```
    public AssocAB(A a, B b) {
      this.a=a; this.b=b;
    }
    public void add(AssocAB ab) { ... }
    public void add(A a, B b) { ... }
    public void remove(AssocAB ab) { ... }
    ...
}
```

9.3.2 시퀀스 다이어그램의 코딩

시퀀스 다이어그램은 객체의 동작적인 측면을 모델링한 것이다. 즉 오퍼레이션의 종류와 클래스 사이의 관계를 파악하기 위하여 각 객체가 주고받는 메서드를 나타낸다. 시퀀스 다이어그램으로부터 클래스의 메서드를 위한 코드 골격을 어떻게 생성하는지 살펴보자.

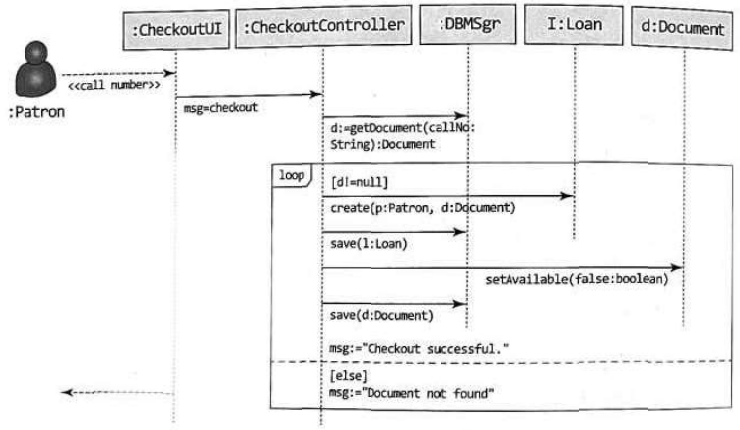

그림 9.3 Checkout Document 사용 사례

기본적인 룰은 A 객체에서 B 객체로 나가는 메시지가 표시되어 있다면 그 메시지는 B 클래스에 정의되어야 한다. 예를 들어, [그림 9.3]에서 checkOut() 메시지는 CheckoutController 클래스 안에 정의되어야 한다. [그림 9.3]에 있는 시퀀스 다이어그램에 표시된 클래스 CheckoutController을 코딩하면 다음과 같다.

```
public class CheckoutConroller {
  Patron p;
  public String checkout(String callNo) {
    BDNgr dbm=new DBMge();
    Document d=dbm.getDocument(callNo);
    String msg=" ";
    if (d!=null) {
      Loan l=new Loan(p, d);
      dbm.save(l);
      d.setAvailable(false);
      dbm.save(d);
      msg="Checkout successful.";
    } else {
      msg="Document not found.";
    }
    return msg;
  }
}
```

위 원시코드는 불완전한 것으로 뼈대만 나타낸 것이다. 프로그래머는 빠진 부분, 즉 Patron 객체 p를 초기화 하는 부분이나 어떤 패키지를 import 하여야 하는지 알아내서 채워야 한다. 시퀀스 다이어그램에서 코드 골격을 만들어 낼 수 있다면 코딩 작업은 훨씬 간단해진다.

9.4 리팩토링

리팩토링은 이미 존재하는 코드의 디자인을 안전하게 향상시키는 기술이라고 할 수 있다. 리팩토링은 문제가 될 수 있는 코드를 찾아내어 여러 가지 측면에서 향상시키는 방법을 제공해 준다. 단순히 코딩 스타일만을 개선하는 것이 아니라 성능과 코드의 구조, 즉 좋은 설계가 되도록 개선시키는 과정을 의미한다.

9.4.1 리팩토링 개념

Martin Fowler가 소개한 리팩토링 개념은 이미 프로그래머들에 의하여 사용되었던 좋은 습관으로 알려져 있다. 특히 Extreme Programming 개념을 주장한 Kent Beck과 같은 사람도 코드를 정리하는 데 노력하여 복잡하고 지저분한 코드보다는 깔

끔하고 변경하기 쉬운 기반이 견고한 코드를 만드는 것을 좋은 습관으로 주장하고 권장하여 왔다.

Kent Beck에 의하면 프로그램은 두 가지의 가치를 지니고 있다고 한다. 하나는 오늘 당장 프로그램이 수행되어 작업에 도움이 되어야 하는 가치이고 또 하나는 내일, 즉 프로그램의 변경이나 개선이 필요할 때를 대비한 가치이다.

프로그래밍 할 때 대부분 오늘 당장, 즉 기능이 수행되기 위한 목표를 제일 우선적으로 여기고 여기에 초점을 맞춘다. 기능을 추가하거나 프로그래밍의 기능을 더 좋게 하여 많이 쓰게 하고 많이 팔리게 하는 것이 주된 관심사이다.

그러나 프로그래밍을 오래 하다보면 오늘을 위한 초점은 일부에 불과하다는 것을 깨닫게 된다. 시스템의 사용 기간이 길어지면서 계속적으로 업그레이드와 새로운 환경의 변화에 대처해 나가야 하는데 이러한 내일을 대비한 코딩을 하지 않으면 상상도 못한 일들을 겪게 되고 처음부터 다시 반복해야 하는 경우도 생긴다.

■ 리팩토링의 목적

리팩토링은 단순히 코드를 깔끔하게 만드는 것은 아니다. 보다 효율적이고 통제된 방법으로 점진적으로 재구성하는 것이다. 리팩토링을 도입하면 코드를 보다 효율적으로 정리하게 된다. 어떤 문제의 기미가 보일 때 어떤 리팩토링을 사용할 것인지 알고 버그를 최소화 하기 위하여 어떤 방법을 사용하는지 알기 때문이다. 또한 리팩토링을 한 후에는 즉시 테스트하기 때문에 시스템의 신뢰성도 향상된다.

리팩토링은 소프트웨어를 보다 쉽게 이해하고 수정하기 쉽도록 만드는 것이다. 겉으로 보이는 동작은 변경시키지 않고 소프트웨어의 많은 것을 고칠 수 있다. 이런 의미에서 성능 최적화, 즉 튜닝도 대조되는 기술이다.

리팩토링을 하는 목적을 다음과 같이 요약할 수 있다.

- 리팩토링은 소프트웨어의 디자인을 개선시킨다. 리팩토링은 코드를 정돈하고 적절하지 않은 코드를 제거하고 구조가 망가지지 않도록 디자인을 유지하고 잘 파악되도록 바꾸는 것이다.
- 리팩토링은 소프트웨어를 이해하기 쉽게 만든다. 프로그램은 사람과 컴퓨터와의 대화 수단이다. 또한 사람과 사람의 대화의 수단이 될 수도 있다. 리팩토링을 통하여 프로그램이 읽기 쉽게 되고 더 효율적인 커뮤니케이션의 도구가 된다면 소프트웨어 개발에 매우 큰 장점이다.
- 버그를 찾는데 도움을 준다. 코드를 잘 이해하면 버그도 쉽게 찾을 수 있다. 리팩토링을 하면 그 과정에 코드가 무슨 작업을 하는지 깊이 이해하게 되고 버그를 더 잘 찾게 된다. 프로그램에 대한 가설을 잘 세우고 그 진위를 빨리 파악할 수 있게 되어 버그가 눈에 잘 들어오게 된다.
- 리팩토링은 프로그램을 빨리 작성할 수 있게 도와준다. 소프트웨어를 빨리 개발하는 데는 좋은 디자인이 가장 중요하고 강력하게 영향을 미치는 요소이다. 좋은 디자인이 되어야 개발 속도가 빨라지는데 좋은 디자인의 필수 요소가 리팩토링이다. 대부분 형편없는 디자인 때문에 작업 속도가 느려진다.

9.4.2 리팩토링 과정

리팩토링 과정은 다음과 같은 순서로 이루어진다.

1. 소규모의 변경 – 단일 리팩토링
2. 코드가 전부 잘 작동되는지 테스트.
3. 전체가 잘 작동하면 다음 리팩토링 단계로 전진.
4. 작동하지 않으면 문제를 해결하고 리팩토링 한 것을 undo하여 시스템이 작동되도록 유지.

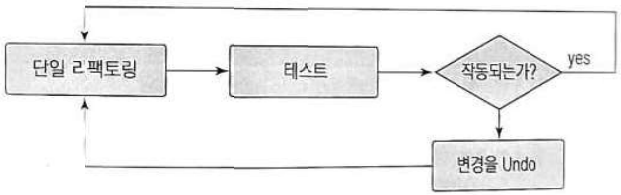

그림 9.4 리팩토링 과정

결국 리팩토링이란 소프트웨어를 보다 쉽게 이해할 수 있고 적은 비용으로 수정할 수 있도록 겉으로 보이는 동작의 변화 없이 내부구조를 변경하는 것이다.

예를 들어 객체지향 프로그램의 코드에서 반복 사용될 수 있는 코드가 있다면 메서드로 따로 떼어 낼 수도 있다. 또 다른 예로 클래스의 속성이 서브클래스 사이에 중복되어 있다면 속성을 끌어올려 슈퍼클래스에 정의할 수 있다. 이렇게 리팩토링은 코드에 드러난 버그라 할 수 없는 문제점들을 조금씩 변경하는 것이다.

9.4.3 코드 스멜

리팩토링은 Kent Beck이 주장한 것과 같이 곤욕스러운 상황에서 빠져나올 수 있는 방법이다. 어제의 결정이 적합하지 않으면 바꾸어 내일의 변경에 대비하여야 한다. 프로그램에 대한 작업을 어렵게 만드는 것을 찾아(리팩토링에서는 이를 나쁜 코드의 냄새라는 의미로 코드 스멜이라 부른다) 고치고 다듬어 나가는 과정이다. 코드 스멜이라 불릴 수 있는 것들은 다음과 같은 것이다.

- 읽기 어려운 프로그램
- 중복된 로직을 가진 프로그램
- 실행 중인 코드를 변경해야 하는 특별한 동작을 요구하는 프로그램
- 복잡한 조건문이 포함된 프로그램

리팩토링이 필요한 시점에 대한 정확한 기준을 제시하는 것은 쉽지 않은 일이다. 경험과 직관이 제일 좋은 기준이라고 할 수 있다. 코드 스멜은 리팩토링으로 해결될 수 있는 문제가 있다는 징후를 알려주는 것이다. 즉 얼마나 많은 변수가 지나친 것인지, 어느 정도의 길이가 지나치거 긴 메서드인지 정도를 감지할 필요가 있다.

Martin Fowler는 코드 스멜에 대한 리팩토링 방법을 [표 9.1]과 같이 제시하고 있다.

표 9.1 코드 스멜과 리팩토링

코드 스멜	설명	리팩토링
중복된 코드	기능이나 데이터 코드가 중복된다.	중복을 제거한다.
긴 메서드	메서드의 내부가 너무 길다.	메서드를 적정 수준의 크기로 나눈다.
큰 클래스	한 클래스에 너무 많은 속성과 메서드가 존재한다.	클래스의 몸집을 줄인다.
긴 파라미터 리스트	메서드의 파라미터 개수가 너무 많다.	파라미터의 개수를 줄인다.
두 가지 이상의 이유로 수정되는 클래스 (Divergent Class)	한 클래스의 메서드가 2가지 이상의 이유로 수정이 되면, 그 클래스는 한 가지 종류의 책임만을 수행하는 것이 아니다.	한 가지 이유만으로 수정되도록 변경한다.
여러 클래스를 동시에 수정 (Shotgun Surgery)	특정 클래스를 수정하면 그때마다 관련된 여러 클래스들 내에서 자잘한 변경을 해야 한다.	여러 클래스에 흩어진 유사한 기능을 한 곳에 모이게 한다.
다른 클래스를 지나치게 애용 (Feature Envy)	빈번히 다른 클래스로부터 데이터를 얻어와서 기능을 수행한다.	메서드를 그들이 애용하는 데이터가 있는 클래스로 옮긴다.
유사 데이터들의 그룹 중복(Data Clumps)	3개 이상의 데이터 항목이 여러 곳에 중복되어 나타난다.	해당 데이터들을 독립된 클래스로 정의한다.
기본 데이터 타입 선호 (Primitive Obsession)	객체 형태의 그룹을 만들지 않고, 기본 데이터 타입만 사용한다.	같은 작업을 수행하는 기본 데이터의 그룹을 별도의 클래스로 만든다.
Switch, If 문장	switch 문장이 지나치게 많은 case를 포함한다.	다형성으로 바꾼다(같은 메서드를 가진 여러 개의 클래스를 구현한다).
병렬 상속 계층도 (Parallel Inheritance Hierarchies)	[Shotgun Surgery]의 특별한 형태로서, 비슷한 클래스 계층도가 지나치게 많이 생겨 중복을 유발한다.	호출하는 쪽의 계층도는 그대로 유지하고 호출당하는 쪽을 변경한다.
게으른 클래스 (Lazy Class)	특정 클래스가 별로 사용되지 않는다.	제거하거나 다른 클래스에 합병한다. 상속인 경우 상속을 없앤다.
지나친 일반화 (Speculative Generality)	지금 당장 필요하지 않은 기능을 위해 미리 만들어 놓은 상속 관계	상속을 없애 버린다.
임시 속성	클래스의 속성들이 한두 번만 사용되는 임시 변수처럼 사용된다.	속성을 메서드의 내부로 옮긴다.
메시지 체인 (Message Chains)	특정 객체를 사용하기 위해 지나치게 많은 클래스를 거쳐야 한다.	메시지 체인을 거치지 않고 직접 사용할 수 있도록 한다.
미들 맨(Middle Man)	클래스가 가진 메서드 대부분이 다른 클래스에 책임을 넘긴다.	미들맨 역할의 객체를 제거한다.
부적절한 친밀성 (Inappropriate Intimacy)	불필요하게 자신의 정보를 다른 클래스에게 노출하고, 다른 객체에 국한된 정보를 알아내려고 한다.	다른 클래스가 레퍼런스를 유지하지 못하게 하고, 이 클래스가 다른 객체의 정보를 알지 않고 기능을 처리할 수 있도록 한다.
미완성된 라이브러리 클래스	다른 사람이 만든 라이브러리 클래스가, 내가 원하는 기능을 온전히 수행하지 못한다.	자신이 만든 클래스와 라이브러리 클래스 간에 래핑(Wrapping) 클래스를 만들어 연결한다.
데이터 클래스	특정한 클래스가 단순히 속성값을 읽거나 쓰기만 하고 아무 일도 하지 않는다.	이 클래스의 데이터를 주로 사용하는 다른 클래스의 메서드를 내부로 가져온다.
상속을 거부함 (Refused Bequest)	상위 클래스의 속성과 메서드가 하위 클래스에서 사용되지 않는다.	상위 클래스와 하위 클래스를 합친다.
주석(Comments)	코드를 이해할 수 없기 때문에 주석 처리를 한다.	주석이 없어도 코드를 이해할 수 있도록 변경한다.

9.4.4 리펙토링 사례

■ 메서드 추출

코드 안에 자주 발생할만한 코드, 즉 재사용 확률이 많은 코드는 메서드로 정의하고 이를 호출한다. 긴 로직이 포함되었다면 다른 곳에서 재사용하기 어렵다. 메서드 추출(extract method) 리팩토링은 코드의 중복을 제거할 수 있는 유용한 방법 중 하나이다.

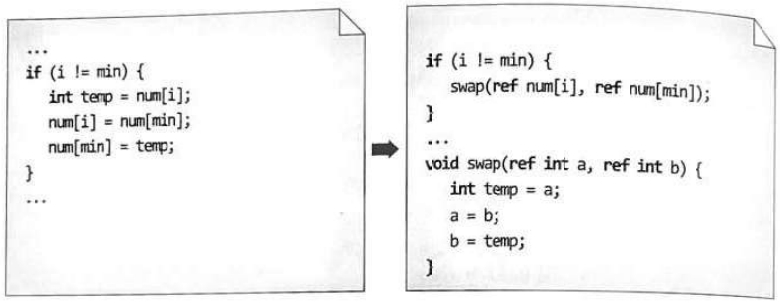

그림 9.5 메서드 추출

■ 클래스 추출

Customer 클래스의 일부로 phone이 포함되어 있는 것은 클래스 하나에 고유한 책임을 갖게 만드는 객체지향 모델의 정신에 맞지 않는다. 따라서 두 개의 단일 책임 클래스로 분할한다.

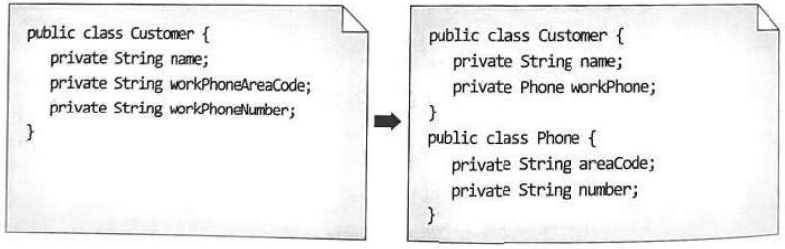

그림 9.6 클래스 추출

■ 서브 클래스 추출

클래스 안에 특정 인스턴스에게만 유용한 속성이나 메서드가 있다면 클래스의 서브 클래스로 만들고 그런 속성이나 메서드를 내려 보낸다. 이렇게 하면 원래의 클래스는 더욱 고유한 추상성을 유지하여 다른 곳에서도 쉽게 재사용할 수 있다.

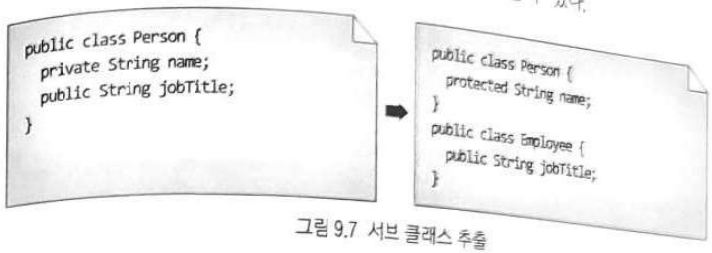

그림 9.7 서브 클래스 추출

■ 메서드 이동

메서드가 정의된 클래스보다 다른 클래스의 기능을 더 많이 사용한다면 다른 메서드로 옮긴다. Student 클래스는 더 이상 Course 인터페이스에 대하여 알 필요가 없다. 또한 isTaking() 메서드는 사용하는 데이터와 더 가까이 있게 되었으며 Course 클래스의 응집이 높아지고 결합이 낮아져 전체적인 설계가 향상되었다.

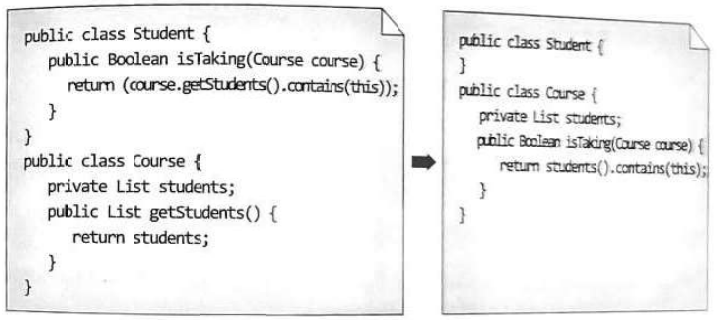

그림 9.8 메서드 이동

■ 인터페이스 추출

아래 사례에서 Customer 클래스의 외부 프로그램이 Customer 클래스 안의 이름을 접근할 수도 있고 또한 외부에서 객체를 XML로 직렬화하고 싶을 수도 있다. 하지만 toXML()을 Customer의 인터페이스로 만드는 것은 인터페이스 분리 원칙을 따르는 것이 좋다. 다목적 인터페이스 하나를 만드는 것보다 다양하게 구현된 특별한 인터페이스가 많아야 한다.

```
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: " + dateObBirth.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: " + dateObBirth.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