DSM을 이용한 소프트웨어 디자인 개선

김준환[O1], 이민순[1], 윤승준[2], 이병수[1]
[1]인천대학교 컴퓨터공학과, [2]㈜ 이지막
{andyhwan, zerone}@incheon.ac.kr, ceo@ezmarc.com, bsl@incheon.ac.kr

Software Design Improvement using DSM

Jun-Hwan Kim [O1], Min-Soon Lee [1], Soong-Joon Yoon [2], Byung-Soo Lee [1]
[1]Dept. of Computer Science and Engineering, University of Incheon, [2]ezMARC Co., Ltd.

요 약

클래스들 사이의 Circular Dependency는 결합도를 높이고 클래스의 독립적인 수정 및 재사용을 어렵게 만든다. 이러한 Circular Dependency는 더 나아가 전체적인 소프트웨어 디자인을 부패시키고, 소프트웨어의 유지보수를 더욱 어렵게 만든다. 또한, 소프트웨어 개발 과정에서는 새로운 요구 사항의 추가나 계획되지 않은 변경이 빈번하게 일어난다. 따라서, 소프트웨어 디자인이 부패하는 것을 방지하려면 클래스 사이에 존재하는 의존관계가 반드시 관리되어야 한다. 본 논문에서는 DSM을 이용하여 소프트웨어를 분석하고, 디자인 패턴을 적용하여 구조를 개선한 후 객체지향 설계원칙에 부합함을 보이고, DSM을 이용하여 개선된 결과를 나타낸다. 개선된 결과를 통하여 DSM이 소프트웨어 디자인 개선 및 유지보수에 있어서 효과적으로 이용될 수 있음을 보인다.

1. 서론

소프트웨어 디자인은 새로운 요구 사항의 추가나 계획되지 않은 변경들에 의해서 퇴화한다. 이러한 변경이 반복될수록 결과적으로 소프트웨어 디자인을 변경하기 어렵고, 변화에 취약하며, 재사용하기 어렵고, 유지보수를 더욱 힘들게 만든다. 따라서 이러한 소프트웨어 디자인의 부패를 방지하기 위해서는 클래스 사이의 의존관계를 관리해야 한다.

Circular Dependency는 둘 이상의 모듈이 올바르게 동작하기 위해 직·간접적으로 의존하는 관계이다. 소프트웨어 디자인에 있어서 Circular Dependency는 반드시 피해야 하며, 클래스들 사이의 Circular Dependency는 결합도를 높여 클래스가 독립적으로 재사용하기 어렵게 만든다. 클래스 하나를 수정하기 위해서 관련된 부분을 모두 이해해야 하고, 수정하려는 클래스에 의존하는 다른 클래스들도 영향을 받게 된다. 이러한 Circular Dependency는 결과적으로는 전체적인 소프트웨어의 디자인을 부패시킬 뿐만 아니라 유지보수를 어렵게 만든다.

개발하고 있는 ezMARC 솔루션은 MARC(MAchine-Readable Cataloging) 정보 구축 및 서비스를 위한 소프트웨어로 개발이 진행될수록 새로운 기능의 추가나 변경이 많은 부분에 영향을 주어 전체적인 개발 진행속도가 느려지는 문제가 있어, 이에 대한 원인 분석 및 해결이 필요하며, 이러한 과정을 본 논문에서 다루고자 한다.

논문의 구성은 다음과 같다. 2장에서는 DSM과 객체지향 설계 원칙을 살펴보고, 3장에서는 소프트웨어 디자인 분석을 위해서 DSM을 적용하고 객체지향 설계원칙에 기반을 두고 문제점을 분석한 다음, 디자인 패턴을 이용하여 문제를 해결한다. 4장에서는 결론과 향후 과제에 대해 논의한다.

2. 관련연구

일반적으로 소프트웨어 개발이 진행될수록 기능이 추가되고 클래스가 늘어나게 되어 클래스나 패키지 사이의 의존관계를 일일이 파악하기 어렵다. 본 논문에서는 클래스 사이의 의존성을 분석하고 개선하기 위해서 DSM(Dependency Structure Matrix)[1]을 이용한다. DSM은 제품 개발 프로세스를 최적화하기 위해서 만들어졌으며, 여러 분야에서 널리 이용되고 있다. 소프트웨어 개발에서는 주로 소프트웨어의 아키텍처 분석 및 관리를 위해 사용된다.

2.1. Dependency Structure Matrix

		1	2	3	4
TASK A	1			X	X
TASK B	2			X	
TASK C	3	X			X
TASK D	4				

그림 1. DSM의 예

DSM은 제품 개발 프로세스를 최적화하기 위해서 Design Structure Matrix라는 이름으로 만들어졌다. Dependency Structure Matrix, Problem Solving Matrix(PSM)라고도 알려졌으며, DSM은 복잡한 시스템이나 프로젝트를 분석 및 관리할 수 있는 작은 Matrix 형태로 나타낸다[2, 3].

그림 1은 4개의 Task로 구성된 DSM이다. 여기서 Task는 네임스페이스나 클래스를 의미한다. DSM은 열을 기준으로 읽는다. 1번 열은 Task A가 Task C에 의존한다는 의미이고, 3번 열은 Task C가 Task A와 Task B에 의존한다는 의미이다. 위의 DSM에서 Task A와 C는 서로 의존하는데 이러한 관계를 Circular Dependency라고 한다. 이 경우 한쪽이 변경되면 다른 한쪽의 변경을 불러오는 상황이 일어날 수 있다. 더욱 심각한 경우에는 의존하는 모든 클래스로 연쇄적 변경이 파급되는 파도 효과(Ripple Effect)를 불러오게 된다[4].

		1	2	3	4
TASK D	1				
TASK A	2	X		X	
TASK C	3	X	X		
TASK B	4			X	

그림 2. Partitioning 적용 후의 DSM

그림 2는 그림 1에 Partitioning이 적용된 후의 DSM을 보여준다. Partitioning은 서로 연관 있는 작업을 그룹별로 정렬된 클러스터 형태로 나타낸다. Partitioning을 적용하면 분산된 태스크 클러스터들 사이의 의존관계를 쉽게 파악하는 데 도움이 된다.

소프트웨어 개발에서도 전체적인 디자인이나 아키텍처를 분석하기 위해서 DSM을 사용하고 있으며 대표적인 상용 소프트웨어로는 Lattix[1, 5], NDepend[6] 등이 있으며, 본 논문에서는 소프트웨어 디자인 분석을 위해 .Net Reflector[7]와 Dependency Structure Matrix Plugin[8]을 사용한다.

2.2. 객체지향 설계원칙

객체지향 설계원칙들은 소프트웨어 디자인을 구성하고, 모듈 사이의 의존성을 관리하는데 도움을 준다. 객체지향 설계원칙들은 다음과 같다[9].

1. 단일 책임 원칙
   - 클래스는 단 한 가지의 변경 이유를 가져야 한다.
2. 개방-폐쇄 원칙
   - 소프트웨어의 개체(클래스, 모듈, 함수 등등)는 확장에 대해 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다.
3. 리스코프 치환 원칙
   - 하위타입(Subtype)은 그것의 기반 타입(Base type)에 대해 치환 가능해야 한다.
4. 의존관계 역전 원칙
   - 상위 수준의 모듈은 하위 수준의 모듈에 의존해서는 안 된다. 둘 다 추상화에 의존해야 한다.
   - 추상화는 구체적인 사항에 의존해서는 안 된다. 구체적인 사항은 추상화에 의존해야 한다.
5. 인터페이스 분리 원칙
   - 클라이언트가 자신이 사용하지 않는 메소드에 의존하도록 강제되어서는 안 된다.

이러한 원칙에 주의를 기울이지 않으면, 새로운 요구사항의 추가나 계획적이지 않은 변경들로부터 발생하는 새로운 의존관계에 의해서 소프트웨어 디자인이 부패하게 된다. 소프트웨어 디자인의 부패 때문에 소프트웨어는 변경하기 어렵고, 변화에 취약하고, 재사용하기 어렵게 되며, 유지보수를 더욱 힘들게 만든다. 따라서 이러한 소프트웨어 디자인의 부패를 미리 방지하기 위해서는 클래스 사이의 의존관계를 반드시 관리해야 한다[10].

3. DSM을 이용한 소프트웨어 디자인 분석 및 개선

분석 대상 소프트웨어는 C#을 이용하여 개발되고 있으며, 주요 기능을 담당하는 클래스들 크기가 너무 크고, 지나치게 많은 멤버 변수와 메소드를 포함하고 있었다. 또한, 주요 클래스에 의존하도록 구성된 클래스들 때문에 새로운 기능 추가나 변경이 예상보다 많은 부분에 영향을 주어 전체적인 개발 진행속도가 느려지는 문제가 있었다.

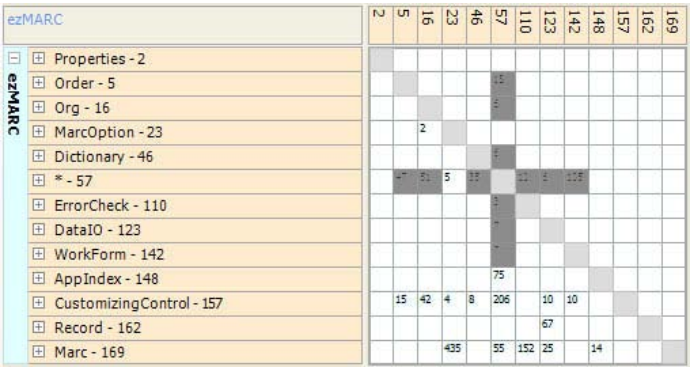

그림 3. MARC 정보 구축 및 서비스 소프트웨어의 DSM

위의 그림은 MARC 정보 구축 및 서비스 소프트웨어 내부의 전체적인 의존관계를 파악하기 위해서 DSM을 적용한 것이다. 그림 3은 네임스페이스들 사이의 의존관계를 나타낸다. 강조된 셀은 네임스페이스들 사이에 존재하는 Circular Dependency를 의미하며, 이는 소스

코드 상의 문제가 아닌 네임스페이스들 사이에 존재하는 의존관계를 나타낸다. Circular Dependency는 부정적인 영향이 더 크기 때문에 피해야 하고, 제거되어야 하는 의존관계이다. 일반적으로 소프트웨어 개발이 진행될수록 기능이 추가되고 클래스가 늘어나게 된다. 따라서 네임스페이스 사이의 의존관계 보다는, 세부적인 클래스들 사이의 의존관계에 대한 분석 및 관리가 더 중요하다.

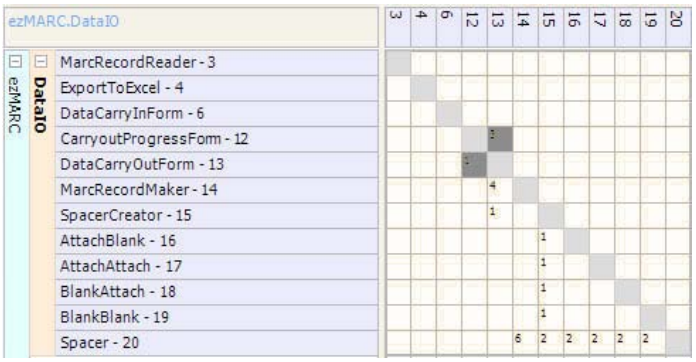

그림 4. 클래스 사이의 Circular Dependency

위의 그림은 그림 3의 DataIO 네임스페이스를 확장한 것으로, CarryOutProgressForm과 DataCarryOutForm 클래스 사이에 Circular Dependency가 형성되어 있는 것을 알 수 있다. 두 클래스는 각각 데이터가 반출 되는 상황을 보여주는 역할, 데이터를 반출하는 역할을 담당한다. 위와 같이, Circular Dependency가 클래스 사이에 존재하는 경우에는 두 클래스 모두 확장하기 어렵고, 변경이 클래스에 의존하는 다른 클래스로 전파되는 파도효과가 발생한다. 또한, 수정이나 유지보수를 어렵게 하고, 전체적인 소프트웨어 디자인을 부패시킨다.

그림 5는 그림 4에서 Circular Dependency가 형성된 클래스들을 UML로 나타낸 것이며, 그림 6은 그림 5의 CarryOutProgressForm 클래스 소스 코드의 일부이다. CarryOutProgressForm은 호출하는 폼에 따라서 동작을 다르게 하는 방식으로 구성되어 있다. 또한, 데이터 반출 상황을 보여주는 역할뿐만 아니라, 호출하는 폼이 담당하는 데이터 형식으로 파일을 반출하는 역할도 가지고 있다.

그림 5. 개선 대상의 UML

```
private void CarryOutProgressForm_Shown(object sender, EventArgs e)
{
    if (Owner is DataCarryOutForm)
    {
        progressBar.Maximum = macList.Count;

        outputThread = new Thread(MacFileOutPut);
        outputThread.Start();
    }
    else if (Owner is SimpleListForm)
    {
        // To-do
    }
}
```

그림 6. CarryOutProgressForm의 일부

결과적으로 CarryOutProgressForm은 호출하는 폼의 형식이 달라지면 그 동작이 새롭게 추가되거나 수정되는 구조이므로 개방-폐쇄 원칙(OCP)을 위반하고, 파일을 반출하는 역할도 추가로 담당하여 단일 책임 원칙(SRP)을 위반한다.

이러한 구조로 인해, CarryOutProgressForm은 재사용이 어렵고, 수정이 필요한 경우, 의존하는 클래스들에 미치는 영향을 고려해야 한다. 위의 구조를 개선하기 위해 Observer Pattern과 Template Method Pattern을 이용하였다. Observer Pattern은 Circular Dependency를 해결하는데 가장 널리 사용되고 있는 방법으로, 일 대 다의 의존관계를 정의해 두어, 한 객체의 상태가 변할 때 그 객체에 의존성을 가진 다른 객체들이 그 변화를 통지받고 자동으로 갱신될 수 있게 만든다[11].

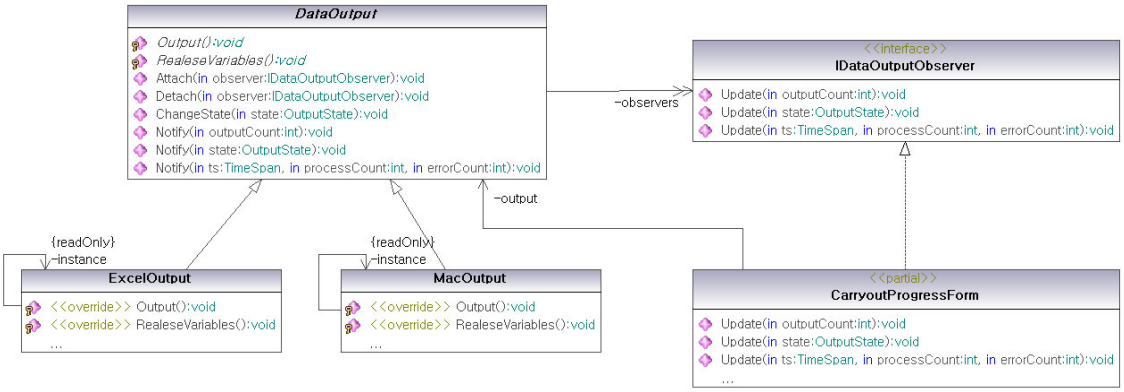

그림 7. 개선된 결과 UML

그림 7은 Observer Pattern을 적용하여 개선된 결과의 UML이다. 그림 7을 보면 Template Method Pattern[11]을 이용하여, DataOutput이라는 추상 클래스를 정의하고, DataOutput을 상속하는 하위 클래스인 MacOutput에서 클래스의 역할에 맞게 동작하는 구조로 개선하였다. 또한, DataOutput을 상속하는 ExcelOutput이라는 클래스를 추가하였다. 이를 통해 관찰 대상 객체를 변경하지 않고도 새로운 관찰 객체를 추가할 수 있었다.

이는 개방 폐쇄 원칙(OCP)을 만족하며, 각각의 기반 타입에 대해 하위 타입 객체로 교체 가능하므로 리스코프 교체 원칙(LSP)도 만족한다.

IDataOutputObserver는 추상화된 인터페이스이고, 하위 타입인 CarryoutProgressForm이 기반 타입에 의존한다. DataOutput의 구체적 메소드도 Observer에 의존하므로 이는 의존관계 역전 원칙(DIP)도 만족한다.

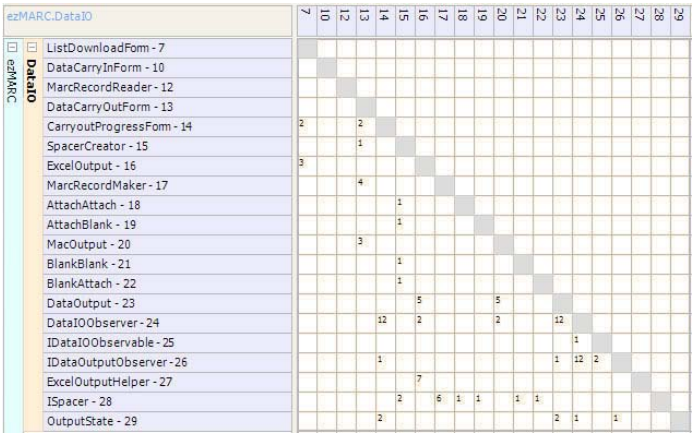

그림 8. Circular Dependency가 해결된 DSM

그림 8은 Observer Pattern을 이용하여 그림 4의 Circular Dependency를 해결한 DSM으로 Circular Dependency가 사라진 것을 알 수 있다.

4. 결론 및 향후 연구

소프트웨어 개발 과정에서는 요구 사항의 추가나 계획되지 않은 변경이 빈번하게 일어나며, 소프트웨어 개발이 진행될수록 클래스가 늘어나는 것이 일반적이다. 클래스나 패키지의 숫자가 많은 경우에는 의존관계를 일일이 파악하는 것이 어려우며, 클래스들 사이에 존재하는 Circular Dependency는 새로운 기능의 추가나 변경을 어렵게 만든다. 따라서 Circular Dependency로 인해 발생할 수 있는 문제를 방지하기 위해서 반드시 제거되어야 하며, 항상 관리되어야 한다.

구성요소 사이의 의존관계를 관리하고 디자인 분석을 위해서 DSM을 이용하는 것은 의존성을 좀 더 쉽게 분석하고 관리할 수 있도록 해주며, Circular Dependency를 파악하는 데 효과적이다.

본 논문에서는 DSM을 이용하여 소프트웨어 내부 구성요소의 전체적인 의존관계를 파악하고 그 일부를 Observer Pattern을 이용하여 Circular Dependency를 제거하고 개선하였다.

향후 연구로는 다양한 디자인 패턴과 리팩토링[12] 기법들을 활용하여 클래스 사이의 의존관계를 관리하는 방법에 대한 연구를 진행할 것이다.

참고문헌

[1] N. Sangal, E. Jordan, V. Sinha and D. Jackson, “Using dependency models to manage complex software architecture,” *Proc. of the 20th Annual ACM/SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA 2005)*, pp. 167-176, 2005.

[2] The Design Structure(DSM), http://www.dsmweb.org

[3] http://en.wikipedia.org/wiki/Design_Structure_Matrix

[4] 손영수, “Dependency로 보는 소프트웨어 아키텍처,” 마이크로소프트웨어 12월호, pp. 270-275, 2009.

[5] Software Architecture, Software Quality, Impact Analysis, Dependency Management and DSM Tools, http://www.lattix.com

[6] NDepend Homepage, http://www.ndepend.com

[7] .Net Reflector, class browser, analyzer and decompiler for .Net, http://www.red-gate.com/products/reflector/

[8] Dependency Structure Matrix Plugin for .Net Reflector, http://tcdev.free.fr/

[9] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003.

[10] R. Martin, “Design Principles and Design Patterns,” http://www.objectmentor.com, 2000

[11] E. Gammar, R. Helm, R. Johnson, J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995.

[12] Martin Fowler, *Refactoring : Improving the Design of Existing Code*, Addison-Wesley, 1999.