법과 GitHub(깃허브)에 업로드되어 있는 이 책의 소스 코드에 대해 알아본다.

■ 부록 A: 적응형 도구들

이 장에서는 적어도 소스 코드를 GitHub에서 다운로드하여 마이크로소프트 비주얼 스튜디오 2013에서 컴파일할 수 있는 수준이 될 수 있도록 Git 소스 관리 도구에 대해 간략하게 설명한다. Git에 대한 전반적인 설명은 제공하지 않지만, 공식 Git 튜터리얼 같은 훌륭한 자료들이 이미 많이 존재한다.

http://git-scm.com/docs/gittutorial

또한, 웹 검색을 통해 더 많은 자료들을 찾을 수 있다.

이 부록에서는 지속적 통합(CI, Continuous Integration)과 개발 환경 같은 다른 개발자 도구들에 대해서도 살펴본다.

■ 부록 B(온라인으로만 제공): GitHub 예제 소스

예제 코드를 한곳에서 관리하기 위해 소스 코드를 GitHub에 업로드하였다. 저장소는 읽기 전용이지만, 부록 A와 부록 B를 통해 코드를 찾아 다운로드하고, 컴파일하고, 실행하며, 로컬 환경에서 변경 사항을 관리하는 방법을 설명한다. 만일 버그를 발견했거나 코드의 변경을 제안하고 싶다면, AdaptiveCode 저장소에 풀 리퀘스트(Pull Request)를 보내주기를 바란다. 기쁜 마음으로 살펴보도록 하겠다. 부록 B는 https://github.com/Jpub/AdaptiveCodeWithCSharp에서 볼 수 있다.

이 책에서 사용하는 규칙

이 책에는 몇 가지 규칙이 반복적으로 적용되어 있다. 주로 마이크로소프트 출판사업부가 출판하는 도서에는 표준처럼 사용되는 것들이지만, 다시 한 번 설명하기로 한다.

예제 코드

필요한 경우 예제 코드를 보여 주며, 예제 1-1처럼 관련된 곳에는 도움말을 표시한다.

예제 1-1 이 부분이 예제 코드이다. 이 책에서 수없이 많이 찾아볼 수 있다.

```
public void MyService : IService
{

}
```

코드의 특정 부분에 좀 더 주목할 필요가 있는 경우(예를 들면, 이전 예제에서 특정 코드가 변경되었을 때) 해당 코드를 볼드체로 표시한다.

노트와 사이드 바

노트(도움말)나 주의 사항은 작은 영역을 차지하는 상자를 이용하여 표시하며, 사이드 바는 본문과 별개로 보다 큰 영역에 표시한다. 아래 예시를 살펴보자

이곳에 도움말이 표시된다. 본문과 관련된 간단한 정보를 주로 표시하며, 간혹 중요한 내용을 언급하기도 한다.

사이드 바

예시에서는 짧게 표현하였지만, 실제 사이드 바는 본문의 내용과 관련된 주제에 대한 더 긴 내용을 설명하기 위해 주로 사용한다.

그림

때에 따라서는 (아무리 갖은 미사여구를 동원해도) 설명만으로는 충분하지 않을 수가 있다. 이럴 때는 그림을 제공한다. 모든 다이어그램은 마이크로소프트 비지오 2013으로 만들어졌으며, 색상 테마 없이 고대비로만 이루어져 설명에 집중할 수 있게 했다. 실행 화면은 고대비 테마를 적용하여 캡처하였다.

시스템 요구 사항

이 책의 예제를 실행해 보기 위해서는 다음의 하드웨어 및 소프트웨어 사양을 갖추는 것이 좋다.

- 윈도즈 XP 서비스 팩 3(스타터 에디션은 제외), 윈도즈 비스타 서비스 팩 2(스타터 에디션은 제외), 윈도즈 7, 윈도즈 서버 2003 서비스 팩 2, 윈도즈 서버 2003 R2, 윈도즈 서버 2008 서비스 팩 2 또는 윈도즈 서버 2008 R2
- 비주얼 스튜디오 2013의 모든 에디션(익스프레스 에디션을 사용하는 경우는 여러 제품을 다운로드해서 설치해야 할 수도 있다)
- 마이크로소프트 SQL 서버 2008 익스프레스 에디션 또는 그 상위 버전(2008 또는 R2 릴리즈) 및 SQL 서버 매니지먼트 스튜디오 2008 익스프레스 또는 그 상위 버전(비주얼 스튜디오에 포함되어 있으며, 익스프레스 에디션의 경우 별도로 다운로드해야 한다).

- 1.6GHz 또는 그 이상의 속도를 제공하는 프로세서가 장착된 컴퓨터(2GHz 이상 권장)
- 1GB(32비트) 또는 2GB(64비트)의 램(가상 머신에서 동작하거나 SQL 서버 익스프레스 에디션 혹은 그 상위 버전을 사용하는 경우는 512MB의 메모리가 더 요구된다).
- 3.5GB의 하드 디스크 여유 공간
- 5,400RPM 하드 디스크
- DirectX 9 및 1024 × 768 이상의 해상도를 지원하는 비디오 카드
- 소프트웨어 및 소스 코드를 다운로드하기 위한 인터넷 연결

독자들이 사용하는 윈도즈 운영체제의 설정에 따라 비주얼 스튜디오 2013과 SQL 서버 2008 제품을 설치할 때 로컬 관리자 권한이 필요할 수도 있다.

다운로드: 예제 코드

필자는 크기가 큰 예제를 구성하는 예제 코드들은 독립적인 애플리케이션으로 실행되는지는 물론 단위 테스트까지 모두 확인하려고 최대한 노력했다. 대부분의 단위 테스트는 크기가 작으며 MSTest를 토대로 작성했으므로 별도의 테스트 실행 도구는 필요하지 않지만, 보다 복잡한 단위 테스트는 NUnit을 이용했다. 모든 코드는 비주얼 스튜디오 2013 Ultimate 에디션에서 작성했다. 일부 코드는 프리뷰 버전에서 작성했지만, 이후 정식 버전에서 컴파일 및 테스트를 모두 마쳤다. 가능하면 비주얼 스튜디오 2003 익스프레스 에디션이 제공하지 않는 기능은 사용하지 않았지만, 일부 주제는 그렇게 할 수 없었다. 이런 코드를 실행하고자 한다면 유료 버전을 설치해야 한다.

코드 자체는 GitHub에 업로드되어 있으며, 아래의 URL을 통해 접근할 수 있다.

- 원서 - https://github.com/garymcleanhall/AdaptiveCode
- 번역서 - https://github.com/Jpub/AdaptiveCodeWithCSharp

부록 A는 Git의 사용법을 설명하며, 부록 B(온라인으로만 볼 수 있다)는 예제 코드의 구성을 자세히 설명한다.

필자에게 다른 코멘트를 남기고 싶다면 아래의 워드프레스 블로그를 방문해 주기 바란다.

http://garymcleanhall.wordpress.com

감사의 글

사실, 이 책은 마감일을 제대로 지키지 못했다. 지금부터 호명할 사람 중 단 한 명이라도 함께 하지 않았다면 필자는 이 책을 마칠 수 없었을 것이다. 모두가 각자의 방법으로 필자를 도와 주었다.

나의 아내, 빅토리아 — 이 책을 쓸 수 있게 해 주었다. 그냥 하는 소리가 아니라 진실이다.

나의 딸, 아밀리아 — 이 책을 미리 읽어 봐 주었으며, 나에게 어마어마한 응원의 메시지를 보내줬다.

아버지, 레스 — 늘 감사합니다.

나의 형, 다린 — 항상 옳은 방향을 안내해 주었다.

Online Training Solutions, Inc의 케시 크라우스 — 이 책을 멋지게 편집해 주었다.

데본 머스그레이브 — 끝까지 이 책을 믿고 기다려 주었다.

오탈자, 수정 및 사후 지원

저희는 이 책과 내용의 정확도를 최대한 높이기 위해 모든 노력을 기울였다. 이 책에 대한 (제보된 오탈자에 대한 목록 및 해당 변경 사항 등) 수정 사항은 아래 URL을 통해 확인할 수 있다.

- 원서 - http://aka.ms/Adaptive/errata
- 번역서 - http://www.jpub.kr의 이 책 소개 페이지

이 목록에서 확인하지 못한 오류를 발견하였다면 같은 페이지를 통해 알려 주기를 바란다.

베타리더 후기

제이펍은 책에 대한 애정과 기술에 대한 열정이 뜨거운 베타리더들로 하여금
출간되는 모든 서적에 사전 검증을 시행하고 있습니다.

강미희(휴인시스텍)

C#과 애자일을 들어만 봤지 실제 프로젝트나 책으로는 처음 접하는 거라 조금 어려운 감이 있었습니다. 그렇지만 마지막 파트의 가상의 사례를 통해 애자일을 실제로 어떻게 사용하는지 알 수 있었고, 또 그 사례를 통해 앞에서 본 내용을 이해할 수 있어서 좋았습니다.

김용균(이상한모임)

국외에서는 C#에 대한 열풍이 엄청나지만, 우리나라에서는 여전히 주류와 거리가 먼 언어라 그런지 국내에는 입문서 수준에서 벗어나는 책이 많지 않은 것 같습니다. 이 책은 그 갈증을 해결해 주는 도서입니다. 애자일부터 고수준의 디자인 패턴, 단위 테스트에 이르기까지 넓고 깊은 지침을 제공합니다. C#을 실무에서 깊이 있게 사용하고자 하는 사람에게 많은 도움이 될 중고급용 도서로서 C#을 사용한다면 꼭 읽기를 추천합니다.

김종욱(카이스트)

이 책은 다양한 기법을 통해 일반 사용자용 프로그램을 체계적으로 만드는 방법을 소개하고 있습니다. 다양한 예제와 사례를 들어가며 왜 이런 패턴과 저런 패턴을 사용해야 하는지, 개발자가 무엇을 피해야 하는지, 그리고 스타트업이나 다양한 팀 단위 일을 진행할 때 각자의 역할과 타협점을 어떻게 도출할지를 잘 서술한 책입니다. 베타리딩하면서 정말 좋았던 점은 친절한 예제를 통해 각각의 경우를 설명하며 독자의 이해를 돕고 있다는 것이었습니다. 기존의 도서들은 단순히 코드를 던져 주고 코드 따로 설명 따로 진행하는 경우가 태반이었는데, 이 책은 이러한 저의 편견을 과감히 무너트리는 새로운 시선을 제공해 주었습니다. 마지막으로, 집필뿐 아니라 번역도 책 전반에 걸쳐 부드러워서 좋았습니다.

김준환(소프트캠프)

소프트웨어의 변경은 소프트웨어의 생명 주기가 끝날 때까지 반복적으로 지속되는, 서로 떼려야 뗄 수 없는 관계인 것 같습니다. 이 책은 애자일 관련 기초 개념부터 SOLID 원칙, 가상의 프로젝트를 통한 활용 예시로 구성되어 있습니다. 변경에 유연하게 대처할 수 있는 소프트웨어를 디자인하고 구현할 때 좋은 참고가 될 수 있는 책이라고 생각합니다.

노승헌(아카마이 테크놀로지스 코리아)

유지보수를 고민하지 않고 난개발(?)된 코드에서의 사소한 변경은 다른 여러 코드에 나쁜 영향을 주곤 합니다. 이 책은 의존성 문제로 발생한 코드의 복잡성을 제거하는 여러 사례와 방법을 소개하고, 소모적인 코드 변경 공수를 줄이고 안전한 코드를 만들 수 있는 지침을 주고 있습니다. 컴퓨터 옆에 두고 필요할 때마다 찾아보고 읽으면 많은 도움이 될 것 같습니다.

석대진(프리랜서)

이 책에는 정말 많은 내용이 담겨 있습니다. 스크럼, 인터페이스, 디자인 패턴, 단위 테스트, 모의 객체 등등…. 저자의 풍부한 개발 경험은 이들을 유기적으로 연결하는 데 적잖은 도움이 된 것 같습니다. 특히, 대화 형식으로 구성된 10, 11, 12장을 읽으면 좋은 스프린트를 경험한 듯합니다(대화형식으로 된 이 스프린트는 한 편의 미드를 보는 것처럼 재미있었습니다). 국내에서 경험 많은 C# 개발자가 읽을 만한 책이 상당히 적은 편인데, 그들의 갈증을 채워 줄 소중한 책이라고 생각합니다.

한홍근(고려대학교 세종캠퍼스)

이번 책을 베타리딩하면서 다양한 개발 방법론을 서로 비교하며 익힐 수 있어서 좋았습니다. 특히, 가상 상황을 통한 대화체 설명이 기억에 남습니다. 팀 프로젝트를 진행할 때는 팀원들과의 소통이 무엇보다 중요하다고 생각하는데, 실제 프로젝트를 하지 않고도 다양한 상황을 예측해 볼 수 있었던 좋은 시간이었습니다. 팀 프로젝트 진행을 앞둔 분들께 권해 드리고 싶습니다. 책 번역도 매우 깔끔하여 번역서의 어색함을 전혀 느낄 수 없었습니다. 그리고 각 장의 끝에 있는 '마치며'라는 부분은 정말 마음에 들었습니다. 한 단원을 공부한 뒤 빠르게 요약하여 복습할 수 있었고 추가적인 설명으로 이해를 높이는 데 도움이 되었습니다.

PART

I

애자일의 기본기 갖추기

1부에서는 애자일(Agile)의 원칙과 사례에 대한 기초를 제공한다.

소프트웨어 개발에서의 핵심은 코드를 작성하는 행위이다. 그러나 실제로 동작하는 코드를 작성하는 방법은 수없이 다양하다. 아주 작은 기능이라 하더라도 이를 구현할 수 있는 다양한 방법이 존재한다는 사실은 얼마나 많은 플랫폼, 개발 언어, 그리고 프레임워크가 존재하는지 굳이 세어 보지 않더라도 알 수 있는 일이다.

소프트웨어 개발 산업은 항상 성공적인 소프트웨어 제품 개발을 목표로 해 왔다. 그러나 최근 들어 개발자들은 반복적이면서도 코드의 품질에 긍정적인 효과를 이끌어 낼 수 있는 패턴의 구현과 실무 경험을 더욱 강조하기 시작했다. 그 이유는 코드의 품질과 소프트웨어 제품의 품질을 더는 별개로 취급할 수 없기 때문이다. 낮은 품질의 코드는 시간이 흐를수록 제품의 품질을 저하시킨다. 꼭 그렇지는 않더라도 최소한 완전하게 동작하는 소프트웨어의 출시 일정을 지연시킨다.

높은 품질의 소프트웨어를 생산하기 위해서는 개발자들이 유지보수가 쉽고, 가독성이 높으며, 테스트를 고려한 코드를 작성하도록 노력해야 한다. 여기에 새로운 요구 사항이 출현했으니, **변화에 순응**할 수 있는 적응형 코드를 작성하자는 요구가 바로 그것이다.

1부의 내용은 최신 소프트웨어 개발 프로세스 및 그에 대한 사례를 제공한다. 이 프로세스와 사례들은 일반적으로 빠르게 방향을 변화시킬 수 있는 능력을 반영하는 **애자일(Agile)** 프로세스와 사례라고 불린다. **애자일 프로세스**는 소프트웨어 개발팀이 피드백을 빠르게 이끌어내고 피드백에 대응하는 데 초점을 맞추도록 변화하는 방법을 제안하며, **애자일 사례**는 소프트웨어 개발팀에게 방향의 전환을 가능하게 하는 코드 작성 방법을 제안한다.

PART I

An Agile foundation

CHAPTER

1

스크럼을 소개합니다

이 장의 주요 내용

- 프로젝트의 주요 인원에게 역할 할당하기
- 스크럼이 요구하고 생성하게 되는 여러 문서와 다른 산출물 구분하기
- 전체 개발 진도에서 스크럼 프로젝트의 진척도 측정하기
- 스크럼 프로젝트의 문제점을 분석하고 해결책 제시하기
- 최고의 효율을 위해 효과적인 방향으로 스크럼 회의 주재하기
- 애자일 및 다른 방법론들보다 스크럼을 채택하는 당위성 확보하기

스크럼(Scrum)은 프로젝트 관리 방법론(project management methodology)이다. 간략히 설명하면, 스크럼은 애자일 방법론의 하나이다. 스크럼은 반복적인 절차를 통해 소프트웨어 제품에 가치를 더하는 개념을 바탕으로 하고 있다. 전체적인 스크럼 프로세스는 소프트웨어 제품의 개발이 완료되었다고 판단되거나 프로세스가 중지되었다고 생각될 때까지 계속해서 반복된다. 이렇게 반복되는 과정을 **스프린트**(sprint)라고 하며, 스프린트가 종료되면 잠재적으로 출시가 가능한 소프트웨어가 완성된다. 모든 태스크는 **제품 백로그**(product backlog) 내에 우선순위가 결정되며, 각 스프린트를 시작할 때마다 개발팀이 모여 새로운 스프린트 기간에 완료해야 할 작업들을 선별하여 **스프린트 백로그**(sprint backlog)를 구성한다. 스크럼의 작업 단위는 **스토리**(story)이다. 제품 백로그는 대기 중인 스토리를 우선순위에 근거하여 나열하며, 각 스프린트는 해당 기간에 개발하게 될 스토리에 의해 정의된다. 그림 1-1은 전체적인 스크럼 프로세스를 보여 준다.

그림 1-1 **스크럼은 소프트웨어 제품의 작은 기능을 개발하는 생산 라인처럼 움직인다.**

스크럼은 문서 산출물, 개발팀 내/외 인력의 역할, 그리고 세레머니, 즉 관련된 부서의 인력들이 참여하는 회의로 이루어진다. 1장의 프로젝트 관리 규칙으로 스크럼이 제공하는 모든 것들을 살펴보기에는 충분하지 않지만, 이번 장을 통해 스프링보드(springboard)에 대한 기본적인 정보는 물론 스크럼의 일상적인 사례들에 대한 기본 개념을 충분히 제공할 것이다.

스크럼은 애자일이다

애자일(Agile)은 프로젝트가 진행 중이더라도 고객의 변경된 요구 사항을 수렴하기 위한 가벼운 소프트웨어 개발 방법론의 집합이다. 애자일은 보다 엄격한 구조의 방법론들로부터 얻은 실패를 바탕으로 발전된 방법론이다. 애자일 선언문은 애자일의 특징을 잘 설명하고 있다. 애자일 선언문은 www.agilemanifesto.org에서 찾아볼 수 있다.

애자일 선언문에는 17명의 개발자가 서명했다. 애자일 방법론은 그동안의 경험을 애자일 환경으로 확장해 나가면서 계속해서 발전해 왔으며, 이제는 소프트웨어 개발을 위한 기본 소양으로 자리잡고 있다. 스크럼은 애자일 프로세스를 구현한 가장 대중화된 방법 중 하나이다.

스크럼 vs. 폭포수

필자는 경험상 소프트웨어 개발의 방법으로서 애자일 접근법은 폭포수(waterfall) 방법론보다 낫다고 생각하며, 애자일 프로세스의 전도에 힘쓰고 있다. 폭포수 방식의 문제점은 너무 엄격하다는 점이다. 그림 1-2는 폭포수 방식을 채택한 프로젝트의 프로세스를 표현하고 있다.

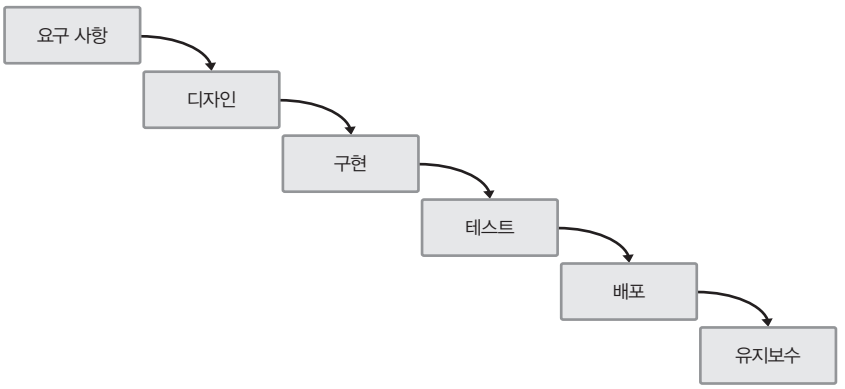

그림 1-2 **폭포수 개발 프로세스**

주목할 점은 한 단계의 산출물이 그 다음 단계에 투입된다는 점이다. 또한, 각 단계는 다음 단계로 이동하기 전에 반드시 완료되어야 한다. 이는 일단 한 단계가 완료되면 어떤 오류나 이슈, 문제점들이 발견되지 않는다는 것을 전제로 한 것이다. 그림의 화살표를 보면 오직 한 방향만을 향하고 있음을 알 수 있다.

또한, 폭포수 프로세스는 한 단계가 완료된 후에 그에 대한 어떤 변경도 발생하지 않는다는 것을 전제로 한다. 이는 경험과 통계에 근거한 여러 정황들과는 사뭇 달라 보이는 가정이 아닐 수 없다. 변화는 소프트웨어 엔지니어링에서만이 아니라 인간의 삶에서도 자연스러운 부분이다. 폭포수 방법론은 변화에 대해 많은 비용을 수반하고, 불필요하며, (가장 마음에 들지 않는 것은) **회피할 수 있는 개념**이라고 받아들이고 있다. 폭포수 방법론은 요구 사항의 정리와 디자인에 더 많은 시간을 할애하면 변화를 충분히 파악할 수 있다고 생각한다. 이런 개념은 **한마디로 말해서 말이 되지 않는다.** 변화란 **항상** 존재하기 때문이다.

애자일은 이러한 사실에 대해 완전히 다른 접근법을 채택하고 있다. 변화를 받아들이고, 변화가 발생하면 어느 누구라도 이를 수용할 수 있도록 허용한다. 애자일이 (스크럼 또한) 프로세스 수준에서 변화를 허용한다고는 하지만, **변화를 수용하기 위한 코딩**은 가장 어려운 일 중 하나이자 가장 중요한 일이며, 근대 소프트웨어 개발에서는 필수적인 개념이다. 이 책은 변화에 순응하여 살아남기 위한, 애자일스러우면서도 충분히 적응 가능한 코드를 만드는 방법을 설명하는 것을 목적으로 한다.

폭포수 방법론은 문서 중심적(document-centric)이기도 해서 엄청난 양의 문서를 만들어 내는데, 사실 이 문서들은 소프트웨어 제품의 질적 향상에 아무런 도움을 주지 못한다. 반면, 애자일은 실제로 동작하는 소프트웨어 자체를 가장 중요한 문서로 여긴다. 무엇보다 소프트웨어

의 동작은 (소스 코드를 설명한 문서가 아니라) 소스 코드를 읽어 보면 알 수 있다. 게다가, 문서란 소스 코드와는 별개의 것이기 때문에 문서의 내용이 소프트웨어 자체와 일치하지 않은 경우가 비일비재하다.

스크럼은 프로젝트의 진행과 전반적인 진행 상태에 대한 피드백을 제공하기 위한 몇 가지 기준을 사전에 정의하고 있으며, 이런 기준들은 제품에 대한 방대한 해설을 담은 문서와는 확연히 다르다. 일반적으로 애자일 환경에서는 최소한의 문서화는 허용하지만, 문서화를 강요하지는 않는다.

어떤 코드는 문서화를 지원할 때 이점을 제공하기도 하지만, 이런 문서도 여러 차례에 걸쳐 작성되었지만 아무도 읽지 않는 경우가 대부분이다. 이런 이유로 스크럼 팀은 대부분 위키(wiki)처럼 사용하기 쉬운 라이브(live) 문서 도구들을 활용한다.

이 장의 나머지 부분은 스크럼의 가장 중요한 관점을 보다 자세히 설명한다. 순수한 스크럼의 시각에서 바라본 내용은 아니지만, 스크럼으로부터 파생된 것들이라고 할 수 있다. 하나의 프로세스로서 스크럼의 목적은 소프트웨어 제품을 반복적으로 재정의하는 것뿐만 아니라 주어진 각자의 상황과 컨텍스트 내에서 프로세스가 잘 동작하고 있는지를 확인하기 위한 약간의 변화를 가미하는 것이다.

스크럼의 구성 요소에 대해 설명한 후에는 그 단점에 대해서도 설명한다. 이번 장은 앞으로 이 책에서 다룰 내용들, 즉 변화에 순응할 수 있으며 스크럼 프로세스에 어울리는 코드를 구현하기 위한 기초를 다지기 위한 것이다. 사실, 어떤 변화를 실제로 코드 수준에서 구현하는 것이 너무나도 어렵다면, 변화에 유연하게 대처할 수 있다고 주장할 만한 프로세스를 채택하는 것 자체가 무의미하다고 할 수 있다.

스크럼의 여러 형태

개발팀이 스크럼 방법론을 따른다고 주장하지만, 사실은 스크럼의 **변종**을 따르고 있는 경우가 많다. 순수한 스크럼은 익스트림 프로그래밍(XP, Extreme Programming) 같은 다른 애자일 방법론들이 가지는 일반적인 사례들을 그다지 많이 포함하고 있지는 않다. 원래의 정의로부터 파생된 스크럼은 다음과 같이 세 종류가 있다.

스크럼이며 또한…(Scrum and…)
단위 테스트를 먼저 작성하거나 두 명이 짝을 이루어 프로그래밍을 하는 등의 사례들은 사실 스크럼의 일부가 아니다. 그러나 이것들은 매우 유용하며, 여러 팀을 위한 프로세스에 가치를 더하기 때문에 스크럼을 보완하기 위한 것으로 받아들여지고 있다. XP나 칸반(Kanban)과 같은 다른 애자일 방법론들의

사례들이 더해지면, 이 프로세스는 "스크럼이며 또한…"의 형태가 된다. 즉, 스크럼에 몇 가지 좋은 사례들이 더해져 스크럼 프로세스를 (해치는 것이 아니라) 확장시키는 형태를 말한다.

스크럼이긴 하지만…(Scrum but…)

일부 개발팀은 스크럼을 표방하기는 하지만 중요한 관점을 놓치는 경우가 있다. 순서대로 백로그를 관리하며, 반복되는 스프린트를 통해 이를 처리하고, 스프린트 회고(retrospective)는 물론 스프린트 회의도 매일 진행한다. 그러나 스토리 점수를 예상하는 것이 아니라 실제 필요한 시간을 예상한다. 이렇게 의미가 퇴색된 버전을 "스크럼이기는 하지만…"이라고 표현한다. 많은 부분에서 스크럼을 표방하고 있기는 하지만 한두 가지 핵심을 놓치는 경우를 말한다.

스크럼이 아닌…(Scrum not…)

개발팀이 스크럼 방법론과는 꽤나 다른 방법을 채택하고 있다면 "스크럼이 아닌" 것으로 평가받을 수 있다. 이런 경우는 특히 팀 구성원들은 애자일 방법론을 기대하고 있지만, 실제 진행 중인 프로세스가 스크럼과는 전혀 닮아 있지 않을 때 문제를 유발하기 쉽다. 필자의 경험상 매일 진행하는 스프린트 회의는 쉽게 도입이 가능하지만, 변화에 대한 긍정적인 태도나 업무 수행에 대한 상대적인 스토리 점수 예측은 도입하기가 훨씬 어렵다. 스크럼의 상당 부분이 경시되거나 채택되어 있지 않다면 이 프로세스를 더는 스크럼이라고 할 수 없다.

역할과 책임

스크럼은 단지 프로세스일 뿐이며, (아무리 강조해도 부족하겠지만) 프로세스의 효율성은 해당 프로세스를 각자가 얼마나 따라주느냐에 따라 결정된다. 이 프로세스를 따르는 사람들은 각자의 행동을 안내하기 위한 역할과 책임을 갖는다.

제품 소유자

제품 소유자(Product Owner, 간혹 PO라고 부르기도 한다)의 역할은 매우 중요하다. 제품 관리자는 고객 또는 클라이언트와 나머지 개발팀을 연결하는 역할을 담당한다. 제품 소유자는 최종 제품에 대한 오너십(ownership)[1]을 가지며, 따라서 다음과 같은 책임도 수반하게 된다.

---

1 역주 여기서 오너십이란, 제품에 대한 소유권을 의미하는 것이 아니라 제품의 방향과 기능에 대한 의사결정을 하는 데 많은 권한과 책임을 가진다는 뜻이다.

- 구현할 기능을 결정하는 책임
- 비즈니스의 가치에 따라 기능의 우선순위를 결정하는 책임
- 태스크(task)의 '완료 여부'에 대한 판단을 내리는 책임

제품 소유자(PO)는 프로젝트 성공의 핵심 이해관계자로서 제품의 비전에 대해 명료하게 커뮤니케이션할 수 있어야 하며, 실제로도 팀과 그렇게 커뮤니케이션해야 한다. 프로젝트의 장기 목표는 개발팀에게 명확하게 공유되어야 하며, 중요한 변화는 시기적절하게 전파되어야 한다. 제품 소유자는 소프트웨어의 릴리스에 포함되어야 하는 기능들과 제품 백로그 내의 우선순위를 결정해야 한다.

제품 소유자의 역할이 아무리 중요하다고 해도 프로세스를 벗어나 무한한 영향력을 행사할 수는 없다. 제품 소유자는 팀이 한 스프린트를 진행하는 동안 얼마나 많은 커밋을 해야 할 것인지에 대해 어떠한 영향력도 행사할 수 없다. 이는 개발팀이 스스로의 역량을 고려해 판단할 사안이기 때문이다. 또한, 제품 소유자는 어떤 작업을 어떻게 진행할 것인지에 대해 언급할 수 없다. 특정 스토리를 기술적으로 어떻게 구현할 것인지 결정할 수 있는 것은 오직 개발팀뿐이다. 스프린트가 진행되는 동안에는 제품 소유자가 스프린트의 목표나 태스크의 수렴 조건을 변경한다거나 스토리를 더하거나 빼는 등의 행위를 해서는 안 된다. 목표가 설정되고 계획 과정에서 해당 스프린트에 스토리를 완료하기로 결정되었다면, 진행이 시작된 스프린트는 변경이 불가능하다. 즉, 스프린트나 프로젝트를 취소하고 완전히 다시 시작해야 하는 변경 사항이 아닌 이상, 모든 변경 사항은 다음 스프린트까지 기다려야 한다. 이렇게 함으로써 개발자는 흔들림 없이 해당 스프린트의 목표를 이루는 것에만 집중할 수 있게 된다.

스프린트가 진행되면서 개발팀이 스토리를 처리하고 완료하면, 제품 소유자는 해당 기능이 어떻게 동작하는지 검증을 요구하거나 진행 중인 태스크에 댓글을 작성할 수 있다. 스프린트가 진행되는 동안 예상하지 못했던 상황이나 혼동이 발생했을 때는 제품 소유자가 개발팀과 의사소통을 위해 시간을 할애하는 것이 중요하다. 이렇게 해야 '처리 완료'로 표시했던 스토리가 사실은 원래의 목적에서 벗어났음을 스프린트가 완료된 후에 뒤늦게 깨닫는 일이 발생하지 않는다. 그러나 제품 소유자는 해당 스토리가 필요한 조건을 만족하는지, 그리고 구현이 완료되어 스프린트 종료 시점에 데모를 할 수 있는 수준인지 아닌지를 판단할 수는 있다.

스크럼 마스터

스크럼 마스터(SM, Scrum Master)는 스프린트 기간에 외부의 간섭으로부터 팀을 보호하며, 팀이 매일 스크럼 회의를 진행하면서 표시해 둔 모든 방해 요소를 제거하는 역할을 담당한다. 이 역할은 팀이 완전히 스프린트 목표에만 집중함으로써 최상의 상태로 수행하며, 스프린트 기간 내내 생산성을 유지할 수 있도록 돕는다.

제품 소유자가 제품(즉, 결과물)에 대한 오너십을 갖는 것과 마찬가지로 스크럼 마스터는 프로세스, 즉 결과물을 어떻게 만들 것인지와 관련된 프레임워크에 대한 오너십을 갖는다. 따라서 팀이 프로세스를 따르도록 만드는 것은 오롯이 스크럼 마스터의 책임이다. 스크럼 마스터는 프로세스를 개선하기 위한 (4주 간격의 스프린트를 2주 간격으로 변경하자든가 하는 등의) 제안을 할 수는 있지만, 스크럼 마스터의 권한은 극히 제한적이다. 예를 들어, 스크럼 마스터는 팀이 스토리를 구현하는 과정이 스크럼 프로세스를 따르는지를 확인하는 수준에서만 간섭할 수 있을 뿐 기술적으로 간섭해서는 안 된다.

프로세스를 책임지는 사람으로서 스크럼 마스터는 매일 스크럼 회의를 주재한다. 스크럼 마스터는 팀이 회의에 참여할 수 있도록 독려하며, 해야 할 일이 처리되지 못하고 있지는 않은지 집중적으로 파악한다. 그러나 팀 입장에서는 스크럼 회의를 하는 동안 스크럼 마스터에게 업무 보고를 하지는 않는다. 그저 현재 프로세스에 참여한 **모든 사람**에게만 내용을 공유하면 된다.

개발팀

이상적인 상황이라면 애자일 팀은 **평준화된 전문가**(generalizing specialist)들로 구성된다. 이는 팀의 각 구성원은 여러 분야에 전문성을 확보하고 있어야 한다는 뜻이다. 즉, 여러 다른 기술을 효율적으로 운영할 수 있으면서도 특정 분야에 재능을 갖추고 있거나 특화된 전문가여야 한다는 말이다. 네 명의 개발자로 구성된 팀을 예로 들어 보자. 이들 각각은 ASP.NET MVC, Windows Workflow, 그리고 WCF(Windows Communication Foundation) 기술을 자유롭게 활용할 수 있는 인력들이다. 그러나 이 중 두 개발자는 Windows Forms 기술이 특기이며, 나머지 둘은 WPF(Windows Presentation Foundation)와 마이크로소프트 SQL 서버를 잘 다루는 사람들이다.

이처럼 팀이 다양한 기술을 갖춤으로써 애플리케이션의 일부에 대한 지식을 한 사람이 독점하는(즉, 각자의 역할을 '웹 개발자', '데이터베이스 담당자' 혹은 'WPF 개발자'처럼 분류하는) 상황을 방지할 수 있다. 한 가지 역할만을 강조하는 것은 프로젝트에 참여하는 모든 사람에게 좋지 않

다. 따라서 가능하다면 특정 업무가 한 사람에게 집중되는 현상을 방지하도록 해야 한다. 스크럼에서 코드는 팀 전체의 것이다. 따라서 특정 부분이 한 사람에게만 집중되는 것은 개발자로 하여금 특정 부분에만 가치를 제공할 수 있는 단일 리소스에 너무 의존하게 하므로 비즈니스 측면에서도 좋지 않다. 또한, 팀 구성원도 자신의 역할을 '내가 할 수 있는 것'에만 한정하게 되어 편협한 사고에 빠지기가 쉬워 개발자 자신에게도 좋지 않다.

소프트웨어 테스터는 소프트웨어가 개발되는 동안 그 품질을 관리하는 책임을 가진다. 테스터는 스토리가 시작되기 전에 해당 기능이 요구 사항을 만족하도록 구현되었는지를 검증하기 위해 테스트를 어떻게 자동화할 것인지를 논의한다. 또한, 그 테스트 계획을 구현하기 위해 개발자들과 협업을 하거나 직접 해당 테스트를 구현하기도 한다. 스토리가 구현되면 개발자는 해당 기능에 대한 테스트를 요청하며, 테스트 전문가는 해당 기능이 요구대로 동작하는지를 검증한다.

돼지와 닭

스크럼 프로세스에서 각각의 역할은 돼지 혹은 닭으로 분류할 수 있다. 이러한 분류는 지금부터 소개할 이야기와 연관이 있다. 어느 날 닭이 친구인 돼지에게 다가가 이렇게 말했다. "이봐, 돼지야. 내게 좋은 아이디어가 있어. 우리 함께 레스토랑을 개업하면 어때?" 이 말을 들은 돼지는 큰 관심을 보이며 이렇게 물었다. "그럼 레스토랑 이름을 뭐라고 하지?" 닭이 대답하기를, "'햄이랑 달걀이랑'으로 하면 어때?" 그러자 돼지는 잠깐 생각하더니 이렇게 소리쳤다. "말도 안 돼. 나는 내 몸의 일부를 팔아야 하는데, 넌 겨우 네가 낳은 달걀이나 팔겠다는 거야?"[2]

이 이야기는 프로젝트 내에서 특정 구성원의 참여도만을 중점적으로 비유한 것이다. 돼지는 프로젝트에 완전히 헌신하며 그 결과물에 대해 책임을 지는 반면, 닭은 오로지 프로젝트에 기여만 하며 보조적인 형태로 참여한다. 제품 소유자, 스크럼 마스터, 그리고 개발팀은 모두 제품의 출시에 직접 관련이 있기 때문에 모두 돼지라고 볼 수 있다. 대개의 경우 고객들은 프로젝트에 기여하므로 돼지보다는 닭의 입장이라고 생각하면 된다.

2 역주 여기서 돼지는 헌신자(committer)를, 닭은 기여자(contributor)를 표현한다. 돼지는 자신의 살을 직접 사용하기 때문에 프로젝트에 헌신하는 역할을, 닭은 자신의 살은 사용하지 않지만 달걀을 제공하기 때문에 프로젝트에 어느 정도 기여하는 역할을 의미한다.

산출물

소프트웨어 프로젝트의 주기를 진행하면서 많은 문서, 그래프, 다이어그램, 차트, 그리고 매트릭스 등을 만들고, 리뷰하며, 분석하고, 논의하게 된다. 이런 관점에서 스크럼 프로젝트는 다른 프로젝트와 별반 다를 바가 없다. 그러나 스크럼의 문서는 다른 종류의 프로젝트 관리 방법론에 의해 산출되는 문서와는 그 종류와 목적이 분명히 다르다. 모든 애자일 프로세스와 다른 프로세스들의 가장 큰 차이점은 문서의 상대적 중요도이다. 예를 들어, 구조화된 시스템 분석 및 디자인 방법론(SSADM, Structured System Analysis and Design Methodology)에서는 엄청난 양의 문서를 작성하는 것을 매우 중요하게 여긴다. 어떤 이들은 이를 가리켜 어마어마한 디자인 우선(BDUF, Big Design Up Front) 방식이라고 놀리기도 한다. 이는 모든 걱정과 불확실성, 그리고 의심은 프로젝트의 문서에 충분한 관심을 두면 제거할 수 있다는, 다소 잘못된 믿음을 바탕으로 한다. 애자일 프로세스는 프로젝트의 성공을 위해 반드시 필요한 것을 제외하고는 문서의 양을 줄이는 것을 목표로 하고 있다. 대신, 애자일에서는 코드로 문서를 대신하고 있다. 코드는 매우 신뢰할 수 있는 문서로서 언제든지 배포, 실행 및 활용이 가능하다. 또한, 문서 작성자의 입장에서는 거의 읽혀지지도 않을 문서를 작성하여 의사소통을 하는 것보다는 직접 만나서 하는 의사소통을 더 선호할 수 밖에 없다. 애자일 프로젝트에서도 문서는 중요한 부분을 차지하지만, 실제로 동작하는 소프트웨어나 구성원 간의 의사소통보다 더 중요하게 여기지는 않는다.

스크럼 보드

스크럼 프로젝트의 일상을 관리하는 중심은 스크럼 보드(Scrum board)이다. 스크럼 보드는 벽에 적당한 크기로 자리해야 한다. 만일 보드가 너무 작으면 중요한 내용을 자꾸 생략하고 싶은 유혹에 빠지기 쉽다. 어쩌면 독자의 사무실 벽은 그만한 공간을 마련하기가 쉽지 않을 수도 있다. 이런 경우에는 그냥 방치된 화이트보드를 스크럼 보드로 활용하면 된다. 자석을 이용하면 금속으로 된 서류함도 스크럼 보드로 사용할 수 있다. 임대 사무실이거나 벽이 조금이라도 손상되면 안 되는 경우에는 '매직' 화이트보드(그냥 닦아내면 내용이 지워지는 종이)를 활용해도 좋다. 일단, 사무실에서 이런 용도로 사용해도 좋은 공간을 찾아보자.

어떤 방법을 사용하든 스프린트를 두어 번 진행해 보면서 뭔가 올바르지 않다고 느껴진다면 얼마든지 변경해도 좋다. 물리적인 스크럼 보드는 반드시 필요하다. 비록 디지털 스크럼 도구들도 유용하게 활용할 수는 있지만, 필자는 이런 도구들은 물리적 스크럼 보드를 보조하는 용도로 사용하는 것이 좋다고 생각한다. 그림 1-3은 전형적인 스크럼 보드의 모습을 보여 준다.

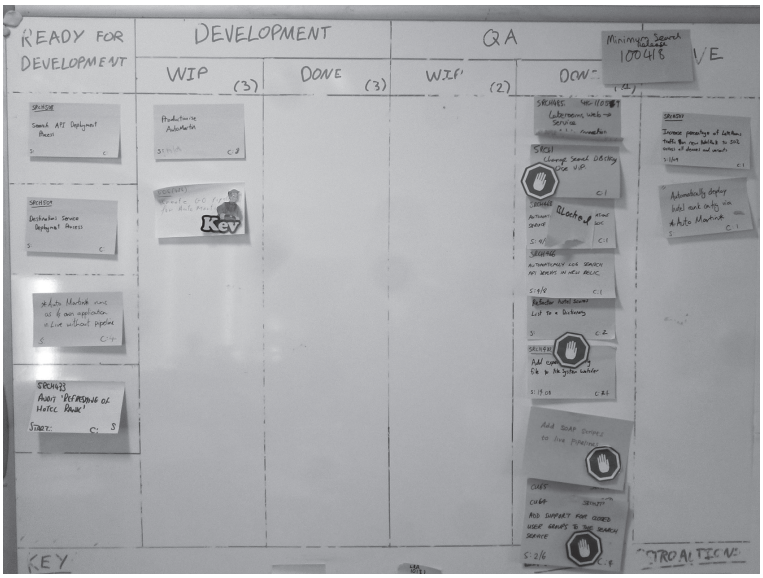

그림 1-3 **스크럼 보드는 현재 개발이 진행 중인 상태에 대한 스냅 샷이라고 할 수 있다.**

스크럼 보드는 정보의 보고이다. 상세한 내용들이 그곳에 기록되어 있으며, 어떤 경우에 위험이 발생할 것인지에 대한 통찰이 들어 있다. 이 절의 나머지 부분에서는 스크럼 보드를 구성하는 각각의 요소를 상세하게 설명하겠다.

⬟ 카드

스크럼 보드에서 가장 중요한 아이템은 카드(card)이다. 이 카드는 아주 작은 태스크로부터 물리적인 소프트웨어의 릴리스에 이르기까지 소프트웨어 제품의 진도를 나타내는 개별적인 요소를 표현한다. 각 카드의 종류는 주요 색상이나 채도로 표현된다. 스크럼 보드는 대체로 공간의 제약으로 인해 현재 스프린트와 연관된 스토리, 작업, 결함 및 기술 부채(technical debt)[3]만을 표현한다.

카드를 색상으로 구분하는 것만으로는 팀의 모든 구성원을 수렴하기에 충분치 않을 수 있다. 예를 들어, 색상을 올바르게 구별하지 못하는 팀 구성원을 위해 색상과 함께 독특한 모양을 함께 적용할 수도 있다.

3 역주 기술적으로 당장 해결이 어려운 일들을 계속 미뤄두는 현상. 기술 부채가 늘어난다는 것은 제품의 기능 자체는 문제가 없을지언정 전체적으로는 SOLID 원칙을 벗어나는 코드가 늘어난다는 것을 의미하며, 향후 리팩토링에 더 많은 시간과 노력을 투자하게 되거나 심한 경우는 아예 리팩토링이 불가능한 수준에 다다를 수도 있다.

구성의 계층구조 그림 1-4는 스크럼 보드 상의 카드들 사이의 상관 관계를 보여 준다. 주목할 점은 이 그림은 제품 자체가 여러 태스크의 조합으로 구성된다는 사실을 암시한다는 점이다. 아무리 복잡한 소프트웨어라 하더라도 결국에는 유한한 수의 개별 태스크로 나눌 수 있으며, 이들을 하나씩 완료함으로써 완성으로 가는 길을 닦아 나갈 수 있다.

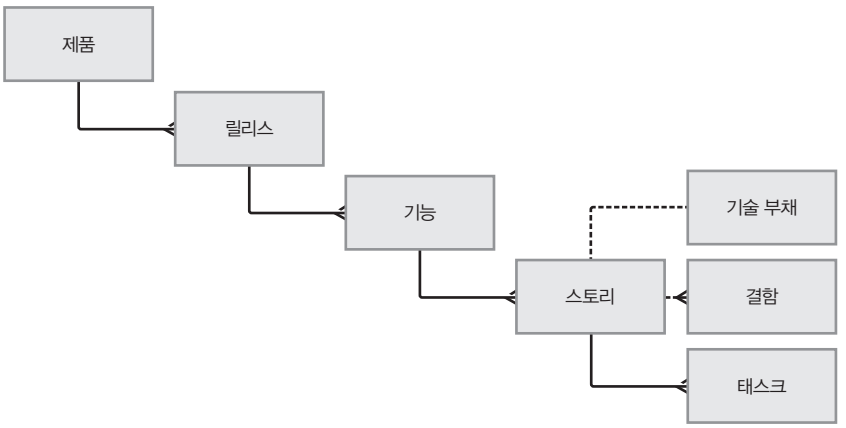

그림 1-4 **스크럼 보드 상의 카드는 제품을 구성하게 되는 각기 다른 부분을 표현한다.**

제품 스크럼 먹이 사슬의 최상위에는 구현해야 할 소프트웨어 제품(product)이 존재한다. 통합 개발 환경(IDE, Integrated Development Environments), 웹 애플리케이션, 회계 소프트웨어, 소셜 미디어 앱 등 제품의 예는 매우 다양하다. 독자들이 개발하는 것은 소프트웨어이며, 이것이 바로 출시하고자 하는 제품이다.

팀은 주로 한 번에 한 제품을 대상으로 작업을 수행하지만, 때로는 여러 개의 제품을 준비해야 할 때도 있다.

릴리스 독자들이 개발하는 모든 제품은 여러 번의 릴리스(release)를 거칠 수 있다. 릴리스는 최종 사용자가 구매하거나 서비스로서 사용하게 되는 소프트웨어의 버전을 말한다. 어떤 경우에는 결함을 수정한 릴리스를 출시하는 경우도 있지만, 핵심 고객에게 가치 있는 기능을 추가하기 위한 것이거나 또는 준비 중인 소프트웨어를 미리 살펴볼 수 있도록 베타 버전을 출시하는 경우도 릴리스라고 볼 수 있다.

웹 애플리케이션은 주로 앞서 릴리스된 제품을 대체하는 코드의 배포를 통해 암묵적으로 버전이 향상된다. 사실, 구글 크롬 브라우저가 매우 인상적인 예가 될 수 있다. 이 소프트웨어는 데스크톱 애플리케이션이기는 하지만, 다른 경쟁 브라우저와는 달리 작은 크기의 릴리스들이

아무런 표시도 없이 데스크톱에 끊임없이 배포된다. 인터넷 익스플로러 8, 9 및 10 버전은 텔레비전을 통해 광고도 했었지만, 크롬은 이런 패턴을 따르지 않는다. 구글은 단지 버전과는 무관하게 브라우저 자체만을 광고할 뿐이다. 이처럼 릴리스를 반복하는 것은 이제 일반적인 현상이 되어 가고 있다. 스크럼은 매 스프린트가 끝날 때마다 문제없이 동작하는 소프트웨어를 출시할 수 있도록 하는 것에 집중함으로써 이런 릴리스 패턴을 이끌어 낼 수 있다.

최소 존속 릴리스

첫 번째 릴리스는 **최소 존속 릴리스(MVR, Minimum Viable Release)**, 즉 기본적인 요구 사항을 수렴하기에 충분하다고 생각되는 기본적인 기능 구현에 부합한다고 볼 수 있다. 예를 들어, 회계 프로그램의 경우는 새로운 고객 정보를 생성하고, 계좌에서 발생하는 (입금 및 출금 모두에 대한) 트랜잭션을 처리할 수 있으며, 총액을 계산할 수 있는 기능이 이에 해당할 것이다. 이에 근간이 되는 개념은 프로젝트의 기반을 확보하여 가능한 빠른 시일 내에 스스로 자립할 수 있도록 만드는 것이다. 이런 바탕이 반드시 MVR의 결과로만 실현되는 것은 아니지만, MVR이 가져다 주는 희망은 제품의 개발이 진행되는 동안 어느 정도의 수익을 가져다 줄 수 있다는 점이다. 뿐만 아니라 어느 정도 제한된 수준의 고객에게만 공개한다 하더라도 제품의 첫 번째 배포를 통해 소프트웨어의 전체적인 방향을 결정지을 수 있는 중요한 피드백들을 얻을 수 있다. 이는 모든 소프트웨어는 변경의 대상이라는 점을 인지함으로써 계속해서 진화하는 소프트웨어 제품을 구현하려는 스크럼의 (그리고 애자일의) 본질이기도 하다.

기능 각 릴리스는 그전까지는 제공되지 않았던 하나 혹은 그 이상의 기능(feature)들로 구성된다. 어떤 소프트웨어이든지 버전 1.0과 2.0의 가장 큰 차이점은 신규 고객의 구매와 기존 고객의 업그레이드에 대한 관심을 끌기에 충분히 매력적이라고 판단되는 새로운 기능들의 구현 여부이다.

최소 시장성 기능(MMF, Minimum Marketable Feature)이라는 단어는 기능을 묘사하고 릴리스를 구성하는 데 도움이 된다. 아래의 목록은 다양한 프로젝트에 적용할 수 있는 일반화된 기능들의 예시들을 실세계에서의 구현에도 적합하도록 구체화한 것이다.

- 애플리케이션 데이터를 XML 기반의 이식 가능한 형식으로 내보내기
- 웹 페이지의 요청에 0.5초 내로 응답하기
- 향후에 참고하기 위해 과거의 데이터 기록하기
- 텍스트를 복사하고 붙여넣기
- 네트워크를 통해 동료와 파일 공유하기

이런 기능들은 고객에게 가치를 제공할 수 있다면 시장성이 있다고 본다. 기능들을 세부적으로 가장 작은 크기로 나누었음에도 여전히 가치를 제공할 수 있다면, 이 기능은 최소화되었다고도 볼 수 있다.

에픽/기능 vs. MMF vs. 테마

스크럼을 설명할 때 **기능(feature)**보다는 **에픽(epic)**이라는 단어에 더 익숙할 수도 있겠지만, 필자는 필요에 따라 두 단어를 적절히 섞어서 사용한다. 에픽과 기능은 '큰 스토리(large story)'라고 생각할 수 있다. 즉, MMF보다 훨씬 크며, 한 스프린트 기간에 마무리할 수 없을 만큼 큰 스토리를 말한다.

기능은 스크럼에서 일반적인 목표를 만족시키는 스토리의 집합을 의미하는 **테마(theme)**와 비슷하다.

기능은 각 릴리스마다 필수(required), 선호(preferred), 바람(desired) 등 크게 세 가지로 분류될 수 있다. 이들은 각 기능에 배정되는 전체적인 우선순위를 반영하는 상호배타적인 옵션이다. 주로 개발팀은 선호되는 기능을 구현하기에 앞서 필수적으로 요구되는 기능을 먼저 구현하며, 바람에 해당하는 기능들은 시간적 여유가 있을 때만 구현한다. 이미 예상했겠지만, 이 분류는 (그리고 각 분류에 해당하는 기능 자체는) 언제든지 변경이 가능하다. 기능의 구현 요청이 취소되거나, 우선순위가 재조정되거나, 요구 사항이 변경되거나, 혹은 다른 기능으로 대체되는 상황은 언제든지 발생할 수 있으며, 팀은 이러한 변화에 유연하게 대응해야 한다(조건에 따라 마감일과 비용 역시 변경되어야 한다). 스크럼에서는 무엇이든 변경될 수 있으며, 따라서 이 책은 독자들이 실세계에서 이러한 변화를 처리하는 데 도움을 주는 것을 목표로 하고 있다.

사용자 스토리 사용자 스토리(user story)는 스크럼의 산출물 중 가장 많은 사람에게 익숙한 개념이겠지만, 아이러니하게도 이는 스크럼에서 정의한 개념이 아니다. 사용자 스토리는 익스트림 프로그래밍(Extreme Programming)의 산출물이지만, 매우 대중적으로 사용되기 때문에 스크럼에서도 사용되는 것뿐이다. 사용자 스토리는 아래의 템플릿을 이용해 정의한다.

> "[사용자의 역할]로서, 나는 [동사 위주의 행위]를 함으로써 [사용자에게 제공할 가치]를 얻고자 한다"

대괄호는 사용자 스토리를 구분하기 위한 일종의 매개변수를 표시하기 위한 것이다. 실제 사례는 다음과 같다.

> "등록은 완료했지만 아직 인증을 받지 않은 사용자로서, 나는 비밀번호 재설정을 통해 비밀번호를 잃어버린 상황에서도 시스템에 로그인할 수 있어야 한다"

이 사용자 스토리에는 주목할 만한 점이 여러 가지가 있다. 첫째, **실제로 어떤 기능을 구현해야 하는지에 대한 충분한 상세 설명을 제공하지는 않는다는 점**이다. 사용자 스토리는 사용자의 관점에서 작성되어야 한다. 그렇게 해야 더 명확하게 요구 사항을 정의할 수 있다. 그러나 사용자 스토리는 너무 많이, 그리고 너무 빈번하게 개발자의 관점에서 작성되곤 한다. 템플릿의 첫 번째 부분인 '**[사용자의 역할]로서**'가 중요한 이유는 바로 이것이다. 마찬가지로, '**[사용자에게 제공할 가치]**' 부분 역시 중요하다. 이 부분이 존재하지 않는다면 사용자 스토리의 존재 가치를 이해할 수 없기 때문이다. 이 부분은 주로 사용자 스토리를 그 상위의 기능과 연결하는 부분이다. 예를 들어, 앞서 살펴본 예제의 경우는 '**분실한 사용자 인증 정보는 복구가 가능해야 한다**'라는 기능과 연결될 수 있다. 또한, 이 스토리는 사용자가 로그인 이름을 잊어버린 경우의 스토리와 사용자가 로그인 이름 및 비밀번호를 모두 잊어버린 경우의 스토리 등과 그룹화할 수 있다.

앞서 살펴본 사용자 스토리는 사실 개발을 시작하기에는 충분하지 않다. 대체 이 스토리를 통해 얻을 수 있는 가치란 무엇일까? 사용자 스토리는 개발팀과 고객 사이의 대화를 표현한다. 이 스토리를 구현할 시기가 되면 이 스토리를 담당하게 된 개발자들은 이 스토리를 사용자에게 제시하고 그들의 요구 사항을 토대로 대화를 하게 된다. 이 분석 기간을 통해 사용자 스토리에 대해 반드시 충족되어야 할 여러 조건들을 도출하고 구현함으로써 비로소 사용자 스토리를 완료된 것으로 처리할 수 있다.

요구 사항을 수집하고 나면 개발자들은 이 요구 사항들을 충족시킬 수 있는 디자인 아이디어들을 수집한다. 이 과정에서는 발사믹(Balsamiq)이나 마이크로소프트 비지오(Microsoft Visio) 등의 도구를 이용하여 사용자 인터페이스 목업(mockup)을 구성한다. 일부 기술적 디자인 컨셉은 주로 UML(Unified Modeling Language) 다이어그램을 통해 기존의 코드 기반이 어떻게 변경되어야 할 것인지에 대한 상세한 내용을 기술하기도 한다.

디자인에 대한 승인을 얻으면 팀은 사용자 스토리를 작은 태스크로 분리한 후 이 태스크들을 수행함으로써 스토리를 구현해 나간다. 스토리가 원하는 대로 동작할 수 있는 정도의 점수에 도달하면 구현한 기능들에 대한 테스트를 요청한다. 마지막 단계인 품질 보증(QA, Quality Assurance) 기간에는 수렴 가능한 조건(acceptance criteria)들에 대한 소프트웨어의 동작을 다시 한 번 점검하고 이에 대한 수용 여부를 결정한다. 구현된 기능들이 수용되면 스토리는 완료된다.

지금까지의 이야기를 다시 한 번 정리해 보자. 사용자 스토리가 마련되면 개발자는 분석 단계에서 요구 사항을 수집하고 디자인을 구상한 뒤, 실제로 동작하는 솔루션을 구현한다. 그런 후에 수렴 가능한 조건에 부합하는지를 테스트한다. 마치 폭포수 접근법과 동일한 것처럼 보이지

않는가! 사실, 그렇다. 이것이 바로 사용자 스토리의 본질이다. 즉, 전체 소프트웨어 개발 주기를 축소하여 각각의 스토리에 적용하는 것이다. 사용자 스토리는 개발자가 소프트웨어 제품과 관련이 있다고 확신을 하기 전까지는 스크럼 보드에서 개발 준비 단계로 이동하는 일이 없기 때문에 불필요한 노력을 방지하는 데 큰 도움이 된다.

사용자 스토리는 스크럼 방식으로 일을 진행하는 데 있어 가장 중요한 관점이다. 사용자 스토리는 스크럼이 팀 구성원들에게 줄 수 있는 인센티브(incentive), 즉 스토리 포인트를 가지고 있다. 팀은 스프린트를 계획하는 동안 사용자 스토리에 적절한 스토리 포인트를 할당한다. 사용자 스토리를 완료하는 것은 스토리 포인트를 획득하는 것이며, 이렇게 획득된 점수는 전체 스프린트 점수에서 뺀다. 스토리 포인트에 대해서는 이 장의 후반부에서 조금 더 자세히 설명하도록 하겠다.

태스크 사용자 스토리보다 더 작은 작업 단위가 바로 태스크(task)이다. 스토리는 여러 개의 관리 가능한 태스크로 분리되어 스토리에 배정된 개발자들에게 할당된다. 필자는 스토리를 태스크로 나누기 직전에 보드에서 떼어 내는 것을 선호하지만, 경우에 따라서는 스프린트 계획 단계에서 스토리를 태스크로 나누는 것을 본 적도 있다.

사용자 스토리는 특정 기능을 구현하기 위해 수직적으로 분할되어야 하지만, 태스크는 팀 내 개발자들의 특성을 활용할 수 있도록 계층 수준으로 분리될 수도 있다. 예를 들어, 기존의 양식에 새로운 필드를 추가해야 한다면 이 작업을 위해 사용자 인터페이스와 비즈니스 로직, 그리고 데이터 액세스 계층을 변경해야 할 수 있다. 이런 경우는 스토리를 세 개의 계층을 대상으로 하는 세 개의 태스크로 분리하여 각각의 태스크를 WPF 개발자, C# 전문가, 그리고 데이터베이스 전문가 등 각 분야의 전문가에게 맡길 수 있다. 물론, 운이 좋아서 전체 구성원이 전체적으로 상향 평준화된 팀을 가지고 있다면 각 태스크를 **지원자**에게 맡길 수도 있다. 이렇게 함으로써 모든 구성원이 코드의 여러 부분에서 작업을 수행할 수 있게 되고, 이를 통해 전체적인 이해도를 향상시켜 결과적으로 자신들의 업무에 대한 만족도를 높일 수 있게 된다.

수직적 분할

필자가 어렸을 때 필자의 아버지는 크리스마스 때마다 트라이플(triple)을 만드셨다. 트라이플은 여러 겹으로 된 영국식 전통 디저트인데, 가장 아랫부분에는 잘게 썬 과일이 있고, 그 위에 스펀지 케이크, 젤리, 그리고 카스타드를 얹은 후 맨 위에 휘핑 크림을 듬뿍 발라 만들었다. 내 동생은 숟가락을 이용해서 여러 겹의 내용물을 깊게 파서 먹었고, 나는 주로 한 겹씩 차례대로 먹곤 했다.

잘 디자인된 소프트웨어 역시 트라이플처럼 여러 계층으로 구성된다. 가장 아랫부분에는 데이터 액세스 계층이 존재하며, 그 위로 객체 관계 매퍼(ORM, Object-Relational Mappers), 도메인 모델, 서비스 그리고 컨트롤러 등의 계층이 있으며, 가장 위쪽에는 사용자 인터페이스가 존재한다. 트라이플을 먹는 것과 마찬가지로, 계층화된 애플리케이션을 여러 부분으로 나누는 방법 역시 두 가지가 존재한다. 바로 수직적 분할(vertical slice)과 수평적 분할(horizontal slice)이 그것이다.

수평적 분할의 경우, 개발자는 각 계층을 맡아 해당 계층 전체에 요구되는 기능들을 구현한다. 그러나 각 계층의 완료 시점을 동일하게 보장할 수 없다. 예를 들어, 아직 구현이 완료되지 않은 하위 계층의 기능이 사용자 인터페이스를 통해 노출될 가능성이 존재한다. 결과적으로, 클라이언트는 각 계층의 상당 부분의 구현이 완료되기 전까지는 애플리케이션을 사용할 수 없다. 이는 애자일 방법론을 도입해 얻을 수 있는 주요 피드백 루프의 지연을 유발하며, 필요 이상의 것이나 잘못된 것을 개발하게 될 가능성이 늘어나게 된다.

수직적 분할이야말로 우리가 목표로 해야 할 방법이다. 각 사용자 스토리는 각각의 계층에 필요한 모든 기능을 포함해야 하며, 최상위 계층인 사용자 인터페이스 계층과 연동되어야 한다. 이 방법을 이용하면 사용자에게 해당 기능을 제공하고 그에 대한 피드백을 빠르게 전달받을 수 있다. 또한, '나는 이번 달에 비용을 납입하지 않은 고객을 데이터베이스에서 조회하고자 한다'와 같이 개발자 중심의 사용자 스토리를 정의하는 경우를 피할 수 있다. 사실, 이 스토리는 마치 태스크처럼 보인다. 주요 미납 고객에 대한 보고서를 생성하는 기능에 대한 것이 더 스토리답다.

사용자 스토리는 스토리 포인트를 가지고 있으며, 이 포인트는 스토리를 구성하는 태스크들로 전이되지 않는다는 점을 인지해야 한다. 즉, 5포인트짜리 스토리를 세 개의 태스크로 분할한다고 해서 1포인트짜리 태스크 두 개와 3포인트짜리 태스크 한 개로 구성되지는 않는다. 그 이유는 부분적으로 완료된 작업에는 어떤 인센티브나 보수도 없기 때문이다. 스토리의 구현이 (하나의 완결된 기능으로서) 완료되었다는 것을 스프린트가 종료되기 전에 QA 프로세스를 통해 증명하지 못하면 해당 스토리가 보유하고 있던 포인트는 전혀 적용되지 않는다. 스토리는 다음 스프린트에서도 계속해서 작업 중인 상태로 남아 있게 되며, 이상적으로는 다음 스프린트 기간 중 비교적 이른 시점에 완료될 것이다. 스토리가 완료되기까지 너무 오랜 기간이 소요되어 오랫동안(즉, 전체 스프린트 기간보다 오랫동안) 진행 중인 상태로 남아 있게 되면, 이는 애당초 너무 큰 작업이었으므로 더 작고 관리 가능한 스토리로 분리되었어야 한다는 것을 의미한다.

기술 부채 기술 부채(technical debt)는 상당히 흥미로운 개념이지만 잘못 이해하고 있는 경우가 허다하다. 기술 부채란, 스토리가 스크럼 보드를 통해 진행되는 동안 만들어지는 디자인과 아키텍처의 타협을 의미하는 단어이다. 기술 부채는 이 장의 후반에서 더 자세히 설명하기로 하자.

결함 결함(defect)을 표현하는 카드는 이미 완료된 사용자 스토리가 수렴 가능한 조건을 만족하지 못할 때마다 생성한다. 이 카드는 자동화된 테스트 환경을 필요로 한다. 특정 스토리를 테스트하기 위해 작성하는 테스트들은 향후 진행될 작업들이 현재의 스토리에 문제를 유발하는 변경 사항을 만들어 내지 못하도록 하기 위한, 일련의 테스트 세트 형태로 작성된다.

기술 부채와 마찬가지로, 결함 카드는 스토리 포인트를 가지지 않기 때문에 결함과 기술 부채 카드를 생성하려면 인센티브를 줄여야 한다. 결함과 기술 부채를 완전히 없애는 것은 사실상 불가능하지만, 개발자들은 어떻게 해서라도 인센티브가 줄어드는 일은 피하고 싶을 것이다.

모든 소프트웨어에는 결함이 존재한다. 이는 소프트웨어 개발에 있어서 단순명료한 사실이며, 어떠한 계획이나 노력도 인간의 실수를 방지할 수는 없다. 결함은 크게 A, B 혹은 C 등 세 가지로 분류할 수 있다. A는 **심각한 결함(apocalyptic defects)**을 의미하며, B는 **행위 오류(behavioral errors)**를, 그리고 C는 **표현 이슈(cosmetic issues)**를 뜻한다.

심각한 결함은 애플리케이션이 완전하게 실패하거나 혹은 사용자가 작업을 계속할 수 없도록 하는 결함을 말한다. 이에 대한 전통적인 예시는 처리되지 않은 예외(uncaught exception)인데, 이런 예외가 발생하게 되면 프로그램을 반드시 종료하고 다시 실행해야 하거나 (웹 환경인 경우에는) 웹 페이지를 반드시 다시 로드해야 하기 때문이다. 이런 결함은 높은 우선순위를 가지고 소프트웨어를 릴리스하기 전에 반드시 수정해야 한다.

행위 오류는 아주 심각하지는 않지만 사용자의 짜증을 유발할 수 있다. 이런 종류의 오류는 단지 애플리케이션의 충돌을 유발하는 것 이상의 피해를 줄 수 있다. 예를 들어, 반올림 계산을 엉뚱하게 하는 잘못된 환율 계산 로직을 생각해 보자. 이 알고리즘이 고객이나 비즈니스에 적용되면 누군가는 돈을 잃게 된다. 물론, 모든 로직상의 오류가 이처럼 심각하지는 않지만, 이런 문제에 중간 우선순위나 높은 우선순위를 주로 할당하는 이유는 쉽게 이해할 수 있다.

표현 이슈는 주로 사용자 인터페이스에서 발생하는 문제이다. 이미지 정렬이 잘못 되었다거나, 창이 전체 화면으로 전환되지 않는다거나, 웹의 이미지가 로드되지 않는 등의 문제들을 말한다. 이런 이슈들은 소프트웨어의 사용 자체에는 영향을 미치지 않는다. 단지 외관상의 문제일 뿐이다. 대게 이런 이슈들에는 낮은 우선순위를 부여하지만, 소프트웨어의 겉모습 또한 사용자의 기대치에 유리하게 작용한다는 점을 잊어서는 안 된다. 만일 사용자 인터페이스의 버튼이 형편 없는 디자인에 제대로 동작조차 하지 않고 이미지 또한 로드되지 않는다면, 사용자 입장에서는 해당 소프트웨어의 내부 동작 역시 신뢰할 수 없을 것이다. 반대로, 멋진 사용자

인터페이스에 여러 부가 기능도 잘 갖춰져 있다면 사용자로 하여금 이 소프트웨어는 내부적으로도 잘 디자인되어 있을 것 같다는 느낌을 줄 수 있다. 평판이 좋지 않은 프로젝트에 통상적으로 적용되는 편법이 바로 사용자 인터페이스를 새로 디자인하여 (심한 경우에는 전체 제품을 리브랜딩(rebranding) 해서라도) 제품에 대한 인식을 개선하고 기대치를 재설정하는 방법이다.

카드의 모양

스크럼 보드의 카드를 사용자 정의 및 개인화하는 방법은 수도 없이 많다.

색상 스키마

카드에는 어떤 색상을 사용해도 무방하지만, 필자의 경험상 몇 가지 색상만을 사용하는 것이 가장 잘 어울린다. 인덱스 카드는 기능 및 사용자 스토리에 이상적이며, 태스크, 결함 및 기술 부채 카드 등은 관련된 스토리에 부착되어야 하기 때문에 접착식 메모지[4]를 사용하는 것이 어울린다. 필자가 추천하는 색상 스키마는 다음과 같다.

- 기능: 녹색 인덱스 카드
- 사용자 스토리: 흰색 인덱스 카드
- 태스크: 노란색 접착 메모지
- 결함: 빨간색/분홍색 접착 메모지
- 기술 부채: 보라색/파란색 접착 메모지

아마도 사용자 스토리와 태스크가 가장 빈번하게 사용되는 종류의 카드일 것이므로 가장 구하기 쉬운 인덱스 카드와 접착 메모지를 사용하면 좋다. 마지막으로 대비해야 할 것은 인덱스 카드를 다 써버리는 상황이므로 가장 쉽게 구할 수 있는 색상을 사용하도록 하자.

누가 카드를 생성할까?

'누가 카드를 생성할까?'라는 질문에 대한 가장 간단한 답변은 '누구든지'이다. 물론, 이 답변에는 약간의 조건이 있다. 누구든 카드를 생성할 수는 있지만, 그 카드의 유효성, 우선순위, 심각도(criticality) 및 기타 다른 사항들은 어느 한 사람이 혼자서 결정할 수 있는 것은 아니다. 모든 기능과 스토리 카드는 제품 소유자의 확인을 거쳐야 하지만, 태스크, 결함 및 기술 부채 카드는 오로지 개발팀이 전담하는 영역이다.

아바타

온라인 상의 포럼과 트위터 등에서 흔히 볼 수 있는 아바타와 마찬가지로, 팀의 다양한 구성원을 표현하기 위한 수단이 필요하다. 팀 구성원 스스로를 표현할 수 있는 아바타는 스크럼 프로세스가 진행되는 동안 하나의 재미 요소가 될 수 있다. 물론, 공격적인 의미를 담고 있는 것은 피해야 하며, 나아가 이를 통해 개개인을 식별할 수 있는 것을 사용해야 한다.

4 역주 포스트잇 또는 유사한 제품

작업 주기(iteration)를 진행하는 동안 이 아바타들은 여기저기로 옮겨 다니며 하루를 기준으로 움직인다. 이미 보드 상에는 인덱스 카드 스토리와 접착 메모지 태스크를 가지고 있기 때문에 아바타는 2인치 정사각형 크기를 넘어설 수는 없다. 아바타를 겹쳐 놓으면 스크럼 보드가 복잡해지는 것을 방지할 수 있으며, 재사용 가능한 끈끈이 테이프나 작은 테이프 조각을 이용해 적당한 곳에 붙이면 된다.

⬟ 스윔레인

스크럼 보드에는 스윔레인(swimlane)을 구분하기 위해 수직으로 선이 그어져 있다. 각각의 스윔레인은 개발 주기에 걸쳐 진행 중인 스토리의 상태를 표시하기 위해 여러 개의 사용자 스토리 카드를 올려놓을 수 있다. 카드는 왼쪽에서 오른쪽 방향으로 이동하며, 기본적인 스윔레인은 백로그(Backlog), 진행 중(In Progress), QA 및 완료(Done)로 구성된다.

백로그 레인(lane)의 스토리는 특정 스프린트에 진행하는 것으로 '승인된' 것이어야 하며, (취소되기 전까지는) 스토리 보드에서 떼어 내고 작업을 시작해야 한다. 이 레인에 올려진 카드들은 우선순위에 따라 정렬하여 가장 위의 아이템을 항상 먼저 구현해야 한다.

스토리를 백로그로부터 떼어 내고 제품 소유자와 함께 기능의 범위와 요구 사항들에 대한 대화를 완료하면, 해당 카드는 다시 (산출된 태스크들과 함께) 스크럼 보드의 진행 중 레인으로 돌아온다. 이 시점에는 스토리에 참여하는 팀 구성원 모두의 아바타가 함께 붙어 있어야 한다. 그런 후, 스토리는 진행 중 기간에 관련된 스윔레인의 카드 개수 제한에 따라 하나씩 진행하게 된다. 예를 들어, 동시에 최대 세 개의 사용자 스토리만을 진행하도록 규칙을 정했다면, 팀이 아직 시작되지 않은 스토리를 시작하기에 앞서 이미 시작된 스토리를 완료하도록 강제할 수 있다. 부분적으로 완료된 작업에는 아무런 인센티브가 없음을 다시 한 번 기억하자.

스토리를 위한 분석과 디자인, 그리고 구현이 모두 완료되면 '개발자 완료(developer complete)' 상태가 되어 품질 보증(QA, Quality Assurance) 스윔레인으로 옮겨갈 수 있는 상태가 된다. 이상적인 경우의 QA 환경은 실제 서비스 환경을 최대한 반영하여 배포 시 발생할 수 있는 작은 차이점으로 인한 환경적 오류를 최소화할 수 있어야 한다. 테스트 분석가는 수렴 가능한 조건들을 바탕으로 스토리를 평가한다. 기본적으로, 테스트 분석가들은 스토리의 정의를 벗어난 행위를 통해 코드가 하지 말아야 할 행위를 수행하는지 여부를 검증한다. 주로 특정 작업에 일반적이지 않거나 오류의 가능성이 있는 데이터를 입력하여 유효성 검사가 올바르게 동작하는지 여부를 파악한다. 또는 보안상의 허점을 이용하여 악의적인 사용자들이 특별한 권한을 얻게 되는 경우가 있지는 않은지 확인하기도 한다. 모든 것이 완벽하게 완료되면 사용자 스토리

는 완료(Done) 스윔레인으로 이동한다. 스토리에 할당된 모든 스토리 포인트를 획득하게 되고, 스프린트 번다운(burndown) 차트(스프린트의 진척도를 보여 주는 그래프)가 수정된다. 이런 산출물들에 대해서는 나중에 더 자세히 설명할 기회를 갖도록 하겠다.

수평 스윔레인 스크럼 보드는 수평 스윔레인(horizontal swimlane)을 통해 구분될 수도 있다. 이 스윔레인을 통해 스토리를 기능 단위로 그룹화하여 모든 사람이 어느 지점에 팀의 역량이 집중되고 있는지, 그리고 나아가 어느 지점에 해소해야 할 병목 현상이 발생하는지를 한눈에 파악할 수 있도록 할 수 있다.

스크럼 보드의 가장 위쪽에는 서둘러야 할 일(Fast-Track) 레인이라는 특별한 레인이 존재하는데, 이 레인에는 매우 높은 우선순위를 갖는 태스크들이 나열된다. 팀 구성원들은 서둘러야 할 일에 나열된 아이템에 대해서는 다른 중요한 업무에 대해 약간의 손실을 감수하더라도 다같이 협력하여 최대한 빨리 마무리할 수 있도록 교육되어 있어야 한다. 이 경우, 현재 팀이 협업하고 있던 문제나 태스크의 우선순위를 조정하고 잠시 중단하게 된다. 이 방식은 유용하긴 하지만, 이와 같은 우선순위 조정이 필요한 일이 발생했을 때에 한해 선별적으로 적용되어야 한다. 실제 서비스 환경에서 발견된 심각한 결함은 대부분 서둘러야 할 일로 분류할 수 있다.

⬟ 기술 부채

기술 부채(technical debt)라는 단어는 조금 더 설명이 필요하다. 사용자 스토리를 구현하는 과정에서는 언제든지 '이상적인 코드'와 '마감일을 맞추기에는 충분한 수준의 코드' 사이의 타협이 발생하게 마련이다. 마감일을 맞추기 위해 형편없는 디자인을 의도적으로 허용해야 한다고(또는 적극적으로 권장해야 한다고) 말하는 것은 아니지만, 향후의 개선점을 염두에 두고 지금 당장은 간단하게 해결하는 방법 역시 가치 있는 방법이다.

좋은 기술 부채와 나쁜 기술 부채 부채는 프로젝트 기간에 서서히 생겨난다. 이 개념을 우리가 어떻게 바라봐야 하는지를 잘 암시하는 단어는 바로 부채(debt)라는 단어이다. 사실, 어떤 종류의 재정적 부채를 지고 있다고 해서 크게 잘못된 것은 없다. 예를 들어, 차를 구입하면서 12개월 할부를 적용했는데 이자를 지불하지 않아도 된다면, 실제로는 부채를 지게 된 것이지만 출퇴근을 위해 차가 필요한데 일시불로 지불할 여력이 없을 경우 매우 좋은 선택이 될 수 있다. 또한, 차를 이용하여 직장에 정시에 출근하게 됨으로써 나머지 비용을 지불할 수 있는 수입을 창출할 수도 있다.

물론, 어떤 부채는 그다지 좋지 않다. 신용카드로 얼마나 많은 부채를 부담해야 할지 계산도 하지 않고 사치스런 물건을 사다 보면, 결국 이자가 싼 대부업체를 찾느라 전전긍긍하게 될 것이다. 나중에 돌이켜 보면 올바르지 못한 경제적 의사결정은 바로 이 좋지 못한 부채에서 기인한 것이 많다. 중요한 것은 선택 가능한 옵션들을 신중히 검토하여 지금 이 부채를 짊어질 만한 가치가 있는지, 아니면 지금 당장 전체 비용을 지불하는 것이 좋을지를 결정하는 것이다.

소프트웨어에서도 동일한 트레이드 오프(trade-off)가 존재한다. 지금 당장 차선책을 구현하여 일정을 맞추거나 추가 근무 시간을 투입하여 디자인을 향상시킬 수 있지만, 아마도 이 경우에는 마감일을 지키지 못할 것이다. 모든 상황에 적합한 정답은 없다. 그러나 좋은 기술 부채와 나쁜 기술 부채를 구분할 수 있는 가이드라인은 존재한다.

기술 부채의 사분면 걸출한 애자일 에반젤리스트인 마틴 파울러(Martin Fowler)는 스토리를 완료하기 위해 필요한 양보와 타협을 기반으로 기술 부채 사분면(quadrant)을 정의하고 있다. 우선 x, y 두 개의 축을 이용하여 평면을 사분할하는데, 이 두 축에는 각각 '이 기술 부채가 합당한 이유를 통해 생성된 것인가?'와 '이 기술 부채를 회피할 대안을 알고 있는가?'라는 두 가지 질문이 연결되어 있다. 첫 번째 질문에 대한 대답이 '예'라면 기술 부채를 신중하게 결정한 것이라고 볼 수 있다. 즉, 이 기술 부채를 감수하는 합당한 이유가 있으며 부끄럼 없는 선택을 한 것이다. 만일 대답이 '아니오'라면 이 부채는 무모한 것이며, 따라서 계속 쌓아 두는 것보다는 지금 당장 처리하는 편이 더 낫다.

두 번째 질문에 긍정적인 답이 가능하다면 대안을 염두에 둔 상태에서 부채를 받아들인 것이라고 볼 수 있다. 그러나 부정적인 답을 할 수밖에 없다면 다른 대안은 존재하지 않는다는 것을 의미한다.

이 질문들의 결과에 따라 선택 가능한 시나리오는 그림 1-5에서 설명하는 네 가지 시나리오이다.

- **무분별함, 의도적:** 이런 종류의 부채는 가장 위험한 것이다. 이런 부채가 있다는 것은 '우리는 디자인에 투자할 시간 따위는 없다'고 말하는 것과 같으며, 근무 환경이 매우 열악하다는 것을 의미한다. 이런 결정이 이루어졌다는 사실은 이 팀은 유연하지 않으며 피할 수 없는 실패를 향해 꾸준히 나아가고 있다는 경고나 다름없다.
- **무분별함, 부주의함:** 이런 종류의 부채는 경험의 부족으로 인해 생겨난 것이 대부분이다. 즉, 근대 소프트웨어 엔지니어링에 대한 사례에 대해 제대로 알지 못한 결과일 뿐이다. 주로 코드가 너저분하고 이전에도 유사한 경우가 있었지만, 개발자가 더 나은 방법을

알지 못해서 그 외에 다른 선택지를 찾지 못하는 경우이다. 이런 경우에는 교육이 답이다. 개발자들이 배울 의지만 있다면 이런 종류의 기술 부채는 더는 생겨나지 않을 것이다.

- **분별 있음, 부주의함:** 이 경우는 기존의 사례를 충실히 따랐지만 그보다 더 나은 방법이 존재한다는 사실이 밝혀져 '이제는 이 작업을 어떻게 처리해야 하는지 알게 된' 경우이다. 바로 앞서 설명한 경우와 유사해 보이지만, 이 경우에는 의사결정 당시 모든 개발자가 현재 채택한 방식이 이 문제를 해결하기 위한 최선이라고 동의했다는 점이 다르다.
- **분별 있음, 의도적:** 가장 무난한 형태의 부채라고 할 수 있다. 가능한 선택 사항에 대해 모두 고민한 후에 이 부채를 짊어질 것을 결정하기 때문에 정확히 어떤 일을 하려는 것인지 (그리고 왜 하려는 것인지) 충분히 인지하고 있는 경우이다. 이런 부채는 주로 '우선 출시 후 처리하는' 형태로 최종 의사결정이 내려진 경우에 발생한다.

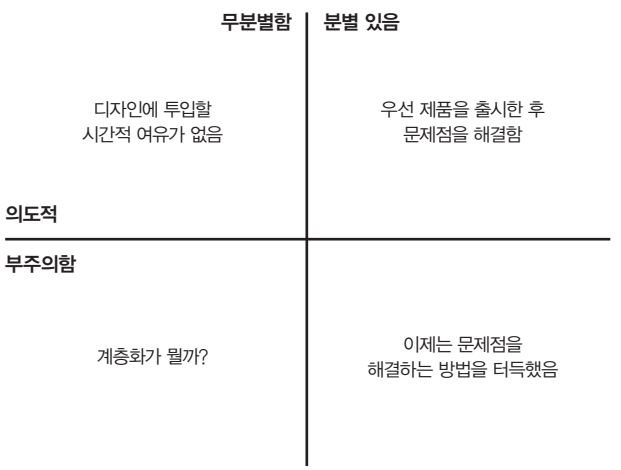

그림 1-5 **마틴 파울러의 설명에 따르면, 기술 부채 사분면은 개발자들이 네 가지 종류의 기술 부채를 시각화하는 데 도움이 된다.**

부채 상환하기 기술 부채는 스토리 포인트와는 무관하지만, 직접적인 인센티브가 없다 하더라도 반드시 상환해야 한다. 가장 좋은 방법은 기술 부채 카드를 스토리에 부착하고 코드를 리팩토링하여 새로운 동작과 관련해 새로운 디자인을 구현하는 것이다. 다음번에 스토리를 보드에서 떼어 낼 때 코드 중에 기술 부채와 연관되어 수정될 코드가 있는지 확인하고, 관련된 기술 부채가 존재한다면 두 가지를 모두 해결하려고 노력해야 한다.

⬟ 디지털 스크럼 보드

디지털 스크럼 보드는 벽에 계속해서 매달아 두지 않으면 프로젝트에 대한 매우 중요한 정보를 보여 주지 못한다. 이 정보를 공개하고 회사 전체가 볼 수 있도록 표시하지 않으면 계속해서 전체 진척 현황에 대한 질문을 받게 될 것이다. 진척 현황을 투명하게 공개하면, 특히 회사에 스크럼을 최초로 도입할 때 큰 이점을 얻게 된다. 업무가 잘 진행되고 있음을 공개함으로써 이해관계자들로부터 신뢰를 얻을 수 있다.

진부한 말이기는 하지만, 사람이라면 누구나 변화를 두려워한다. 두려움은 불확실한 것에 대한 자연스러운 반응이다. 사람들에게 여러분이 무엇을 하고 있는지, 그리고 특정 차트가 무엇을 의미하는지(게다가 한쪽 벽이 왜 인덱스 카드로 가득 차 있는지)를 교육하고 이해시킴으로써 가치를 매길 수 없는 협업과 의사소통의 정신을 육성할 수 있다. 특히, 문외한들에게 이런 개념들을 설명하는 것은 여러분 본인에게도 도움이 된다. 그 이유는 그렇게 함으로써 여러분 스스로도 전체 프로세스에 대해 더욱 자세히 이해할 수 있게 되기 때문이다.

지금까지 설명한 모든 도구 중에서도 가장 좋은 것은 지속적인 접촉(high-touch)을 통해 저항이 누그러든(low-resistance) 것들이다. 이런 도구들은 매우 빈번하게 활용되며, 사용하는 데 특별히 방해가 되는 요소들도 존재하지 않는다. 만일 어떤 도구가 사용하기에 조금이라도 불편해진다면 이 도구는 점점 방치될 것이며, 비록 처음에는 빈번하게 사용되었고 지속적으로 업데이트가 되어 왔다 하더라도 더는 관심을 끌지 못하고 빠르게 뒤쳐지게 된다.

⬟ 완료에 대한 정의

모든 프로젝트에는 완료에 대한 정의(DoD, Definition of Done)가 필요하다. 이는 모든 사용자 스토리를 완료로 처리하기 위해 반드시 정의해야 하는 표준이다. 아마도 여러분들은 개발자들이 다음과 같이 말하는 것을 수천 번은 들어 봤을 것이다.

> "일단 일은 마쳤고요, 이제 테스트만 남았습니다"
>
> "일은 마무리됐습니다만 수정해야 할 결함을 하나 발견했네요"
>
> "일은 마쳤는데요, 디자인이 마음에 안 들어서 인터페이스를 조금 바꿔 볼까 합니다"

필자 역시 과거에는 이런 말들을 하곤 했다. 만일 스토리가 진정으로 완료되었다면 여기에는 어떤 경고나 조건, 추가 필요사항이 없어야 한다. 이 예시들은 개발자들이 예측을 잘못하거나 미처 예상하지 못한 문제로 인해 시간을 조금 더 확보해야 할 필요가 있을 때 주로 하는 말들이다.

'완료'에 대한 정의를 모두가 동의하고 이를 완수하기 위해 분발해야 한다. 사용자 스토리가 조건에 부합하지 않는다면 완료 상태가 될 수 없다. 물론, 이 완료 조건에 부합하지 않는다면 스토리 포인트 역시 확보할 수 없다.

완료에 대한 정의에는 어떤 것들이 필요할까? 이는 여러분 본인과 여러분이 속한 팀, 그리고 품질 보증 절차가 얼마나 엄격한지에 따라 다르다. 다음은 보편적인 DoD의 정의를 보여 준다.

사용자 스토리를 완료하기 위해서는 반드시 다음의 사항을 준수해야 한다.

- 코드 실행의 성공 및 실패의 경우를 모두 커버할 수 있는 단위 테스트를 작성하며, 모든 테스트는 실패 없이 통과해야 한다.
- 모든 코드는 지속적 통합 서버에 전달되어 (오류 없이) 빌드 및 컴파일되어야 하며, 모든 테스트를 통과해야 한다.
- 제품 소유자와 함께 수렴 가능한 조건들에 대한 제품의 동작을 검증한다.
- 해당 스토리에 참여하지 않은 개발자와 짝을 이루어 코드를 리뷰한다.
- 의사소통에 필요한 만큼만 문서를 작성한다.
- 무분별한 기술 부채를 해결한다.

지금까지의 항목들을 바탕으로 자유롭게 규칙을 추가, 수정 혹은 제거해도 되지만, 이렇게 정의된 항목들에 대해서는 강력하게 규제해야 한다. 한 스토리가 모든 조건을 만족하지 못하면 스토리가 모든 조건을 만족하도록 수정하거나, 또는 완료를 정의한 조건 중 달성이 어려운 항목을 제거하면 된다. 예를 들어, 코드 리뷰어가 모호하거나 너무 보수적이라면 완료 조건에서 해당 조건을 생략해도 된다.

차트와 측정

스크럼 프로젝트는 다양한 종류의 차트를 이용해 진척도를 모니터링한다. 스크럼 차트를 통해 스크럼 프로젝트의 현재 상태와 진행 과정의 궤적을 확인할 수 있으며, 달성 가능한 향후의 성과를 예측할 수 있다. 이런 모든 차트는 몇 발자국 떨어져서도 쉽게 확인할 수 있는 크기로 스크럼 보드에 항상 표시되어 있어야 한다. 이 차트는 팀의 역량을 공개적으로 측정하고 있음을 보여 주기 위한 것이 아닐 뿐더러, 관리자들을 위해 프로젝트의 진척도를 측정하고 있다는 것을 보여 주기 위한 것도 아니다. 이는 진척도의 측정 방법을 모두에게 공개하는 것이며, 역량을 평가하기 위한 것이 아니라 전체 프로젝트의 문제점을 분석하기 위한 것임을 명확히 해야 한다.

관련 노트에는 개발자별로 달성한 스토리 포인트 등과 같은 개인적인 평가는 절대 언급되어서는 안 된다. 이런 평가는 팀에 좋지 않은 메시지를 전달할 뿐이며, 개발자들은 팀 전체의 진척도보다는 개인의 진척도를 우선시하게 될 것이다. 개발자들은 이런 종류의 평가에 금방 적응하여 포인트가 높은 스토리를 차지하여 그 포인트를 독차지하려 할 것이다. 어떤 방식을 장려할 것인지는 신중히 결정하기 바란다.

측정 항목에 대해서는 항상 신중하기 바란다. 이와 관련해서 '관찰자 효과(observer effect)'가 존재한다. 예를 들어, 어떤 항목은 최초 측정 항목을 바꾸지 않고서는 측정이 불가능한 것들이 있다. 자동차의 타이어 공기압 측정을 예로 들어 보자. 타이어 공기압을 측정하기 위해서는 먼저 타이어에서 약간의 공기를 빼내지 않으면 측정하기가 매우 어렵다. 그런데 이렇게 함으로써 애초에 측정하려 했던 공기압이 바뀌게 된다. 이와 동일한 원칙이 인간 본성에도 그대로 적용된다. 팀이 어떤 조건에 의해 평가될 것인지를 알게 되면 구성원들은 자신들의 통계자료를 개선하기 위해 무엇이든 하려 할 것이다. 여러분이 권모 술수에 능한 팀 구성원들을 관리하고 있다는 이야기를 하려는 것이 아니라, 스토리 포인트가 평가의 기준이 된다는 것을 팀 구성원들이 알게 되면 그들은 같은 노력으로 더 많은 포인트를 얻을 수 있는 스토리를 원할 것이라는 말이다. 이런 현상을 방지하기 위해서는 (이 장 후반부의 '스프린트 회고' 절에서 설명할) 삼자 평가(triangulation)를 통해 예상되는 노력치와 실제 노력치 사이의 간극을 조율해야 한다.

⬟ 스토리 포인트

스토리 포인트는 매 스프린트에 비즈니스 가치를 부여하여 팀을 독려하기 위한 장치이다. 스토리 포인트는 전체 팀이 스프린트 계획 회의를 진행하는 동안 사용자 스토리에 할당한다(이번 장 후반부의 '스프린트 계획' 절을 참고하기 바란다). 스토리 포인트는 사용자 스토리가 표현하는 특정 기능을 구현하는 데 필요한 상대적인 노력의 정도를 의미한다. 또한, 요구 사항 분석, 기술적 디자인 및 단위 테스트를 포함한 코드의 구현을 비롯하여 수용 조건에 대한 품질 보증 및 스테이지(stage) 환경으로의 배포 등 전체 개발 과정을 완료하기 위해 필요한 노력의 정도를 모두 포함한다. 모든 스토리는 스프린트 기간 내에 완료할 수 있도록 충분히 작은 범위로 나뉘어야 하지만, 그 크기에 따라 다양하게 분류할 수 있다.

가장 작은 크기의 스토리는 '1점 스토리(one-point story)'로, 최소한의 노력으로 구현할 수 있는 스토리이다. 스토리 포인트에 대한 가장 중요하면서도 흥미로운 사실은 스토리가 할당된 팀 내부적으로만 의미를 갖는다는 점이다. 예를 들면, 같은 스토리에 대해 어느 팀은 1점을, 다른 팀은 3점을 할당할 수도 있다. 스토리를 위한 적절한 노력의 정도를 합의하는 일은 수차례 스프린트를 진행하는 동안 계속해서 해야 하는 일이다.

반면, 스토리 포인트는 해당 스토리를 완료하는 데 필요한 노력을 시간으로 측정하여 표현한 것은 아니다. 다만, 대략적으로 그림 1-6에서 보여 주는 시간의 범위에 대응한다. 이 차트를 보면 수직 막대는 예상 시간을 표현하며, 이 막대에 덧붙여진 수평 막대는 해당 스토리의 포인트를 위해 투입할 실제 노력을 표현한다.

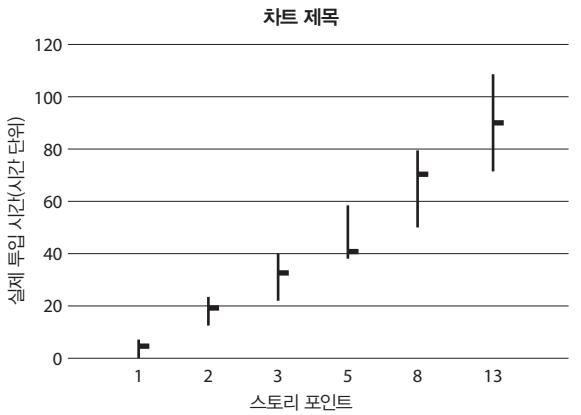

그림 1-6 **최소/최대/평균 차트는 예상된 노력 대비 실제 투입한 노력 간의 상관관계를 보여 준다.**

이 차트를 통해 알 수 있는 것은 범위가 큰 스토리일수록 그에 상응하게 많은 범위의 시간을 소모한다는 점이다. 즉, 범위가 큰 스토리일수록 완료하는 데 얼마나 많은 시간을 소요하게 될 것인지를 정확히 예측하기가 더 힘들다.

⬟ 속도

스프린트를 연속해서 진행하다 보면 스토리 포인트를 달성하기 위해 평균적으로 소요되는 시간을 계산할 수 있게 된다. 팀이 세 개의 스프린트를 진행하는 동안 각각 8점, 12점, 11점이 할당된 스토리를 완료했다고 가정해 보자. 이때 총 스토리 포인트는 31점이며, 평균 스토리 포인트는 10점이다. 이것을 팀의 속도(velocity)라고 하며, 두 가지 방법으로 활용할 수 있다.

먼저, 팀의 속도는 다음 스프린트에 팀이 커밋해야 할 포인트의 한계치이다. 만일 팀이 스프린트당 평균 10포인트를 획득한다면, 한 번의 작업 주기 내에 그보다 많은 양의 커밋을 발생시키는 것은 단순히 낙관적인 목표라고만 할 수는 없다. 이로 인해 팀 전체의 사기를 떨어뜨리는 실패를 경험하게 될 수도 있다. 불가능한 초과 목표를 설정하고 진행하다가 도중에 목표치를 낮추는 것보다는 달성 가능한 목표를 수립하고 그만큼을 달성하는 것이 훨씬 나은 방법이다. 만일 팀이 10포인트의 목표를 설정하였는데 실제로 11포인트를 달성했다면, 팀의 속도는 (12 + 11 + 11) / 3

으로 계산하여 11로 변경된다. 이것이 바로 스크럼의 반복적인 피드백의 결과이다.

속도를 이용하는 두 번째 방법은 소프트웨어의 출시에 대한 문제를 분석하는 것이다. 한 스프린트가 지난 이후 팀의 속도가 눈에 띄게 감소했다면 이는 아마도 그 스프린트 기간에 뭔가 좋지 않은 일이 발생했기 때문일 것이며, 이에 대한 조정이 필요할 것이다. 어쩌면 스토리가 너무 비대했지만 실제 크기를 제대로 예상하지 못해서 그 스토리가 오랜 기간 진행 중 상태로 남아 있었거나, 심한 경우 다음 스프린트까지 이어졌을 수도 있다. 또는 설명이 너무 단순했을 수도 있다. 너무 많은 핵심 인력이 동시에 휴가를 가서(또는 아파서) 전반적으로 진행이 늦어졌을 수도 있다. 한편으로는 이미 동작하는 기존 코드에 대해 리팩토링에 너무 많은 시간을 할애하느라 시스템에 새로운 기능을 더할 충분한 여력이 없었을 수도 있다. 이유가 무엇이든 팀의 속도가 25% 하락한다고 해서 항상 뭔가 안 좋은 일이 발생하는 것은 아니지만, 조만간 최대한 빨리 해결해야 하는 문제가 생길 것이라는 암시이기는 하다. 속도가 주 단위로 줄어드는(지속적으로 감소하는) 현상은 분명히 문제점이며, 아마도 코드가 변화를 수용하지 못하고 있음을 의미하는 것이다. 이 책은 이런 문제를 해결하는 데 도움을 주기 위한 책이다.

⬟ 스프린트 번다운 차트

매 스프린트를 시작할 때마다 2차원의 데카르트 그래프를 만들고 이를 스크럼 보드에 붙여 둔다. 이 그래프의 y축에는 총 스토리 포인트를 표시하며, x축에는 총 업무일수를 표시한다. 그런 후에 그림 1-7과 같이 이상적인 진척도를 의미하는(최적선이라고 알려진) 직선을 대각선으로 그린다.

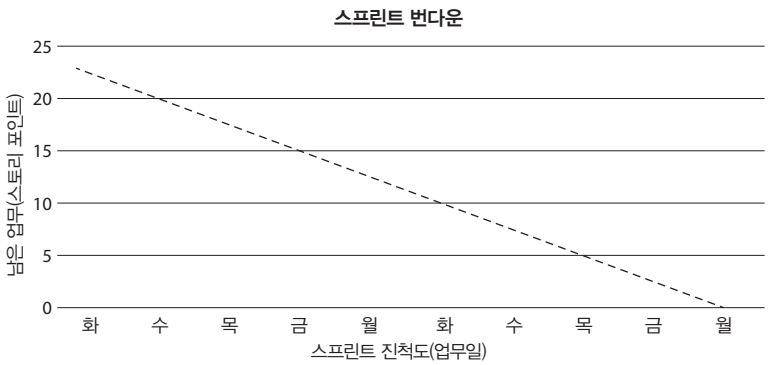

그림 1-7 **스프린트 시작 시점의 스프린트 번다운 차트(sprint burndown chart). 직선은 스프린트 목표에 대한 '최적선'을 보여 준다(예제의 경우 총 23스토리 포인트가 목표로 설정되어 있다).**

PART

III

적응형 예제

제10장 적응형 예제 — 소개

제11장 적응형 예제 — 스프린트 1

제12장 적응형 예제 — 스프린트 2

3부에서는 소프트웨어 제품 개발의 기본 절차들을 되돌아 본다. 3부를 구성하는 세 개의 장에서는 가상의 팀과 프로젝트를 설정하고, 팀 구성원들의 대화를 통해 이들이 어떤 의사결정을 하게 되는지 엿볼 수 있다.

코드 예제는 1부와 2부에서 지금까지 살펴봤던 패턴과 사례들 중 일부를 선택하여 반영하게 될 것이다. 모든 패턴과 사례가 언급되지는 않지만, 구현에 대해 가장 일반적으로 갖게 되는 의문들에 대한 해법을 제시한다.

이 책의 나머지 부분에서 다루는 예제의 실제 코드는 Github(깃허브)에서도 찾아볼 수 있다. 부록 A '적응형 도구'에서는 소스 제어를 위한 Git의 사용법을 설명한다. 웹으로만 제공하는 부록 B 'Github 코드 예제'에서는 각 장에서 사용했던 예제 코드가 Github 저장소의 어느 브랜치에 위치하는지를 찾아볼 수 있다.

PART III

Adaptive sample

CHAPTER

10

적응형 예제 — 소개

이 장의 주요 내용

- 적응형 예제 애플리케이션을 개발하는 팀에 대해 알아본다
- 적응형 예제 애플리케이션의 제품 기능에 대해 이해한다.
- 가장 첫 번째 스프린트(sprint zero)에서 애플리케이션의 최초 제품 백로그를 작성한다.

이번 부를 구성하는 세 개의 장에서는 스크럼과 적응형 디자인 원칙을 적용하여 실제로 동작하는 애플리케이션을 구현해 나간다. 지금까지 학습했던 내용을 모두 활용하여 하나의 완벽한 그림을 그려 나갈 것이다. 남은 내용을 읽는 동안 지금까지 학습했던 내용들을 모두 고려하면서 읽어 나가길 권한다. 그렇지 않는다면 코드를 올바르게 이해하기 어려울 것이다. 마찬가지로, 마이크로소프트 비주얼 스튜디오 솔루션 파일 전체를 다운로드하지 않는다면 앞으로 살펴볼 예제 코드만으로는 전체 그림의 아주 작은 부분밖에 이해하지 못할 것이다.

이번 장은 실세계를 충분히 반영할 수 있는 시나리오를 바탕으로 하지만, 간결함과 명료함을 위해 약간 간소화된 시나리오를 반영한다. 이번 장에서는 가상의 스크럼 팀을 소개하고 우리가 개발하게 될 제품을 개략적으로 살펴본다.

트레이 리서치

예제 애플리케이션은 트레이 리서치(Trey Research)라는 가상의 회사가 개발하는 제품이다. 이 회사는 변화를 수렴할 수 있는 적응형 코드를 작성하는 것에 대해 큰 자부심을 가지고 있다.

팀

예제 애플리케이션은 스크럼 프로세스를 사용하여 개발할 것이다. 따라서 개발 업무를 담당한 팀에는 스크럼에서 언급하는 모든 역할을 소화할 수 있는 구성원들이 필요하다. 스크럼 프로세스와 필요한 역할에 대해 다시 한 번 되짚어 보려면 제1장 '스크럼을 소개합니다'를 참고하기 바란다.

이 팀은 애플리케이션의 기획 단계부터 릴리스 단계에 이르기까지 필요한 모든 역할을 수행할 구성원들로 구성되어 있다. 제품 소유자는 애플리케이션이 어떻게 동작해야 하는지, 각 기능의 우선순위는 무엇인지, 그리고 어떤 기능이 회사의 수익에 가장 크게 기여하는지 등에 대한 충분한 지식을 가지고 있다. 스크럼 마스터는 팀이 활용 중인 프로세스에 집중한다. 스크럼 마스터는 프로세스가 실제로 팀 내에 잘 반영되고 있는지, 팀의 업무에 방해가 되는 요소는 없는지, 그리고 제품 소유자가 사용자 스토리를 개발하는 과정에서 발생한 모든 이슈에 대해 인지하고 있는지 등에 집중한다. 트레이 리서치의 개발팀은 스토리를 구현하는 개발자와 테스트를 디자인하고 구현된 사용자 스토리가 제품 릴리스 조건에 부합하는지를 테스트하는 테스트 분석가로 구성된다.

⬟ 제품 소유자

제품 소유자의 이름은 페트라(Petra)이다. 그녀는 베테랑 비즈니스 분석가로 회사에 합류한 지는 얼마 되지 않았다. 그녀의 특기는 제품 소유자라면 반드시 갖추어야 할 능력, 즉 고객이 원하는 것을 정확히 파악하는 것이다. 그녀는 애자일 프로세스에 대해서는 문외한이지만, 자신의 새로운 역할을 이해하고자 하는 열의는 대단하다.

개발 프로세스가 진행되는 동안 페트라는 고객과 접촉하며 고객이 어떤 기능을 왜 필요로 하는지를 파악한다. 또한, 고객에게 제공할 다양한 기능의 가치를 정확히 산출하여 개발팀의 업무 우선순위 확립에 도움을 줄 것이다.

⬟ 스크럼 마스터

스티브(Steve)는 회사에서 두 가지 역할을 수행한다. 그는 스크럼 마스터인 동시에 개발팀의 리더이다. 회사로서는 가까운 시일 내에 이 부분을 개선하여 스티브가 자신이 원하는 스크럼 마스터 역할에 집중할 수 있도록 도움을 주고자 한다. 그러기 위해 회사는 개발팀의 리더 역할을 수행할 수 있는 경험이 풍부한 개발자를 구인 중에 있다.

스크럼 마스터로서 스티브의 역할은 팀이 스크럼 프로세스를 준수하도록 유도하며, 팀 구성원들이 현재의 프로세스에 만족하도록 만들어 주는 것이다. 그는 제품 소유자나 고객 어느 누구와도 정직하고 투명하게 일하고 있는 것에 대해 자부심을 가지고 있으며, 업무의 범위를 변경하거나 무분별한 릴리스 계획을 세우지 않는다.

스티브는 코드를 작성할 시간은 거의 없지만 디자인 회의에는 꾸준히 참석하여 개발팀이 올바른 방향으로 나아가는 데 도움을 주고 있다.

⬟ 개발자들

데이빗(David)과 다이엔(Dianne)은 이 회사에 근무하는 개발자들이다. 데이빗은 대학을 갓 졸업하고 입사한 초급 개발자이며, 다이엔은 데이빗보다는 경험이 있는 중급 개발자이다.

스티브가 데이빗을 고용한 이유 중 하나는 데이빗이 계속해서 프로그래밍 사례와 기법들을 스스로 공부하고 있기 때문이다. 데이빗은 항상 최신 트렌드를 파악하고 공부하려는 노력을 게을리하지 않는다. 하지만 이런 새로운 기술이나 기법을 마치 마법봉처럼 여기고 있어서 여기저기에 적용하려 하는 성향을 지니고 있다. 데이빗 입장에서는 더할 나위 없이 좋은 연습과 공부가 되겠지만, 불필요한 부분들로 인해 코드가 엉망이 되는 경우가 잦다는 것이 단점이다.

다이엔은 데이빗에 비해 경험이 더 풍부하지만, 최근 몇 년간 새로운 기술이 계속해서 등장하면서 이들을 쫓아가기엔 다소 지친 듯한 모습을 보이고 있다. 다이엔은 데이빗이 현재 진행 중인 업무를 함께 하고 있다. 다이엔은 개발팀의 리더 역할을 하기 위해 노력하고 있으며, 승진 자격에 대해서는 이미 충분히 검증된 상황이다. 또한, 데이빗을 도와 함께 일하는 것에 대해서도 만족해하고 있다.

⬟ 관심 주기

IT 조사 및 컨설팅 회사인 가트너(Gartner)가 개발한 관심 주기(The Hype Cycle)는 새로운 기법과 기술들의 성장 과정을 평가할 수 있는 매우 유용한 도구이다. 우선 그림 10-1을 살펴보자.

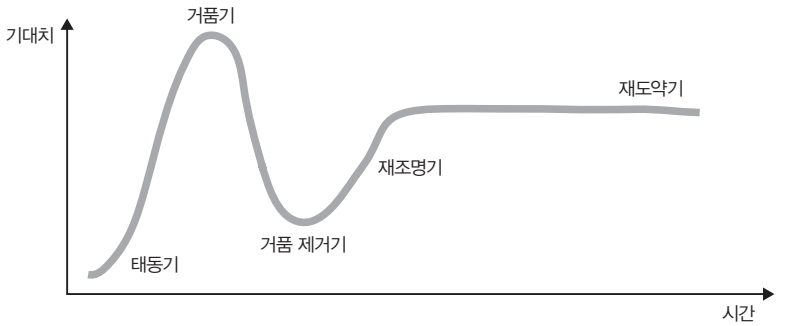

그림 10-1 **관심 주기는 새로운 기법과 기술에 대한 관심의 정도를 설명하는 그래프이다.**

이 그래프의 X축은 시간의 흐름을 표현하며, Y축은 사람들의 기대치를 표현한다. 보통은 새로운 기술적 발견이나 유용한 새로운 기법 또는 절차가 등장하게 되면 사람들의 기대치는 거품기라고 표시된 지점에 도달할 때까지 가파르게 상승한다. 이 지점에 도달하면 기술이나 기법이 처음에 기대했던 것에 미치지 못하는 것처럼 보이게 마련이다. 그런 이유로 기술에 대한 기대치가 급속도로 빠져나가는, 이른바 거품 제거기를 맞이하게 된다. 하지만 모든 거품이 빠져나가는 것이 아니라 재조명기를 거쳐 재도약기에 이르게 된다. 이 지점에 도달하면 기술이나 기법이 어느 정도 정립되어 현실적인 기대치를 반영하게 된다.

개발자들의 새로운 프로그래밍 기법이나 기술에 대한 기대치는 대체로 이 그래프상의 어딘가에 위치한다. 예를 들어, 트레이 리서치의 경우 데이빗은 디자인 패턴, SOLID 원칙, 단위 테스트 및 리팩토링 등의 주제에 대해 거품기에 가까운 기대치를 가지고 있다. 반면, 다이엔은 거품이 다 빠져나간 거품 제거기 근처에 있다고 볼 수 있다. 잠시 후에 스프린트 회의에 대해 자세히 살펴보겠지만, 다이엔이 던지는 질문과 반응에 대해 주의 깊게 살펴보고 이들이 관심 주기 그래프의 어느 지점에 위치하고 있는지에 대해 생각해 보자.

스티브의 경험들은 그래프상의 재도약기에 해당하는 영역 어딘가에 있다. 뿐만 아니라 스티브는 이 관심 주기에 큰 영향을 받지 않는다. 그는 생산성이나 품질, 효율성 등을 향상시킨다고 주장하는 새로운 기술을 도입할 때는 신중을 기한다. 그렇기 때문에 스티브는 새로운 기술이나 기법을 마주하더라도 흔들림이 없다.

⬟ 테스트 분석가

톰(Tom)은 팀의 테스트 분석가이다. 그는 테스트 자동화 업무를 담당하고 있으며, 소프트웨어 제품의 약점을 찾는 것과 제품의 신뢰도 향상을 통한 전체적인 사용자 경험의 개선에 뛰어난

능력을 발휘한다. 또한, 제품의 기능을 블랙박스처럼 취급하는 것을 좋아해서 해당 기능이 어떻게 구현되었는지보다는 요구 사항에 맞게 동작하는지를 더 중요하게 여긴다.

톰은 개발팀이 구현한 기능들 중 일부가 출시에 부적합한 경우가 지속되면서 자신의 업무가 점점 많아지고 있다고 생각하고 있다. 자신이 기대하는 대로 사용자 스토리가 구현되었는지를 테스트하고, 기능이 제대로 구현되지 않은 경우 부적합 판정을 내리게 되면 나중에 해당 기능을 다시 테스트해야 하기 때문이다. 하지만 그로 인해 고객의 장비에 결함을 가진 소프트웨어가 배포되는 일을 최소화하고, 자동화된 테스트를 통해 결함을 미리 찾아내는 일에 자부심을 가지고 있다.

제품

트레이 리서치는 고객의 요구에 따라 프로즈웨어(Proseware)라는 새로운 온라인 채팅 애플리케이션을 개발하고 있다. 이 애플리케이션은 웹 기반 애플리케이션으로, 사람들이 채팅을 통해 소통할 수 있는 기능을 제공한다. 페트라는 고객과의 첫 회의를 통해 고객이 이 프로젝트에 대해 좋은 아이디어를 많이 가지고 있으며, 제품을 지속적으로 올바르게 이끌어가기 위해 이 아이디어들을 잘 활용하고 싶어한다는 점을 간파했다. 트레이 리서치는 점진적인 제품 개발 스타일과 요구 사항의 잠재적인 변화에 대응할 수 있는 능력이 충분하므로 이 제품 개발에 적합한 회사였다.

페트라는 고객에게 프로젝트를 시작하기 전에 개발팀에게 어떤 제품을 개발해야 하는지에 대한 정보를 제공해 줄 것을 요청했다. 고객은 트레이 리서치가 프로즈웨어를 호스팅하기를 원했으므로 플랫폼과 도구는 팀 구성원들이 가장 적합하다고 판단되는 것이라면 어떤 것이든 자유롭게 선택할 수 있었다. 또한, 페트라는 고객이 프로즈웨어에 대해 어느 정도의 고객 수용력을 기대하는지에 대해서도 확인을 해 두었다. 페트라 역시 고객이 원하는 대로 최소 20명의 동시 접속자를 수용할 수 있어야 한다는 점에 동의했으며, 따라서 이 요구 사항이 제품의 기능외 요구 사항 중에서는 핵심적인 요구 사항이라고 할 수 있다.

팀이 코드를 작성하기에 앞서 페트라는 전체 팀 회의를 주관하여 팀 구성원들이 프로젝트에 대해 파악하고 사용자 스토리에 대한 기본적인 백로그를 작성하는 시간을 마련해 주었다. 이렇게 스토리들을 결정해야 팀이 업무를 시작하고 매 스프린트마다 데모가 가능한 결과물을 만들어 낼 수 있게 될 것이다.

최초의 백로그

고객은 페트라와 팀에게 프로즈웨어의 기능 목록을 전달했다. 이 목록은 장황한 글로 표현되어 있어서 팀은 회의를 통해 이 요구 사항을 하나 혹은 그 이상의 사용자 스토리로 변환하기로 했다.

> 우리는 프로즈웨어를 사람들이 채팅하고 싶을 때 가장 먼저 방문하게 되는 사이트로 만들고 싶습니다. 그러나 로마가 하루 아침에 건설되지 않았듯이 우리의 요구 사항 역시 현재로서는 완벽하지 않다는 것을 잘 알고 있습니다. 다만, 최대한 빠른 시일 내에 결과물을 보고 싶습니다.
>
> 우리는 사용자들이 서로 이미지나 파일을 주고받을 수 있게 하고 싶지만, 이보다 더 중요한 기능은 텍스트를 바탕으로 한 커뮤니케이션입니다. 채팅방에 접속한 사람이라면 서로가 주고받는 텍스트 메시지를 모두 볼 수 있어야 합니다.
>
> 텍스트 기반의 채팅은 매우 중요한 요소이기 때문에 채팅방에 있는 참여자들이 HTML처럼 텍스트를 포장해서 보낼 수 있으면 좋겠습니다. 물론, 의미 없는 이미지나 동영상을 잔뜩 보내서 채팅방을 어지럽히는 일은 방지했으면 합니다.
>
> 프로즈웨어에서 이루어지는 대화 중 일부는 특정 사용자에 대해서는 편집이 불가능해야 하며, 일부 유저는 메시지를 읽을 수만 있을 뿐 전송할 수는 없도록 했으면 합니다.

페트라는 회의에서 이 요구 사항을 개발팀에 전달했고, 개발팀은 이를 바탕으로 사용자 스토리를 찾아내기로 했다.

사용자 스토리 찾기

고객들은 자신들이 기대하는 결과물을 앞서 살펴본 것과 같이 장문의 글로 표현한다. 개발팀은 이 글에서 자신들이 구현해야 할 사용자 스토리를 추출해 내야 한다. 트레이 리서치 팀은 고객이 전달한 프로즈웨어에 대한 요구 사항으로부터 사용자 스토리를 찾아내기 위해 회의를 열었다.

> 페트라: 자, 다들 요구 사항 문서를 읽어 봤겠죠? (모두가 고개를 끄덕이며 관심을 보인다.)
>
> 스티브: 요구 사항이 꽤 많은데요?
>
> 데이빗: 아뇨, 그렇지 않은 것 같아요. 이 정도면 한 스프린트 내에 다 끝낼 수 있을 것 같아요. (스티브가 활짝 미소를 짓는다.)
>
> 페트라: 이 요구 사항에서 어떻게 사용자 스토리를 찾아낼 수 있을까요?

다이엔: 먼저, 동사와 명사를 분류해야 하지 않을까요? 보내기, 받기, 형식 맞추기, 스팸, 채팅방, 대화, 사용자, 가입자, 채팅 등으로요.

스티브: 맞아요. 그거 좋은 생각이네요.

페트라: 사용자로서 나는 OO를 원합니다. 뭐 이런 건가요?

다이엔: 이 제품을 사용하는 사용자의 역할이 일반 사용자 말고는 없나요?

스티브: 아뇨, 일반 사용자 외에도 다른 역할이 더 있을 것 같네요.

데이빗: 사용자라고 표현한 부분이 있고, 참여자라고 표현한 부분이 있네요.

톰: 후자는 채팅방에 참여하고 있는 사람들을 의미하는 게 아닌가요?

스티브: 맞을 거에요. 하지만 우리는 고객과 동일한 언어를 사용해야 해요. 일단, 대화를 읽을 수만 있는 역할과 그렇지 않은 역할이 있는 건 확실한 거죠?

다이엔: 네. 이 두 가지 역할이 권한 관리 시스템의 힌트가 되어 줄 것 같네요.

페트라: 그럼 스토리를 하나 끄집어 내 볼까요?

데이빗: 참여자로서 나는 다른 사람이 볼 수 있는 텍스트를... 음... 예쁘게 꾸미고 싶습니다. 이 정도?

스티브: 일단은 시스템 내의 역할과 각 역할별 행위에 먼저 집중해 볼까요? 페트라, 여기에 비즈니스적으로 조금 더 가치를 부여할 수 있겠죠?

페트라: 네. 가능하죠.

다이엔: 좋네요. 하지만 이건 좀 분량이 커 보이는데요. 사용자 스토리라기보다는 에픽(epic)에 가까운거 같아요.

페트라: 아, 에픽이 정확히 뭐죠?

데이빗: 그냥 분량이 큰 스토리라고 생각하면 되요. 다이엔은 이 스토리가 한 스프린트 내에 마치기에는 너무 크다고 생각하는 것 같네요. 그런데 저는 개인적으로 권한 시스템의 분량을 어떻게 줄일 수 있을지 잘 모르겠네요.

다이엔: 음, 예쁘게 꾸민 텍스트보다 조금 더 간단한 게 뭘까요? HTML은 일단 고객이 예를 든 것뿐이니까 무시하고요. 방법이 뭐가 됐든 일단 꾸며진 텍스트라고 생각해 보자구요. 이것보다 간단한게 뭘까요?

데이빗: 어... 꾸며지지 않은 텍스트?

스티브: 보통은 평문(plain text)이라고 하죠. (데이빗이 흠칫하며 웃는다.)

다이엔: 채팅방 참여자로서 나는 채팅방의 다른 참여자에게 평문 텍스트 메시지를 보내고 싶습니다. (페트라가 이 의견에 반대하는 사람이 있는지 살펴본다. 다들 말없이 동의하는 눈치다.)

톰: 짜잔, 드디어 첫 번째 사용자 스토리가 정해졌네요!
(페트라를 뺀 나머지 팀 구성원들이 박수를 친다.)

페트라: 하지만 고객은 텍스트를 꾸며서 보낼 수 있기를 원했잖아요?

스티브: 걱정 말아요, 페트라. 그건 다른 스토리로 구분해서 일단 평문을 전송할 수 있게 한 다음 나중에 구현하면 되요.

페트라: 알았어요. 채팅방 참여자로서 나는 채팅방의 다른 참여자에게 꾸며진 텍스트 메시지를 보내고 싶습니다.

스티브: 두 번째 사용자 스토리도 정해졌네요. 다른 의견 또 있나요?

데이빗: 나는 채팅방을 만들고 싶습니다?

다이엔: 맞아요. 그리고 그 사람이 방장이 되어야죠. 아닌가요? 방장으로서 나는 대화를 분류할 수 있도록 채팅방을 개설하고 싶습니다.

스티브: 좋아요.

톰: 고객 요구 사항을 보니 이미지랑 파일도 전송하고 싶다고 써 있네요.

페트라: 그러네요. 이것도 또 다른 스토리가 되겠군요.

스티브: 하나 더 있어요. 나는 읽기 전용 대화를 생성하고 싶습니다.

다이엔: 그거면 마지막 요구 사항도 수렴할 수 있겠군요. 이제 어느 정도 정리가 됐네요.

스토리 점수 예상하기

이제 팀 구성원들이 고객의 요구 사항을 바탕으로 사용자 스토리를 추출해 냈다. 그리고 아래의 항목을 바탕으로 사용자 스토리 카드를 만들었다.

- 나는 채팅방의 다른 참여자에게 평문 텍스트 메시지를 전송하고 싶습니다.
- 나는 대화를 분류하기 위해 채팅방을 개설하고 싶습니다.
- 나는 채팅방의 다른 참여자에게 꾸며진 텍스트 메시지를 전송하고 싶습니다.
- 나는 이미지나 파일을 전송하고 싶습니다.
- 나는 읽기 전용 대화를 생성하고 싶습니다.

그리고 완성된 카드를 테이블에 펼쳐 스토리 점수를 할당하기로 했다. 구현해야 할 스토리가 다섯 개밖에 없었기 때문에 각 스토리에 투입될 노력의 정도를 예상하기 위해 계획 포커(planning poker)를 사용하기로 했다.

페트라: 평문 메시지를 보내는 건 별로 어렵진 않겠죠?

데이빗: 나라면 한 시간이면 끝낼 수 있어요!
(스티브와 다이엔은 눈썹을 움찔거렸고, 톰은 눈을 동그랗게 떴다.)

스티브: 아, 제 생각에는 이 스토리를 먼저 시작할 수 없을 것 같네요. 채팅방을 개설할 수 없으면 참여자도 없을 테니까요.

다이엔: 맞아요. 이 스토리들 사이에는 의존 관계가 있어요. 하지만 전체적인 분량에 영향이 있을 것 같진 않네요. (스티브가 동의하듯 끄덕거렸다.)

페트라: 다들 점수를 줄 준비가 되었나요?

카드가 몇 번 돌고 돌더니 팀 구성원들은 각자의 점수를 다음과 같이 결정했다.

페트라	톰	스티브	다이엔	데이빗
3	3	5	5	1

페트라: 음, 다 제각각이군요. 데이빗은 왜 1점이라고 예상하죠?

데이빗: 별로 할 일이 없어 보여서요. 입력창 한두 개를 놓고 입력된 텍스트를 화면에 보이기만 하면 되잖아요.

다이엔: 아니에요, 데이빗. 이건 그렇게 단순한 일이 아니거든요. 메시지를 전송하려면 입력된 메시지를 어딘가에 저장했다가 나중에 다시 읽을 수 있어야 해요. 채팅방의 다른 참여자들도 마찬가지로 그 메시지를 읽을 수 있어야 하고요.

스티브: 다이엔 말이 맞아요. 이 건에 대해서는 아키텍처에 대해서도 고민을 해야 할 필요가 있어요.

데이빗: 아, 그래요? 전 그냥 텍스트를 브라우저 창에 내보내기만 하면 되는 건 줄 알았어요. 아, 그런데 그렇게 하면 채팅 메시지를 보낸 사람만 볼 수 있겠군요. 와, 이거 생각처럼 간단한 일이 아니군요!

페트라: 좋아요, 그럼 예상 점수를 다시 공유해 볼까요?

페트라	톰	스티브	다이엔	데이빗
5	5	5	5	5

스티브: 이 스토리를 '읽기' 부분과 '쓰기' 부분으로 나눌 수 있을까요?

다이엔: 죄송하지만 무슨 말씀인지 이해를 잘 못하겠어요.

톰: 제 생각에는 보내진 메시지를 채팅방의 다른 참여자에게 표시하는 스토리와 채팅방에 메시지를 전송하는 스토리를 나누자는 말씀 같은데요?

스티브: 맞아요. 그렇게 하면 업무 분량을 비교적 작게 해서 시작할 수 있을 것 같아요. 스토리 점수 5점이 그렇게 높은 점수는 아니지만, 개발을 시작하는 단계에서는 그렇게 낮은 점수도 아닌 것 같거든요.

다이엔: 좋은 생각이네요. 그러면 메시지를 보내는 부분에 대해서만 다시 예상 점수를 공유해 볼까요?

페트라	톰	스티브	다이엔	데이빗
3	3	3	3	3

다이엔: 그리고 채팅방에 보내진 메시지를 표시하는 기능에 대해서도 예상을 해 보죠?

페트라	톰	스티브	다이엔	데이빗
2	3	2	2	1

스티브: 톰하고 데이빗, 2점이면 괜찮겠지요?

톰: 네, 좋습니다.

데이빗: 저도요.

스티브: 좋아요. 그럼 이 스토리에는 2점을 할당합니다. (스티브가 해당 스토리 점수에 2점을 기록하고 다음 카드의 주제를 읽는다.) '나는 대화를 분류하기 위해 채팅방을 개설하고 싶습니다.'

다이엔: 이 주제를 조금 더 명확히 할 수 있을까요?

톰: 페트라와 얘기해 봤는데, 일단 필요한 기능 중에 하나는 이미 만들어진 채팅방의 목록을 볼 수 있어야 한다는 거에요.

다이엔: 이 스토리도 '읽기'와 '쓰기' 스토리로 나눠야 할까요?

스티브: 네, 그래 보이네요.

다이엔: 그럼, 읽기 부분부터 먼저 예상치를 공유해 보죠. 나는 각각 대화가 진행 중인 채팅방의 목록을 보고 싶습니다.

팀 구성원들은 다음과 같은 예상 점수를 내놓았다.

페트라	톰	스티브	다이엔	데이빗
2	2	2	2	2

다이엔: 와, 만장일치네요. 그럼 새로운 채팅방을 생성하는 스토리는요?

이에 대한 예상 점수는 다음과 같다.

페트라	톰	스티브	다이엔	데이빗
2	2	2	2	1

스티브: 데이빗, 이 스토리에 2점을 부여해도 괜찮겠죠?

데이빗: 네. 뭐, 좋아요.

스티브: 다음은... 나는 채팅방 참여자들에게 꾸며진 텍스트를 전송하고 싶습니다.

페트라: 이건 그냥 텍스트를 꾸미는 기능만 추가하면 되는 걸까요?

스티브: 네. 아마 평문 텍스트를 보내는 스토리를 완료하면 이 기능에 필요한 대부분의 기능들은 구현될 거에요.

톰: 텍스트를 어떻게 꾸미고 싶다는 걸까요? 이미지를 첨부하거나 뭐 그런 걸 말하는 걸까요?

데이빗: 이미지는 별개의 스토리에요. 제 생각에는 글씨를 볼드체로 표시하거나, 기울이거나, 밑줄을 긋거나 뭐 그런 것들을 말하는 것 같네요.

페트라: 맞아요. 고객이 그렇게 말했어요. HTML은 고객이 알고 있는 게 그것밖에 없어서 한 이야기일 테고, 우리가 굳이 HTML을 적용할 필요는 없을 거에요. 간단한 몇 가지면 충분하지 너무 많은 기능을 제공할 필요는 없을 거에요.

스티브: 자, 그럼 각자 예상 점수를 볼까요?

팀 구성원들은 아래와 같은 점수를 들어 보였다.

페트라	톰	스티브	다이엔	데이빗
1	1	1	1	1

스티브: 야, 이번에도 만장일치네요. 다음은 이미지나 파일을 전송하고 싶습니다.

다이엔: 이미지나 파일 중 하나를 전송하고 싶다는 건가요? 아니면 둘 다 전송하고 싶다는 걸까요? 조금 헷갈리네요.

톰: 음, 이미지와 파일을 전송하는 데 차이점이 있나요?

페트라: 고객과 이야기를 해 보니 이미지는 브라우저에 표시하고 싶어 하고, 파일은 사용자 컴퓨터에 다운로드할 수 있으면 좋겠다고 하더군요.

데이빗: 그럼, 이것도 두 개의 스토리로 분리해야겠네요. 그렇죠? 하나는 파일을 다운로드하는 기능이고, 다른 하나는 이미지를 보여 주는 기능이고요.

다이엔: 맞아요. 지금 스토리는 너무 커서 분리하는 게 좋겠어요. 둘 중에서는 파일 전송이 조금 더 쉬워 보이니까 파일 전송을 먼저 구현하고 이미지 전송은 이어서 구현하는 게 어떨까요?

스티브: 좋은 생각입니다. 그러면 스토리를 두 개로 분리하기로 하죠. 나는 채팅방의 다른 참여자들에게 파일을 전송하고 싶습니다. 그리고 나는 채팅방의 다른 참여자들에게 이미지를 전송하고 싶습니다. 각자 예상 점수를 말해 볼까요?

팀 구성원들은 첫 번째 스토리에 대한 예상 점수를 다음과 같이 결정했다.

페트라	톰	스티브	다이엔	데이빗
5	5	3	3	3

스티브: 음, 왜 5점이나 필요한 거죠?

톰: 테스트하기가 어려울 것 같아요. 고려해야 할 테스트 케이스가 많아 보이거든요. 예를 들면, 전송하려는 파일의 용량이 너무 크다거나 파일이 바이러스에 감염되어 있으면 어떻게 하죠? 그리고 이 파일들은 계속 우리 서버에 저장해야 하나요? 그래야 한다면 얼마나 오래 보관하고 있어야 할까요?

스티브: 음, 좋은 지적이에요. 전 그냥 개발에 어느 정도 노력이 필요한지에 대해서만 생각했네요. 그런 것들을 모두 고려하자면 시간이 많이 필요하겠네요.

데이빗: 점수를 다시 예상해 볼까요? 아니면 그냥 5점을 부여할까요?

스티브: 전 5점을 주는 게 좋다고 생각해요.

다이엔: 저도요.

스티브: 이미지 전송은요?

페트라: 이건 그냥 웹 페이지에 이미지를 보여 주는 것뿐이잖아요?

스티브: 맞아요. 파일을 업로드하는 기능은 이미 구현이 된 상황일 테니까 이건 그냥 페이지에 이미지를 보여 주는 기능만 구현하면 되겠죠.

팀 구성원들의 예상 점수는 각각 다음과 같았다.

페트라	톰	스티브	다이엔	데이빗
3	3	3	3	5

스티브: 데이빗이 5점을 할당했네요?

데이빗: 네. 제 생각에는 보기보다 어려울 것 같아요. 이미지가 부적절한 이미지이거나 사용자가 너무 많아서 이미지를 다운로드하는 데 시간이 오래 걸리면 어떻게 하죠? 우리 서버에 부하가 발생할 것 같아요.

다이엔: 저도 그렇게 생각해요. 그런 문제들에 대해서도 생각해 봐야겠죠. 하지만 당장은 기능 자체의 범위에서 벗어난 주제 같아요. 페트라, 고객에게 연락해서 콘텐츠 필터링에 대해서 어떻게 생각하는지 한번 물어봐 주겠어요? 이미지뿐만이 아니라 텍스트도요. 당장은 20명의 사용자만 지원하면 되니까 서버의 부담은 그렇게 크지 않을 거에요.

페트라: 콘텐츠 필터라... 좋은 생각이네요, 데이빗.

스티브: 데이빗, 그럼 3점을 주면 충분할까요?

데이빗: 음, 지금까지 이야기를 생각해 보면 이제 2점으로도 충분할 것 같지만... 네. 뭐, 3점도 좋아요.

스티브: 좋습니다. 이제 마지막 스토리네요. 나는 읽기 전용 대화를 생성하고 싶습니다.

팀 구성원들의 예상 점수는 다음과 같았다.

페트라	톰	스티브	다이엔	데이빗
8	1	5	8	3

스티브: 이런, 다 제각각이네요. 톰, 왜 1점이라고 생각해요?

톰: 아, 미안해요. 분석과 개발에 필요한 부분을 생각을 못했네요. 그냥 다른 스토리들에 비해 테스트하기가 엄청 쉬울 것처럼 보여서요.

스티브: 괜찮아요, 페트라. 다이엔은 8점이라고요?

다이엔: 현재로서는 역할이나 권한에 대해 전혀 고려를 하고 있지 않잖아요? 하지만 분명 필요한 부분일 테고요. 제 생각에는 이 부분에 해야 할 일이 상당히 많을 것 같아요.

데이빗: 아, 역할이나 권한이 필요할 거라고는 생각을 미처 못했네요. 전 그냥 대화에 읽기 전용이라고 표시하는 정도라고만 생각했어요.

스티브: 읽기 전용 기능은 사용자 단위로 지정할 수 있어야 할 거에요. 사실, 역할이나 권한을 구현하려고 뭔가 대단한 인프라스트럭처를 도입해야 할 필요성은 아직 못 느끼겠네요. 우선은 최대한 간단하게 구현하고 나서 나중에 개선하는 게 어떨까요?

다이엔: 좋은 생각이에요.

페트라: 저도요. 여러분들을 믿습니다.

스티브: 평균 점수가 정확히 5점이네요. 5점을 부여해도 될까요? (모두가 동의한다)

스티브: 데이빗, 이게 아직도 한 스프린트에 끝낼 수 있는 일 같은가요?

데이빗: 어... 아뇨.

이렇게 회의가 끝났다.

마치며

회의를 진행하면서 팀은 고객의 간략한 요구 사항을 바탕으로 여러 개의 사용자 스토리를 도출해 냈다. 그런 후 각 스토리의 구현에 필요한 노력의 정도를 예측하고 이를 토대로 우선순위가 적용된 백로그를 생성했다. 이로써 개발을 시작하기 위한 준비는 끝났다.

지금까지 대화가 진행되면서 분석과 구현, 그리고 테스트 등 개발 과정의 모든 측면이 고려되었다는 점이 중요하다. 개발팀은 각 스토리마다 해당 스토리를 완료하기 위해 어느 정도의 노력이 필요할 것인지에 대한 합의를 이룰 수 있었다.