제 3 장 소프트웨어 개발 단계와 테스트

3.1 개 요

소프트웨어 테스트는 다음과 같이 매우 다양한 방법으로 분류할 수 있다(표 3.1 참조).

- 테스트 레벨에 따른 테스트 분류
- 테스트 설계 방식에 따른 분류
- 테스트 유형(품질 특성)에 따른 분류

표 3.1 소프트웨어 테스트의 분류

분류	테스트 종류		설 명
테스트 레벨	컴포넌트/단위 테스트		각각의 컴포넌트를 테스트한다.
	통합 테스트		컴포넌트 간의 인터페이스를 테스트한다.
	시스템 테스트		전체 시스템이 목적을 만족시키는지 테스트한다.
	인수 테스트		사용자의 요구사항을 만족하는지 확인한다.
테스트 설계	동적 테스트	명세 기반 테스트	명세를 바탕으로 테스트 케이스를 생성한다.
		구조 기반 테스트	프로그램 코드 정보를 바탕으로 테스트 케이스를 생성한다.
		경험 기반 테스트	테스터의 경험을 기반으로 테스트 케이스를 생성한다.
	정적 테스트	리뷰	산출물에 존재하는 결함을 검출하거나 프로젝트의 진행 상황을 점검한다.
		정적 분석	자동화된 도구를 이용하여 산출물의 결함을 검출하거나 복잡도를 측정한다.

테스트 유형	기능 테스트		기능적 요구사항 측면의 결함 검출 및 충족 여부를 확인한다.
	비기능 테스트	기능 적합성 테스트	사용자의 요구사항을 만족하는 기능이 제공되는 정도를 테스트한다.
		성능 효율성 테스트	시스템의 응답시간이나 처리량을 테스트한다.
		호환성 테스트	다른 시스템과의 상호 연동 능력이나 공존성을 테스트한다.
		사용성 테스트	사용자가 이해하고 배우기 쉬운 정도를 테스트한다.
		신뢰성 테스트	규정된 조건/기간에 오동작 없이 수행하는 능력을 테스트한다.
		보안성 테스트	시스템의 정보 및 데이터를 보호하는 능력을 테스트한다.
		유지보수성 테스트	소프트웨어 유지보수의 용이성을 테스트한다.
		이식성 테스트	다양한 플랫폼에서 운영될 수 있는 능력을 테스트한다.

이 장에서는 테스트 레벨에 따른 테스트 방법에 대해 살펴본다. 컴포넌트(단위) 테스트, 통합 테스트, 시스템 테스트, 인수 테스트는 목적이나 방법이 서로 다르므로 각각에 대해 자세하게 알아볼 필요가 있다.

그림 3.1은 컴포넌트 테스트, 통합 테스트, 시스템 테스트 레벨의 각 레벨별 테스트 대상을 보여준다.

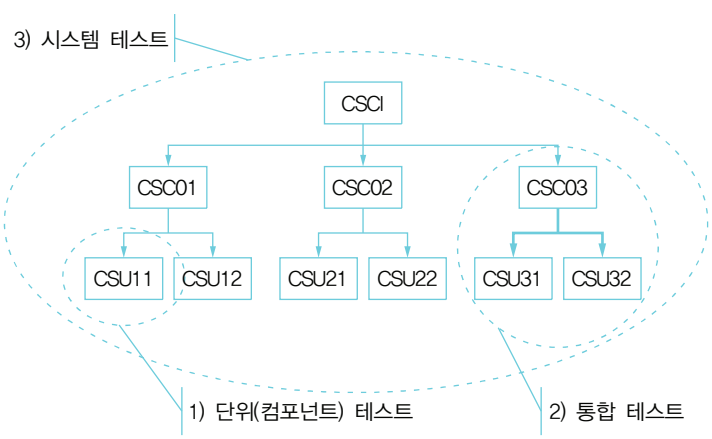

그림 3.1 테스트 레벨과 테스트 대상

이 그림에서 CSCI, CSC, CSU 등은 시스템을 구성하는 주요 컴포넌트를 나타낸다. 시스템은 컴포넌트 간의 연동으로 동작하며, 이 그림에서는 각 컴포넌트 간의 연동이 화살표로 표현되어 있다. 예를 들어, CSCI 컴포넌트는 CSC01, CSC02, CSC03 컴포넌트를 호출한다.

시스템이 구성되면 이를 바탕으로 우리는 컴포넌트 테스트, 통합 테스트, 그리고 시스템 테스트의 대상을 결정할 수 있다. (참고로, 이런 구조도를 아키텍처 설계도라고 부른다) 따라서, 만약 우리가 시스템에 대한 테스트 레벨과 테스트 대상을 결정하고자 한다면 우선 위와 같이 간단한 형태라 하더라도 시스템의 구조가 어떤 모습인지를 알아야 한다.

그림 3.1을 기준으로 한다면 컴포넌트 테스트를 수행할 때 각 개별 컴포넌트가 테스트 대상이 된다. 그리고 통합 테스트는 화살표로 표시된 연동 관계가 있는 컴포넌트 간의 연결이 테스트 대상이 되고, 시스템 테스트를 수행할 때는 전체 시스템 자체를 테스트 대상으로 삼는다.

표 3.2는 각 테스트 레벨별 테스트 대상을 예시로 보여준다. 컴포넌트 테스트는 각 컴포넌트를 대상으로 하므로 테스트 대상의 수는 총 10개이고, 통합 테스트는 컴포넌트 간의 연동을 대상으로 하므로 총 9개가 테스트 대상이 된다. 시스템 테스트는 시스템 자체이므로 1개가 테스트 대상이다.

표 3.2 테스트 레벨별 테스트 대상 예

테스트 레벨	테스트 대상 수	테스트 대상
컴포넌트 테스트	10	CSCI, CSC01, CSC02, CSC03, CSU11, CSU12, …
통합 테스트	9	CSCI → CSC01, CSCI → CSC02, CSCI → CSC03 CSC01 → CSU11, CSC01 → CSU12, …
시스템 테스트	1	시스템 자체

3.2 컴포넌트 테스트

3.2.1 개요

컴포넌트(단위) 테스트는 개별적인 모듈(또는 컴포넌트)의 테스트를 말하며, 구현 단계에서 각 모듈을 구현한 후에 수행한다(물론 테스트 주도 개발처럼 코드가 개발되기 전에 테스트 케이스를 먼저 생성할 수도 있다). 개별적인 모듈에 대해 컴포넌트 테스트를 수행하려

면 모듈을 단독으로 실행할 수 있는 환경이 필요하다. 테스트 환경은 테스트 베드(Test bed)라고도 한다. 테스트 환경의 주요 구성 요소로 테스트 드라이버(Driver)와 테스트 스텁(Stub)이 있다.

테스트 드라이버와 스텁에 대한 이해를 돕기 위하여 3개의 컴포넌트로 구성된 시스템을 대상으로 컴포넌트 테스트를 그림 3.2와 같이 표현하였다. (b), (c), (d)에서 컴포넌트 0, 컴포넌트 1, 컴포넌트 2는 각각 테스트 대상이다.

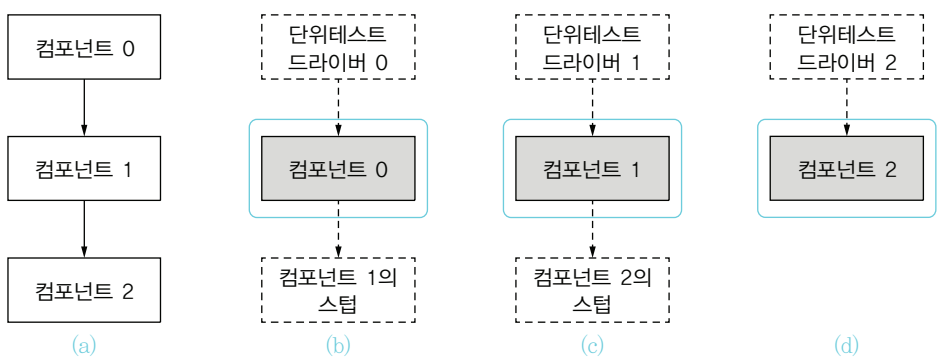

그림 3.2 컴포넌트 테스트를 위한 테스트 환경

컴포넌트 테스트는 시스템을 구성하는 컴포넌트를 독립적인 방식으로 테스트를 수행하는 것이 특징이다. 예를 들어, 컴포넌트 0을 테스트할 때 컴포넌트 1 및 컴포넌트 2에 의존하지 않는 방식으로 컴포넌트 테스트가 수행된다. 그러므로 그림 3.2의 (b)에서 볼 수 있듯이, 컴포넌트 0을 대상으로 하여 컴포넌트 테스트를 수행하는 경우, 컴포넌트 1을 호출하는 대신에 이에 대한 스텁을 사용한다. 마찬가지로 컴포넌트 1을 테스트 대상으로 할 때는 컴포넌트 2 대신 스텁을 이용한다. 이런 방식을 통해서 개별적인 컴포넌트의 독립적인 동작을 확인할 수 있다.

3.2.2 모의 객체 생성 프레임워크

객체 지향 프로그램에서는 컴포넌트 테스트를 수행할 때 테스트 되는 메소드가 다른 클래스의 객체에 의존할 수 있다. 이런 경우에는 메소드를 고립화하여 테스트하는 것이 불가능하다. 따라서 독립적인 컴포넌트 테스트를 위해서는 절차 지향적 프로그래밍에서 스텁과 같은 개념이 필요하다. 모의(Mock) 객체는 스텁의 객체 지향 버전이라 할 수 있다.

모의 객체는 개발자가 처음부터 수작업으로 만들거나 모의 객체 생성 프레임워크를 이용하여 만들 수 있다. 여기에서는 Mockito 모의 객체 생성 프레임워크를 이용하여 모의 객체를 만들어 컴포넌트 테스트를 수행하는 방법에 대해 간략하게 살펴본다.

예를 들어, 인자로 은행의 계좌 정보(Account 객체)를 사용하는 클래스 Foo의 메소드 perform()을 테스트하는 상황을 생각해보자. Account 인터페이스는 그림 3.3에 정의되어 있다.

```
interface Account {
              Money balance();
              boolean withdraw(Money m);
              void deposit(Money m);
}
```

그림 3.3 Account 인터페이스

메소드 perform()을 수행하는 데 500달러가 소요된다고 가정하면 은행 잔고가 최소 500달러 이상이어야 한다는 사실을 알 수 있다. 실제의 계좌 객체는 은행과 네트워크로 통신하여 인터페이스에 정의된 기능을 수행해야 하지만 모의 객체를 이용하면 실 객체 대신에 여러 상황을 별다른 어려움 없이 간단하게 테스트할 수 있다.

그림 3.4는 perform() 메소드가 정상적으로 동작하는 시나리오를 고려하여 작성한 JUnit 테스트 케이스이다:

```
@Mock
Account acctMock;
@InjectMocks
Foo foo;
@Test
public void testPerformInFoo() {
   1: MockitoAnnotations.initMocks(this);
   2: when(acctMock. balance()).thenReturn((Money)850);
   3: foo.perform();
   4: verify(acctMock, times(1)).balance();
   5: verify(acctMock, times(1)).withdraw((Money)500);
}
```

그림 3.4 모의 객체를 이용한 Junit 테스트 케이스

테스트 케이스는 크게 생성, 동작, 확인 세 부분으로 구성된다. 생성은 1번, 2번 문장에 해당하는데, 테스트에 필요한 상황을 생성하는 부분(Setup 부분)이고 fixture라고도 한다. 이 부분에서는 테스트에 필요한 객체를 생성하거나 자원을 할당한다. 1번 문장은 테스트 대상이 되는 객체와 계좌 모의 객체를 생성한다. 그리고 생성된 모의 객체(AcctMock)를 테스트 대상이 되는 객체(Foo)에 주입한다. 2번 문장은 모의 객체의 행위를 지정한다. 메소드 balance()가 호출되었을 때 850달러를 반환하도록 작성되어 있다.

두 번째 부분인 동작은 실제 테스트하고자 하는 기능을 호출하고, 3번 문장이 이에 해당한다. 3번 문장은 Foo 클래스의 perform() 메소드를 호출한다.

마지막으로 4번, 5번 문장은 확인에 해당한다. 여기서는 동작의 실행 결과와 기대 결과를 비교한다. 만약, 테스트 케이스에서 이 부분이 빠지면 테스트 케이스 역할을 전혀 수행하지 못할 것이다. 메소드 perform()이 실행되면 계좌 객체에 대해 balance()가 호출되고 balance()가 실행된 후에는 실제 예금 인출이 이루어져야 하므로 withdraw() 함수가 호출된다. 만약, 이렇게 예상된 함수 호출이 이루어지지 않으면 결함이 있는 것이다.

심화노트 모의 객체

모의 객체는 다음과 같이 분류하기도 한다.

- 더미(Dummy) 객체는 테스트할 때 객체만 필요하고 해당 객체의 기능까지는 필요하지 않은 경우에 사용된다. 더미 객체의 메소드가 호출되면 정상 동작은 수행하지 않고 예외를 던진다.
- 테스트 스텁(Stub)은 더미 객체에 단순한 기능성을 작성하며 추가객체의 특정 상태를 가정해서 특정한 값을 리턴하거나 특정한 메시지를 출력하게 한다.
- 테스트 스파이(Spy)는 주로 테스트 대상 클래스(CUT)와 협력하는 클래스로 가는 출력을 검증하는 데 사용하며 CUT가 실행되는 동안 특정 협력 클래스로의 호출(또는 호출의 결과)을 잡아내 실행이 끝난 후 정상 호출되었는지 검사한다.
- 가짜(Fake) 객체는 실제 협력 클래스의 기능을 대체해야 할 경우에 사용하며 실제 협력 클래스의 기능 중 전체나 일부를 훨씬 단순하게 구현한다. 실제 협력 클래스가 구현되지 않았거나 너무 느리거나 테스트 환경에서는 사용할 수 없을 때 가짜 객체를 사용한다.

모의 객체는 앞서 기술한 모든 형태를 포함하는 의미로 사용된다.

3.2.3 FIRST 원칙

컴포넌트 테스트를 잘 수행하는 것은 매우 중요하다. 컴포넌트 테스트는 통합 테스트나 시스템 테스트, 인수 테스트보다 쉽게 수행할 수 있으며 테스트 수행에 따른 피드백이 빠르다. 또한, 결함이 발견되었을 때 결함을 발생시키는 부분을 쉽게 식별하여 수정할 수 있으

므로 컴포넌트 테스트 케이스를 잘 설계하고 수행하여야 한다.

컴포넌트 테스트를 잘 수행하기 위한 FIRST 원칙이 있다. 이 원칙은 다음과 같은 5가지 원칙으로 구성되어 있다.

- Fast: 컴포넌트 테스트는 빠르게 수행되어야 한다. 이 원칙은 매우 중요하다. 예를 들면, 500개의 컴포넌트 테스트가 있을 때 각 테스트가 실행되는 데 0.5초가 걸린다면 컴포넌트 테스트 모두를 수행하는 데 약 4분이 넘게 소요된다. 컴포넌트 테스트의 주요 목적 중 하나가 리그레션 테스트인데, 코드를 변경할 때마다 4분 넘는 테스트가 수행된다면 개발자는 테스트를 수행하지 않고자 하는 유혹에 빠질 수 있고, 코드에 새로운 기능이 추가될수록 테스트 실행 시간은 증가될 것이다. 결국, 이는 코드의 품질에 심각한 영향을 주게 된다. 특히, 데이터베이스나 네트워크처럼 외부 자원에 의존하는 경우에는 테스트 수행 시간이 길어질 가능성이 크다. 이 경우에는 모의 객체 생성 프레임워크를 이용하여 테스트 시간을 단축하는 것이 좋다.

- Isolated: 컴포넌트 테스트가 다른 컴포넌트 테스트에 의존하지 않도록 해야 한다. 어떤 특정한 컴포넌트 테스트 집합이나 컴포넌트 테스트 하나를 독립적으로 수행할 수 있어야 한다. 만약, 한 컴포넌트 테스트가 다른 컴포넌트 테스트의 수행 결과에 의존한다면 테스트 실행 순서에 따라 다른 결과가 나올 것이다. 즉, 전체 컴포넌트 테스트 집합을 실행한 결과와 개별적으로 컴포넌트 테스트를 실행한 결과가 달라서는 안 된다.

- Repeatable: 테스트를 몇 번 실행해도 동일한 결과가 나오도록 해야 한다. 만약 테스트를 실행할 때마다 다른 결과가 나온다면 더 이상 테스트 결과를 신뢰하지 않고 테스트를 수행하지 않게 될 것이다. 이 문제는 테스트가 랜덤 함수나 시간 또는 날짜에 의존하는 경우에 발생할 수 있다. 또한, 데이터베이스를 사용하여 테스트를 하는 경우도 주의해야 한다. 만약, 데이터베이스의 내용이 이미 다른 개발자가 수행한 테스트에 의해 변경되었을 때 이 변경으로 인하여 테스트의 결과가 달라질 수 있다. 개발자 자신만이 사용할 수 있는 샌드박스를 구축하여 테스트를 수행하는 것이 좋다.

- Self-Validating: 사람의 개입 없이 테스트가 통과되었는지 알 수 있도록 작성해야 한다. 테스트 결과를 판단하기 위하여 사람이 개입하도록 작성하면 매우 많은 시간이 소요될 여지가 있고 많은 위험이 따른다. 또한, 테스트에 필요한 데이터 준비 작업이나 설정 파일 조작과 같은 작업도 자동화하여 사람이 개입할 필요가 없는 것이 좋다.

• Timely: 컴포넌트 테스트는 제때 수행되어야 한다. 여기서 제때란 테스트 대상이 되는 코드가 작성되는 시점을 의미한다. TDD에서는 코드 작성 바로 전이다.

3.3 통합 테스트

3.3.1 개요

통합(Integration) 테스트는 컴포넌트를 통합하는 과정에서 수행되는 테스트이다. 컴포넌트 테스트는 개별적인 모듈・컴포넌트의 기능이 올바르게 작동하는지 테스트하는 반면에 통합 테스트는 컴포넌트 간의 상호 연동이 제대로 수행되는지 검사하는 테스트이다. 개별적인 컴포넌트에 대해 테스트가 수행되었더라도 실제로 컴포넌트들을 통합한 후에 결함이 발생할 수 있다.

통합 테스트에서는 서로 다른 컴포넌트가 통합되어 호출된다. 그림 3.5는 3개의 컴포넌트에 대한 통합 테스트 방법을 나타낸 것이다. (a)에서는 컴포넌트 1과 컴포넌트 2를, (b)에서는 3개의 컴포넌트를 모두 통합한 모습을 보여 준다.

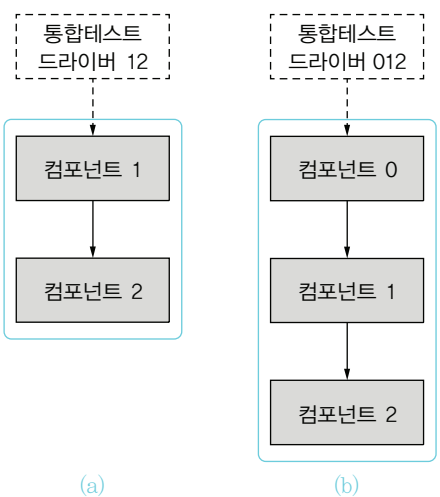

그림 3.5 통합 테스트

통합 테스트를 수행할 때는 각 테스트별로 통합 대상이 되는 컴포넌트가 먼저 결정되어야 한다. 예를 들어, 그림 3.5의 (a)에서는 컴포넌트 1과 컴포넌트 2가 통합 대상이고, 그림

3.5의 (b)에서는 컴포넌트 0, 컴포넌트 1, 컴포넌트 2가 통합 대상이 된다.

통합 테스트의 목적에 대해서는 두 가지 다른 관점이 있다. 자료에 따라 통합 테스트가 두 컴포넌트 간 연결의 정확성에만 초점을 두기도 하고, 연결된 두 컴포넌트의 기능적인 측면에 초점을 두기도 한다.

그림 3.6은 컴포넌트 간의 상호작용에만 초점을 두고 수행하는 통합 테스트의 예시이다. 이 통합 테스트는 컴포넌트 1과 컴포넌트 2를 대상으로 수행된다. 하지만 실제 테스트는 음영으로 표시한 것과 같이 컴포넌트 1의 Output1과 컴포넌트 2의 Input2 사이의 데이터 연결에만 초점을 두고 있다. 즉, 컴포넌트 1에서 컴포넌트 2로 전송한 데이터가 누락되는지, 일부 변경이 발생하는지, 데이터의 전달 순서가 변경되는지 등에 대한 테스트를 수행하는 것이다.

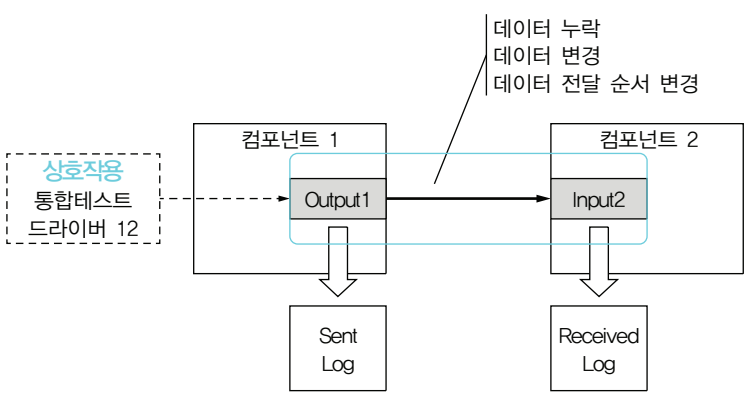

그림 3.6 상호작용에 초점을 둔 통합 테스트

이와 같이 상호작용에만 초점을 두고 통합 테스트를 수행할 때는 두 컴포넌트 간에 전송·수신된 데이터를 중심으로 본다. 그러므로 그림에서 볼 수 있듯이 컴포넌트 1의 데이터를 전송하는 Output1에서 전송 데이터에 대한 로그를 생성하고, 컴포넌트 2의 데이터를 수신하는 Input2에서 수신 데이터에 대한 로그를 생성하여 이 두 로그의 값을 비교하는 방식으로 두 컴포넌트의 상호작용 적합성을 확인할 수 있다.

그러나 시스템에 따라서 전송·수신 로그를 생성하는 것이 용이하지 않을 수도 있다. 예를 들어, 임베디드 소프트웨어는 로그를 기록하기 위한 메모리와 파일시스템 공간이 제한적이므로 로그 크기를 최소화해야 한다. 이런 경우에는 실제 전달되는 데이터 전체를 로그로 남기는 대신에 핑거 프린트(Finger print)를 사용해서 데이터의 변경 여부만을 판단하기

위한 최소한의 로그를 생성할 수 있다.

그림 3.7은 다른 관점의 통합 테스트를 보여준다. 이 통합 테스트는 컴포넌트 간 데이터 전달의 적합성만을 보는 것이 아니라 연결된 두 컴포넌트의 기능적인 측면에서 적합성을 확인한다. 이 경우에는 컴포넌트 1과 컴포넌트 2 전체가 테스트 대상이라고 볼 수 있다.

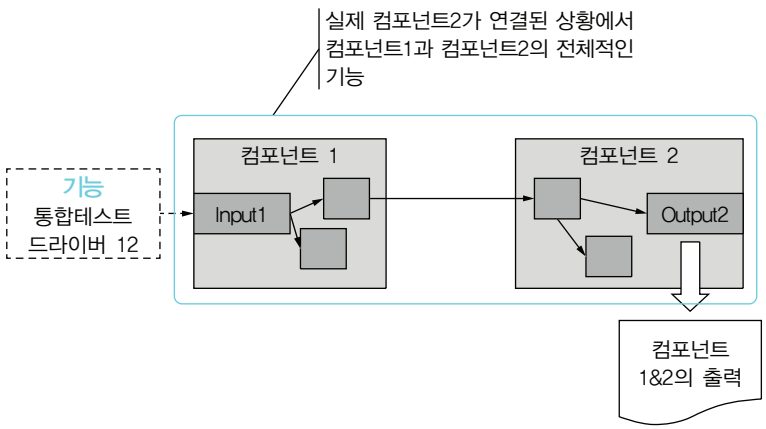

그림 3.7 기능에 초점을 둔 통합 테스트

컴포넌트 테스트와 비교해 보면, 컴포넌트 1을 대상으로 컴포넌트 테스트를 수행할 때는 컴포넌트 2를 대신해서 스텁을 사용했지만, 통합 테스트에서는 실제 컴포넌트 2를 컴포넌트 1과 연결해서 테스트를 수행하는 것이다. 그래서 컴포넌트 1을 동작시키면 컴포넌트 2가 동작되고 컴포넌트 1과 컴포넌트 2의 종합적인 동작 결과를 확인할 수 있다.

첫 번째 관점과 같이 상호작용에 초점을 둘 것인지, 아니면 두 번째 관점과 같이 기능에 초점을 둘 것인지는 상황에 따라 달라질 수 있다. 만약, 컴포넌트 간의 연결 자체에서 오작동이 발생할 가능성이 크다면, 첫 번째 방식의 통합 테스트를 먼저 수행하는 것이 효율적일 수 있다. 하지만 컴포넌트 간의 통신에 문제가 있을 가능성이 작다면 두 번째 방식처럼 바로 기능 위주의 통합 테스트를 수행하는 것이 더 효과적이다.

3.3.2 점진적 통합

통합 대상 컴포넌트가 많은 경우, 전체 컴포넌트를 한 번에 통합하여 테스트하는 방법을 빅뱅(Big-bang) 방식이라고 부른다. 예를 들어, 그림 3.5 (b)는 빅뱅 방식이라고 볼 수 있다. 빅뱅 방식의 통합 테스트는 한 번에 많은 수의 컴포넌트가 통합되므로 테스트를 통해서 오동작이 확인되었을 때 어떤 컴포넌트가 오동작의 원인, 즉, 결함을 가지고 있는지 판단

하기 어렵다. 그림 3.5에서 (b)의 경우에는 컴포넌트가 3개뿐이므로 오동작의 원인이 되는 컴포넌트를 찾기가 어렵지 않지만, 만약 30개의 컴포넌트를 한 번에 통합한 경우에는 이 중에서 결함을 가진 컴포넌트 찾기가 쉽지 않을 것이다.

이런 경우에는 전체 컴포넌트를 한 번에 통합하는 빅뱅 방식 대신에 적은 수의 컴포넌트를 차례로 통합하는 점진적(Incremental) 방식을 적용하는 것이 효과적이다. 점진적 방식은 오작동의 원인이 되는 컴포넌트를 찾기 쉬운 반면 테스트 드라이버 및 스텁을 여러 번 개발해야 한다.

만약, 컴포넌트 0, 컴포넌트 1, 컴포넌트 2 이렇게 3개의 컴포넌트가 있다면, 점진적 방식을 이용하여 다음과 같이 2회에 걸쳐 통합테스트를 수행할 수 있다. 이때, 1차, 2차 통합 테스트 수행 시 드라이버와 스텁을 각각 개발해야 한다.

- 1차 통합 테스트: 컴포넌트 1, 컴포넌트 2
- 2차 통합 테스트: 컴포넌트 0 추가

점진적인 통합 테스트 수행 방식에는 호출 관계의 하위에 있는 컴포넌트들을 시작으로 해서 상위에 있는 컴포넌트들을 통합하는 상향식 통합과 시스템을 구성하는 컴포넌트들의 계층 구조에서 가장 상위에 있는 컴포넌트부터 시작하여 하위에 있는 컴포넌트들을 점진적으로 통합하는 하향식 통합, 그리고 상향식과 하향식 두 방법을 결합하여 시스템을 통합하는 샌드위치 통합 방식이 있다. 그림 3.8은 상향식 통합과 하향식 통합의 차이를 보여준다.

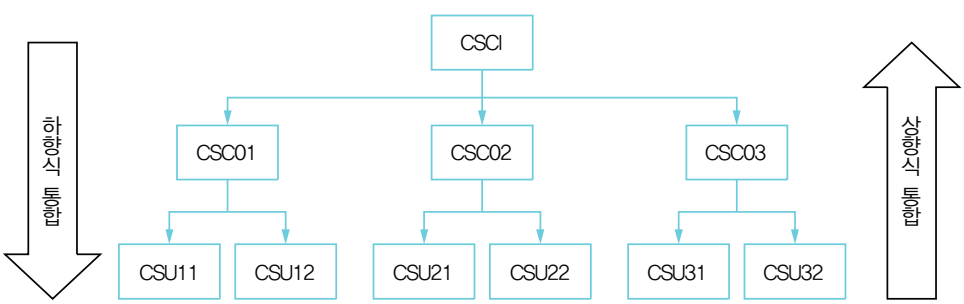

그림 3.8 점진적 통합 순서 : 상향식과 하향식

당연히 상향식 통합 테스트와 하향식 통합 테스트는 선택되는 컴포넌트의 순서가 다르다. 그림 3.5의 (b)를 예로 들면, 통합 방식에 따라서 선택되는 컴포넌트가 표 3.3과 같이 달라진다.

표 3.3 상향식 통합과 하향식 통합의 예

통합 순서	상향식 통합	하향식 통합
1차	컴포넌트 1 컴포넌트 2	컴포넌트 0 컴포넌트 1
2차	컴포넌트 0 컴포넌트 1	컴포넌트 1 컴포넌트 2

상향식 통합 과정을 보면 특별한 기능을 제공하는 하위의 컴포넌트(모듈)를 식별하여 그룹화·클러스터링한 후에 테스트 드라이버를 작성하여 테스트를 수행한다. 이때, 여러 모듈의 묶음을 '클러스터(Cluster)' 또는 '빌드(Build)'라 한다. 클러스터를 테스트한 후에 테스트 드라이버를 제거하고 실제 모듈과 결합한다. 이와 같은 과정을 시스템이 완전히 통합될 때까지 반복한다. 그림 3.9는 이러한 상향식 통합 과정을 보여 준다.

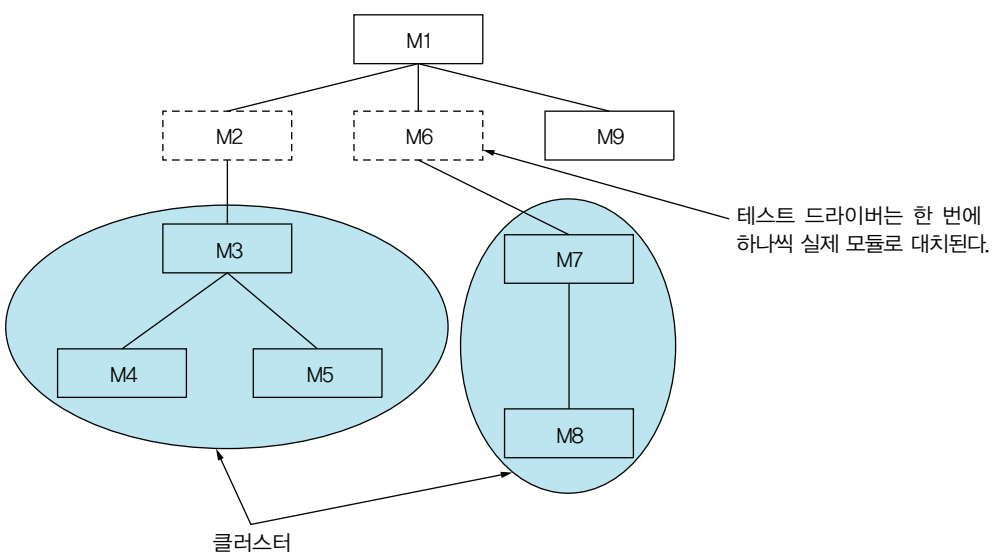

그림 3.9 상향식 통합 테스트

상향식 통합 테스트의 장점은 하위 컴포넌트를 충분하게 테스트할 수 있다는 점이다. 일반적으로 컴포넌트 의존 관계에서 하위에 있는 컴포넌트는 시스템이 제공하는 서비스에 필요한 공통적인 기능을 제공하는 역할을 한다. 따라서 컴포넌트가 하위에 있을수록 여러 상위 컴포넌트가 빈번하게 사용하는 코드를 갖는다고 간주할 수 있으며, 통합이 진행될수록 이 코드들은 그만큼 빈번하게 테스트 된다. 또한, 하향식 통합에서 필요한 테스트 스텁을 제공하는 비용이 들지 않는다는 장점도 있다.

하향식 통합 테스트 방식을 사용하여 시스템을 테스트하는 과정은 다음과 같다. 우선, 가장 상위에 있는 컴포넌트를 테스트하기 위해 하위 컴포넌트를 테스트 스텁으로 대치한 후 테스트를 수행한다. 깊이 우선 방식이나 너비 우선 방식을 사용하여 테스트 스텁을 한 번에 하나씩 실제 컴포넌트로 대치하고, 대치된 컴포넌트가 실제 호출하는 하위 컴포넌트를 테스트 스텁으로 대치한다. 이 경우에 테스트 스텁이 실제 모듈로 대치되어 시스템에 변경이 발생하였으므로 리그레션 테스트를 수행한다. 이 과정을 시스템이 완전히 통합될 때까지 반복한다(그림 3.10 참조).

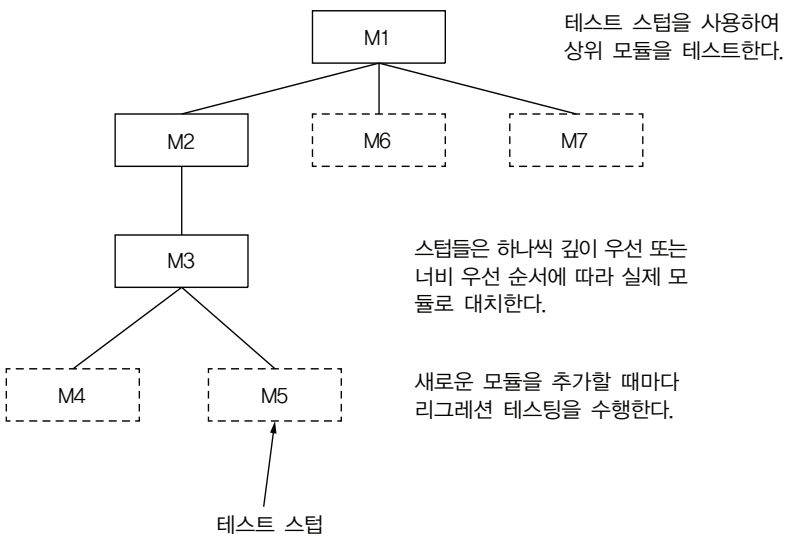

그림 3.10 **하향식 통합 테스팅 과정**

컴포넌트 종속 관계에서 기본적으로 상위 컴포넌트는 시스템의 기능을 결정하고, 하위 컴포넌트는 시스템이 제공하는 기능을 보조하는 역할을 한다. 즉, 상위 컴포넌트의 결함은 시스템 설계의 문제가 나타난 것으로 해석할 수 있다. 상위 컴포넌트의 결함은 상위 컴포넌트를 반복적으로 테스트하는 하향식 통합 테스트 방식으로 빠르게 발견할 수 있다. 반면, 하향식 통합 테스트 방법은 많은 수의 테스트 스텁이 필요하므로 만약 테스트 스텁 구현 비용이 많이 드는 경우라면 효과적인 테스트 방법이 아니다.

상향식 통합 테스트와 하향식 통합 테스트 방식을 결합하여 시스템을 통합할 수 있는데, 이러한 방식을 샌드위치 통합 테스트 방식이라고 한다.

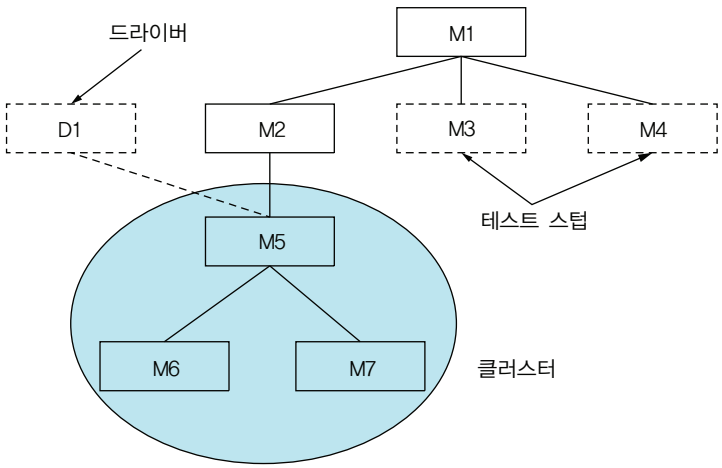

그림 3.11 샌드위치 통합

그림 3.11은 샌드위치 통합 테스트 전략을 이용하여 모듈을 통합하는 과정을 보여준다. 상위 컴포넌트 M1은 M2, M3, M4에 해당하는 테스트 스텁을 사용하여 테스트하고, M5, M6, M7은 클러스터링하여 테스트 드라이버 D1을 사용하여 테스트한다. 클러스터에 대한 테스트가 완료되면 실제 컴포넌트 M2를 통합하고 나머지 컴포넌트 M3, M4도 추가 통합하여 완전한 시스템을 구축한다.

3.4 시스템 테스트 및 인수 테스트

시스템 테스트(System test)는 통합 테스트가 완료된 후에 전체 시스템이 시스템 명세에 따라 개발되었는지 검증하기 위해 수행하는 테스트이다. 시스템 테스트의 목적은 컴포넌트 테스트나 통합 테스트와는 다르다. 컴포넌트 테스트나 통합 테스트는 기능이 올바르게 수행되는지 검증하는 것에 중점을 두지만, 시스템 테스트는 시스템의 기능 측면뿐만 아니라 성능(Performance), 호환성(Compatibility), 사용성(Usability), 신뢰성(Reliability), 보안성(Security), 유지보수성(Maintainability), 이식성(Portability) 등과 같은 비기능적인 요구사항을 만족하는지도 검증한다.

시스템 테스트가 완료되면 실제 사용자의 요구사항을 만족하는지 확인하기 위한 인수 테스트(Acceptance test)를 수행해야 한다. 인수 테스트의 주목적은 결함 검출이 아니라 시스템을 인수해도 되는지 고객의 입장에서 평가하는 것이다. 인수 테스트에서 사용되는 테스트 케이스는 사용자 또는 소프트웨어 구입자가 제시할 수도 있고 시스템 테스트에서 사

용했던 테스트 케이스를 사용할 수도 있다.

실제 사용자가 시스템을 사용하는 방식은 개발자가 시스템을 테스트할 때 사용한 방식과 차이가 있을 수 있으므로, 개발자가 수행한 테스트로 발견되지 않은 결함이 인수 테스트 단계에서 발견될 가능성이 있다.

인수 테스트의 유형에는 알파 테스트(Alpha test)와 베타 테스트(Beta test)가 있다. 알파 테스트는 선택된 사용자(회사 내의 다른 사용자 또는 실제 사용자)가 개발자 환경에서 통제된 상태로 수행하는 반면, 베타 테스트는 일정 수의 사용자에게 소프트웨어를 사용하게 하고 피드백을 받는다. 보통 베타 테스트에는 개발자가 참여하지 않는다.

3.5 리그레션 테스트

유지보수 단계에서도 소프트웨어가 수정된 후에 변경이 올바르게 되었는지 검사하기 위하여 리그레션 테스트를 수행한다. 유지보수 단계에서는 다음과 같은 이유로 소프트웨어 수정이 이루어진다:

- **결함 수정 작업**(Corrective maintenance): 소프트웨어를 사용하는 도중에 발견된 결함을 수정하기 위해 소프트웨어를 변경하는 유지보수 활동
- **기능 보강 작업**(Perfective maintenance): 소프트웨어 기능을 추가하거나 성능을 개선하기 위해 소프트웨어를 변경하는 유지보수 활동
- **적응 작업**(Adaptive maintenance): 소프트웨어 시스템을 새로운 운영환경에 적응시키기 위해 소프트웨어를 변경하는 유지보수 활동
- **예방 작업**(Preventive maintenance): 더 나은 유지보수를 위해 기존의 소프트웨어 시스템에 대한 문서를 준비하거나 시스템 구조를 유지보수하기 용이한 새로운 구조로 변경하는 작업 활동

그림 3.12는 유지보수 단계에 상기 네 가지 유지보수 활동이 차지하는 상대적 비율을 나타낸다.

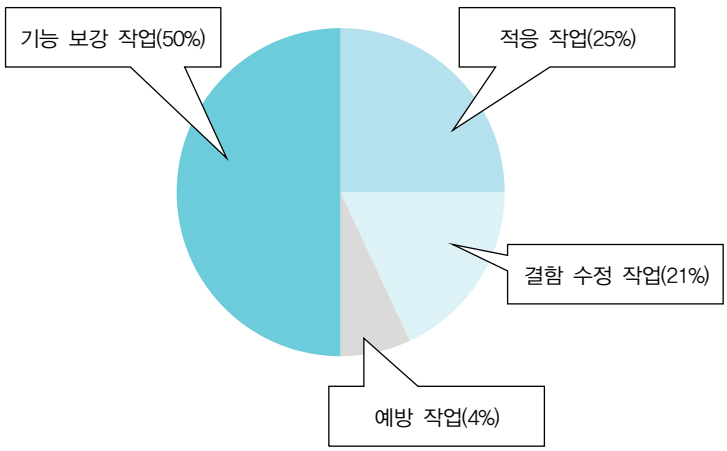

그림 3.12 소프트웨어 유지보수 활동 비율

이러한 유지 보수 활동에는 필연적으로 소프트웨어 수정 작업이 수반된다. 따라서 소프트웨어가 변경되었을 때 변경 작업으로 인해 새로운 결함이 유발되었는지 살펴보아야 한다(그림 3.13 참조).

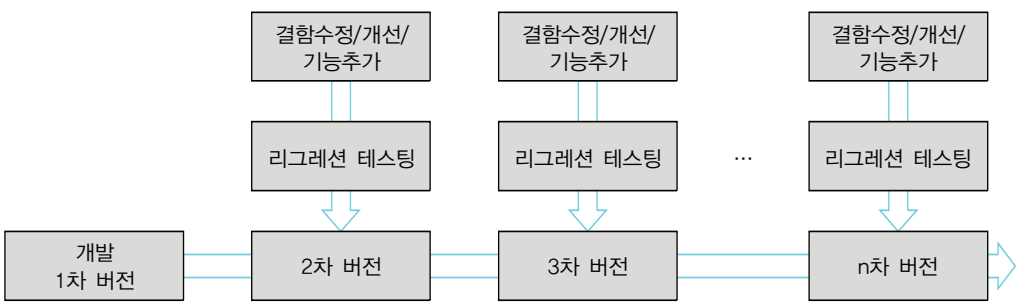

그림 3.13 리그레션 테스트 사이클

보통 유지보수 단계에서 리그레션 테스트를 수행하는 방법은 개발 단계에서 사용한 테스트 케이스를 이용하는 것이다. 개발 단계에서 사용한 테스트 케이스를 수정된 프로그램에 실행하여, 그 결과가 수정되기 전 프로그램에서 실행한 결과와 차이가 있는지 비교해본다. 만약 다르다면 이 변화가 의도한 것인지 확인하고 의도한 결과가 아니라면 프로그램에 결함이 있다고 판단할 수 있다. 현재 시장에 나와 있는 대부분의 테스트 도구 및 환경은 이러한 형태의 리그레션 테스트를 지원한다.

리그레션 테스트에서 중요하게 고려해야 할 점은 테스트 케이스의 규모이다. 시간이 갈수록 새로운 기능이 추가되는 시스템이라면 리그레션 테스트에 소요되는 비용과 시간도 늘

어나게 될 것이다. 표 3.4는 기능 추가에 따라 소요되는 리그레션 테스트 시간의 변화를 보여 준다.

표 3.4 기능 추가에 따른 리그레션 테스트 시간 변화

추가 기능	테스트 케이스 집합	테스트 실행 시간	리그레션 테스트 케이스 집합	리그레션 테스트 실행 시간
F1	TS1	T1	–	–
F2	TS2	T2	TS1	T1
F3	TS3	T3	TS1+TS2	T1+T2
F4	TS4	T4	TS1+TS2+TS3	T1+T2+T3

리그레션 테스트에도 여러 방식이 있다. 그중 기존에 개발된 모든 테스트 케이스를 사용하는 방식이 Retest-All 방식이다. 이 방식은 복잡한 테스트 절차를 요구하지 않지만 너무나 많은 시간과 자원이 필요하다.

현실적으로 시간과 예산이 한정되어 있으므로 확보된 테스트 케이스의 일부만을 사용해서 리그레션 테스트를 수행하는 방법이 있다. 그림 3.14는 Retest-All 방식을 포함한 여러 리그레션 테스트 방식을 보여준다.

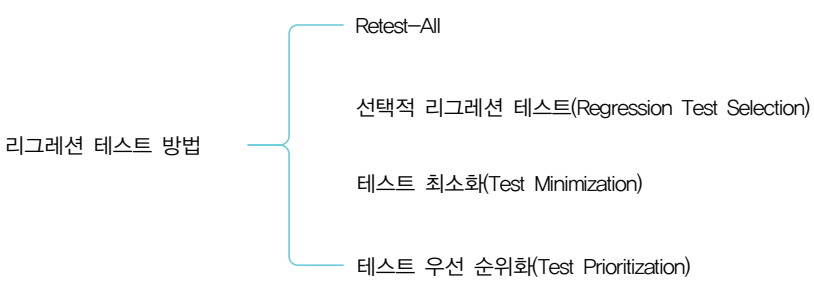

그림 3.14 리그레션 테스트 방식

선택적 리그레션 테스트는 기존의 테스트 케이스 중에서 일부만 선정하는 방식이다. 이 방식은 슬라이싱 기법, 자료 흐름 분석 기법과 같은 변경 영향 분석(Change impact analysis)을 통해 원래의 프로그램과 변경된 프로그램이 서로 다른 결과를 출력할 가능성이 있는 테스트 케이스를 식별하여 리그레션 테스트를 수행한다.

예를 들어, 슬라이싱 기법을 이용하여 선택적 리그레션 테스트를 수행할 때, 테스트 케이스가 실행한 문장들로부터 출력 결과에 영향을 미칠 수 있는 부분을 분석하고, 이들이 변경

된 부분을 포함하는지 파악한다. 만약 포함한다면 해당 테스트 케이스를 리그레션 테스트 케이스로 사용한다.

테스트 최소화 방식은 중복된 테스트 케이스를 제거하여 테스트 케이스의 수를 줄이는 방식이다. 이 방식은 중복된 테스트 케이스를 식별하기 위해 커버리지 개념을 사용한다. 예를 들어, 하나의 테스트 케이스 t1이 실행한 문장들이 다른 테스트 케이스 t2가 실행한 문장들을 포함하거나, 동일한 함수를 실행했다면 테스트 케이스 t2를 제거할 수 있다. 테스트 최소화 방식은 선택적 리그레션 테스트와는 달리 실제 테스트 케이스가 제거되기 때문에 변경되는 부분이나 변경에 의하여 영향받는 부분을 테스트하는 테스트 케이스가 제거될 위험이 있으므로 주의하여야 한다.

테스트 우선순위화 방식은 테스트 케이스에 우선순위를 두어 우선순위가 높은 테스트 케이스만을 활용하는 방식이다. 테스트 케이스 우선순위화 방식은 가능한 한 빨리 많은 결함을 검출할 수 있도록 테스트 케이스의 실행 순서를 결정한다.

테스트 케이스 우선순위의 효과성을 평가하기 위한 척도로 APFD(Average Percentage of Faults Detected)가 많이 사용된다. APFD는 테스트 케이스의 실행 수 대비 검출된 결함의 비율을 측정한다. 따라서 높은 APFD는 많은 결함을 빨리 검출하였다는 의미이며, 테스트 케이스 우선순위화 방식은 가능한 높은 APFD를 갖도록 테스트 케이스에 우선순위 등급을 부여하는 것이다. 다음은 APFD를 계산하는 공식으로, 그 값은 0부터 100까지 범위를 가진다.

$$\text{APFD} = (1 - \frac{Tf1 + Tf2 + \ldots + Tfm}{mn} + \frac{1}{2n}) \times 100$$

n: 테스트 케이스의 수
m: 결함의 수
$TCfi$: 결함 i를 검출하는 테스트 케이스의 위치

Exercise 01	테스트 케이스의 실행 순서에 따라 APFD가 어떻게 변화하는지 살펴보자. 표 3.5는 5개의 테스트 케이스와 이들이 검출한 5개의 결함을 보여준다.

표 3.5 결함 매트릭스

테스트 케이스	결함				
	f1	f2	f3	f4	f5
TC1	×				
TC2	×	×			×
TC3				×	
TC4		×	×		
TC5		×		×	×

우선 테스트 케이스가 TC1, TC2, TC3, TC4, TC5의 순서대로 실행되었을 때 APFD를 계산해보자.

$n=5,\ m=5$

$$\text{APFD}=\left(1-\frac{1+2+4+3+2}{5\times5}+\frac{1}{2\times5}\right)\times100=62$$

만약 테스트 케이스가 TC2, TC4, TC3, TC1, TC5 순서대로 실행되었을 때 APFD를 계산하면 다음과 같다.

$n=5,\ m=5$

$$\text{APFD}=\left(1-\frac{1+1+2+3+1}{5\times5}+\frac{1}{2\times5}\right)\times100=78$$

예제에서 볼 수 있듯이 테스트 케이스의 실행 순서에 따라 APFD가 큰 차이를 보인다. APFD가 높다는 것은 더 적은 수의 테스트 케이스를 실행하여 많은 결함을 빠르게 검출할 수 있음을 의미한다. 만약, 리그레션 테스트에 소요되는 시간과 비용이 제한된 상황이라면, 가능한 한 결함 검출을 빠르게 할 수 있는 테스트 케이스의 우선순위를 높게 설정하여 테스트의 효용성을 높일 수 있다.

결함 검출률이 높은 테스트 케이스 식별은 현실적으로 매우 어려운 일이다. 보통은 비즈니스 중요도, 리스크, 테스트 케이스의 실행 시간, 커버리지, 결함 검출 내역 등의 요인들을 고려하여 테스트 케이스 우선순위 등급을 설정한다.

리그레션 테스트는 컴포넌트 테스트, 통합 테스트, 시스템 테스트를 비롯한 모든 단계에서

수행된다. 그림 3.15는 애플리케이션에 변경이 이루어진 후에 수행되는 리그레션 테스트의 절차를 보여준다.

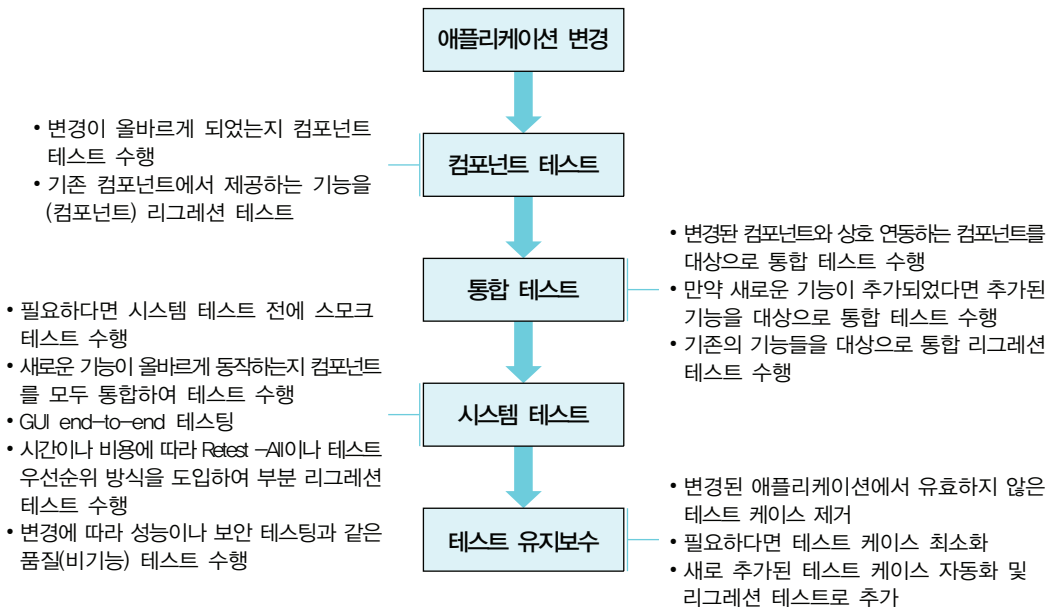

그림 3.15 리그레션 테스트 절차

테스트 유지보수 단계에는 더 이상 필요 없는 테스트 케이스를 제거하는 과정이 있다. 만약, 어떤 기능이 더 이상 비즈니스에 필요하지 않아 애플리케이션에서 제거되었다면, 이를 테스트하는 테스트 케이스도 리그레션 테스트 케이스 집합에서 제거되어야 한다. 이 경우에 해당 테스트 케이스를 제거하지 않는다면 이로 인하여 리그레션 테스트를 수행할 때 잘못된 테스트 결과가 보고되고 테스트의 신뢰성을 잃게 만드는 요인이 될 것이다.

또한, 요구사항이 변경되어 입력 출력의 관계가 변할 수 있다. 이 경우에도 해당하는 테스트 케이스를 요구사항 변경에 따라 변경하는 작업이 필요하다. 새로 추가된 테스트 케이스가 있다면 이 테스트 케이스를 자동화하는 작업도 추가될 수 있다. 리그레션 테스트는 많은 테스트 케이스를 자주 실행하므로 가급적 자동화하는 편이 효율적이다. 따라서 새로 추가된 테스트 케이스를 자동화하는 작업도 중요한 테스트 유지보수 작업에 해당한다.

다음은 리그레션 테스트 프로세스를 수행하는 예이다.

Exercise 02	기존에는 일반 택배 서비스만 제공하던 온라인 샵이 이번 릴리즈에서 퀵 서비스를 제공하는 기능을 추가하였다. 이 경우에 적용되는 리그레션 테스트 프로세스를 살펴보자.

기능 추가 후 배송지 정보를 입력하고 저장하는 퀵서비스 윈도우에 대한 컴포넌트 테스트를 수행한다. 또한, 택배 서비스 화면에 기존의 택배 서비스와 추가된 퀵 서비스 옵션이 나타나는지 검증한다. 이때 기존의 택배 서비스 인터페이스 API 서버와 퀵 서비스 인터페이스 API를 제공하는 서버는 모의 객체로 대신한다.

통합 테스트 단계에서 퀵 서비스 인터페이스 API를 제공하는 서버와 통합하여 적절하게 호출되었는지 검증하고 배송지 정보가 문제없이 전달되었는지 확인한다. 또한, 기존의 택배 서비스를 대상으로 통합 리그레션 테스트를 수행하여 퀵 서비스 추가가 기존 택배 서비스에 어떤 영향이 있는지 검증한다.

통합 테스트가 통과된 후에는 GUI를 이용한 퀵 서비스 시스템 테스트를 수행하고, 기존 택배 서비스에 대해서도 리그레션 테스트를 수행한다. 시간과 비용에 제약이 있다면 테스트 우선순위화 방식을 이용하여 우선순위 등급 1, 2에 대해서만 기존의 택배 서비스에 대한 GUI 테스트를 수행한다. 이 경우에는 비기능적 테스트는 수행하지 않는다.