```
public class Customer {
  private String name;
  public String getName(){
    return name;
  }
  public void setName(String string) {
    name = string;
  }
  public String toXML(){
    return "<Customer><Name>" +
    name + "</Name></Customer>";
  }
}
```

```
public class Customer implements SerializableToXml {
  private String name;
  public String getName(){
    return name;
  }
  public void setName(String string) {
    name = string;
  }
  public String toXML(){
    return "<Customer><Name>" + name + "</Name></Customer>";
  }
}
interface SerializableToXml {
  public abstract String toXML();
}
```

그림 9.9 인터페이스 추출

■ 템플릿 메서드 형성

각 서브 클래스에 있는 두 메서드가 완전히 같지 않지만 비슷한 작업을 한다면 하나로 통일하여 슈퍼 클래스로 올린다.

```
public abstract class Party {
}
public class Person extends Party {
   private String firstName;
   private String lastName;
   private Date dateOfBirth;
   private String nationality;
   public  void printNameDetails() {
      System.out.println("Name: " + firstName + " " + lastName);
      System.out.println("dateOfBirth: " + dateOfBirth.toString() +
                         "Nationality:" + natoinality);
   }
}
public class Company extends Party {
   private String Name;
   private String CompanyType;
   private Date incorporated;
   public  void PrintNameDetails() {
      System.out.println("Name: " + name + " " + companyType);
      System.out.println("incorporated: " + incorporated.toString());
   }
}
```



```
public abstract class Party {
   public void PrintNameAndDetails() {
      printName();
      printDetatils();
   }
   public abstract void printName();
   public abstract void printDetails();
}
public class Person extends Party {
   private String firstName;
   private String lastName;
   private Date dateOfBirth;
   private String nationality;
   public  void printDetails() {
      System.out.println("dateOfBirth: " + dateOfBirth.toString() +
                         "Nationality:" + natoinality);
   }
   public void printName() {
      System.out.println("Name: " + firstName + " " + lastName);
   }
}
public class Company extends Party {
      ... 생략
```

그림 9.10 템플릿 메서드

9.5 코드 품질 향상 기법

프로그래머가 모듈을 위한 코드를 작성한 후에는 다른 곳에서 사용되기 전에 오류가 없는지 검사하고 테스트 하여야 한다. 코드의 품질을 높이는 방법은 흔히 테스팅이 가장 일반적인 방법이라고 생각하고 있는데 더욱 효과적인 방법들이 있다.

프로그램을 수행시켜보는 것 대신에 읽어보고 눈으로 확인하는 방법이 있는데 이를 인스펙션이라고 한다. 테스팅의 일종으로 볼 수도 있지만 프로그램에 데이터를 주어 실행시켜 보는 것이 아니라 수행되지 않는 데드 코드가 없는지, 선언이 되지 않고 사용한 변수가 없는지 등을 검사하는 정적 분석도 있다.

극도의 신뢰성을 요구하는 시스템의 핵심 부분은 프로그램의 정확성을 증명하는 방법을 사용하기도 한다. 또한 애자일 방법에서는 프로그래밍을 하기 전에 먼저 테스트를 준비하여 프로그램을 작성하면서 계속 테스팅을 병행해 나가는 테스트 중심 개발 방법

도 사용하고 있다.

9.5.1 인스펙션

인스펙션은 설계든 코드든 결함을 찾아내고 이를 확인하려는 작업이다. Fagan이 제안한 이 방법은 품질 개선과 결함을 초기에 발견하기 위한 효과적인 방법으로 알려져 있다.

코드 인스펙션은 프로그램이 성공적으로 컴파일 되고 정적 분석 도구에 의하여 검사된 후에 이루어진다. 그 이유는 컴파일러나 정적 분석기가 자동으로 찾아낼 오류들을 사람이 찾는 노력과 시간을 절약하기 위한 것이다.

코드 인스펙션을 하는 방법은 먼저 리뷰할 코드와 설계 문서에 대한 인스펙션을 담당할 멤버들에게 보낸다. 인스펙션은 프로그래머뿐만 아니라 설계자, 테스트 엔지니어도 참여한다.

코드 인스펙션의 목적은 코드에 묻혀있는 결함을 찾아내는 것이다. 결함과 함께 다른 품질 이슈, 예를 들면 효율성, 코딩 표준의 준수 여부 등을 검사하기도 한다. 코드 인스펙션 하는 동안 관심을 가지는 결함의 종류를 체크리스트 형태로 정리하여 미리 인스펙션에 참여하는 멤버들에게 배포한다. 코딩 단계에는 [표 9.2]와 같은 체크리스트를 사용한다.

표 9.2 코딩 단계의 체크 리스트

대상	체크 항목
클래스 전체	C1. 클래스의 이름은 적절한가? 요구분석이나 설계와 일치하는가? 충분히 세분화/일반화 되었는가? C2. 추상화될 수 있는가? C3. 헤더주석이 목적을 잘 나타내고 있는가? C4. 헤더주석이 관련된 요구나 설계요소를 언급하고 있는가? C5. 소속된 패키지를 기술하였는가? C6. 충분히 private인가? C7. final이 되어야 하나? C8. 문서표준이 적용되었는가?
속성	A1. 속성이 필요한가? A2. static이 될 가능성은? 모든 인스턴스가 이 변수를 가져야 하나? A3. final이 되어야 하나? 값이 변경되는가? get 메서드만 필요하지 않은가? A4. 명명 규칙이 적절히 적용되었는가? A5. 충분히 private한가? A6. 속성들이 가능한 독립적인가? A7. 초기화 정책이 적당한가? 선언할 때 생성자 호출시 static{} 사용 위 사항의 조합
생성자	CO1. 생성자가 필요한가? 팩토리 메서드가 필요하지 않은가? CO2. 모든 속성을 초기화하는가? CO3. 충분히 private한가? CO4. 상속된 생성자를 적절히 사용하였는가?
메서드 헤더	MH1. 메서드 이름을 적절히 잘 정하였는가? MH2. 충분히 private한가? MH3. static이 될 수 있는가? MH4. final이 되어야 하는가? MH5. 헤더주석이 메서드의 목적을 잘 나타내고 있는가? MH6. 메서드 헤더 주석이 관련되는 요구와 설계를 언급하고 있는가? MH7. 모든 필요한 변경조건을 잘 기술하였는가? MH8. 모든 전제조건이 기술되었는가? MH9. 모든 결과조건이 기술되었는가? MH10. 문서표준이 적용 되었는가? MH11. 파라미터 타입이 제한되어 있는가?
메서드 내부	MB1. 알고리즘이 상세 설계와 일치하는가? MB2. 코드가 전제 조건만을 가정하고 있는가? MB3. 결과조건의 모든 것을 만족하고 있는가? MB4. 코드가 요구한 불변조건을 잘 지키고 있는가? MB5. 반복구조가 모두 종료되는가? MB6. 요구하는 표준 표시법을 따르고 있는가? MB7. 한 줄 한 줄 완벽히 체크하였는가? MB8. 괄호가 모두 짝이 맞는가? MB10. 코드가 정확한 타입을 리턴하는가? MB11. 코드에 대한 주석이 완벽한가?

인스펙션의 결과 중 심각도는 매우 중요하다. 결함의 우선순위를 나타내어 작업 스케줄을 정하기 때문이다. 고치기 쉬운 결함은 인스펙션 회의를 가질 시간적 여유가 없다. 쉬운 결함을 다루는 좋은 방법은 인스펙터가 원시코드에 직접 표기하여 원작자에게 주는 것이다.

인스펙션을 한 후 결함에 대하여 표시할 때 포함되는 사항 중 다른 하나는 타입이다. 결함의 타입은 다음과 같이 구별할 수 있다.

- 로직 문제(case 또는 step이 빠짐, 중복 로직, 조건이 빠진 경우, 불필요한 기능, 잘못된 해석, 잘못된 조건 테스트, 틀린 변수의 체크, 반복 루프의 결함 등)
- 컴퓨팅 문제(잘못된 수식, 오차, 부호 오류)
- 인터페이스/타이밍 문제(인터럽트 처리 부정확, I/O 타이밍 부정확, 서브루틴/모듈 불일치)
- 데이터 처리(초기값 부정확, 데이터 접근 저장 부정확, 자료 값의 스케일 또는 단위 부정확, 벡터 데이터의 차원 부정확)

9.5.2 정적 분석

프로그램 텍스트를 조직적으로 분석하여 결함을 찾아내는 것을 정적 분석(static analysis)이라 한다. 정적 분석은 소프트웨어 도구를 이용하여 자동으로 할 수 있다. 정적 분석하는 동안 프로그램은 실행되지 않는다. 다만 프로그램 텍스트가 도구에 입력된다. 정적 분석의 목적은 코드에 있는 오류나 잠재적인 오류를 찾아내고 디버깅에 유용한 정보를 생성하기 위함이다.

정적 분석 도구가 결함을 찾는 데는 두 가지 방법을 사용한다. 첫째는 코드에 존재하는 결함으로 나타날 비정상적인 패턴이나 원하지 않는 패턴을 찾는 방법이다. 또 다른 방법은 실행할 때 프로그램의 고장을 일으킬 코드 상에 존재하는 결함을 직접 찾는 방법이다.

첫 번째 방법인 비정상적인 패턴을 찾는 방법은 자료 흐름이나 논리 흐름을 분석하는 것이다. 자료 흐름이란 자료가 어디서 정의되고 어디서 사용되었는지를 나타내는 그래프이다. 자료 흐름도를 분석하면 자료가 정의되지 않고 사용되거나 정의되고 사용되는 비정상적인 패턴을 찾을 수 있다. 이를 자료 변칙(data anomaly)이라 한다.

다음과 같은 프로그램의 자료 흐름을 분석해 보자.

```
(1)   scanf(x, y);
      if (y < 0)
(2)       pow = 0 - y;
(3)   else  pow = y;
(4)   z = 1.0;
(5)   while (pow != 0)
(6)       { z = z * x; pow = pow - 1; }
(7)   if ( y < 0 )
(8)       z = 1.0/ z;
(9)   printf(z);
```

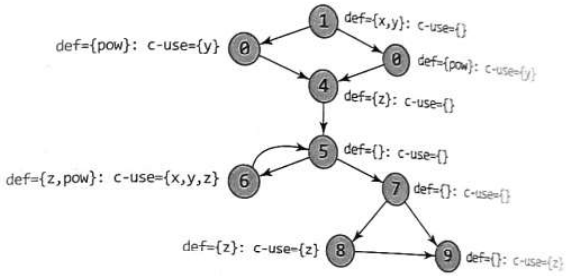

그림 9.11 자료 흐름도

[그림 9.11]의 각 노드에 표시된 것은 정의된 변수(def)와 사용된 변수(c-use)를 집합 형태로 표현한 것이다. 예를 들어 3번 노드에서는 pow라는 변수가 선언되었고 y라는 변수가 사용되었다. 만일 어떤 사용된 변수가 정의되었는지를 판단하려면 제어 경로를 따라 올라가 그 변수가 def 집합에 있는지를 조사해 보면 된다. 프로그램 시작부분까지 올라갔지만 def 집합에 그 변수를 찾을 수 없다면 정의 변칙(define anomaly) 오류임을 알 수 있다.

반대로, 정의되고 사용되지 않는 변수를 찾으려면 정의된 노드로부터 아래 경로를 추적하여 use 집합에 있는지 검사한다. 결국 각 변수에 대하여 정의된 노드로부터 사용된 노드까지의 경로(이를 du-경로라 함)가 있는지를 판단하면 된다.

프로그램에 있는 불필요한 부분(redundant)은 또 하나의 잠재된 결함이다. 불필요한 데드 코드(dead code)는 대체로 컴파일러에 의하여 검출되지 않는다. 실행 경로가 없어 결코 실행될 일이 없는 코드도 결함이다.

```
int global;
void f ()
{
  int i;
  i = 1;          /* dead store */
  global = 1;     /* dead store */
  global = 2;
  return;
  global = 3;     /* unreachable */
}
```

```
class GFG {
    public static void main(String args[])
    {
        System.out.println("I will get printed");
        return;

        // it will never run and gives error
        // as unreachable code.
        System.out.println("I want to get printed");
    }
}
```

그림 9.12 데드 코드와 실행 불가능 코드

9.5.3 테스트 중심 개발

일반적으로는 프로그램을 작성한 후 테스팅을 한다. 하지만 테스트 중심 개발(Test-Driven Development)은 테스트를 위한 코드를 작성한 후 기능을 구현하도록 요구하고 있다. 테스트 코드를 쓰기 위해서는 프로그램의 기능과 명세를 잘 이해하여야하기 때문이다.

테스트는 주로 클래스 안에 있는 메서드를 시험하기 위한 코드, 즉 단위 테스팅을 위한 것이다. 코딩 하기 전에 미리 테스트를 생각하면 메서드 구현 과정에 발생하는 오류를 많이 줄일 수 있어 코드 품질을 향상 시킬 수 있다.

■ 개발 과정

TDD를 진행하는 과정은 [그림 9.13]과 같다. 객체지향 프로그램의 기본 단위인 클래스의 기능에 대한 코딩과 테스트 코드의 작성을 병행하여 점증적으로 작성하는 것이다. 테스트를 위한 간단한 코드는 클래스의 메서드를 구동시키고 결과값을 체크하는 목적을 가진다.

1. **TDD를 준비** - 개발하려는 프로그램의 뼈대를 만든다. 예를 들면 클래스의 골격 코드가 될만한 인스턴스 변수의 선언과 메서드의 선언을 작성한다. 또한 테스트 커버리지 수준을 결정한다. 커버리지는 품질 기준이 될 수 있는 것으로 예를 들어 100% 문장 커버리지는 프로그램의 모든 문장을 적어도 한 번씩 테스트 하는 기준이다.
2. **테스트 코드 작성** - 구현할 기능을 선정하고 동시에 기능을 테스트할 방법을 설계하고 코딩한다. 테스트 코드는 테스트 하려는 대상을 셋업하고 구동시킨 후 결과를 비교하는 명령으로 구성되어 있다.
3. **반복하여 기능을 구현하고 테스트** - 미리 작성한 테스트 코드에 의하여 통과될 수 있도록 기능을 조금씩 정확히 구현한다. 테스트 코드를 이용하여 구현된 기능을 테스트 한다. 테스트를 통과할 때까지 코드를 발전시키고 테스트를 수행하는 일을 반복한다.
4. **테스트 커버리지 측정** - 테스트가 수행되어 프로그램의 실행이 확인된 비율을 측정하여 원하는 수준이 미치지 못하였다면 테스트 케이스를 추가한다. 추가된 테스트 코드를 사용하여 프로그램을 테스팅 하여 원하는 수준의 커버리지에 도달하면 종료한다.

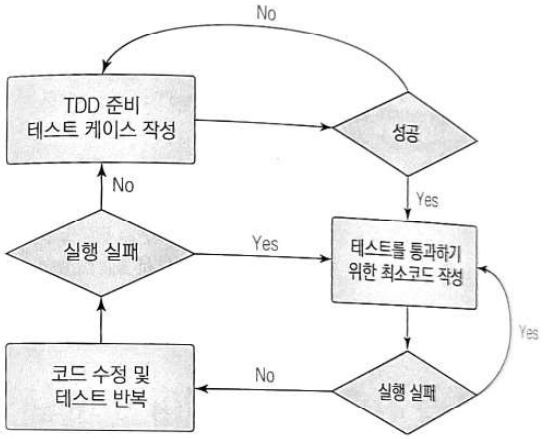

그림 9.13 TDD 작업 과정

■ 테스트 코드

TDD에서 사용하는 테스트 코드는 클래스의 메서드를 시험하는 단위 테스트 목적을 가지고 있다. 따라서 클래스의 메서드를 구동시키기 위하여 객체를 생성하여 테스트를 준비하는 코드와 메서드를 실행시키는 코드, 실행 결과 값을 예상 값과 비교하는 코드로 구성된다.

예를 들어, MyUnit이라는 스트링을 이어 붙이는 간단한 다음 프로그램의 테스트 코드를 작성해 보자.

```
public class MyUnit {
   public String concatenate(String one, String two){
      return one + two;
   }
}
```

클래스를 테스트하려면 public 메서드를 테스트하는 단위 테스트가 필요하다. concatenate() 메서드를 테스트하는 JUnit 단위 테스트는 다음과 같다.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class MyUnitTest {
   @Test
   public void testConcatenate() {
      MyUnit myUnit = new MyUnit();
      String result = myUnit.concatenate("one", "two");
      assertEquals("onetwo", result);
   }
}
```

단위 테스트 클래스 안에 있는 메서드 testConcatenate()은 Concatenare() 메서드를 테스트 하려는 코드다. 어노테이션 @Test는 JUnit에게 신호를 보내는 것으로 실행되어야하는 단위 테스트임을 나타낸다.

테스트 코드 안에서 먼저 테스트 대상인 MyUnit의 인스턴스를 생성한 후 concatenate() 메서드를 두 개의 테스트 문자열 값으로 호출한다. 마지막으로 assertEquals() 메서드의 호출이 실제 테스트 수행의 핵심이다. 실제 호출된 메서드 concatenate()의 출력을 예상하는 출력과 비교한다. 즉, "onetwo"(예상 출력)와 concatenate() 메서드에 의해 반환된 값을 비교하는 것이다.

두 값이 같으면 assertEquals() 메서드는 정상적으로 리턴되고 실행이 계속된다. 하지만 같지 않으면 예외가 발생하고 테스트가 중단된다. 단위 테스트 클래스에는 여러 가지 테스트 케이스를 커버하는 메서드가 있을 수 있다. JUnit 테스트 러너는 이를 모두 찾아서 실행한다.

짝 프로그래밍

짝 프로그래밍(pair programming)은 두 사람이 같은 컴퓨터를 사용하면서 다른 역할(개발과 테스팅)을 한다. 한 프로그래머가 기능을 개발하면 다른 프로그래머는 리뷰와 테스팅을 담당한다. 정기적으로 또는 원하면 두 프로그래머의 역할을 바꿀 수도 있다.

개발 팀의 프로그래밍 작업을 작은 작업으로 나누고 여기에 짝 팀을 배정된다. 각 작업은 클래스 단위가 될 수도 있고 또는 유스케이스, 패키지 단위로 이루어진다. 중요한 점은 짝 팀 사이에 진도와 이슈, 변경 등에 대한 의견을 교환하기 위하여 커뮤니케이션이 지속되어야 한다는 것이다.

테스트 중심 개발에서는 파트너들이 함께 테스트 케이스를 작성하고 기능을 구현하고 테스트를 수행하며 프로그램을 수정하고 리팩토링한다. 파트너들은 다음에 무엇을 하고 왜 해야 하며 어떻게 할 것인지 상의한다. 다음에 구현 담당 프로그래머는 아이디어를 구현하고 리뷰어는 코드의 정확성, 일관성, 완벽성 등을 체크하고 개선 의견을 제시한다.

■ 장점

- 스트레스와 성공의 기쁨을 나눌 상대가 있기 때문에 프로그래밍에 재미가 있고 스트레스를 줄일 수 있다.
- 짝 프로그래밍을 하는 동안 파트너와 아이디어를 교환하기 때문에 팀의 소통을 향상시킬 수 있다.
- 짝 프로그래밍은 팀 구성원이 서로를 이해하고 여러 가지 방법으로 서로에게서 배우기 때문에 상호 이해와 협력이 향상될 수 있다.
- 토론이 창의적인 사고로 발전하여 문제 해결에 도움이 되므로 간단하고 효율적인 해법을 만들어 낼 수 있다.

■ 단점

- 모든 사람에 맞지 않는다. 혼자 작업하는 것을 더 좋아하는 프로그래머도 있다.
- 적절히 다루지 못하면 파트너와의 대화에 시간이 너무 많이 걸린다. 따라서 대화는 문제의 해결과 작업 완성에 초점을 둔다.
- 파트너와의 교육, 경험, 문제 해결 방법, 코딩 스타일 등 차이점에 적응하기 위해 시간이 걸릴 수 있다.
- 파트너와 같은 장소에 있어야 하며 스케줄이 맞지 않아 미팅 시간을 찾기 어려울 수 있고 결함 발견을 위해 인스펙션과 같은 체계적인 리뷰만큼 효과적이지 않을 수 있다는 제약이 있다.