14 모델의 무결성 유지

Domain-Driven Design, Modeling, Design

- 모델의 가장 근본적인 요구사항
  - 모델의 내적으로 일관성 유지
  - 모델의 용어는 언제나 의미가 동일
  - 모델에는 어떠한 모순되는 규칙도 없어야 한다

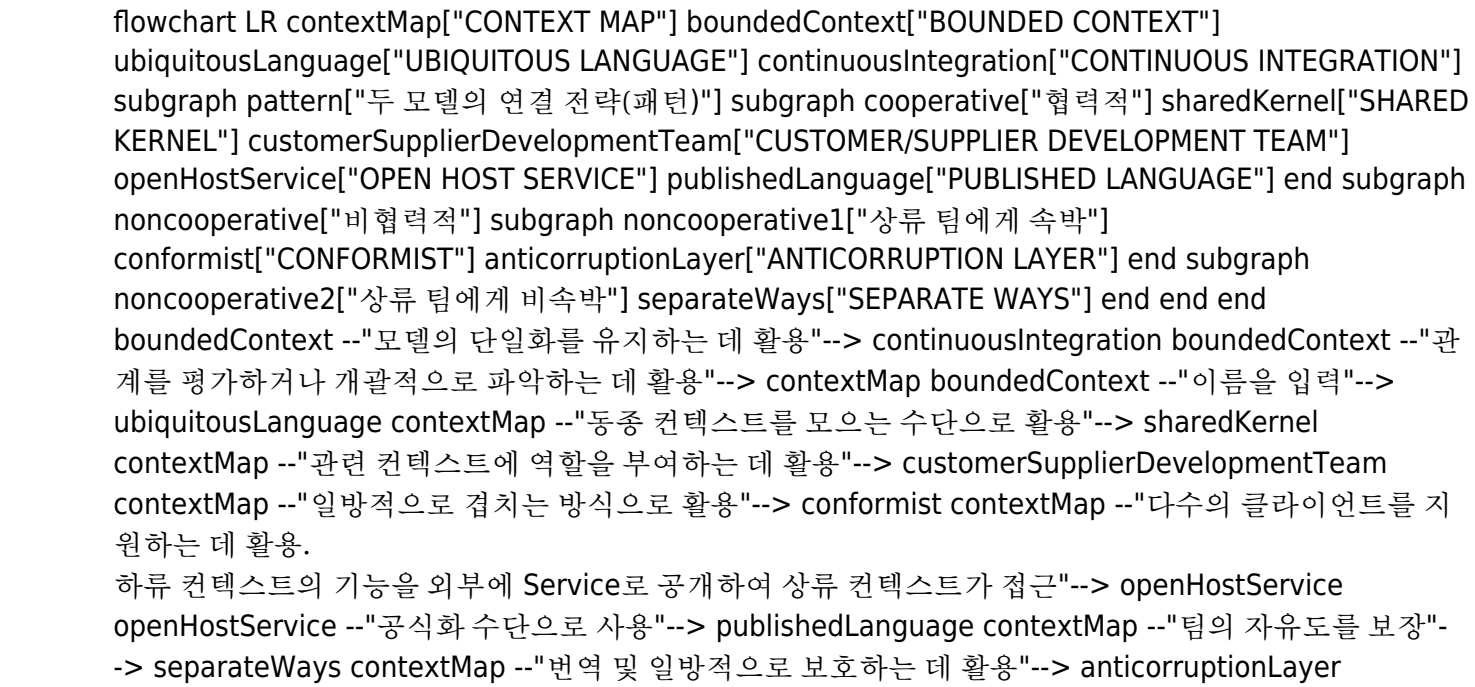
각 용어가 모호하지 않고 모순되는 규칙이 없는 모델의 내적 일관성을 **단일화(unification)**라 한다.

하지만 대규모 시스템 개발이라는 세계는 이상적인 세계가 아니다. 전사적 시스템 차원에서 단일화를 유지하기란 생각보다 복잡하다. 여러 모델이 서로 다른 부분에서 개발되게끔 해야 할 필요도 있지만 시스템의 어느 부분을 나누고 나눈 부분들 간에는 어떤 관계를 맺을지 결정하는 데는 신중을 기해야 한다. **대규모 시스템의 도메인 모델을 완전하게 단일화한다는 것은 타당하지 않거나 비용 대비 효과적이지 않을 것이다.**

- 복수 모델을 단일화 모델로 변경하려고 할 때, 고려할 위험 요소
  1. 한 번에 지나치게 많은 레거시를 교체하려 할지도 모른다.
  2. 대규모 프로젝트에서는 능력에 비해 조율에 드는 비용이 너무 커서 난관에 처할지도 모른다.
  3. 특화된 요구사항이 있는 애플리케이션에서는 요건을 완전하게 충족하지 못해 애플리케이션의 행위를 다른 곳에 둘 수 밖에 없는 모델을 사용해야 할지도 모른다.
  4. 이와 반대로 단일 모델로 모두를 만족시키려 해서 모델을 사용하기 어렵게 만드는 복잡한 대안으로 이어질지도 모른다.

- 모델 분화의 원인
  - 기술적 관심사
  - **정치적 분열**
  - **다양한 경영상의 우선순위**

우리에겐 다른 여러 모델 간의 **경계**와 **관계**를 표시해줄 수단이 필요하다. 우리는 의식적으로 전략을 결정해야 하며, 그리고 나서 이러한 전략을 지속적으로 따라야 한다.



BOUNDED CONTEXT

모델을 올바른 상태로 유지하는 데 실패했는가는 결국 실행 중인 코드가 정상적으로 동작하지 않을 때 드러나지만 문제의 원인은 **팀이 조직되는 방식**과 사람들이 **상호작용**하는 방식에 있다. 그러므로 모델의 컨텍스트를 명확하게 만들려면 **프로젝트**와 **산출물(코드, 데이터베이스 스키마 등)**을 모두 살펴봐야 한다.

모델은 **컨텍스트**에 적용된다. 컨텍스트는 코드의 특정 부분일 수도, 개별 팀이 수행하는 업무일 수도 있다. 브레인스토밍 회의를 거쳐 만들어진 모델의 경우, 회의에서 오간 대화로 컨텍스트를 국한시킬 수 있다. **모델 컨텍스트란 모델에서 사용된 용어를 특정한 의미로 의사소통하기 위한 조건의 집합이다.**

모델이 적용되는 컨텍스트를 **명시적으로 정의**하라. 컨텍스트의 경계를 팀 조직, 애플리케이션의 특정 부분에서의 사용법, 코드 기반이나 데이터베이스 스키마와 같은 **물리적인 형태의 관점에서 명시적으로 설정**하라. 이 경계 **내**에서는 모델을 엄격하게 **일관된 상태**로 유지하고 경계 바깥의 이슈 때문에 초점이 흐려지거나 혼란스러워져서는 안 된다.

BOUNDED CONTEXT는 팀 구성원이 어떤 부분에서 일관성을 지녀야 하고 다른 CONTEXT와 어떤 식으로 관련돼 있는가를 서로 명확하게 이해할 수 있게 특정 모델의 적용 범위를 제한한다.

모델이 적용되는 BOUNDED CONTEXT를 파악하려면 **프로젝트에서 일어나고 있는 일을 살펴야 한다.** 염두에 둘 것은 프로젝트를 **있는 그대로**봐야지, 이상적인 프로젝트를 생각해서는 안 된다는 것이다.

두 팀이 같은 CONTEXT에 있지 않다면 어느 정도의 변화가 생기기 전까지는 **코드를 공유하려 해서는 안된다.**

- BOUNDED CONTEXT를 정의해서 얻을 수 있는 장점?
  - CONTEXT 안 : 업무의 **명확함**
  - CONTEXT 밖 : 동일한 모델을 사용해야 하는지에 대한 **자유로움**. 생각할 필요가 없다.

BOUNDED CONTEXT 안의 균열 인식

모델의 단일화가 깨지는 징조.

- 코드로 작성된 인터페이스가 서로 맞지 않는 경우
  - 자동화된 테스트를 이용한 CONTINOUS INTEGRATION 프로세스가 이러한 문제점을 발견하는 데 도움이 될 수 있다.
  - 그러나 **초기 징후**는 대개 **언어를 혼동한 상태**로 구사하는 데서 나타난다.
    - 의사소통 오류 → 모델의 잘못된 이해 → 코드 반영 → 인터페이스 불일치
- 뚜렷이 구분되는 모델 요소를 결합할 경우
  두 가지 종류의 문제가 일어나게 되는데,
  중복된 개념과 **하위 동족 언어(false congnates)**[1)]
  - 중복된 개념 : 같은 개념을 나타내는 두개의 모델 요소(그리고 그것에 따르는 구현)가 존재하는 것
    - 변경이 일어나면 두 군데 다 갱신하고 변환해야 한다
      - 심지어 데이터까지 다른 두 가지 버전의 객체가 존재할 수 있다.
  - 하위 동족 언어 : 그리 일반적으로 나타나지는 않지만 좀 더 교묘한 방식으로 해를 끼친다.
    서로 같은 것을 이야기하고 있다고 생각하지만 실제로는 그렇지 않는 경우를 말한다.

CONTINUOUS INTEGRATION

BOUNDED CONTEXT를 건전한 상태로 유지하는 방법.

다수의 사람이 동일한 BONDED CONTEXT 내에서 작업할 경우 모델이 **단편화**될 가능성이 높다. 팀의 규모가 커지면 문제도 증폭되지만 서너 명 정도에 달하는 소수의 인원으로 구성된 팀도 심각한 문제에 마주칠 수 있다. 그렇다고 시스템을 더 작은 CONTEXT로 분할한다면 결국 가치 있는 수준의 **통합**과 **응집성**을 잃게 되는 결과를 초래한다.

이따금 다른 사람이 모델링한 객체나 상호작용의 의도를 완전히 이해하지 못한 채로 객체를 수정해 원래의 목적으로 사용할 수 없게 만들 때가 있다. 간혹 다루고 있는 개념이 이미 모델의 다른 부분에 포함돼 있다는 사실을 알지 못해서 동일한 개념과 행위를 (부정확하게) 중복시킬 때가 있다. 때로는 다른 표현 방식을 알고 있지만 기존에 정상적으로 수행되고 있는 기능에 오류를 추가할지도 모른다는 **두려움** 탓에 함부로 손을 댈 수 없어 개념과 기능을 중복시키기도 한다.

규모와 상관없이 통합된 시스템을 개발하는 데 필요한 수준의 의사소통을 유지하기란 매우 어려운 일이다.

바로 이런 상황에서 익스트림 프로그래밍(XP, Extreme Programming)이 진가를 발휘한다. XP에서 제시하는 각종 실천사항의 목표는 많은 사람들에 의해 끊임없이 수정되는 **응집도** 높은 설계를 유지하는 문제를 해결하는 데 있다.
가장 순수한 형태의 XP는 단일 BOUNDED CONTEXT에 포함된 모델의 무결성을 유지하는 데 적합하다.
그러나 XP의 활용 여부와 관계없이 일정 수준의 CONTINUOUS INTEGRATION 프로세스를 보유하는 것은 대단히 중요하다.

CONTINUOUS INTEGRATION은 내부적으로 균열이 발생할 때 이를 빠르게 포착하고 정정할 수 있을 정도로 컨텍스트 내의 모든 작업을 빈번하게 **병합**해서 **일관성**을 유지하는 것을 의미한다. DOMAIN-DRIVEN DESIGN의 다른 기법과 마찬가지로 (1) **모델 개념의 통합**과 (2) **구현 수준에서의 통합**이라는 두 가지 수준에서 작용한다.

- 통합을 위한 가장 효과적인 프로세스의 공통적인 특징
  - 단계적이고 재생 가능한 병합/빌드 기법
  - 자동화된 Test suit
  - 수정사항이 통합되지 않은 상태로 존재할 수 있는 시간을 적당히 짧게 유지하는 규칙
- 비록 공식적으로 포함되는 경우는 그리 많지 않지만
  효과적인 프로세스의 또 다른 측면으로 **개념적인 통합**이 있다.
  - 모델과 애플리케이션에 관해 논의할 때 **UBIQUITOUS LANGUAGE**를 지속적으로 사용

MODEL-DRIVEN DESIGN에서 개념의 통합은 구현을 통합하는 방법을 좀더 용이하게 하는 반면, 구현 통합은 모델의 유효성과 일관성을 입증하고 발생한 균열을 드러낸다.

그러므로

단편화가 발생했다는 사실을 빠르게 알려줄 수 있는 **자동화된 테스트**와 함께 모든 코드와 그 밖의 구현 산출물을 **빈번하게 병합하는 프로세스**를 수립하라. 개념이 각자의 머릿속에서 발전해감에 따라 모델에 관한 시각의 차이를 해소하기 위해 끊임없이 **UBIQUITOUS LANGUAGE**를 사용하라.

(eg. SonarQube + Jenkins ?)

마지막으로 필요 이상으로 일이 커지지 않게 한다. CONTINUOUS INTEGRATION은 **오직 하나의 BOUNDED CONTEXT 내에서만 필수적**이다. 번역을 비롯해 인접한 CONTEXT와 관련된 설계 쟁점을 똑같은 수준으로

다룰 필요는 없다.

CONTEXT MAP

CONTEXT간의 관계를 번역을 사용하여 전체를 조망하는 방법.

graph LR subgraph context1["Context"] model1 model2 end subgraph context2["Context"] model3 model4 model5 end model1 --번역--- model3 model2 --번역--- model4 model2 --번역--- model5

CONTEXT MAP은 **프로젝트 관리**와 **소프트웨어 설계** 영역 사이에 걸쳐 있는 개념이다. 대개 CONTEXT의 경계는 팀 조직의 윤곽을 따라 정해진다. (조직 구성, 물리적인 사무 공간 등)

> 프로젝트상의 유효한 모델을 식별하고 각 BOUNDED CONTEXT를 정의하라. 여기에는 비객체지향적인 하위 시스템에 대한 **암시적인 모델**도 포함된다. 각 BOUNDED CONTEXT에 **이름을 부여**하고 이 이름을 **UBIQUITOUS LANGUAGE**의 일부로 포함시켜라.
>
> 
>
> 의사소통을 위해 컨텍스트 간의 **번역에 대한 윤곽**을 명확하게 표현하고 컨텍스트 간에 **공유해야 하는 정보를 강조**함으로써 모델과 모델이 만나는 경계 지점을 서술하라.
>
> 각 컨텍스트의 **현재 영역을 나타내는 지도**를 작성하라. 컨텍스트의 배치를 바꾸는 일은 나중에 하라.

MAP을 특정한 형식으로 문서화할 필요는 없다. **MAP을 어떤 형식으로 작성하건 프로젝트에 속한 모든 사람들은 MAP을 이해하고 공유해야 한다.**
MAP은 BOUNDED CONTEXT의 **명확한 이름을 제공**해야 하며, 경계 지점과 경계 지점의 **특성을 명확하게 표현**해야 한다.

- 실전 지침

CONTEXT MAP이 항상 **현재 상태 그대로의 상황**을 표현한다는 사실을 염두에 둔다면 컨텍스트에서 발견하게 되는 관계가 처음에는 패턴과 딱 맞아 떨어지지 않을 수도 있다. 유사성이 눈에 띈다면 패턴 이름을 사용하고 싶어질 수도 있지만 이를 **강요해서는 안된다.**
단지 발견한 관계를 서술하기만 한다.

만약 균열(서로 완전히 뒤얽혀 있지만 비일관성을 내포하는 모델)을 발견했다면 **지도에 모른다고 적어 놓고 거기서 서술을 중단한다. 그러고 나서 정확한 전체적인 뷰를 가지고 혼란스러운 지점을 설명한다.**

하지만 이 같은 필수적인 수선 작업이 **전체적인 구조를 재구성하는 작업으로 이어지게 해서는 안 된다.** 수행해야 하는 모든 작업을 일부 BOUNDED CONTEXT로만 제한하고 연결된 모델 간의 관계가 명확한 모호하지 않은 CONTEXT MAP을 보유하기 전까지는 **명백하게 드러나는 모순만 변경한다.**

실제로 변경이 완료되기 전까지는 MAP을 수정해서는 안 된다!

CONTEXT 경계에서의 테스트

다른 BOUNDED CONTEXT와 **접촉하는 지점**은 테스트할 때 특히 중요하다. 대개 테스트는 컨텍스트의 경계에 존재하는 **번역의 미묘한 차이**와 **낮은 수준의 의사소통**을 보완하는 데 기여하는 **조기경보체계**

CONTEXT MAP의 조직화와 문서화

CONTEXT MAP을 조직화하고 문서화할 때는 아래의 두 가지 사항이 중요하다.

1. BOUNDED CONTEXT의 **이름**은 **해당 BOUNDED CONTEXT에 관해 이야기할 수 있는 것**이어야 한다. 그러한 이름은 팀의 **UBIQUITOUS LANGUAGE**에 들어가야 한다.
2. 모든 이들이 경계가 **어디에 위치하는지** 알아야 하며, 어떠한 코드나 환경의 CONTEXT도 **인식**할 수 있어야 한다.

중요한 것은 팀의 모든 구성원이 서로 동일한 방식으로 개념적 경계를 이해하도록 개념적 경계에 관해 활발히 **의사소통**하는 것이다.

BOUNDED CONTEXT 간의 관계

이어지는 패턴들은 전사적인 차원을 포괄하도록 구성할 수 있는 **두 모델의 연결을 위한 전략**을 폭넓게 다룬다.

이 같은 패턴은 두 가지 목적을 충족한다.

- 개발 업무를 성공적으로 조직화하기 위한 대상을 제공
- 기존 조직을 기술하기 위한 어휘를 공급

이어지는 패턴들은 몇 가지 가장 흔히 나타나고 중요한 경우를 다루며, 이를 바탕으로 다른 경우에는 어떻게 대처할지 참고할 수 있다.

SHARED KERNEL

기능 통합에 한계가 있는 경우 CONTINUOUS INTEGRATION에 따라는 비용이 너무 높다고 판단할 수 있다.

- 팀 내에 CONTINUOUS INTEGRATION을 유지할 수 있는 기술이나 정치적인 조직이 갖춰지지 않은 경우
- 단일 팀의 규모가 너무 크고 비대한 경우

팀 간의 협력이 조율되지 않음 → 결과물을 조합하기 어려움 → 처음부터 CONTINUOUS INTEGRATION을 적용했을 때 보다 더 많은 시간을 번역 계층을 개발하고 구조를 개선하는데 허비 → 공통 UBIQUITOUS LANGUAGE 구축 작업이 중복되고 UBIQUITOUS LANGUAGE로 얻을 수 있는 이점을 잃음

두 팀 간에 공유하기로 한 도메인 모델의 **부분집합을 명시**하라. 물론 여기에는 모델의 부분집합뿐 아니라 연관된 코드나 데이터베이스 설계의 부분집합까지도 포함된다. 명시적으로

공유하는 부분들은 특별한 상태를 가지며, **다른 팀과의 협의 없이는 변경할 수 없다.**

목표는 **중복을 줄이고(하지만 오직 하나의 BOUNDED CONTEXT가 존재하는 경우처럼 중복을 완전히 제거하는 것은 아니다) 두 하위 시스템 간의 통합을 비교적 용이하게 만드는 것**이다.

CUSTOMER/SUPPLIER DEVELOPMENT TEAM

업무의 의존성이 “Upstream→Downstream” 단방향으로 흐르는 Context를 가질 때 사용하는 패턴.

- Downstream team : 하류 팀 - 소비자, 고객의 가까운 업무 수행
- Upstream team : 상류 팀 - 생산자, 공급자의 가까운 업무 수행

이런 형태의 조직이면 속박과 억제가 이뤄질 수 있다…

두 팀 간에 명확한 고객/공급자 관계를 확립해라. 계획 회의에서 **하류 팀이** 상류 팀에 대한 **고객 역할**을 맞게 하라. 하류 요구사항에 대한 작업을 협상하고 이에 대한 예산을 책정해서 모든이들이 일정과 약속을 이해할 수 있게 하라.

결과로 예상되는 인터페이스를 검증하게 될 *자동화된 인수 테스트(acceptance test)**를 함께 개발하라. 이 테스트를 상류 팀의 Test suit에 추가해서 지속적인 통합의 일부로 실행되게 하라. 이러한 테스트를 토대로 상류 팀은 하류 시스템에서 발생할지도 모르는 Side-effect 를 두려워하지 않고 자유로이 코드를 변경할 수 있을 것이다.**

이 패턴에는 두 가지 중요한 요소가 있다.

1. 관계는 **고객과 공급자** 간의 관계여야 하며, 이는 **고객의 요구사항이 가장 중요**하다는 것을 의미한다. 하류 팀이 유일한 고객은 아니므로 협의를 거쳐 서로 다른 고객의 요구사항 간에 균형을 맞춰야 한다(그래도 우선순위는 유지해야 한다).
   이 상황은 하류 팀이 상류 팀에 필요한 사항을 부탁해야만 하는 초라한 동반자(poor-cousin) 관계와는 대조적이다.
2. 상류 팀이 하류 시스템을 망가뜨릴지도 모른다는 두려움 없이 코드를 수정하고
   하류 팀이 지속적으로 상류 팀을 감지하지 않고도
   자신의 작업에 집중할 수 있게 **자동화된 Test suite**가 마련돼 있어야 한다.

계주에서 다음 주자는 바통을 전달받기 위해 줄곧 뒤를 돌아보면서 확인할 수 없다. 주자는 전달자가 정확하게 바통을 전달하리라는 사실을 **신뢰**할 수 있어야 하며, 그렇지 않다면 팀은 무력하게 와해되고 말 것이다.

CONFORMIST

준수자. **협력에 관심이 없는 팀과의 통합** 문제를 다룬다.

관리 계층상 멀리 떨어져있거나 규모가 큰 회사 큰 회사 내부에서 공동 관리자가 두 팀 관계에 무심하여 CUSTOMER/SUPPLIER DEVELOPMENT TEAM을 적용할 수 없을 때 선택할 수 있는 길이 세 가지가 있다.

- 상류 팀에서 제공하는 기능의 사용을 전적으로 포기 → **SEPARATE WAYS**
- 상류 소프트웨어를 사용하는 데서 얻는 가치가 너무 커서(또는 팀에 변경할 권한이 없는 정치적인 결정으로) 의존관계를 유지해야 할 때가 있다.
  이 경우 두 가지 길이 있다.
  선택은 상류팀이 제작한 소프트웨어 소프트웨어 설계의 품질과 스타일에 달려 있다. (캡슐화, 추상화, 패러다임등)
  - (수용이 어려운 경우) 자체적인 모델을 만들고 복잡해질 가능성이 있는 **번역 계층**을 개발하고 유지보수할 책임을 전부 맡아야 한다. → **ANTICORRUPTION LAYER**
  - (수용이 가능한 경우) 독립적인 모델을 포기하는 것이 최선. 상위 팀 모델을 따른다. → **CONFORMIST**

flowchart TB subgraph coperative["협력적"] sharedKernel["SHARED KERNEL"] customerSupplierDevelopmentTeam["CUSTOMER SUPPLIER DEVELOPMENT TEAM"] end subgraph noncooperative["비협력적"] subgraph noncooperative1["상류 팀에게 속박"] conformist["CONFORMIST"] anticorruptionLayer["ANTICORRUPTION LAYER"] end subgraph noncooperative2["상류 팀에게 비속박"] separateWays["SEPARATE WAYS"] end end coperative <--> noncooperative

맹목적으로 **상류 팀의 모델을 준수**해서 BOUNDED CONTEXT 간의 **번역에 따른 복잡도를 제거**하라. CONFORMIST를 따를 경우 하류 팀 설계자들의 설계 형식이 상류 팀에 속박되고 애플리케이션을 위한 이상적인 모델을 만들지는 못해도 통합 자체는 매우 단순해질 수 있다. 또한 SUPPLIER 팀과 UBIQUITOUS LANGUAGE를 공유할 수 있다. SUPPLIER가 주도적인 위치에 있으므로 SUPPLIER를 위해 **의사소통**을 용이하게 하는 것이 좋다. SUPPLIER가 여러분과 정보를 공유하게 하는 데 필요한 것은 **이타주의** 정도면 충분할 것이다.

ANTICORRUPTION LAYER

자체적인 모델을 보유한 레거시나 타 시스템을 통합할 때 **번역**을 사용하는 패턴.

클라이언트 고유의 도메인 모델 측면에서 기능을 제공할 수 있는 **격리 계층**을 만들어라. 격리계층은 기존에 이미 존재하는 인터페이스를 거쳐 다른 시스템과 통신하므로 다른 시스템을 거의 수정하지 않아도 된다. 해당 계층에서는 내부적으로 필요에 따라 두 모델을 상대로 양방향으로 번역을 수행한다.

개념적인 객체와 행위를 하나의 모델과 프로토콜에서 다른 모델과 프로토콜로 **변환**하기 위한 메커니즘

ANTICORRUPTION LAYER의 인터페이스 설계

여러 개의 SERVICE(또는 간혹 ENTITY)

ANTICORRUPTION LAYER의 구현

- 통신 및 전송 메커니즘
- FACADE Pattern
- ADAPTER Pattern
    - 번역기

flowchart LR subgraph newSystem["차세대 시스템"] newInterface["개선된 인터페이스"] end subgraph anticorruptionLayer["ANTICORRUPTION LAYER"] service subgraph adapters["어댑터들"] adapter1["어댑터 1"] translater["번역기"] adapter2["어댑터 2"] end facade["퍼사드"] end subgraph legacySystem["레거시 시스템"] oldInterface["레거시 인터페이스"] end newInterface --> service["서비스"] service --> adapters adapter1 --> translater adapter2 --> translater adapter1 --> facade adapter2 --> facade facade --> oldInterface

SEPARATE WAYS

통합에는 언제나 비용이 많이 든다. 때로는 통합의 혜택이 적은 경우도 있다.

BOUNDED CONTEXT가 다른 것과 아무런 관계도 맺지 않도록 선언해서 개발자들이 이 작은 범위 내에서 단순하고 특화된 해결책을 찾을 수 있게 하라.

OPEN HOST SERVICE

각 하위 시스템에 대한 번역기가 많은 경우 사용하는 패턴.

여러 종류의 Downstream Bounded Context를 고려하여 설계되는 Upstream Bounded Context.

하류 컨텍스트의 기능을 외부에 Service로 공개하여 상류 컨텍스트가 접근.

예를 들어, 상류 Context “네이버 검색”에서 하류 Context인 블로그, 카페, 웹페이지 컨텍스트의 기능을 이용할 것이다. 이 때, 하류 Context N개 만큼 상류 Context에 맞춰 번역 레이어가 필요할 것이고, 유사한 코드가 반복될 것이다. 이를 외부 서브시스템을 서비스의 제공자로 바라보는 관점으로, 외부 서브시스템을 서비스로 감싼다.

하위 시스템 접근과 관련된 프로토콜을 일련의 SERVICE로 정의하라 프로토콜을 공개해서 개발 중인 시스템과 통합하고자 하는 모든 이들이 해당 프로토콜을 사용할 수 있게 하라. 새로운 통합 요구사항을 처리하게끔 프로토콜을 개선하고 확장하되 특정한 한 팀에서 요청해 오는 **독특한 요구사항은 제외하라**. 그와 특수한 경우에는 일회성 번역기로 프로토콜을 보

강해 공유 프로토콜을 단순하게 일관되게 유지하라.

PUBLISHED LANGUAGE

OPEN HOST SERVICE에서 사용하는, 두 BOUNDED CONTEXT의 모델 간에 이뤄지는 번역의 공통 언어.

> 한 모델을 데이터 교환 언어로 사용한다면 해당 모델은 본질적으로 굳어질 테고 새로운 개발 요구사항에 대응하지 못할 것이다.

필요한 도메인 정보를 표현할 수 있는 **적절히 문서화된 공유 언어**를 공통의 의사소통 **매개체**로 사용해서 필요에 따라 해당 언어로, 또는 해당 언어로부터 **번역**을 수행하라.

코끼리 통일하기

다수의 서로 대립되는 도메인 모델을 인식하는 것이야말로 **현실을 직시**하는 것이다. 각 모델이 적용되는 컨텍스트를 명확하게 정의하는 식으로 각 모델의 무결성을 유지하면서도 두 모델 사이에서 여러분이 만들고자 특정한 인터페이스가 의미하는 바를 명확하게 확인할 수 있다. 장님이 코끼리의 전체적인 모습을 확인할 길은 없지만 **자신의 인식이 불완전한 점만 인정**해도 문제를 해결할 기미가 보일 것이다.

모델의 컨텍스트 전략 선택

실무 적용 지침

팀 의사결정 또는 그 이상

먼저 BOUNDED CONTEXT의 정의, 각 BOUNDED CONTEXT 간의 관계를 결정하는데 **전원이 이러한 사안을 알고 있어야 한다**. 실제로 이러한 의사결정은 종종 해당 팀의 범위를 넘어서는 **합의**를 수반할 때가 있다.

실제로 팀 간의 *정치적 관계*로 말미암아 시스템의 통합 방식이 결정될 때가 많다. 기술적으로 이로운 단일화가 보고체계 탓에 불가능해질 수도 있다. 경영진에서 현실적이지 못한 합병을 지시할지도 모른다. 원하는 것을 항상 얻을 수는 없겠지만 적어도 발생하는 비용을 평가 및 전달하고, 이를 완화할 조취를 취할 수는 있을 것이다. 현실적인 CONTEXT MAP에서 시작해서 그것의 변형을 선택할 때 **실용주의를 견지**[2]**해야 한다.**

우리 자신을 컨텍스트에 배치하기

CONTEXT MAP을 구성할 때 우리가 편견을 인식하고, 언제 MAP의 적용 가능성의 한계를 벗어나 있는지 유념하자.

경계의 변형

BOUNDED CONTEXT의 경계를 세우는 데는 무수히 많은 상황과 선택사항이 있다. 하지만 대개 문제는 아래의 요인들 사이에서 균형을 유지하는 데 있다.

규모가 큰 BOUNDED CONTEXT가 선호되는 경우

- 사용자의 작업 흐름이 **단일화된 모델**을 토대로 처리될 때 더 매끄럽게 진행된다
- 개별적인 두 모델의 매핑을 더하는 것보다 **일관성 있는 하나의 모델**을 이해하기가 더 쉽다.
- 두 모델 간의 **번역**이 어려울 수 있다(불가능한 경우도 있다).
- **공유 언어**를 토대로 팀의 의사소통이 명확해진다

규모가 작은 BOUNDED CONTEXT가 선호되는 경우

- 개발자 간의 의사소통에 따른 과부하가 줄어든다.
- 소규모 팀과 코드 기반을 토대로 CONTINUOUS INTEGRATION이 쉬워진다.
- 대형 컨텍스트에서는 용도가 다양한 추상화 모델을 요구할 수도 있는데, 이 경우 제공하기 힘든 기술이 필요할 때가 있다.
- 각기 다른 모델은 특수한 요구사항을 해결하는 데 도움되거나 UBIQUITOUS LANGUAGE의 특화된 방언과 전문적인 사용자 집단의 전문 용어를 포괄할 수 있다.

변경할 수 없다는 사실을 인정하기: 외부 시스템의 묘사

레거시 시스템과 새로운 외부 시스템을 통합할 때 의미상 모순이 되지 않도록 각별히 주의를 기울여야 한다.

외부 시스템과의 연계

1. 먼저, SEPARATE WAYS를 고려해본다. ☞ **사용자가 두 시스템에 모두 손쉽게 접근하게 해주는 것만으로 충분한가?**
2. CONFORMIST ☞ **확장에만 주력하고 기존 모델을 변경해서는 안된다.**
3. ANTICORRUPTION LAYER ☞ 기존 시스템(다른 시스템에 대한 인터페이스의 규모가 작거나 다른 시스템 설계가 열악한)을 확장하는것보다 설계 중인 시스템의 기능이 더 복잡해질 경우 사용

설계 중인 시스템

전체 시스템에 대한 **단일한 BOUNDED CONTEXT**를 갖도록 노력하자.

개별 모델의 특수한 요구사항 충족하기

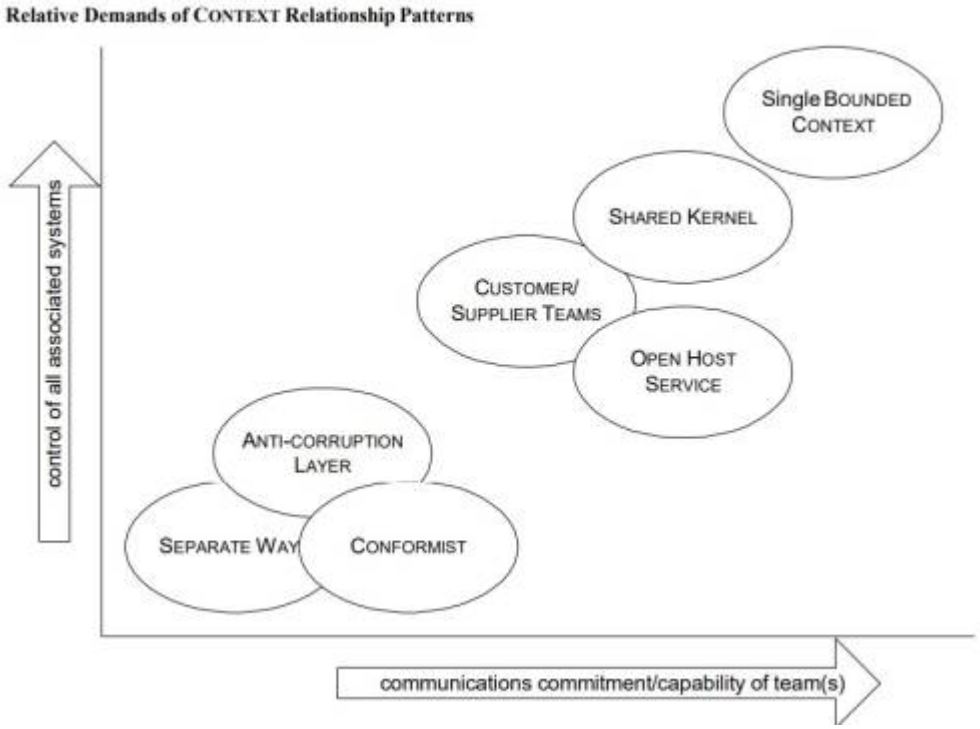
무엇보다 **이러한 사용자 집단의 특별한 전문 용어가 얼마나 가치 있는가?**
번역의 위험에 비춰 팀의 좀더 독립적인 활동이 지닌 가치를 따져봄으로써 가치가 없는 용어상의 **변종을 합리화하는 활동을 주시**해야 한다.

Deployment

BOUNDED CONTEXT의 전략을 선택하는 것은 배포에 영향을 미친다.

타협점

독립적인 활동과 순조로운 의사소통을 맞바꾸는 셈



프로젝트가 이미 진행 중일 때

현재 상태에 따라 BOUNDED CONTEXT를 정의하라. 일단 실제 BOUNDED CONTEXT의 현재 상태를 서술하고 현존하는 관계를 기술했다면 다음으로 **현재 조직을 중심으로 팀의 업무 관행을 밀접하게 정비**해야 한다.

1. CONTEXT 내 CONTINUOUS INTEGRATION을 개선한다.
2. 빗나간 번역 코드를 ANTICORRUPTION LAYER로 들어가게끔 리팩터링한다.
3. 기존 BOUNDED CONTEXT에 이름을 부여하고 반드시 이러한 BOUNDED CONTEXT가 프로젝트의 UBIQUITOUS LANGUAGE에 속하게 한다.

변형

Transformations

이미 결정한 CONTEXT의 경계를 실제로 변경하는 방법

CONTEXT 병합: SEPARATE WAYS → SHARED KERNEL

번역 따르는 과부하가 높은 경우.

두 CONTEXT에서 중복되는 부분을 포함하는 규모가 작은 하위 도메인을 선택하여 병합한다.

p420 참조.

CONTEXT 병합: SHARED KERNEL → CONTINUOUS INTEGRATION

SHARED KERNEL이 확장되어 단일 BOUNDED CONTEXT로 통합할 가능성이 있을 때 사용.

p422 참조.

레거시 시스템의 단계적 폐기

반복적인 배포 프로세스 중에 ANTICORRUPTION LAYER에서 불필요한 부분을 파악해 제거.

p423 참조.

OPEN HOST SERVICE → PUBLISHED LANGUAGE

접근을 원하는 시스템이 늘거나 상호작용이 점차 이해하기 어려워져 유지보수 부담이 가중될 경우.

표준적인 언어를 정의하라.

p425 참조.

---

[1] 발음과 의미가 같거나 유사하지만 다른 어원적 뿌리를 가진 단어 또는 그 단어들의 조합. 출처 : 위키백과
[2] 어떤 견해나 입장 따위를 굳게 지니거나 지킴. 굳게 지지함.