비기능성 기반 디자인 재구성

권재은[+] ○ 김형호[+] 배두환[+]
[+]한국과학기술원 전자 전산학과

Nonfunctionality-driven Design Refactoring

Kwon, Jae-Eun[+] ○ Kim, Hyung-Ho[+] Bae, Doo-Hwan[+]
[+]Dept. of Electrical and Computer Science, KAIST

요 약

소프트웨어 설계 단계에서 기능성(functionality)과 더불어 비기능성(non-functionality)에 관한 고려가 매우 중요하다. 그럼에도 불구하고 비기능성 고유의 복잡성으로 인해 두가지를 함께 해결하기는 매우 어려운 문제였고, 비기능성을 구현하는데 많은 난점이 있어왔다. 우리는 이러한 문제를 해결하기 위해 기능성의 유지를 보장하는 소프트웨어 수정 방법인 재구성(refactoring)에 착안하였다. 이를 사용하여 기능성과 독립적으로 비기능성을 획득하도록 설계를 수정할 수 있도록 도와주는 체계적인 비기능성 기반 디자인 재구성(Nonfunctionality-driven Design Refactoring)을 제안한다. 또한 우리가 제시한 방법을 간단한 웹 쇼핑몰 시스템 예제에 적용하여 확장성이 증가하도록 디자인 재구성을 하여본다.

1. 서론

소프트웨어 설계 단계에서 기능성(functionality)과 더불어 비기성(non-functionality)에 관한 고려가 충분히 이루어져야 한다. 하지만 이러한 고려는 비기능적 요구 사항 고유의 복잡성으로 인하여 기능적 요구 사항과 같이 체계적인 방법을 가지기 어려운 문제이다. 그로 인해 이제까지 전문적인 개발자에 의해 임의로 설계를 수정하는 과정을 통해 이루어졌다. 그 결과 소프트웨어 설계에 비기능성을 추가하는 과정은 복잡하고 많은 경험이 요구되는 작업이었다. 또한 이러한 과정을 통해 도출된 결과가 원래의 기능성을 그대로 유지하고 있는지, 원하던 비기능성을 향상시켰는지 판단하기 어려웠다.

이러한 문제점을 해결하기 위해 관점 기반 프로그래밍(Aspect-Oriented Programming)[2]과 원시 객체 규약(Meta-Object Protocol) 기법이 제안되었다. 이 기법들은 기능적 요구 사항과 비기능적 요구 사항을 다른 요소로 구현되도록 완전히 구분하여 비기능적 요구 사항을 획득하였다. 그러나 이 기법들은 그들 고유의 프로그래밍 언어를 비롯한 적용 환경에 관한 강한 제약 조건으로 인해 널리 적용되기 어려운 실정이다.

'재구성(refactoring)'은 기능성의 변질 없이 소프트웨어를 설계를 비롯 코드까지 자동적으로 전이시킬 수 있는 방법이다. 재구성에 관한 연구는 데이터 베이스 스키마 전이 연구에서 비롯하여 많은 규칙들과 케이스 툴 개발에 이르기까지 다양한 연구가 이루어졌다[1, 3, 6]. 우리는 이러한 재구성 방법을 사용하여 원하는 비기능적 요소를 증가시키도록 고안된 전략에 따라 소프트웨어 설계를 재구성하는 방법을 제안하겠다.

하지만 기존의 재구성에 관한 연구는 소프트웨어 수정 전후의 기능성의 유지는 보장해주었지만 수정에 관한 체계적인 가이드가 없이 개발자의 요구에 따라 임의로 수정하도록 하였다. 우리는 특정한 비기능성을 증가시키기 위해 기존의 재구성에서 제안된 방법들을 조합하고, 비기능성에 따른 전략으로 집중시킨 "비기능성 기반 디자인 재구성(Nonfunctionality-driven Design Refactoring : 이하 NDR)" *을 제안하겠다. 또한 서버 시스템의 확장성 향상을 위한 방법을 보임으로써 우리가 제안한 방법을 효용성을 보이도록 한다.

다음 장에서는 기존의 재구성 연구에 대해 살펴보고, 세부 구조와 동작에 관 해 3장에서 설명한다. 4장에서는 간단한 웹 쇼핑몰에 관한 사례 연구를 통해 서버 시스템의 확장성을 향상시켜보고, 5장에서 우리 연구의 결론과 향후 연구 방향을 살펴보도록 한다.

2. 기반 연구 : Refactoring

재구성(refactoring)은 소프트웨어의 기능성을 그대로 유지하면서 소프트웨어를 수정하는 방법을 말한다[3]. 이러한 재구성 적용 이후의 기능성 유지라는 성질에 착안하여 비기능성 향상을 위한 모델 수정에 재구성을 도입하게 되었다.

기존의 재구성 연구는 특정 예비 조건(pre-condition)을 만족할 때 기능성을 만족한다는 것이 증명된 많은 규칙들을 제시하고 있다[3]. 예를 들어 어떤 같은 이름의 클래스가 존재하지 않으면 새로운 이름의 클래스를 만들 수 있다는 규칙이 있다.

이러한 재구성의 진화 모드는 스키마 전이, 마이크로 아키텍처 디자인 패턴의 도입, 핫-스팟 기반 재구성의 세가지로 분류되고 있다[6].이 중 우리는 스키마 전이 형태의 재구성을 시도하고 있으며 설계 수준에서의 재구성에 관심을 두고 있다.

기존의 재구성에 관한 연구들은 재구성의 목적보다는 재구성의 방법 자체에만 관심을 두었다. 따라서 재구성을 위해 어떤 규칙을 어디에 사용할 것인가에 관한 문제에 관한 가이드라인이 부족하다.

3. 비기능성 기반 디자인 재구성 (Nonfunctional-driven Design Refactoring)

본 연구에서는 비기능성 기반 디자인 재구성(Nonfunctional-driven Design Refactoring:NDR) 프레임웍을 개발함으로써 기존의 컴포넌트 기반 † 소프트웨어 모델을 사용자의 요구에 따라 특정 비기능성을 향상시키는 체계적인 방법을 제시한다.

그림 1은 NDR을 이용한 모델 수정의 전체적인 구조를 나타내고 있다. 먼저 컴포넌트 기반의 재구성하고자 하는 '모델'과 추가하고자 하는 '비기능성'을 NDR 프레임웍에 입력한다. 그러면 NDR 내부에서는 각각의 비기능성에 관한 재구성 전략을 사용하면서, 재구성 과정마다 개발자와의 정보 교류를 통해 가장 효율적인 방향으로 융통성 있게 재구성을 위한 선택을 내린다. 이렇게 반복적인 재구성의 결과로 최종적으로 재구성이 끝난 모델이 산출된다.

* 본 연구에서는 설계 수준의 전이에 관심을 가지고 앞으로 '디자인 재구성'으로 한정짓는다.

† 우리는 재구성 대상이 되는 모델을 컴포넌트 기반 소프트웨어 모델로 가정한다.

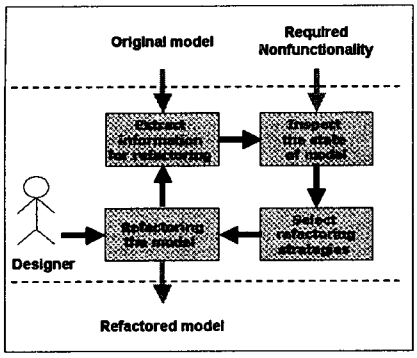

그림 1: Nonfunctional-driven Design Refactoring의 구조

그림 1의 가운데 부분에 나타난 바와 같이 NDR의 자세한 동작 구조는 1.정보 추출 2. 모델의 상태 점검 3. 재구성 전략 결정 4. 모델 재구성 단계로 이루어진다. NDR은 위 네가지 단계를 최종적으로 원하는 상태 즉, 더이상 재구성이 필요하지 않은 상태에 만족할 때까지 반복한다. 이 네가지 단계를 아래에서 각각의 역할과 동작에 관해 자세히 설명한다.

3.1 정보 추출(Extract information for refactoring)

정보 추출의 단계에서는 모델의 재구성을 위해 첫번째로 재구성을 위해 필요한 정보들을 입력 모델로부터 수집한다. 그리고 재구성의 단계마다 새롭게 수정된 정보들을 다시 조합한다.

모델의 재구성을 위해 필요한 정보들은 아래와 같다.

- 재구성 진행 과정을 판단하기 위한 컴포넌트의 '이름'과 '타입'
- 메소드 사이의 의존관계를 보이기 위한 인터페이스에 나타난 메소드의 '이름'과 '관계'
- 재구성 규칙을 적용하기 위한 모델 요소들의 '이름'과 '타입'
- 모든 구성 요소들의 의존관계 파악을 위한 '컨트롤과 데이터 의존성'

이렇게 수집되고 재구성 과정에서 변형되는 정보들은 NDR 적용 과정에서 재구성을 위한 모델의 상태와 재구성 전략 결정에 이용된다.

3.2 모델의 상태 점검(Inspect the state of model)

다음으로 재구성 되는 모델의 정보를 바탕으로 어떤 부분을 재구성해야 할지, 재구성이 앞으로 어떤 방향으로 진행되어야 할지, 언제 재구성을 끝내야 할지 등을 판단하기 위해 현재 재구성을 위한 모델의 상태를 점검하는 과정이 필요하다.

재구성 상태를 가지는 모델을 구성하는 단위로써 컴포넌트와 클래스, 속성과 메소드를 고려한다. 컴포넌트와 클래스는 재구성 전략을 적용하는 단위, 속성과 메소드는 재구성 규칙을 적용하는 단위가 된다.

이런 각 모델의 단위마다 재구성을 위한 상태를 판단하는 기준으로 유동점과 안정점을 판단하게 된다.

- 유동점(floating spot) : 현재 재구성 되어야할 상태에 있는 요소
- 안정점(stable spot) : 현재 재구성 될 필요가 없는 요소

매 재구성의 단계마다 재구성 전략에 따라 모델의 유동점과 안정점을 판단하여 유동점을 없애기 위한 재구성을 시도한다.

Definition .1 (속서와 메소드의 사례) 재구성 전략에 따라 재구성이 필요한 상태일 때 유동점이 되고 그렇지 않으면 안정점이 된다.

Definition .2 (커군포지트와 클래스의 사례) 자신의 모든 구성 요소가 안정점이면 안정점이고 그렇지 않으면 유동점이 된다.

3.3 재구성 전략 결정(Select refactoring strategies)

재구성 전략은 특정 비기능성을 증가시키기 위한 재구성을 이끌어내는 체계적인 가이드라인이다. 따라서 NDR에는 재구성 전략으로써 특정 시스템과 환경 하에서 특정 비기능성을 향상시키기 위해 재구성 되어야할 상태, 재구성된 결과 상태와 그에 도달하기 위한 알고리즘을 정의한다.

안정점, 유동점에 관한 상태 정의와 전략은 각각의 비기능성에 따라 새롭게 정의된다. 이러한 정의는 안정점의 상태는 비기능성이 향상된 상태이도록 해주어야 한다. 따라서 모델은 모든 컴포넌트 또는 클래스가 안정점일 때 재구성이 완료된 상태 즉, 원하는 비기능성이 향상된 상태가 된다.

```
1   loop1{
2     Extract and reorganize information;
3     Find new floating components;
4     Designers choose a component;
5     loop2{
6       Find next floating element;
7       Choose and apply the specific strategy and rules;
8       if(no next floating element) break;
9     }
10    Make the component to be stable;
11    if(all components are stable
12      or designers want to to stop) break;
13  }
```

그림 2: NDR pseudo 알고리즘

재구성은 유동성 컴포넌트의 수를 줄이고, 안전성 컴포넌트로 대체하려는 방향으로 진행되지만 하나의 컴포넌트를 안정성으로 바꾸기 위해 현재 안정성인 다른 컴포넌트들이 유동성으로 전이되어 버리는 문제가 발생할 수 있다. 우리는 이것을 '전략 역전 문제'라 부르겠다. 이러한 전략 역전 문제를 방지하기 위해 그림 2의 loop2에서 한번 시도한 재구성에 의해 발생된 모든 관계된 유동점들을 연속적으로 재구성 하도록 하고 있다.

그림 2의 유동점을 선택하거나, 특정 전략을 적용하는 단계는 각각의 비기능성에 관계된 것으로 확장성 증가에 관한 예는 4장에서 설명된다.

하나의 재구성 전략은 관련된 많은 재구성 규칙의 조합을 포함한다.

- 재구성 규칙(Refactoring rules)

  재구성 전략이 비기능성을 고려한 상위 레벨의 정책이라면 재구성 규칙은 어플리케이션과 독립적으로 모든 재구성에 일반적으로 적용될 수 있는 재구성 전후에 있어 기능성의 유지가 보장된 하위 레벨의 법칙이라 할 수 있다.

  재구성 규칙은 2장에서 설명한 바와 같이 클래스, 어트리뷰트, 메소드의 생성, 삭제, 이동 등에 관해 다양하게 제시되어있다.

  이러한 재구성 규칙은 각각의 비기능성에 따른 재구성 전략에 의해 조합되어 매 단계의 재구성에 적용된다.

  재구성 규칙은 기능성의 유지가 증명된 법칙들이므로 매 단계의 재구성마다 재구성에 의한 기능의 변화를 우려할 필요가 없다.

3.4 모델 재구성(Refactoring the model)

설계 모델에 적용할 재구성 전략이 결정되었으면 이 전략을 모델에 적용한다. 재구성 전략 알고리즘은 재구성을 실행하는 디자이너와의 상호작용을 통해 디자이너의 기호에 따라 유연하게 결정될 수 있다.[1] 따라서 모델 재구성 단계에서는 개발자의 결정과 함께 재구성이 이루어지게 된다.

모델 재구성 후 NDR은 모델이 재구성을 끝내야할 상태인지 확인하게 된다. NDR 알고리즘에 따라 모든 컴포넌트가 안정점이거나 개발자가 원하는 경우 재구성을 종료한다.

[1] 그림 2의 진하게 표기된 부분은 디자이너의 선택이 개입되는 부분이다.

4. 사례 연구 : 간단한 웹 쇼핑몰 모델에 적용

NDR을 컴포넌트 기반의 간단한 웹 쇼핑몰 서버 모델의 확장성 증가를 위한 재구성에 적용해본다. 컴포넌트 기반 서버 시스템의 확장성 증가를 위해서는 서버가 클라이언트의 정보, 즉 상태를 유지 하지 않도록 하는 방법이 있다. 컴포넌트가 상태를 유지하지 않는다면 이 컴포넌트는 사용되지 않을 때는 유지하지 않고 필요한 경우에 새로운 컴포넌트를 생성시키면 된다. 즉 상태를 유지하는 경우보다 훨씬 적은 양의 메모리를 하게 된다. 바꾸어 말해 같은 양의 자원으로 많은 수의 클라이언트를 동시에 관리할 수 있는 더 확장성이 좋은 서버가 된다.[4, 5, 7]

이러한 방법을 기반으로 NDR에서는 '상태 유지 컴포넌트(stateful component)' 를 '상태 비유지 컴포넌트(stateless component)'로 대체하는 전략을 취한다. 상태 유지 컴포넌트는 메소드 호출 사이에 상태 정보를 유지하면서 변환시키는 즉, 인스턴스 변수를 갖는 컴포넌트이고 상태 비유지 컴포넌트는 어떠한 정보도 유지하지 않는 즉, 인스턴스 변수를 갖지 않는 컴포넌트이다.[7]

4.1 서버 시스템의 확장성 증가를 위한 재구성 전략

상태 유지 서버 컴포넌트가 갖는 클라이언트의 정보는 클라이언트의 '주키(primary key)'와 '일시 정보군(transient)'으로 구분될 수 있다. 이러한 정보를 상태 유지 컴포넌트로부터 추출하여 다른 곳으로 이동함으로써 상태 비유지 컴포넌트를 만들 수 있다.

따라서 상태 유지 서버 컴포넌트의 주키와 일시 정보군이 재구성의 대상 즉, 유동점이 되고, 이러한 정보를 가지는 상태 유지 컴포넌트 또한 유동성을 띤 컴포넌트가 된다. 이 외의 요소들은 모두 안정점이 되고 서버 시스템의 확장성 증가를 위한 재구성은 모든 컴포넌트가 안정점 즉, 상태 비유지 컴포넌트가 될 때 모두 종료된다.

우리는 클라이언트의 주키는 컴포넌트의 상대적인[§] 클라이언트로 옮기고, 일시 정보군은 데이터베이스로 옮기도록 하였다.

4.2 웹 쇼핑몰 서버 모델에 적용

위에서 정의한 재구성 전략을 간단한 웹 쇼핑몰 서버 모델의 확장성을 증가시키도록 적용해본다.

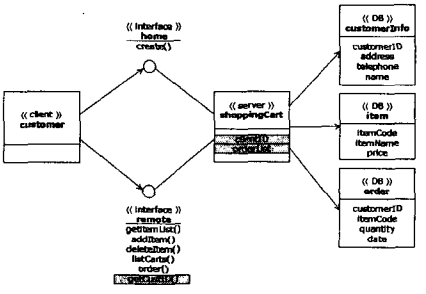

그림 3: 재구성 전에 상태 유지 서버 시스템 디자인

그림 3은 재구성을 위해 입력된 상태 유지 서버 모델이다. 이 모델의 확장성을 증가시키기 위해 아래와 같은 재구성 단계를 거쳐 그림 4의 상태 비유지 서버 모델을 생성할 수 있다.

1. 클라이언트의 primary key인 clientID를 찾는다.
2. 클라이언트의 transient인 orderList를 찾는다.
3. 위에서 찾은 두 개의 유동점을 바탕으로 유동 컴포넌트 shoppingCart를 재구성의 대상으로 선택한다.
4. clientID를 클라이언트인 customer로 옮긴다.
5. clientID와 관계된 getCustID 메소드를 삭제한다.

[§] 컴포넌트에게 클라이언트 역할을 하는 모든 요소를 말한다.

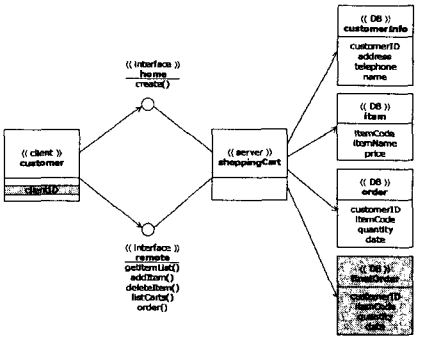

그림 4: 재구성 후의 상태 비유지 서버 시스템 디자인

6. orderList를 데이터베이스로 옮긴다.
7. 더이상 유동점이 없으므로 재구성을 완료한다.

5. 결론 및 향후연구

우리는 설계 단계에서 비기능성에 관한 고려를 쉽게 반영할 수 있는 비기능성 기반 디자인 재구성 방법을 제안하였다. NDR은 개발자의 경험을 '전략'이라는 체계적인 방법으로 재사용함으로써 기존에 소프트웨어 설계 시 비기능성의 반영을 위하여 필요하던 숙련된 전문성을 줄이고자 하였다. 또한 재구성(refactoring)이라는 널리 받아들여지고 있는 기술을 사용하여 새로운 방법의 도입을 용이하게 하였다.

마지막으로 우리는 서버 시스템의 확장성을 증가시키는 전략을 보이고, 간단한 웹 쇼핑몰 서버 시스템 모델을 NDR을 사용하여 재구성해보았다. 이를 통해 컴포넌트 기반 소프트웨어 설계를 기능성에 대한 영향 없이 체계적인 방법으로 특정 비기능성을 증가시킬 수 있음을 보였다.

NDR을 도입함에 있어 비기능성의 특성상 재구성 전략을 수립하는 일이 매우 어렵고, 대상 시스템의 형태와 비기능성의 종류에 따라 매우 많은 수의 개별적인 전략이 필요하다는 단점이 개선하기 위해 향후 재구성 전략 수립을 위한 일반적인 방법에 대한 연구가 필요하다. 더불어 앞으로 성능, 재사용성 등의 다른 비기능성을 지원하도록 전략을 확충하고, 코드 수준까지 재구성 가능하도록 확장할 수 있다.

참고 문헌

[1] W. G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring", ACM Transactions in Software Engineering and Methodology, Vol 2. No 3, 1993

[2] G. Kiczales, et al., "Aspect-Oriented Programming", European Conference on Object-Oriented Programming, 1997

[3] W. F. Opdyke, "Refactoring Object-Oriented Frameworks", PhD thesis, University of Illinois, 1992

[4] D. Reed, T. Trewin, Mai-lan Tomsen, "Microsoft Transaction Server Helps You Write Scalable, Distributed Internet Apps", Microsoft systems journal, 1997

[5] A. Thomas, "Enterprise JavaBeans Technology", SUN White paper, 1998

[6] L. Tokuda and D. Batory, "Automating Three Modes of Evolution for Object-Oriented Software Architectures", Conference on Object-Oriented Technologies and Systems, 1999

[7] T. Valesky, "Enterprise JavaBeans", Addison-Wesley 1999