16 대규모 구조

Domain-Driven Design, Modeling, Design

> 프로젝트의 규모가 너무 크고 복잡하여
> Module을 분해하고, 일정 수준의 디스틸레이션을 거치는 것만으로
> 복잡성을 줄일 수 없다면 어떻게 해야 할까?

나무보다 숲.

넓은 시각으로 시스템에 관해 토의하고 이해하게끔 돕는 언어 고수준 개념이나 규칙, 또는 둘 모두는 전체 시스템에 대한 설계 패턴을 확립한다.

전체 시스템을 포괄하고 각 부분의 책임을 자세히 알지 못해도 **전체적인 관점**에서 해당 부분의 위치를 어느 정도 이해하는 데 도움을 주는 **규칙 또는 규칙과 관계의 패턴**을 고안하라.

대부분의 대규모 구조는 UML로 표현할 수도 없고 그럴 필요도 없다.
대부분의 대규모 구조는 모델과 설계의 형태를 만들고 설명하지만 UML에 나타나지는 않는다. 대규모 구조는 **설계에 관한 또 다른 수준의 의사소통**을 제공한다.

flowchart LR modelDrivenDesign["MODEL-DRIVEN DESIGN"] systemMetaphor["SYSTEM METAPHOR"] responsibilityLayer["RESPONSIBILITY LAYER"] knowledgeLevel["KNOWLEDGE LEVEL"] pluggableComponentFramework["PLUGGABLE COMPONENT FRAMEWORK"] evolvingOrder["EVOLVING ORDER"] ubiquitousLanguage["UBIQUITOUS LANGUAGE"] modelDrivenDesign --"사고를 이끄는 수단으로 활용"--> systemMetaphor modelDrivenDesign --"계층화의 대상으로 사용"--> responsibilityLayer modelDrivenDesign --"인공적인 행위를
분리하는 대상으로 사용"--> knowledgeLevel modelDrivenDesign --"개별 기여도를 분리"--> pluggableComponentFramework modelDrivenDesign --"구조를 발견.정제하는 데 활용"--> evolvingOrder systemMetaphor --"추가"--> ubiquitousLanguage responsibilityLayer --"이름을 추가"--> ubiquitousLanguage

EVEOLVING ORDER

> 문제는 참고할 규칙이 얼마나 엄격하고 어디에서 유래했느냐다.
> 설계를 관장하는 규칙이 정말로 상황에 맞아떨어진다면
> 방해가 되지 않고 실질적으로 도움이 되는 방향으로 개발을 이끌 뿐 아니라 일관성도 제공할 것이다.

개념적인 대규모 구조가 애플리케이션과 함께 발전하게 해서 발전 과정에서 전혀 다른 형식의 구조로도 **변화할 수 있게 하라.**
반드시 세부적인 지식을 토대로 내려야 할 세부적인 설계 및 모델과 관련된 의사결정을 **과도하게 제약해서는 안 된다.**

CONTEXT MAP과 달리 대규모 구조는 **선택사항**이다. 사실 MODULE로 분해했을 때 충분히 이해할 수 있는 정도로 시스템이 간단하다면 대규모 구조가 필요하다지 않다. **대규모 구조는 어떤 구조를 개발하는 데 부자**

연스러운 제약조건을 강제하지 않고도
시스템을 명확하게 만드는 것으로 판명될 때 적용해야 한다. 맞지 않는 구조는 차라리 없느니만 못하므로 종합적인 구조를 얻으려 애쓰기보다는 발생한 문제를 해결하는 최소한의 집합을 찾는 편이 가장 낫다. **적을수록 더 많은 법이다.**

SYSTEM METAPHOR

객체 패러다임과 조화를 이루는, 느슨하고 쉽게 이해할 수 있는 대규모 구조

eg. 방화벽, 계층, 중앙, 커널, 부모/자식, 마스터/슬레이브 등등

XP의 핵심 실천사항 가운데 하나라서 유명한 접근법으로 자리잡았다.

어떤 시스템의 구체적인 비유가 나타나 팀원의 상상력을 포착하고 유용한 방향으로 사고를 이끄는 것으로 보인다면 그것을 대규모 구조로 채택하라. 이러한 은유를 중심으로 설계를 구상하고 **UBIQUITOUS LANGUAGE**로 흡수하라. SYSTEM METAPHOR는 시스템에 관한 의사소통을 촉진하고, 더불어 해당 시스템의 갭라도 이끌 것이다. 이렇게 되면 시스템의 여러 부분과, 심지어 여러 BOUNDED CONTEXT에 걸쳐 일관성이 증대된다. 그러나 모든 은유는 부정확하므로 지속적으로 은유가 지나치거나 적절치 못한가를 재점검하고, 방해가 된다면 언제든 버릴 준비를 한다.

"미숙한 은유"와 그것이 필요없는 이유

XP 커뮤니티 일각에서는 도메인 모델 자체를 “미숙한 은유”라고 이야기해오고 있으나 성숙한 도메인 모델은 결코 미숙하지 않다.

RESPONSIBILITY LAYER

MODEL-DRIVEN DESIGN을 만들고 CORE DOMAIN을 정제하는 데 성공했지만, 설계가 구체화되면서 조율에 어려움을 겪고 있을 때 사용한다. 이를 **책임별 계층**으로 나눈다. 이는 UML로 표현이 되지 않는다.

RESPONSIBILITY LAYER에 가장 잘 부합하는 계층화 패턴은 RELAXED LAYERED SYSTEM[1]이라고 하는 계층화의 일종인데, 이것은 한 계층의 구성요소가 바로 아래에 있는 것만이 아니라 모든 하위 계층에 접근하는 것을 허용한다.

모델에 존재하는 개념적 의존성과 도메인의 여러 부분에 대한 다양한 **변화율**과 **변화의 근원**을 검토하라. 도메인에서 자연적인 층을 식별하면 그것을 **광범위한 추상적 책임**으로 간주하라. 이러한 책임은 시스템의 높은 수준에서의 목적과 설계에 관한 이야기를 들려줄 것이다. AGGREGATE, MODULE과 같은 각 도메인 객체의 책임이 한 계층의 책임 안에서 말끔히 맞아떨어지도록 모델을 리팩터링하라.

예제

심화 예제: 해운 시스템의 계층화. p488

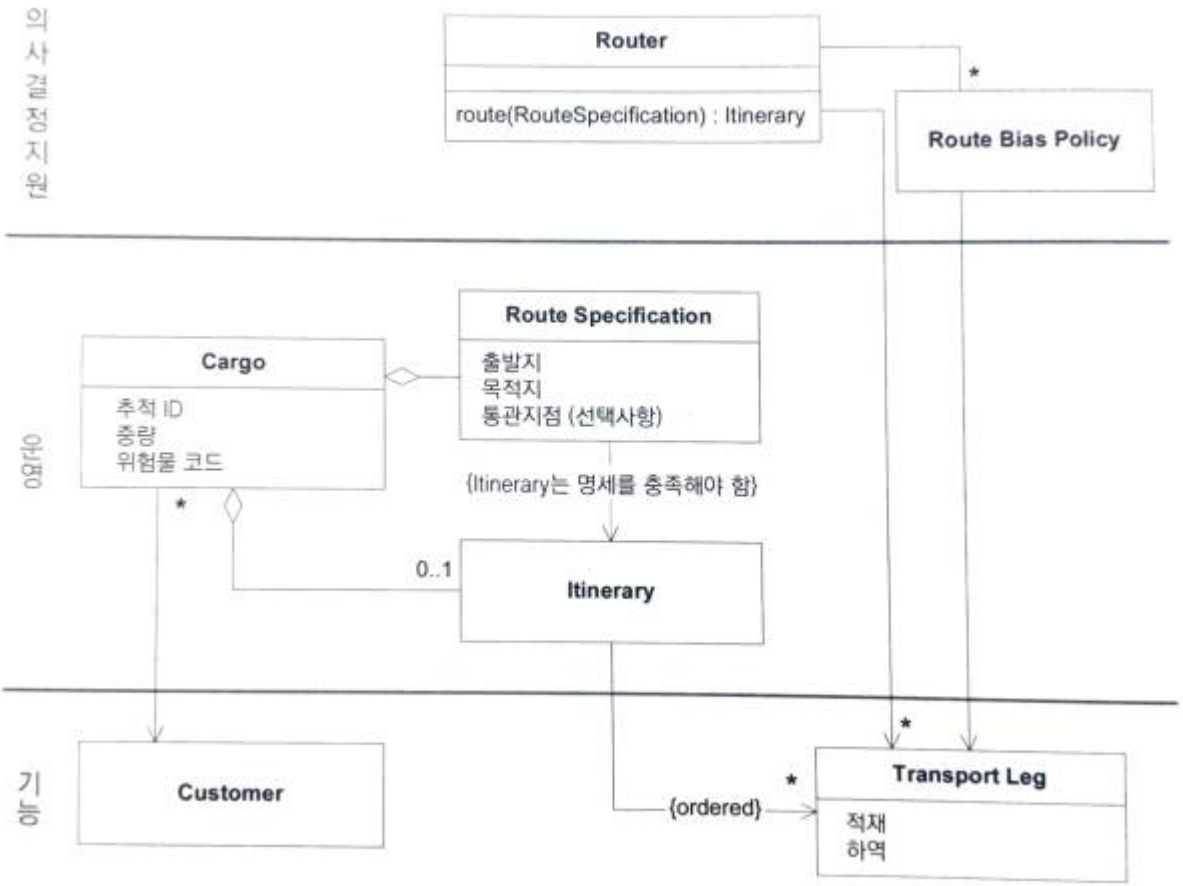


그림 16-8 | 모델을 재구성하고 리팩터링한 결과

적절한 계층의 선택

알맞은 RESPONSIBILITY LAYER나 대규모 구조를 찾을 때는 **문제 도메인을 이해하고 실험**하는 것이 중요하다. (실험 과정에서) **계층**이 바뀌고 병합되고 나뉘고 재정의될 때 찾아서 지켜야 할 몇 가지 유용한 특징은 다음과 같다.

- **스토리텔링,** 계층은 도메인의 기본적인 실제 상황과 우선순위를 전해줘야 한다. 대규모 구조를 선택하는 것은 기술적 결정이라기보다는 업무와 관련된 모델링 결정이다. 계층은 **업무의 우선순위**를 드러내야 한다.
- **개념적 의존성.** "상위" 계층에 있는 개념은 "하위" 계층을 배경으로 하는 의미를 지녀야 하고, 동시에 하위 계층의 개념은 독자적인 의미를 지녀야 한다.
- **CONCEPTUAL CONTOUR.** 다양한 계층에 놓인 객체가 변화율이나 변화의 근원이 서로 다르다면 계층은 그러한 객체 간에 구획을 짓는 일에 일조한다.
  (If the objects of different layers should have different rates of change or different sources of change, the layer accommodates the shearing between them.)

새로운 모델을 위한 개층을 정의할 때마다 꼭 처음부터 다시 시작해야 하는 건 아니다. 어떤 계층은 관련 도메인군 전체에 나타나기도 한다.

계층화 시스템은 **단순**하게 유지하는 것이 가장 좋다. 계층이 4~5개를 넘어가면 다루기가 힘들어진다. 계층이 너무 많으면 이야기할 때만큼 효과적이지 않고 대규모 구조가 풀어야 할 복잡성의 문제가 새로운 형태로 다시 나타날 것이다. 대규모 구조는 반드시 철저하게 정제되어야 한다.

KNOWLEDGE LEVEL

다른 객체 집단이 **어떻게 행동해야 하는지**를 기술하는 객체의 그룹[2)]

사용자가 개발 과정에서 의도하지 않은 시나리오대로 소프트웨어를 사용할 때, 포용할 수 있는 설계 기법

더 넓은 범위의 규칙으로 제약되기 전까지 모델의 특정 부분을 사용자의 손에 맡겨야 할 때 생기는 문제를 해결한다.

KNOWLEDGE LEVEL은 구성 가능한[3)] 행위를 갖춘 소프트웨어의 요구사항을 다룬다.

(다른 분석 패턴이 그러하듯) KNOWLEDGE LEVEL은 도메인을 모델링하기보다 모델을 구조화한다.

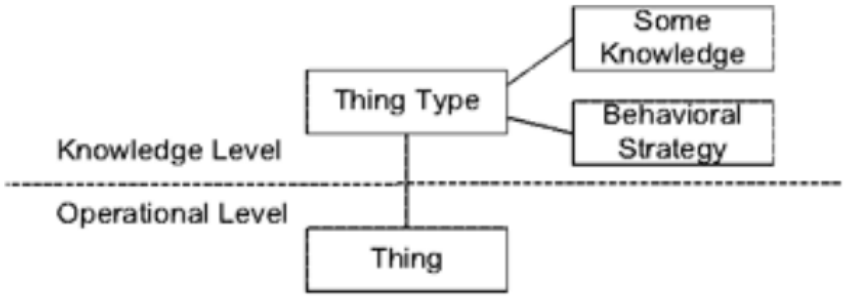
ENTITY 간의 역할과 관계가 각 상황마다 다양하게 작용하는 애플리케이션에서는 복잡성이 폭발적으로 증가할 수 있다.
완전히 일반화된 모델이나 고도로 맞춤화가 가능한 모델도 사용자의 욕구를 충족하지 못한다.
결국 객체는 **다양한 경우**를 다루고자 **다른 타입을 참조**하거나, 아니면 각종 상황에서 **여러 가지 방법으로 쓰일 속성**을 갖게 된다.
데이터와 행위가 **동일한 클래스**가 **단지 다양한 조립 규칙을 수용할 목적**으로 크게 늘어날지도 모른다.

우리의 모델에 자리잡고 있는 것은 해당 모델에 **관한** 또 다른 모델이다. KNOWLEDGE MODEL은 그와 같은 자기 규정적(self-defining)인 측면을 분리해서 해당 측면의 제약조건을 명시적으로 만든다. (≒ REFLECTION)

적용 방법

KNOWLEDGE LEVEL은 REFLECTION 패턴의 도메인 계층을 (개념적으로) 응용한 것. (REFLECTION 패턴은 POSA 참고)

어떤 사물에 대해서 유형(Type)에 따라 행위가 부과되는가?



[A KNOWLEDGE LEVEL is] a group of objects that describes how another group of objects should behave. [Martin Fowler, “Accountability,” www.martinfowler.com]

Folower 용어	POSA 용어	설명
Knowledge Level	Meta Level	소프트웨어의 구조와 행위에 대한 지식
Operational Level	Base Level	소프트웨어를 애플리케이션의 운영적 책임을 수행

기본적인 모델의 구조와 행위를 서술하고 제약하는 데 쓸 수 있는 별도의 객체 집합을 만들어라. 이러한 관심사를 두 가지 “수준”으로 분리해서 하나는 매우 구체적으로 만들고 다른 하나는 사용자나 관리자의 맞춤화가 가능한 규칙과 지식을 반영하게 만들어라

적용시 단점

모호함

KNOWLEDGE LEVEL이 복잡해지면 사용자와 마찬가지로 개발자들도 시스템의 행위를 이해하기가 힘들어진다.
시스템을 구성하는 사용자(또는 관리자)에게는 결국 프로그래머 수준의(그것도 메타 수준의 프로그래머 수준으로) 기량이 필요해질 것이다. 사용자가 실수라도 한다면 애플리케이션은 올바르게 동작하지 않을 것이다.

데이터 이전

KNOWLEDGE LEVEL 내의 구조가 바뀌면 기존 운영 수준에 위치한 객체도 그에 맞게 다뤄야 한다.
기존 객체와 신규 객체가 공존하는 것도 가능할지 모르지만 어떻게든 신중하게 분석할 필요가 있다.

PLUGGABLE COMPONENT FRAMEWORK

Liskov Substitution Principle과 연관.

인터페이스와 상호작용에 대해 ABSTRACT CORE를 정제하고 그러한 인터페이스의 다양한 구현이 자유롭게 대체될 수 있는 프레임워크를 만들어라. 이와 마찬가지로 컴포넌트가 ABSTRACT CORE의 인터페이스를 통해 정확히 작동하는 한 어떤 애플리케이션에서도 그러한 컴포넌트를 사용할 수 있게 하라.

“Domain-Specific Application Frameworks (Wiley, 2000)“에서 이 패턴을 의욕적으로 시도한 사례를 자세히 살펴보고 있다. 가장 성공적인 사례는 다수의 전문적인 애플리케이션이 **완전히 개발되고 난 후**에 나타났다.

불리한 점

- **매우 적용하기 힘들다.**
  이 패턴을 적용하려면 인터페이스를 설계할 때 정밀함과 ABSTRACT CORE에 필요한 행위를 포착할 만큼 충분히 심층적인 모델이 필요하다.
- **애플리케이션에 선택의 여지가 제한된다.**
  애플리케이션에서 CORE DOMAIN에 대해 매우 다른 접근법을 필요로 한다면 해당 구조가 방해될 것이다. 모든 다양한 컴포넌트의 프로토콜을 변경하지 않고는 ABSTRACT CORE를 변경하지 못하고, 더 심층적인 통찰력으로 향하는 리팩터링이 거의 진행을 멈추게 된다.

구조는 얼마나 제약성을 지녀야 하는가?

제약은 양날의 검. 단일 규칙으로 쉽게 해석할 수 있게 만드는 반면, 개발자가 필요로 하는 유연함을 앗아갈 수 있다. 그러므로 (제약성을 가진) 프레임워크를 만들고 대규모 구조의 구현을 엄격히 통제하고자 하는 유혹을 이겨내야 한다.

대규모 구조가 공헌하는 가장 중요한 바는 **개념적 응집성과 도메인에 대한 통찰력**을 주는 것이다.,

각 구조적 규칙은 개발을 용이하게 해야 한다.

잘 맞아떨어지는 구조를 향한 리팩터링

EVOLVING ORDER에 전념하는 팀은 프로젝트 생명주기 내내 대규모 구조를 과감히 재고(fearlessly rethink)해야 한다.
그 팀은 도메인이나 요구사항을 매우 잘 이해하는 사람이 아무도 없었을 때인 초기에 구상한 구조에 안주해서는 안된다.

아쉽게도 그러한 발전이 의미하는 바는 최종 구조가 처음에는 사용할 수 없는 것을 의미하며, 진행하면서 이를 적용하기 위해 리팩터링을 해야하는 것을 의미한다.

비용을 통제하고 이익을 극대화하는 방법들은 다음과 같다.

최소주의

구조를 간단하고 가볍게 유지하는 것이다.. **포괄적인 구조로 만들려고 하지 마라.**
가장 중요한 관심사만 다루고 나머지는 사례별로 처리하게 한다.

의사소통과 자기 훈련

UBIQUITOUS LANGUAGE의 사용과 팀의 해당 언어 연습.

재구조화가 유연한 설계를 낳는다

Restructuring Yields Supple Design

> 새 가죽 재킷은 뻣뻣하고 입기가 불편하지만
> 처음 입고 난 후로는 팔꿈치 부분이 몇 차례 구부러져 접힌다.

안정적인 측면은 단순화되는 반면 계속 늘어나는 지식은 모델에 녹아들고 **변화의 중심축이 식별되어 유연해진다.**

디스틸레이션은 부하를 줄인다

지속적인 디스틸레이션은 다양한 방식으로 **구조 변경**의 어려움을 덜어준다.

---

[1] Pattern-Oriented Software System. Buschmann et al. 1996
[2] Martin Fowler, “Accountability[책임추적성]” www.martinfowler.com
[3] 여러 ENTITY간의 역할과 관계를 초기화 시점이나 실행 시점에서도 변경할 수 있어야 한다는 의미