제 6 장 소프트웨어 생명 주기 모델과 테스트

소프트웨어 생명 주기는 소프트웨어를 개발하는 체계에 관한 추상적 표현으로, 순차적 또는 병렬적인 일련의 단계로 구성된다. 소프트웨어를 개발하는 전형적인 방법은, 우선, 분석가가 개발하고자 하는 소프트웨어에 대한 요구사항을 수집하고 문제를 이해 분석하는 단계에서 시작하여 설계 단계 및 시스템을 구성하는 모듈들을 구현하는 단계를 거친다.

그러나 소프트웨어를 개발할 때 반드시 이와 같은 절차에 따라 개발할 필요는 없다. 소프트웨어는 그 규모나 특성에 따라 매우 다양한 방식으로 개발될 수 있다. 예를 들면, 학교 과제물로 나오는 프로그램과 같이 매우 소규모인 프로그램을 작성하는 경우는 별도의 요구사항 분석이나 설계 과정 없이 바로 코딩 작업에 들어갈 수 있다. 또한, 결함을 검출하기 위한 테스트 작업을 별도로 분리해 수행하지 않고 디버깅 작업의 한 부분으로 수행할 수도 있다. 이와 같이 소프트웨어를 개발하는 방식을 Code-and-Fix 모형이라고 한다. 이 모형은 혼자 사용하거나 곧바로 폐기 처분되는 소프트웨어를 개발하는 경우에는 타당한 접근 방식일 수 있지만, 오랫동안 많은 다양한 사용자들이 사용하는 시스템이거나 비교적 규모가 큰 시스템인 경우에는 적합하지 않은 개발 모형이다.

이 장에서는 현재 사용되고 있는 주요한 소프트웨어 생명 주기 모형들을 기술하고 각 생명 주기 모형에서 테스트 역할에 대해 설명한다.

6.1 순차적 개발 모델

순차적 개발 모델에는 대표적으로 폭포수 모델과 V-모델이 있다. 그 중 폭포수 모델은 모든 소프트웨어 생명 주기 모형 중에서 가장 오래된 전통적인 모형으로, 소프트웨어 개발을 요구사항 분석에서 시작하여 설계, 코딩, 테스트, 유지보수의 전 과정을 체계적이고 순차적으로 접근한다. 그림 6.1은 일반적인 폭포수 모델을 보여 준다.

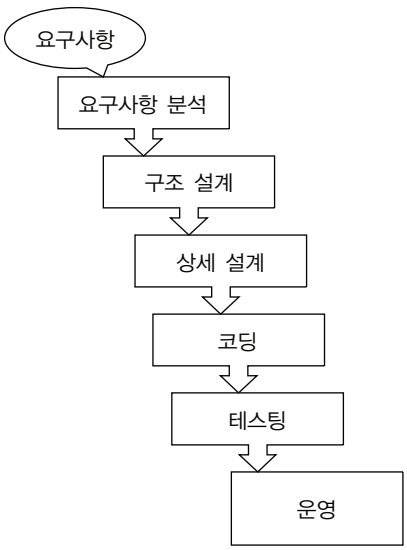

그림 6.1 **폭포수 모델**

다음은 폭포수 모델을 구성하는 주요 단계에 관한 간략한 설명이다.

- **요구사항 분석**: 개발하고자 하는 소프트웨어에 대한 요구사항을 수집하고 문제를 이해 및 분석하여 이를 명세화하는 단계이다. 소프트웨어 분석가는 요구되는 기능, 성능 및 인터페이스를 이해하여 프로그램의 특성을 파악해야 하며, 이 단계의 주요 산출물로는 요구사항 명세(Requirements specification)가 있다.

- **구조 설계 단계**: 소프트웨어의 전체적인 구조를 결정하는 단계이다. 이를 위하여 시스템을 구성하는 요소(모듈, 데이터베이스 등)들 간의 의존성을 파악하여 상호 연결하는 단계이다. 이 단계의 주요 산출물은 시스템의 전체적인 아키텍처를 보여주는 설계 명세이다.

- **상세 설계 단계**: 이 단계에서는 각 모듈의 알고리즘 세부 사항, 구체적인 데이터 표현, 루틴과 데이터 간의 인터페이스를 결정한다. 이 단계의 주요 산출물은 모듈 명세를 포함하는 상세 설계 명세이다.

- **코딩**: 프로그래밍 언어 등을 사용하여 실제 기계가 해독할 수 있는 형태로 변환하는 단계이다. 이 단계의 주요 산출물로는 프로그램을 들 수 있다.

- **테스팅**: 완성된 시스템의 결함을 검출하기 위해 테스트를 수행하는 단계이다.

기본적으로 폭포수 모델은 사용자의 요구사항이 개발자에게 익숙한 경우나 요구사항 변경이 개발 도중에 빈번하게 이루어지지 않는 경우에 적합한 개발 모형이다.

소프트웨어 테스트 관점에서도 폭포수 모델은 좋은 점이 있다. 개발과정을 거치면서 소프트웨어에 관해 문서와 정보가 많이 산출되므로 코딩이 완료된 후 테스트 작업에 필요한 정보를 쉽게 얻을 수 있다. 그러나 폭포수 모델은 기본적으로 테스트 작업을 코딩 단계 후의 한 단계로만 취급한다. 이는 전체 프로젝트의 비용 및 개발 일정에 심각한 영향을 끼칠 수 있다. 그 이유는 개발 초기에 결함을 발견하였을 때보다 개발이 거의 완료될 무렵에 결함을 발견하여 수정할 때 비용과 시간이 훨씬 많이 들기 때문이다. 만약, 발견된 문제가 시스템의 본질적인 구조 설계상의 문제라면 개발 완료 시점에서 이를 수정하거나 올바르게 변경할 수조차 없는 경우가 발생할 수도 있다.

폭포수 모델이 개발 중심 모델인 데 반해 V-모델은 테스트를 개발과 동등하게 취급한 모델이다. 폭포수 모델은 테스트를 하나의 개발 단계로만 간주하지만, V-모델은 생명 주기를 크게 개발에 관련된 단계들과 테스트에 관련된 단계들로 명확하게 구분하고, 그들 간의 관계를 명시적으로 나타낸다. 그림 6.2는 V-모델을 나타낸 것이다. 그림에서 V의 왼쪽 단계들은 개발 관련 단계(Software Development Life Cycle, SDLC)이고 오른쪽은 테스트 관련 단계(Software Testing Life Cycle, STLC)이다.

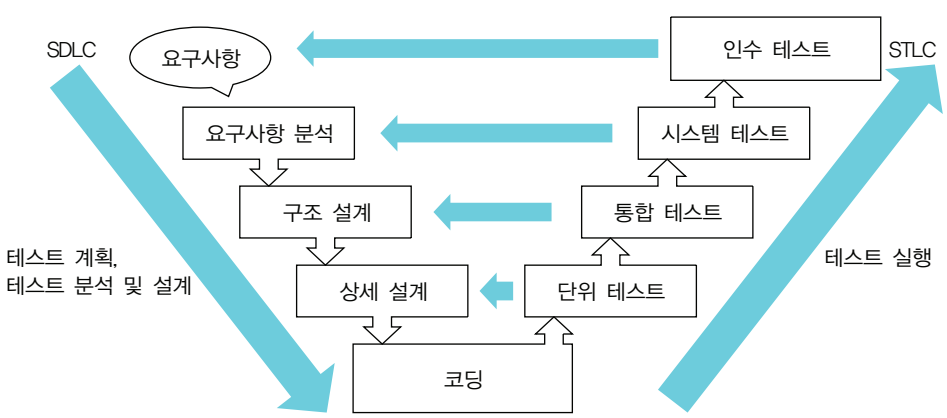

그림 6.2 V-모델

V-모델에서 테스트 활동은 개발이 시작됨과 동시에 시작된다. 개발자가 프로젝트 계획을 세우고 사용자에게서 요구사항을 수집하는 동안 인수 테스트 계획(Acceptance testing plan)을 수립한다. 또한, 시스템 테스트 계획을 수립하기 위해 단위 및 통합 테스트 단계가 종료되길 기다릴 필요가 없다. 그리고 각 개발 단계에서 나오는 산출물을 이용하여 테스트에 필요한 정보를 획득할 수 있다. 예를 들면, 인수 테스트나 시스템 테스트를 위한 테스트 케이스를 사용자의 요구사항에 맞게 개발하고, 통합 테스트를 수행할 때 사용되는 테스트

케이스를 구조 설계도에 맞추어 개발할 수 있다.

V-모델은 각 개발 단계에서 발생하는 결함을 검출하기 위한 테스트 레벨을 명시적으로 보여준다. 단위 테스트에서는 기본적으로 각 모듈이 올바르게 기능을 수행하는지 판별한다. 단위 테스트를 하는 데 구현 정보 및 상세 설계 정보를 이용할 수 있다. 통합 테스트는 모듈 간의 인터페이스를 테스트하는 것이 주목적으로, 모듈 간의 의존관계를 보여주는 구조 설계 문서가 필요하다. 시스템 테스트는 모듈들을 통합하여 완전한 시스템이 구성될 때 개발자가 수행하는 테스트이다. 개발된 시스템이 시스템 명세에 맞게 구현되었는지 시스템 전체를 검사한다. 인수 테스트는 결함 검출보다는 사용자 관점에서 요구사항에 맞게 개발되었는지를 확인하는 테스트이다.

이러한 동적 테스트 외에도 V-모델에서는 개발 산출물에 대한 정적 테스트도 수행된다. 요구사항 명세서, 구조 설계 명세서, 상세 설계 명세서, 그리고 소스 코드 등에 대해서 리뷰와 정적 분석을 수행하여 결함을 검출한다.

심화노트 폭포수 모델과 V-모델

- 폭포수 모델은 개발 중심 모델로, 테스트를 하나의 개발단계로만 간주하는 데 반해, V-모델은 개발과 테스트를 동등하게 보고 명확하게 구분한다.
- V-모델은 개발이 시작됨과 동시에 테스트 계획 및 설계에 필요한 활동이 시작되지만, 폭포수 모델은 코딩 후에 테스트가 수행된다.

테스트 관점에서 V-모델과 폭포수 모델의 가장 중요한 차이점은 무엇일까? 코딩 단계가 완료된 후에 테스트 수행이 이루어진다는 점은 같지만, 폭포수 모델은 모든 테스트 관련 작업이 코딩 후에 이루어지는 반면, V-모델은 테스트 계획이나 테스트 설계 과정이 개발단계와 거의 동시에 시작한다는 점에서 확연히 다르다는 점을 알 수 있다.

V-모델에서 V는 V&V(Verification & Validation)를 의미한다. 여기에서는 'Verification'을 '검증'으로 번역하고, 'Validation'을 '확인'으로 번역한다. 이 두 개념의 차이는 무엇일까? 검증은 시스템이 명세를 만족하는지 검사하는 프로세스를 의미하고, 확인은 시스템이 사용자의 요구사항을 만족하는지 검사하는 프로세스를 의미한다. 즉, 검증은 시스템을 올바르게 만들고 있는지 판별하는 과정(Are we building the system right?)이고, 확인은 우리가 올바른 시스템을 만들고 있는지 판별하는 과정(Are we building the right system?)이다. 따라서 확인은 사용자와 그들의 요구사항을 직접적으로 다루는 반면, 검증

은 구현된 시스템이 해당 명세에 따라 구현되었는지를 본다.

이러한 관점에서 인수 테스트는 사용자 의도대로 시스템이 구현되었는지 판별하는 것이 주목적이므로 대표적인 확인 방법이라 할 수 있다. 반면에 시스템 명세, 구조 설계, 상세 설계는 시스템 테스트, 통합 테스트, 단위 테스트로 각각 검증되며 이 테스트들은 검증 방법에 속한다. 만약, 시스템 명세에 사용자의 운영환경 정보가 반영되어 시스템 테스트에 반영된다면 이 경우에는 시스템 테스트도 확인 방법 중의 하나로 볼 수 있다.

6.2 진화적 개발 모델

현실적으로 사용자의 요구 사항이 프로젝트 시작부터 모두 명확하게 정의되기 어렵고, 시간에 따라 요구사항도 변경되기 마련이다. 순차적 개발 모델은 요구사항이 개발 초기에 완전하게 정의되어 있을 때는 적합하지만, 불확실한 요구사항 또는 요구사항 변경이 빈번하게 발생하는 경우에는 적합하지 않다. 진화적 모델은 이와 같이 요구사항이 명확하지 않은 경우에 적용되는 개발 모델이다.

진화적 개발 모델은 이터레이션(Iteration)과 점진적(Incremental) 개발 원칙에 바탕을 두고 있다. 이 개발 모델은 시스템의 구성요소 중 핵심 부분을 개발한 후, 각 구성요소와 추가 요구사항을 여러 이터레이션을 통해 개선 발전시켜 최종 완성품을 개발한다. 이와 같이 진화적 개발 모델은 많은 이터레이션으로 구성되며 각 이터레이션의 결과물이 고객에게 전달되어 평가받는다. 이 평가를 바탕으로 소프트웨어가 개선된다. 그림 6.3은 진화적 개발 모델을 보여 준다.

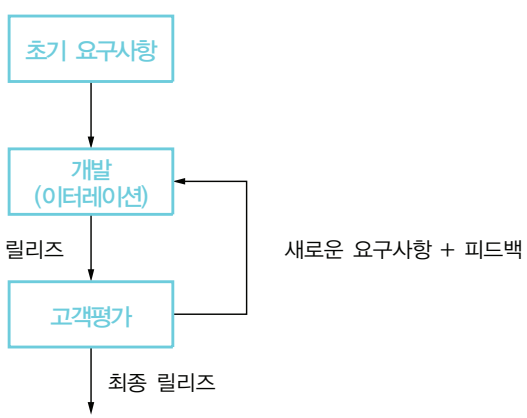

그림 6.3 진화적 개발 모델

진화적 개발 모델은 수행하는 각 이터레이션마다 테스트 수행 계획을 작성하며 이 계획에 따라 테스트를 수행한다. 각 이터레이션은 순차적 모델과 같이 요구사항 분석, 설계, 구현, 테스팅과 같은 단계들로 구성되며 각 개발 단계에서 테스트 관련 프로세스가 수행된다.

요구사항 분석 단계와 설계 단계에서 테스트 대상은 요구사항 문서나 설계 문서이다. 이러한 문서들을 기반으로 위험을 분석하여 위험도에 따라 검토 방법을 선정・수행하거나 경우에 따라서는 도구 지원을 받는 정적 분석을 수행할 수 있다. 이러한 정적 테스트는 동적 테스트에 앞서 수행하여 개발 초기에 결함을 발견하고 수정할 수 있으므로 구현이 완료된 개발 후반부에 결함을 발견하였을 때 초래할 수 있는 위험을 상당히 줄일 수 있다.

코드가 구현되면 컴포넌트(단위) 테스트, 통합 테스트, 시스템 테스트가 수행된다. 물론 경우에 따라 성능 테스트와 같은 비기능 테스트를 수행할 수 있다.

진화적 모델처럼 반복적・점진적으로 개발하는 모델의 예로 나선형 개발 모델을 들 수 있다. 나선형 모델은 요구사항이 개발 초기에 완전하게 정의되어 있지 않고 부분적으로 정의된 경우에 반복적으로 요구사항을 정제하고 확장하는 과정을 사용자가 받아들일 수 있는 완전한 시스템이 개발될 때까지 반복한다.

나선형 모델의 일반적인 방법은 기술적으로 어렵거나 고객의 비즈니스 가치를 최상으로 만드는 요구사항들에 대해 우선 프로토타입을 개발하고, 프로토타입에 대한 테스트 및 사용자의 평가를 거쳐 다음 개발 주기를 시작한다(그림 6.4 참조). 이때 필요에 따라 이전 주기에 개발했던 프로토타입이 기능적으로 더욱 확장되거나 아예 폐기되어 새로운 프로토타입이 개발될 수도 있다. 또한, 새로운 개발 주기가 시작될 때마다 위험 분석을 수행하기 때문에 잠재적인 위험 분야를 파악하여 해결해 나갈 수 있다. 이런 과정을 반복하여 시스템이 제공해야 하는 기능들이 파악되는 시점에서 V-모델에 따라 시스템을 개발한다. 즉, 나선의 각 타원에서 한 가지 모델만 채택할 필요는 없으며 생명 주기 모델들을 여러 개 혼합하여 개발할 수 있다. 이러한 점 때문에 나선형 모델을 메타 생명 주기 모델이라고도 한다.

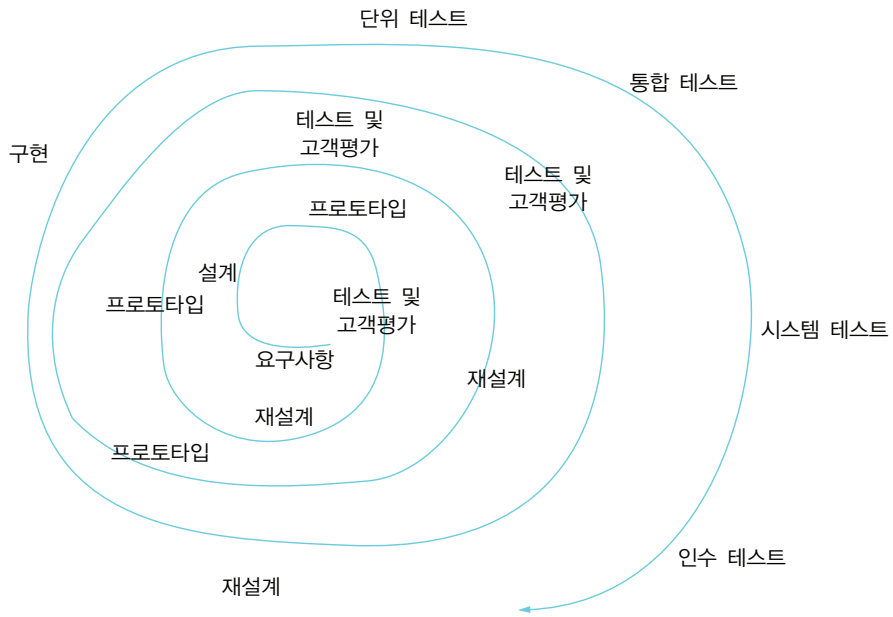

그림 6.4 나선형 모형

나선형 모델은 매 단계에서 적당한 테스트가 이루어지므로 개발 과정에서 발생하는 많은 문제점을 해결할 수 있는 기회를 제공한다. 또한, 나선형 모델은 나선의 한 타원으로 표현되는 소프트웨어 프로세스 한 주기마다 사용자에게 피드백을 받기 때문에 폭포수 모델처럼 개발이 거의 완료가 된 후에야 심각한 결함이 발견되는 문제가 생길 가능성이 거의 없을 뿐만 아니라 다음 개발 주기에 수행되는 설계나 테스트에 고객 평가를 반영할 수 있는 자료를 획득할 수 있다.

6.3 애자일 개발 모델

'Agile'은 가볍고 민첩하다는 사전적 의미가 있다. 이는 폭포수 모델처럼 문서 중심의 매우 복잡한, 또는 프로세스 위주의 방법론과 대치되는 개념이다. 애자일 방법론이 무엇이다라고 정의하는 것보다 애자일 방법론이 추구하는 가치를 먼저 살펴보도록 하자.

- 사람 및 상호 의사 교환이 프로세스나 도구보다 우선한다.
- 동작하는 소프트웨어가 포괄적인 문서보다 우선한다.
- 고객과의 협력이 계약 협상보다 우선한다.
- 변화에 반응하는 것이 계획을 따르는 것보다 우선한다.

위의 네 가지 가치는 애자일 선언(Agile Manifesto)이라고도 하며 애자일 개발 방법론이 추구하는 가치와 기존 개발 방법론과의 차이점을 극명하게 보여준다. 이와 같은 가치를 추구하기 위해 애자일 개발 방법론은 진화적 개발 모델과 같이 반복적이면서 점진적인 개발 접근 방식(Iterative and Incremental Development, IDD)을 따른다.

IDD는 소프트웨어 개발 주기를 여러 개의 이터레이션으로 구분한다. 각 이터레이션은 요구분석, 설계, 구현 및 테스트와 같은 활동들로 구성된 소규모 프로젝트로 생각할 수 있다. 각 이터레이션이 종료되면 부분적으로 완성된 시스템이 산출된다. 그러나 이터레이션에서 산출된 시스템은 내부 개발자가 관리하는 것이며, 사용자에게 외부적으로 릴리즈되는 것은 최종 반복 주기의 산출물이다. 소프트웨어 개발 주기를 구성하는 각 반복 주기는 보통 1주에서 4주이며 주기마다 새로운 요구사항이 추가되어 개발(Incremental development)된다(그림 6.5 참조).

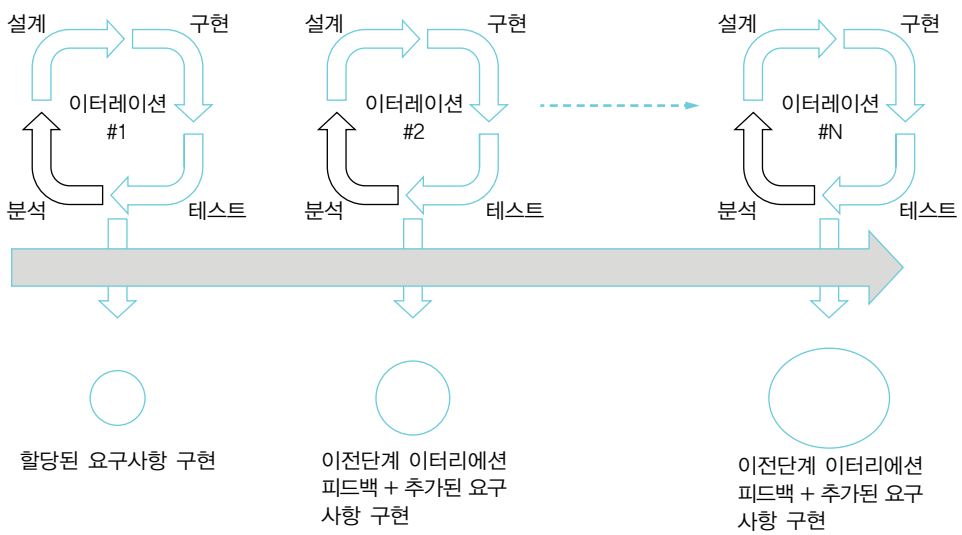

그림 6.5 애자일 방법에서의 IID

이와 같이 이터레이션이 짧기 때문에 실제 요구사항 정의부터 시작하여 개발까지 오랜 시간이 걸리는 순차적 개발 모델과는 달리 고객이 실제 동작하는 소프트웨어를 빠르게 볼 수 있어 일의 진척 정도를 눈으로 확인할 수 있는 기회가 많다(동작하는 소프트웨어가 포괄적인 문서보다 우선한다).

각 이터레이션에서 개발할 요구사항은 고객이 각 반복주기 시작 전에 선택하는데 일반적으로 고객에게 가장 높은 비즈니스 가치를 가져다주는 요구사항에 우선순위를 높게 둔다

(고객과의 협력이 계약 협상보다 우선한다). 각 반복주기가 시작할 때마다 이러한 요구사항을 선정하므로 자주 변경될 수 있는 고객의 요구사항을 탄력적으로 처리할 기회를 제공한다(변화에 반응하는 것이 계획을 따르는 것보다 우선한다). 물론 일단 이터레이션이 시작되면 이제 요구사항 변경은 받아들이지 않아야 한다.

애자일 선언에서 가장 첫 번째로 나오는 가치는 '사람 및 상호 의사 교환이 프로세스나 도구보다 우선한다.'이다. 이것은 프로그래밍은 본질적으로 사람의 활동이라는 사실을 상기하게 한다. 대표적인 애자일 방법의 하나인 XP(eXtreme Programming)는 지속적인 개발을 위해 프로그래머가 과도한 작업을 피하는 것이 매우 중요하다는 사실을 강조한다. 즉, 프로그래머의 과도한 작업이 오히려 역효과를 불러일으킨다는 점을 인지하고 있는 것이다.

또한, 연구 결과에 따르면 개인에 따라 많게는 10배까지 생산성의 차이가 있다는 사실이 밝혀졌다. 이것이 아주 능력이 출중한 프로그래머를 고용해야 한다는 사실을 의미하는 것은 아니다. (보통 이러한 프로그래머는 어느 한 곳에서 지속적으로 개발하는 성향을 보이지 않는다.) 대신에 지속적인 교육과 개발자의 멘토링에 더 많은 가치를 두어야 한다는 의미이다. XP는 짝 프로그래밍(Pair Programming)을 통해 개발자 간의 지식 전달 및 공유를 꾀하고 있다.

실화노트 짝 프로그래밍

XP의 실현 방안(Practice) 중 하나로 개발자 두 명이 짝을 이루어 코딩하는 방식이 있다. 짝 중 한 명이 코드를 입력하는 동안 다른 한 명은 코드를 검토하는 등의 작업을 수행한다. 즉, 한 명이 키보드를 잡고 있을 때 다른 한 명은 뒤에서 작업 상황을 실시간으로 지켜보게 된다. 언뜻 보면 한 명이 할 일을 두 명이 매달리니까 업무 효율이 다소 떨어질지도 모르고, 또 다른 일을 할 수 있는 한 명의 인력을 낭비하는 것처럼 보이기까지 한다.

그러나 이러한 짝 프로그래밍에는 많은 이점이 있다. "Two are better than one."이라는 격언을 상기하라. 두 명의 머리가 한 작업에 집중을 하므로 시너지 효과가 발생한다. 작업하는 도중에 생겨나는 아이디어도 두 배가 될 것이고, 두 명이 각각 가지고 있는 경험이 한 코드에 녹아들게 된다. 또한, 키보드를 잡은 사람이 저지르게 되는 코딩상의 사소한 실수도 다른 한 명이 발견하여 코딩 후 컴파일 시 일어나는 문법상 결함을 제거하느라 소모되는 시간도 줄일 수 있다. 그리고 언젠가는 그 작업물이 팀에서 공유되어야 하는데 그때 다시 다른 사람이 그 코드를 보며 이해하려고 노력하는 시간을 줄일 수 있다. 게다가 혼자서 작업하면 딴 길로 새는 경우가 많은데 뒤에서 누가 지켜보고 있기 때문에 작업 집중도가 높아질 수밖에 없다.

애자일 방법론은 소프트웨어 테스트를 매우 강조한다. XP에는 소프트웨어 테스트에 관련된 중요한 개념이 있다. 바로, 테스트 주도 개발(Test-Driven Development, TDD)이라는 개념이다. TDD는 프로그램에 대한 테스트 케이스를 먼저 작성하고, 이 테스트 케이스로 테스트 되는 실제 프로그램의 코드를 나중에 작성하는 방식이다. 이 방식을 사용하면 테스트 되지 않는 코드가 없어 결함의 발생 가능성을 상당히 줄일 수 있다. TDD는 코드 설계에도 영향을 준다. 테스트를 먼저 고려해서 코딩하기 때문에 테스트 용이성이 높은 코드가 산출되며 변경 요구에 쉽게 대응할 수 있도록 설계가 되는 효과가 있다. 또한, 테스트 케이스는 대상 테스트 모듈을 사용하는 입장에서 작성하게 되기 때문에 테스트 케이스 자체가 요구 사항을 분명하게 드러나게 해주는 효과가 있다.

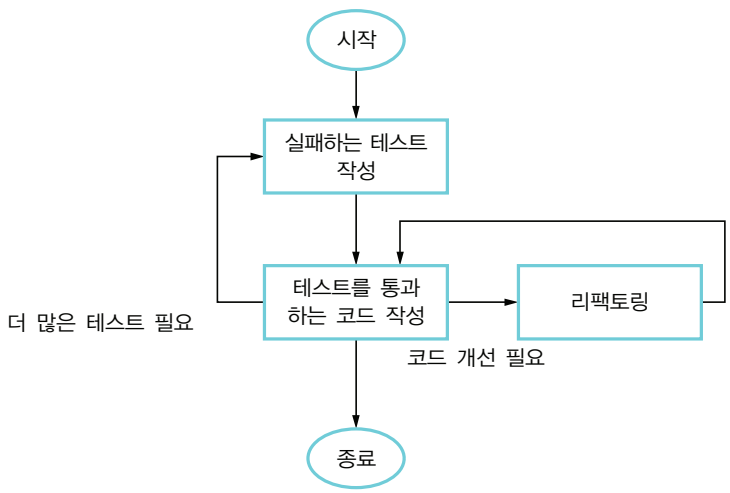

그림 6.6 TDD 프로세스

그림 6.6은 TDD의 작업 흐름을 보여준다. TDD에서 가장 첫 번째로 수행하는 작업은 테스트 케이스를 작성하는 것이다. 물론 이 테스트는 실패할 것이다. 테스트가 실패하는 이유는 테스트할 대상이 아무것도 작성되어 있지 않기 때문이다.

다음 단계는 테스트를 통과하는 코드를 작성하는 것이다. 필요하다면 TDD를 수행하는 도중에 리팩토링(Refactoring)을 수행한다. 리팩토링은 기능을 변경하지 않고 코드의 내부 구조를 개선하는 작업이다. 코딩을 하다 보면 중복된 부분이나 로직이 복잡해질 위험성은 언제든지 발생할 수 있으므로 TDD를 수행하는 중간 중간에 리팩토링을 수행하여 더욱 단순하게 코드를 만들어 줄 필요가 있다. 모든 항목이 구현될 때까지 이러한 과정을 반복한다.

론 제프리스(Ron Jeffries)는 TDD를 수행하면 코드가 단순해지고, 이러한 코드를 "동작

하는 깨끗한 코드(Clean code that works)"라고 하였다. 여기에서 「동작한다」는 의미는 테스트를 통과하는 코드란 의미이며 「깨끗하다」라는 의미는 리팩토링을 수행하여 중복된 코드가 없음을 의미한다.

심화노트 TDD

TDD에 대한 이해를 돕기 위해 아래에 나오는 간단한 예시로 설명하려고 한다(물론 TDD를 지원하는 테스트 프레임워크로 JUnit과 같은 도구를 사용할 수 있지만, 여기에서는 개념적 설명을 위해 이러한 도구에 관한 설명은 생략한다).
기본적으로 TDD는 단위 테스트를 위한 것이다. 객체 지향 프로그램에서 단위는 보통 하나의 클래스를 의미한다. 예를 들어, 일련의 정숫값들을 더하는 클래스를 구현한다고 가정하자. 이를 위해 다음과 같은 (JUnit)테스트 케이스를 작성하였다(TDD는 코드를 구현하기 전에 테스트 케이스를 작성한다는 사실을 명심하라).

```
        @Test
void testAdd() {
          SumInt s=new SumInt(); //SumInt는 개발할 클래스이다.
}
```

현재 작성되어 있는 코드가 없기 때문에 이 테스트 케이스를 실행하면 컴파일되지 않는다. 우선 컴파일만 될 수 있게 최소한의 코드만 작성해보자. 컴파일 오류를 피하기 위해서는 SumInt 클래스를 정의해야 한다.

```
class SumInt {
}
```

테스트가 통과되었으므로 실패하는 테스트를 추가하자.

```
@Test
void testAdd() {
          SumInt s = new SumInt(); // SumInt는 개발할 클래스이다.
          int r = s.add(10, 20);
}
```

실제 다시 한번 테스트를 실행시켜보면 add 함수가 선언되어 있지 않아 컴파일 오류가 발생한다. 테스트를 통과하기 위해서는 SumInt 클래스에 add 함수를 정의하여야 한다.

```
class SumInt {
public int add(int x, int y) { return 0;} // 0은 기본 반환값
}
```

컴파일 오류는 더는 발생하지 않기 때문에 실패하도록 테스트 코드를 추가하자.

```
@Test
void testAdd() {
          SumInt s = new SumInt(); //SumInt는 개발할 클래스이다.
          int r = s.add(10, 20);
          assertEquals(30, r);
}
```

테스트를 수행하면 단정문 "assertEquals(30, r)"을 통과하지 못함을 알 수 있다. 단정문을 통과하기 위한 코드를 다음과 같이 작성해보자.

```
class SumInt {
public int add(int x, int y) { return x+y; }
}
```

다시 테스트를 실행하면 이제 더는 오류가 검출되지 않을 것이다.

지속적 통합(Continuous Integration, CI)은 TDD와 더불어 애자일 개발에서 중요한 실천 규칙이다. 말 그대로 지속적 통합은 통합이 어느 한 시점에 이루어지는 것이 아니라 지속적으로 통합을 수행하는 것을 말한다. 그렇다면 얼마나 자주 통합이 이루어져야 '지속적'이 될 것인가? 지속적 통합에서는 각 개발자가 작업한 코드의 업데이트를 코드 저장소에 반영할 때마다 통합이 이루어진다. 하루에도 몇 번씩 이루어질 수 있다는 의미이다.

지속적 통합의 개념 안에는 코드 통합 작업뿐만 아니라 코드 품질을 평가하는 테스트와 같은 여러 품질관리 활동들이 포함되어 있다. 지속적 통합은 통합이 빈번하게 이루어질 뿐만 아니라 통합되었을 때 즉시 잠재적인 문제가 있는지 바로 결과를 알 수 있기 때문에 소프트웨어 품질을 높이고 통합의 위험을 줄이는 대표적인 방법이며, 지속적으로 품질을 관리하는 방법으로 볼 수 있다. 지속적 통합은 다음과 같은 이점이 있다.

- 통합 지연에 따른 비용 증가를 막을 수 있다.
- 빠른 결함 발견으로 비용을 감소할 수 있다. (결함이 늦게 발견될수록 비용이 증가한다.)
- 항상 빌드 가능한 소프트웨어 버전이 있기 때문에 소프트웨어 품질에 대한 확신이 있다.