Rational Unified Process 를 통한 대규모 시스템 개발

Maria Ericsson

Rational Software 백서

TP 156

Rational®
the software development company

목차

이력 1

상호 연결된 시스템의 시스템 1

소프트웨어 개발 라이프사이클 2

시스템 개발의 워크플로우 및 결과물 3

상호 연결된 시스템에서 시스템 개발 4

- 분해 기준 4
- 조직 5
- 상위 시스템의 라이프사이클 5
- 하위 시스템의 라이프사이클 8

상호 연결된 시스템의 시스템 유스 케이스 11

상호 연결된 시스템의 시스템 설계 모델 12

상호 연결된 시스템의 시스템 정보 세트 13

상호 연결된 시스템의 시스템 구조 14

시스템 간의 관계 15

어플리케이션 영역 16

- 대규모 시스템 16
- 분산 시스템 17
- 레거시 시스템의 재사용 17
- 사전 작성된 패키지 사용 17

요약 17

참조서 18

이력

이 문서는 Jacobson, Karin Palmkvist 및 Susanne Dyrhage [1]가 1995 년 5 월-6 월에 ROAD 에 발표한 "Systems of Interconnected Systems"에서 발췌한 것입니다. 이 문서는 몇 가지 대규모 시스템 개발 프로젝트의 입력에서 이익을 얻고 Rational Unified Process 버전 5.1 [2] 및 UML(Unified Modeling Language) [3]에 맞추려고 한 것입니다.

상호 연결된 시스템의 시스템

대규모 시스템 개발시 복잡도가 상당히 증가하게 됩니다. 더욱 복잡한 결과물 세트를 포함할 뿐 아니라 더 큰 자원 세트가 필요하므로 오버헤드가 발생할 수 있습니다. 이 문서는 추가된 복잡도 오버헤드를 제어하는 데 도움을 주기 위해 사용되는 구조적 패턴을 설명합니다. [4]에서 논의된 다른 위치 사이의 구조적 패턴을 **상호 연결된 시스템의 시스템**이라고 합니다.

이 구성은 명령 및 제어 시스템 또는 고도로 통합된 IT 솔루션과 같이 크거나 복잡한 시스템 빌드시 유용합니다. 대부분의 경우에 이런 유형의 "수퍼 시스템"은 각각이 분리된 시스템으로 독립적으로 개발되는 몇 개의 별도 파트로 이루어집니다. 수퍼 시스템은 수퍼 시스템의 임무를 수행하기 위해 서로 통신하는 상호 연결된 시스템 세트를 통해 구현됩니다. 이런 시스템 중 하나가 전체 성능을 나타내며 이 시스템을 **상위 시스템**이라고 합니다. 나머지 시스템은 전체 파트를 나타내며 이런 시스템을 **하위 시스템**이라고 합니다. 상위 시스템은 이를 구현하는 하위 시스템과 분명히 다릅니다. 다른 유형의 시스템 사이의 관계는 별개로 작성됩니다. 상위 시스템 관점에서 하위 시스템은 서브시스템입니다. 그림 1 을 참조하십시오.

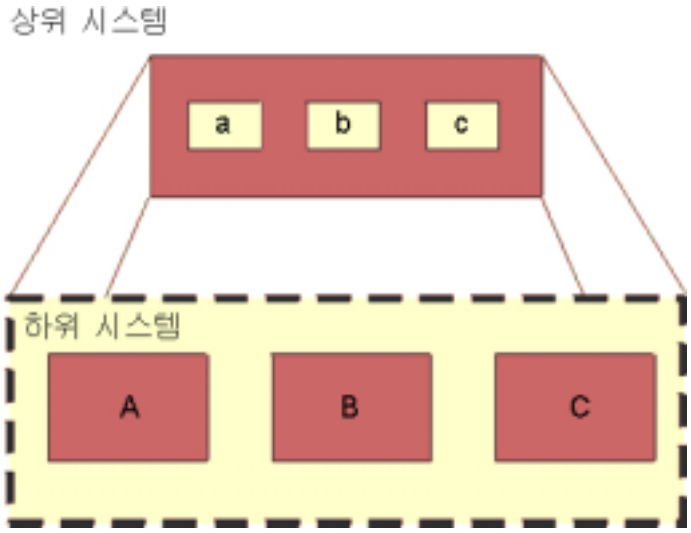

그림 1. 상위 시스템의 스펙은 상호 연결된 시스템의 시스템에 의해 구현됩니다. 시스템 A, B 및 C 는 상위 시스템의 의 서브시스템 a, b 및 c 각각의 구현입니다.

상위 시스템을 하위 시스템에서 분리하는 것은 몇 가지 장점을 가집니다.

- 하위 시스템은 영업 및 결과를 포함한 모든 라이프사이클 활동 중에 개별적으로 관리될 수 있습니다.
- 이것은 하위 시스템을 사용하여 하위 시스템을 상호 연결된 시스템 중 다른 시스템으로 플러그인함으로써 다른 상위 시스템을 구현하는 것을 용이하게 합니다.

- 시스템 빌드를 시작할 때 시스템이 상호 연결된 시스템 중 하나인지 여부를 항상 알고 있지는 않습니다. "단순" 시스템 관점에서 작업을 시작하고 라이프사이클의 후반부에서 상호 연결된 시스템 패턴의 시스템을 적용해야 하는지 여부를 판별할 수 있습니다.
- 이것은 상위 시스템의 새 버전을 개발하지 않고도 하위 시스템에 내부 변경을 작성할 수 있게 합니다. 상위 시스템의 새 버전은 주요 기능 변경시에만 필요합니다.

각 하위 시스템에는 결과물 간의 명확한 추적성을 포함하는 관련 결과물 세트가 있습니다. 또한 하위 시스템의 결과물 세트에서 대응되는 상위 시스템의 결과 세트까지 유지보수되는 추적성이 있습니다. 각 하위 시스템은 자체 라이프사이클 단계(초기, 구현, 구축 및 이행)를 포함하는 별도 개발 프로젝트로 관리될 수 있습니다.

빌드 중인 "수퍼 시스템"이 큰 규모인 경우, 하위 시스템이 더 세부적으로 분할될 필요가 있을 수 있으므로 상호 연결된 시스템의 시스템으로 간주됩니다.

소프트웨어 개발 라이프사이클

Rational Unified Process 에서 개발 라이프사이클은 관리 관점 및 개발 관점의 두 가지 관점으로 표시되고 논의됩니다. 그림 2 를 참조하십시오.

관리 관점에서는 시스템 또는 새로운 세대의 시스템을 개발하기 위해 4 가지 라이프사이클 단계를 통과합니다. 개발 관점에서는 점진적으로 완성되어 가는 시스템 버전을 반복적으로 개발합니다. 반복 단계에서 수행되는 활동은 Rational Unified Process 에서 핵심 워크플로우 세트로 그룹화되어 있습니다. 각 핵심 워크플로우는 일부 시스템 측면을 설명하고 시스템의 모델 또는 문서 세트를 생성하는 데 중점을 둡니다.

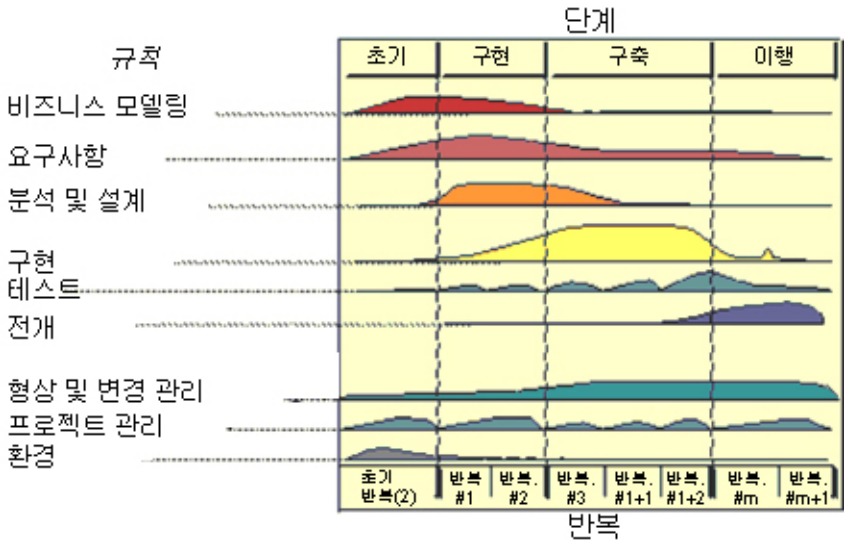

그림 2. 반복 모델

이것을 상호 연결된 시스템의 시스템에 적용하여 각각의 하위 시스템 전체와 상위 시스템이 자체 라이프사이클을 통과하며 흔히 개별 프로젝트로 간주됩니다.

그림 3. 각각의 상위 및 하위 시스템이 자체 라이프사이클을 통과합니다.

물론 라이프사이클에는 종속성이 있습니다. 이런 종속성을 올바르게 관리하는 것은 상호 연결된 시스템 내의 시스템을 개발하는 데 중요한 도전 중 하나입니다. 종속성의 유형은 다음과 같습니다.

- 라이프사이클은 시간에 종속됩니다. 상위 시스템의 라이프사이클이 먼저 시작됩니다. 상위 시스템이 최소한 하나의 반복을 마치고 하위 시스템의 인터페이스가 비교적 안정되면 하위 시스템의 라이프사이클이 시작될 수 있습니다. 사실, 상위 시스템에서 최소한 하나의 반복을 마칠 때까지 하위 시스템이 어느 것인지 알지 못할 수도 있습니다.
- 하위 시스템의 인터페이스가 안정적이 되면 상위 시스템의 라이프사이클은 유지보수에 들어갈 수 있습니다. 이것은 하위 시스템의 인터페이스 변경을 필요로 하는 문제가 발생하는 경우에만 활성 개발이 수행되지 않음을 의미합니다.
- 하위 시스템의 인터페이스는 상위 시스템 개발자가 소유합니다. 인터페이스에 대한 자세한 정보는 [3] 및 [5]를 참조하십시오.
- 하위 시스템의 인터페이스를 구현하는 클래스는 하위 시스템 개발자가 소유합니다.

시스템 개발의 워크플로우 및 결과물

하위 시스템뿐 아니라 상위 시스템도 비복합 시스템에 대해 일반적인 것과 같이 동일한 결과물 세트를 사용하여 개발될 수 있고 동일한 워크플로우를 통해 개발될 수 있다고 가정합니다. 계속 진행하기 전에 수행 방법을 표시하는 이런 결과물 및 워크플로우가 도입되어야 합니다. Rational Unified Process 에서는 다음과 같은 다섯 가지 핵심 프로세스 워크플로우를 도입합니다. 그림 4 를 참조하십시오.

- 비즈니스 엔지니어링 - 시스템이 사용되는 조직을 평가하고 시스템에서 해결될 요구 및 문제점에 대해 더욱 잘 이해하는 것이 목적입니다. 결과는 비즈니스 유스 케이스 모델 및 비즈니스 객체 모델입니다. 이 워크플로우는 선택적입니다. 시스템이 사용될 조직이 단순한 경우 중요하지 않을 수 있습니다.
- 요구사항- 사용성에 중점을 두고 요구사항을 캡처하고 평가하는 것이 목적입니다. 이로 인해 시스템과 통신하는 외부 단위를 나타내는 액터 및 트랜잭션 순서를 나타내고 측정 가능한 결과 값을 액터에게 내주는 유스 케이스를 포함하는 유스 케이스 모델이 생성됩니다.

- 분석 및 설계 – 의도된 구현 환경 및 시스템 구성에 미치는 영향을 조사하는 것이 목적입니다. 이로 인해 유스 케이스의 플로우를 수행하기 위해 객체가 통신하는 방법을 표시하는 유스 케이스 구현을 포함한 객체 모델(설계 모델)이 생성됩니다. 이것은 제공된 조작에 따라 책임을 지정하여 클래스 및 서브시스템에 대한 인터페이스 정의를 포함할 수 있습니다. 이 객체 모델은 구현 언어, 분배 등에 따라 구현 환경에도 채택됩니다. 때때로 독립된 모델의 분석 결과를 조사하는 데 유용하여 분석 모델이라고 합니다.

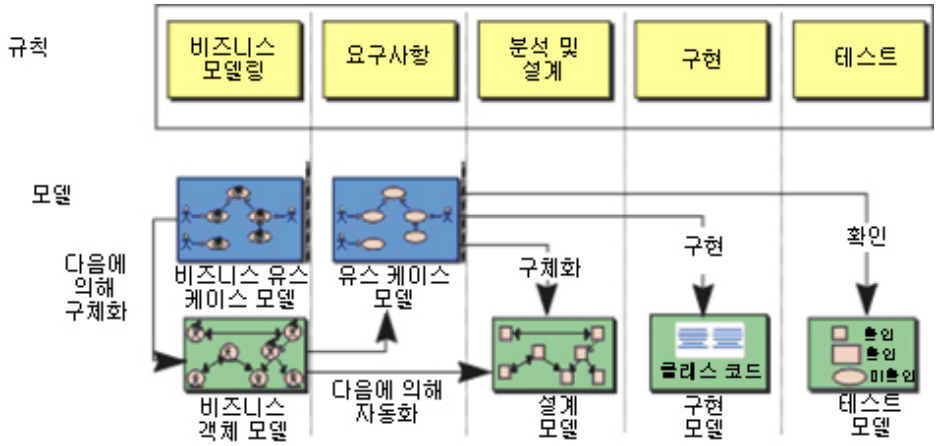

그림 4. 각 핵심 프로세스 워크플로우는 특정 모델 세트와 관련되어 있습니다.

- 구현 – 규정된 구현 환경에 시스템을 구현하는 것이 목적입니다. 이로 인해 소스 코드, 실행 파일 및 파일이 생성됩니다.
- 테스트 – 시스템이 의도된 것인지, 구현에 오류가 없는지 확인하는 것이 목적입니다. 따라서 인증된 시스템에서 산출용으로 생성됩니다.

상호 연결된 시스템에서 시스템 개발

실제로 수행해야 하는 것은 시스템의 책임을 여러 시스템에 분배하여 시스템 각각이 잘 정의된 책임 서브세트를 담당하게 하는 방법을 정의하는 것입니다. 이것의 주요 목표는 하위 시스템 간의 인터페이스를 정의하는 것입니다. 이를 완료했을 때 나머지는 "분할 정복" 원칙에 따라 각 하위 시스템에 대한 별도의 작업을 수행할 수 있습니다. 그러므로 이것은 구현이 수행된 후 테스트와는 별도로, 전체적으로 시스템에 대해 수행해야 하는 모든 것을 입니다.

분해 기준

시스템을 상호 연결된 시스템의 시스템으로 분해할지 여부를 어떻게 결정합니까? 다음과 같은 몇 가지 특성이 고려되어야 합니다.

- 상당히 규모가 크고 복잡한 시스템의 경우, 문제점을 단번에 이해하기 쉽도록 작게 나눌 수 있습니다.
- 물리적으로 분리된 시스템을 처리하고 있습니까?(종종, 레거시 시스템 또는 레거시 구조로 작업할 경우)
- 분해를 통해 시스템 파트 간의 일반적이고 한정된 인터페이스를 정의할 수 있습니다.
- 몇 가지 주요 COTS(Commercial-Off-The-Shelf) 제품을 사용하여 시스템의 일부분을 구현하기로 결정할 수 있습니다. 분해를 통해 COTS 제품의 사용 방법을 명확히 할 수 있습니다.

- 분할은 분산된 개발 조직에서 최상의 것을 얻고 지리적으로 분산된 몇몇 팀 간에 작업을 명확하게 분할할 수 있게 합니다.

다음과 같은 위험 요소도 고려되어야 합니다.

- 과도한 분해로 모든 세부사항에 대한 전반적인 문제점을 숨길 수 있습니다.
- 물리적으로 분리된 시스템 또는 물리적으로 분리된 팀을 사용하면 모든 양식을 재사용을 없게 되며, 결국에는 고정된 스토브 파이프(stove-pipe) 시스템이 됩니다.

조직

위에서 언급된 위험을 완화하려면 인력 그룹이 전체 개발 노력을 감시하도록 지정하는 것이 중요합니다. 종종 이 그룹을 구조 팀이라고 하며, 이 그룹은 다음과 같은 주요 관심사에 중점을 둡니다.

- 정의된 전체 구조가 있고 하위 시스템에서 이 구조를 수행하는 것.
- 재사용 및 하위 시스템 간 경험 공유에 중점을 두는 것.
- 어떤 결과물이 생성되는지와 하위 및 상위 시스템 결과물 간의 관계가 무엇인지에 대해 명확하게 이해하는 것.
- 효율적인 변경 관리 전략이 정의되고 모든 팀이 이 전략을 따르는 것.

항상 그렇지는 않지만, 구조 팀이 상위 시스템 개발을 소유합니다. 조직에 대한 완전한 논의는 [6]을 참조하십시오.

상위 시스템의 라이프사이클

먼저, 시스템의 컨텍스트에 대한 더 나은 이해를 위해 선택적으로 **비즈니스 엔지니어링** 수행을 선택할 수 있습니다. 다음과 같은 경우에 더욱 가치가 있습니다.

- 개발자가 조직에 대해 좀 더 충분히 이해해야 한다는 요구가 있습니다.
- 조직 자체가 조정될 필요가 있는 비즈니스, 용어 및 프로세스를 수행하는 방법에서 이질적입니다.
- 소프트웨어 엔지니어링 노력은 비즈니스 리엔지니어링 노력과 관련하여 수행됩니다.

[6]도 참조하십시오.

이 노력의 결과로 비즈니스 유스 케이스 모델 및 비즈니스 객체 모델이 생성될 수 있습니다. 대신, 비즈니스 도메인의 핵심 개념만을 살펴보는 제한된 비즈니스 엔지지어링을 수행하고 이런 개념을 비즈니스 객체 모델에 문서화도록 선택할 수 있습니다. 이것을 종종 도메인 모델링이라고 부릅니다.

비즈니스 모델 세트를 "준비"했으면 전체 시스템에 대한 **요구사항**을 도출하기 시작해야 합니다. 상호 연결된 시스템 중 한 시스템의 요구사항 모델링에 대한 요구는 다른 모든 시스템의 요구와 동일합니다. 유스 케이스 모델은 결과를 표시하는 자연스러운 방법으로, [7]을 참조하십시오. 이 상위 유스 케이스 모델을 살펴보는 가장 수월한 방법은 상위 유스 케이스 모델이 시스템의 작동 요구사항을 완전하게 캡처한다고 가정하는 것입니다. 그러나 이것은 거의 사용되지 않는 케이스입니다. 시스템을 다른 시스템과 함께 구현해야 하므로 전체 시스템이 다소 복잡할 것입니다. 따라서 이 레벨에서 철저하려고 하는 것은 좋은 생각이 아닙니다. 보통 상위 유스 케이스 모델은 시스템의 기능적 요구사항에 대해 완전하지만 단순한 그림을 제공합니다. 세부사항 모델링이 구현되는 각 하위 시스템 내에서 수행되므로 이 레벨에서 자세한 세부사항은 필요하지 않습니다. 또한 몇몇 서브시스템을 잘라내는 상위 유스 케이스에 많은 요구사항이 보여야 할 필요는 없습니다. 이런 요구사항은 서브시스템에 대한 "로컬"이라고 할 수 있습니다.

분석 및 설계의 목적은 견고한 시스템 구조를 이루는 것입니다. 이것은 물론 상호 연결된 시스템의 시스템에서 극히 중요합니다. 상위 시스템의 개발자는 자체 시스템의 내부 구조를 전혀 걱정할 필요가 없지만 하위 시스템의 견고한 구조를 달성해야 합니다. 따라서 시스템의 한 부분을 서브시스템을 사용하여 더 작은 파트로

모델화합니다. 올바른 서브시스템 세트를 얻고 이런 서브시스템에 상위 시스템의 책임을 분배시키는 방법에 대한 좋은 아이디어를 얻기 위해 분석 모델을 개발합니다. 상위 레벨 유스 케이스가 수행될 때 분석 클래스는 시스템의 파트에 의해 수행되는 역할을 표시해야 합니다. 따라서 분석 모델은 상위 레벨 유스 케이스 모델에서 유추하여 전체 객체 구조의 단순화된 그림을 제공합니다.

기능적으로 연관된 분석 클래스는 함께 서브시스템으로 그룹화됩니다. 따라서 기능적 기준만을 기반으로 한다는 의미에서 이상적인 서브시스템 그룹을 얻게 됩니다(예: 분배 요구사항 고려하지 않음). 주로 매우 영향력이 있는 요인은 레거지 시스템입니다. 레거시 시스템은 분석 모델에서 정의되는 일부 책임 또는 대부분의 책임을 이행할 수 있습니다. 이런 시스템의 존재는 기존 성능을 최대한 재사용할 수 있도록 분석에서 발견된 책임을 다시 분배하기도 합니다.

설계의 결과는 분석 중에 기능적 기준에 기반하여 정의한 서브시스템 구조와 매우 다른 것일 수 있습니다. 결국 각각이 하위 시스템에서 구현되는 설계 서브시스템의 구조로 종료됩니다. 그림 5 를 참조하십시오. 이와 같은 각각의 시스템에 대해 개별적으로 개발 작업을 계속할 수 있도록 각 서브시스템에 대한 인터페이스가 정의됩니다. 사실, 인터페이스가 하위 시스템의 개발에 대한 규칙을 제공하므로 인터페이스 정의는 상위 레벨에서 수행되는 가장 중요한 활동입니다. 설계 클래스가 정의되지 않은 경우 수행하는 유일한 작업은 설계 서브시스템의 인터페이스를 정의하는 것입니다.

구현화는 상위 시스템 라이프사이클의 일부로 수행되지 않습니다. 그보다는 시스템의 특정 설명 측면을 탐색하기 위한 일부 프로토타입 작업일 수 있습니다.

최종 워크플로우는 **테스트**이며 이 경우에는 여러 하위 시스템이 어셈블된 경우의 통합 테스트를 의미합니다. 또한 협력하는 상호 연결된 시스템의 스펙에 따라 모든 상위 유스 케이스가 수행되는 테스트를 의미합니다.

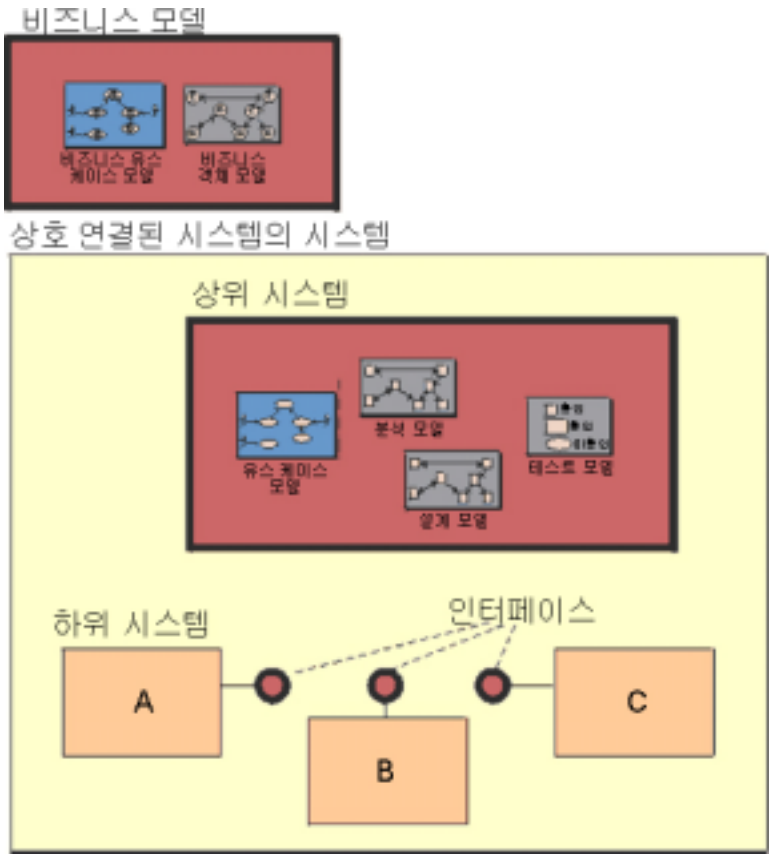

그림 5. 상위 시스템은 상위 레벨 설계 모델에서 정의된 서브시스템이 하위 모델 시스템에 의해 구현되는 모델 세트로 설명됩니다. 하위 시스템의 인터페이스는 상위 시스템이 소유합니다.

상위 시스템에 대해 작업하는 방법을 표시하기 위한 몇 가지 샘플 반복 계획이 있습니다. 하나는 상위 시스템 라이프사이클의 초기 단계에서의 반복이고 또 하나는 구현화 단계에서의 반복입니다. 반복 계획을 설명하기 위해 활동 다이어그램을 사용합니다. 이런 다이어그램의 조치 상태는 Rational Unified Process 에 정의된 워크플로우 세부사항에 대응됩니다.

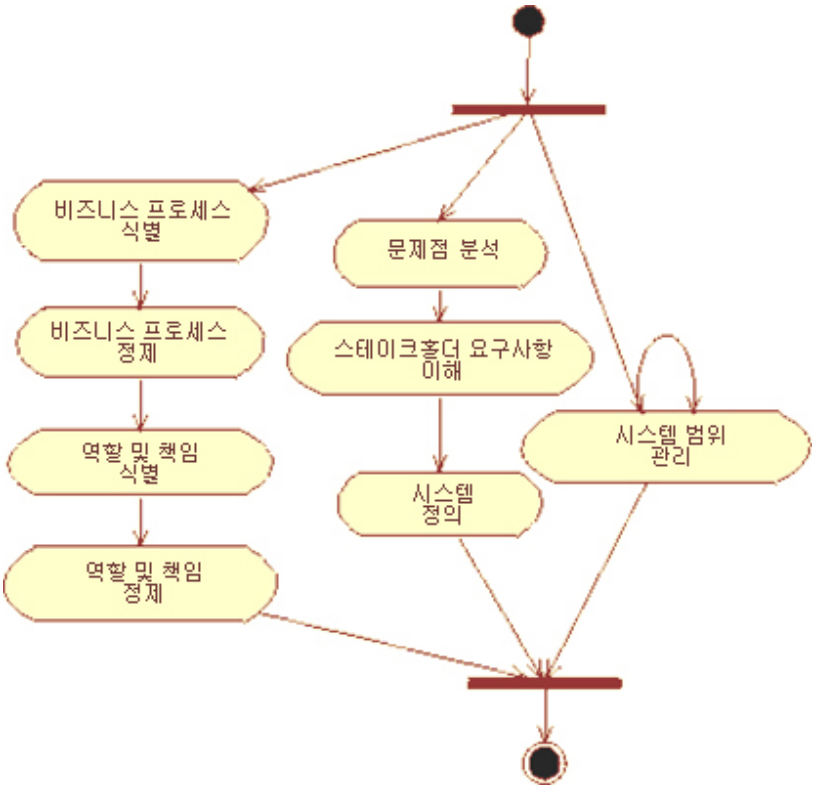

그림 6. 상위 시스템에 대한 초기 반복 계획의 예를 설명하는 활동 다이어그램.

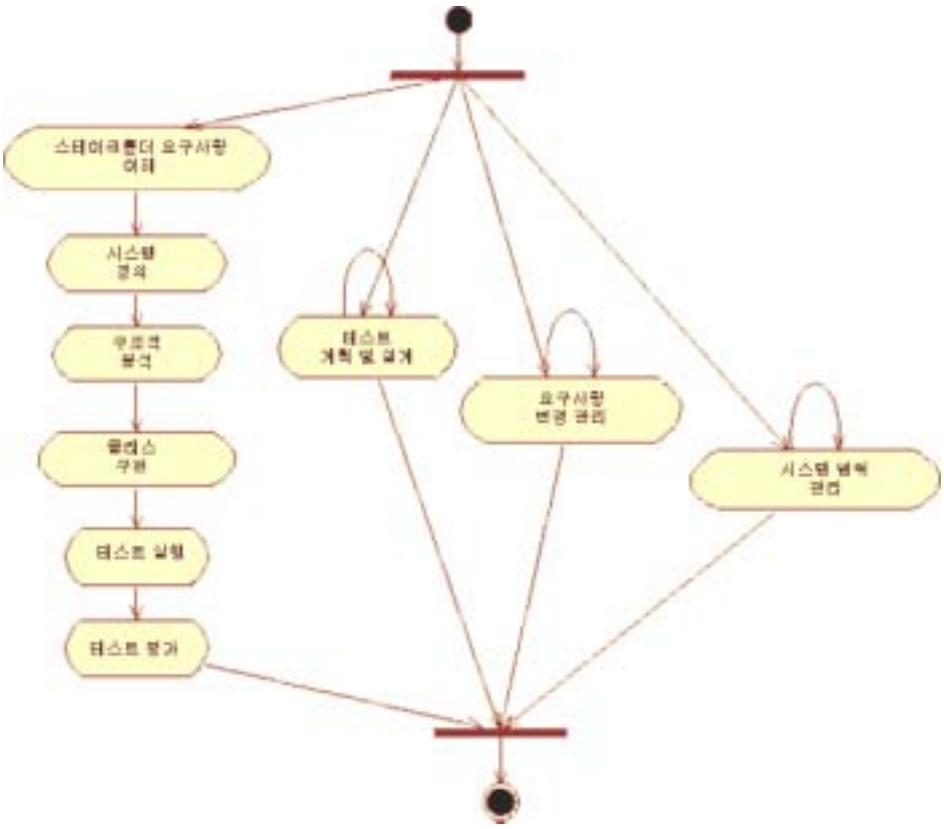

그림 7. 상위 시스템에 대한 구현화 반복 계획의 예를 설명하는 활동 다이어그램. 시스템의 설명적 측면을 탐색하기 위해 일부 제한된 프로토타입 구현화를 수행할 수도 있으므로 여기에 "클래스 구현화" 조치 상태가 있습니다.

하위 시스템의 라이프사이클

각 하위 시스템은 보통 액터로서 통신하는 다른 시스템을 주시하는 블랙 박스로 개발됩니다. 위에 설명된 대로 이와 같은 각 시스템에 대해 통상적인 활동 세트를 수행하고 통상적인 모델 세트를 개발합니다. 상위 레벨에 있는 모델이 모든 세부사항에 정의된 경우, 다른 레벨에 있는 모델 간에 전체 반복을 얻게 되지만 위에 언급된 대로 이것은 실제로는 거의 없는 경우입니다.

하위 시스템에 대해 **요구사항** 워크플로우를 수행합니다. 상위 시스템의 인터페이스 및 유스 케이스는 하위 시스템의 경계와 액터가 무엇인지에 대해 이해하기 위한 기본 입력이 됩니다.

하위 시스템에 대해 **분석 및 설계**를 수행할 때 상위 시스템에 정의된 인터페이스는 상위 레벨 유스 케이스와 함께 "경계 조건"이 됩니다.

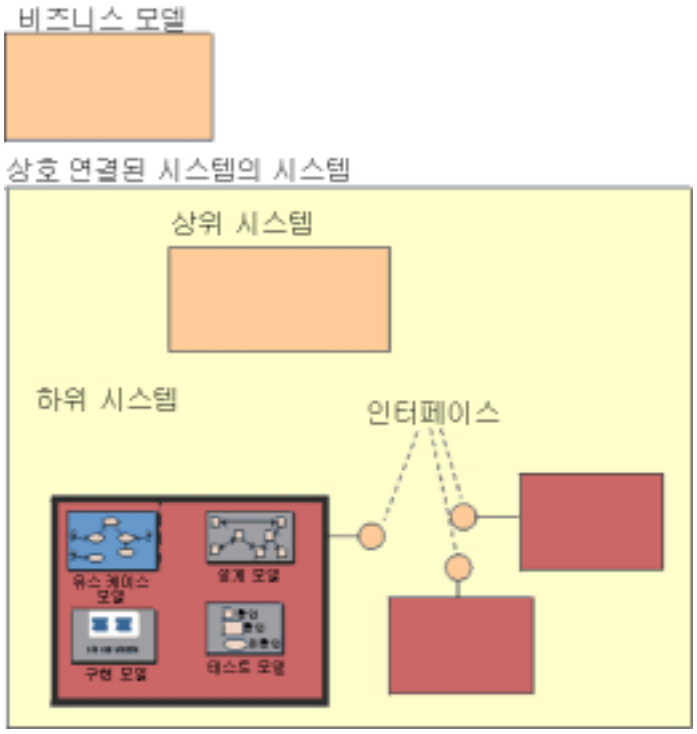

그림 8. 하위 시스템은 자체 모델 세트로 설명됩니다.

하위 시스템에 대해 작업하는 방법을 표시하기 위해 여기서는 라이프사이클에서의 두 가지 샘플 반복 계획을 설명하고 있습니다.

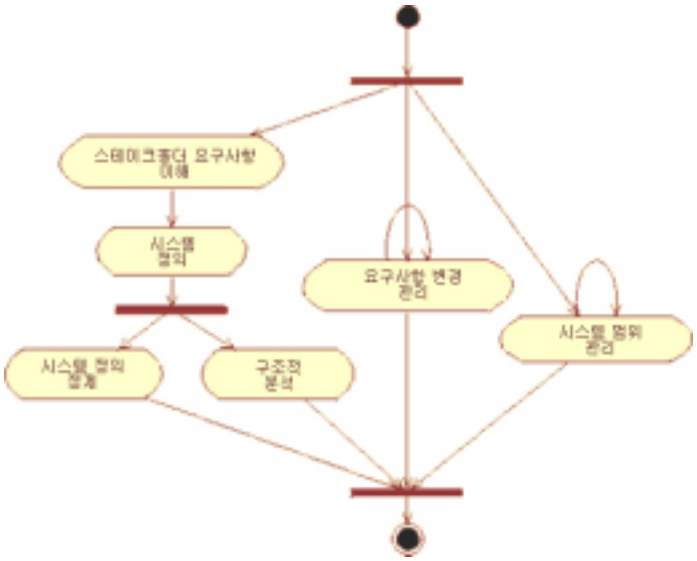

그림 9. 하위 시스템에 대한 샘플 초기 반복 계획. 실행 파일이 생성되지 않으므로 이것은 불완전한 반복입니다.

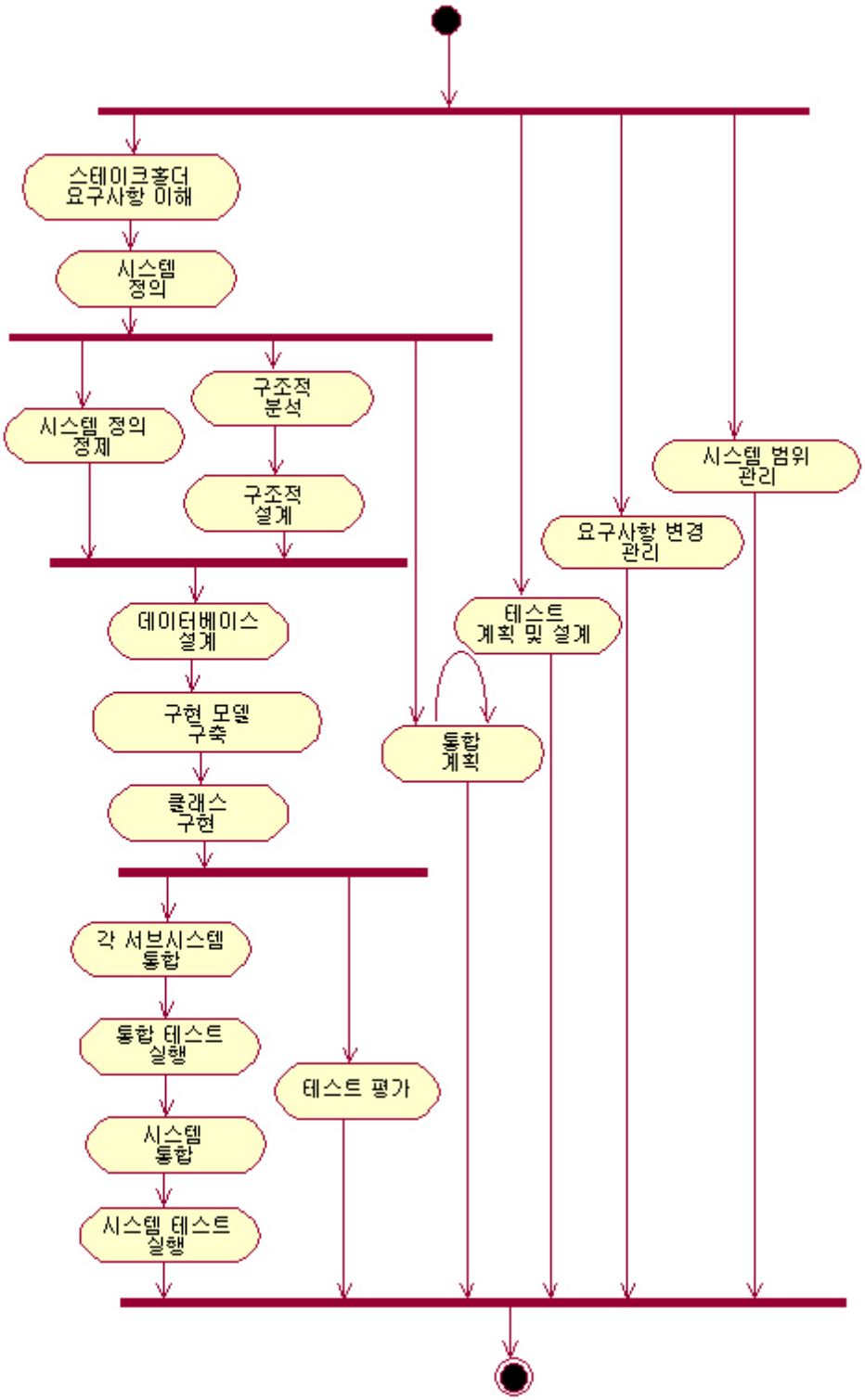

그림 10. 하위 시스템에 대한 샘플 구현화 반복 계획. 구현화에서는 정제된 시스템 정의 및 구조를 완료하는 데 중점을 둡니다.

상호 연결된 시스템의 시스템 유스 케이스

상호 연결된 시스템의 시스템에서 각각의 상위 및 하위 시스템 모두에 대해 하나의 유스 케이스 모델을 빌드해야 합니다. 이런 시스템들은 다음과 같은 방법으로 종속됩니다(그림 11 참조).

- 상위 시스템의 상위 레벨 유스 케이스는 (항상 그러한 것은 아니지만 자주) 서브시스템으로 분리됩니다. 하위 시스템의 경우 각 "분할"은 모델의 유스 케이스가 됩니다. 그림 11 을 참조하십시오.
- 한 하위 시스템의 관점에서 다른 하위 시스템은 유스 케이스 모델의 액터입니다. 그림 12 를 참조하십시오.

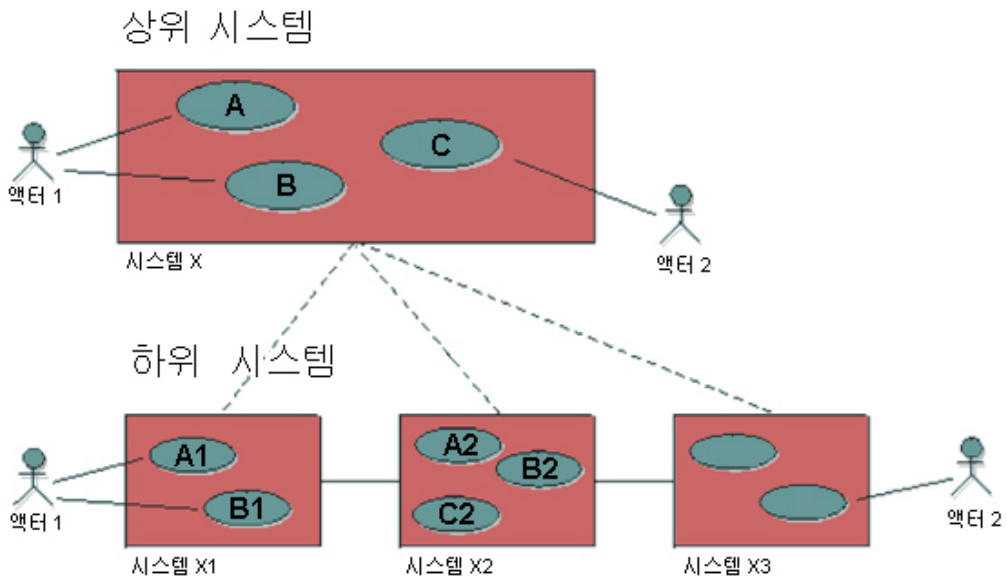

그림 11. 상위 시스템의 상위 레벨 유스 케이스 간 관계 및 하위 시스템의 유스 케이스 세부사항.

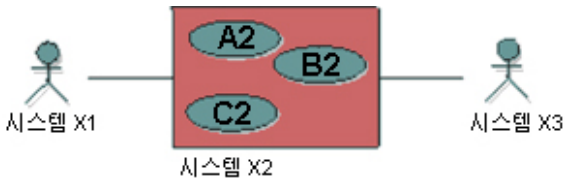

그림 12. 하위 시스템 X2 의 유스 케이스 모델에서 다른 하위 시스템 X1 및 X3 는 액터로 보입니다.

상위 시스템을 설명하는 유스 케이스에 대한 몇 가지 특별한 고려사항이 있습니다. 각 하위 시스템에 모든 요구사항을 어느 정도까지 다시 설명하므로 이런 유스 케이스에 대해 지나치게 세부적으로 조사하는 것은 의미가 없습니다. 보통 경우에, 설명적인 텍스트에 세부적으로 설명하지 않고 상위 레벨 유스 케이스의 이벤트 플로우에 대한 단계별 아웃라인을 쓰는 것만으로 충분합니다.

이 유스 케이스 모델에서 어떤 유스 케이스 관계(일반화, 확장, 포함)도 사용해서는 안됩니다. 일반적으로, 다음과 같은 이유로 가치를 더하지 않습니다.

- 상위 레벨 유스 케이스 세부사항을 설명하지 않으므로 몇몇 곳에서 나타나는 텍스트에 대해 우려하지 않아도 됩니다.
- 하위 시스템에 있는 상위 레벨 유스 케이스를 "분리"할 때 어떤 방법으로든 정보를 구성합니다. 다른 구조 메커니즘과의 혼합은 혼란을 줄 수 있습니다.

상호 연결된 시스템의 시스템에서 재사용 가능한 컴포넌트를 찾으려고 하는 경우 중요한 예외가 있습니다. 일반 유스 케이스를 찾기 위한 상위 유스 케이스 모델 구축은 재사용 가능한 컴포넌트를 찾을 수 있는 강력한 방법입니다. 이 주제에 대한 세부사항은 [6]을 참조하십시오.

상호 연결된 시스템의 시스템 설계 모델

상호 연결된 시스템의 시스템에서 각각의 상위 및 하위 시스템 모두는 자체 설계 모델을 가져야 합니다. 설계 모델은 다음 방법으로 관련됩니다.

- 하위 시스템에 대한 설계 모델에 있는 서브시스템은 하위 시스템의 경계를 정의합니다.
- 상위 시스템의 서브시스템을 정의하는 조작은 하위 시스템에 인터페이스 정의에 대해 입력하는 것입니다.

상위 시스템의 설계 모델은 하위 설계 모델보다 덜 자세하게 설명됩니다. 다음을 생성할 수 있습니다.

- 간략하게 설명된 서브시스템.
- 서브시스템이 협력하는 방법에 따른 유스 케이스 구현화. 이런 상위 레벨 유스 케이스 구현을 문서화하는 일반적인 방법은 순서 다이어그램을 그리는 것입니다. 하위 시스템에 있는 상위 레벨 유스 케이스의 "분리"를 정의하는 다이어그램을 생성하는 것입니다. 그림 13 을 참조하십시오.
- 서브시스템의 조작.
- 서브시스템에 대한 인터페이스 정의.

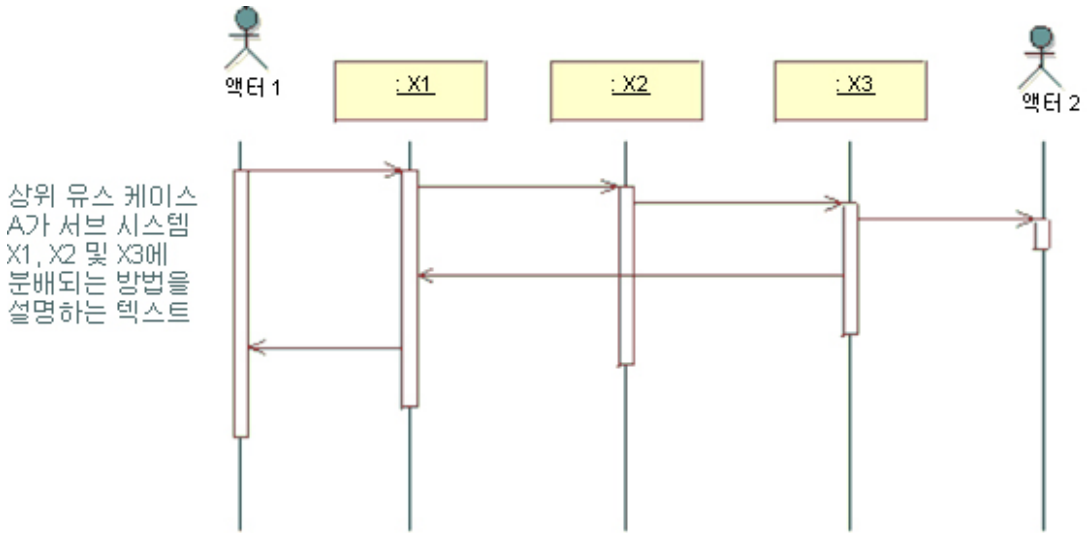

그림 13. 상위 유스 케이스 A 의 구현화에 대한 순서 다이어그램.

상호 연결된 시스템의 시스템 정보 세트

대부분의 조직이 많은 노력을 들이는 부분은 결과물을 관리하는 방법을 이해하고 이들의 종속성에 대해 올바르게 이해하는 것입니다. 이전 섹션에서 특히 상위 및 하위 유스 케이스 모델과 설계 모델 간의 종속성에 대해 논의했습니다. 고려되어야 할 일반 종속성 문제도 있습니다.

시스템이 라이프사이클을 통과할 때 [8] 정보 세트로 구성될 수 있는 결과물을 생성합니다. 그림 14 를 참조하십시오. 이런 세트는 결과물이 "함께" 전개되는 것에 기반하여 구성됩니다.

- 빌드 중인 어플리케이션의 유형에 따라 각 세트의 정확한 컨텐츠를 사용자 정의할 수 있지만 세트는 동일한 것으로 남아 있습니다.
- 결과물 간의 추적성을 효과적인 방법으로 유지보수할 수 있도록 세트 사이의 종속성을 이해해야 합니다.

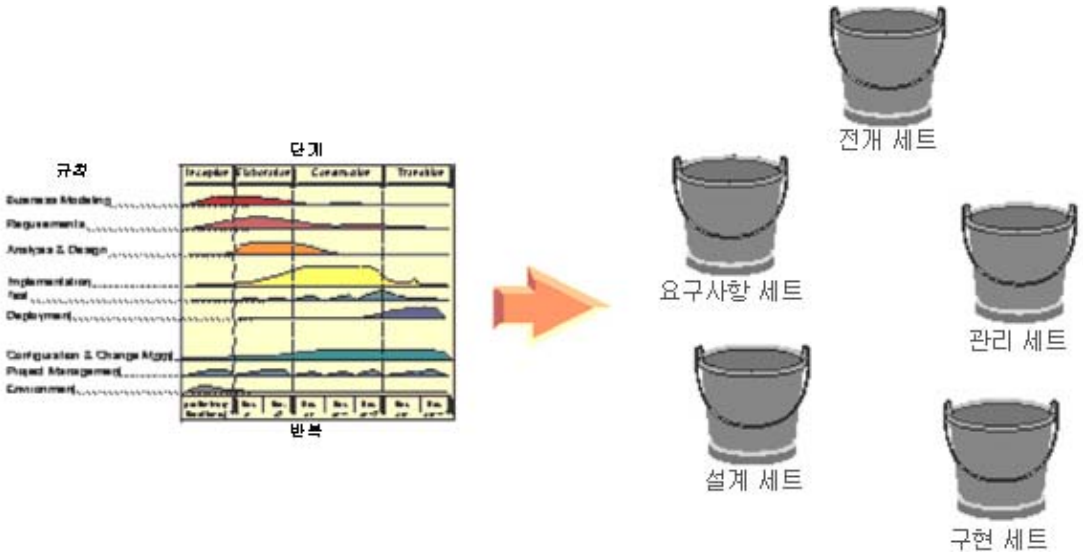

그림 14. 시스템의 라이프사이클이 정보 세트를 생성합니다.

상호 연결된 시스템의 시스템에서 상위 시스템과 각각의 하위 시스템은 자체 정보 세트를 생성합니다. 그림 15 를 참조하십시오.

- 하위 정보 세트는 대응되는 상위 정보 세트에 대해 종속성을 가집니다.
- 어플리케이션 유형이 다를 수 있으므로 대응되는 정보 세트의 컨텐츠 유형이 다를 수 있습니다.
- 대응되는 하위 정보 세트는 상위 시스템에서 정의된 것과 동일한 인터페이스를 이행하는 경우를 제외하고 독립적이어야 합니다.

상위 시스템의 결과물과 하위 시스템의 결과물 간의 추적성을 유지보수하는 데 드는 노력은 최저치로 유지되어야 합니다. 시스템에 내의 추적성을 유지보수하는 것이 우선되어야 합니다.

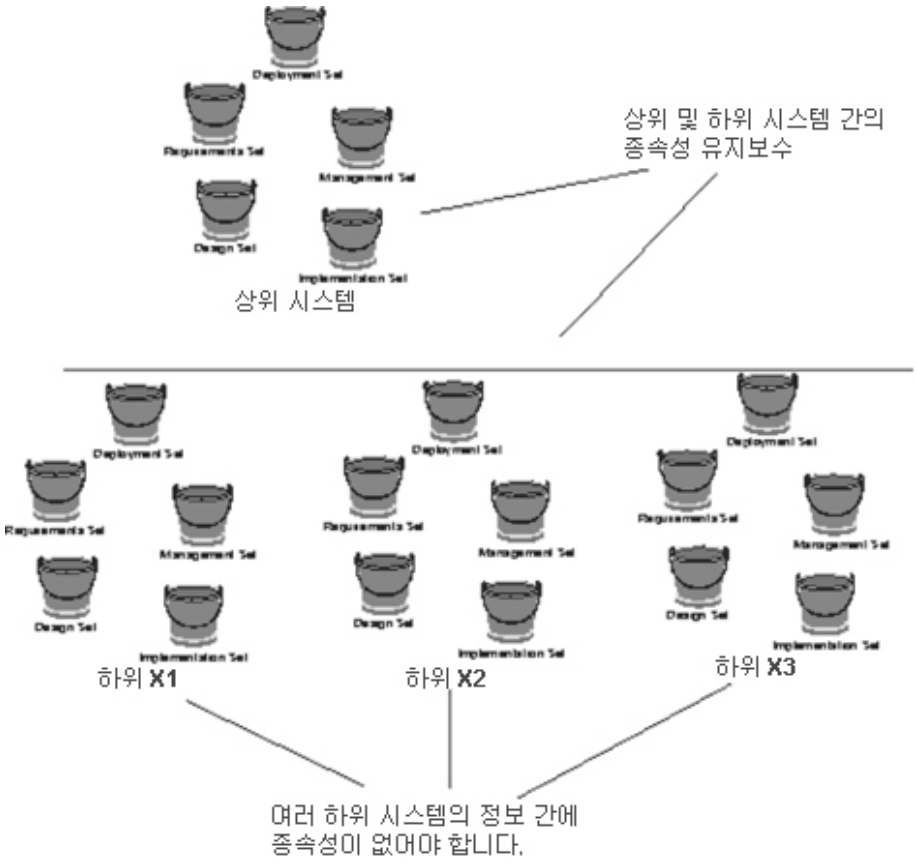

그림 15. 상호 연결된 시스템의 시스템에서 각각의 시스템은 자체 정보 세트를 생성합니다.

상호 연결된 시스템의 시스템 구조

상호 연결된 시스템의 시스템에서 각각의 상위 및 하위 시스템 모두는 정의된 구조를 가져야 합니다.

상위 시스템의 구조 문서에서는 다음에 대해 논의되어야 합니다.

- 상위 시스템에 대한 주요 유스 케이스 또는 시나리오.
- 상호 연결된 시스템의 시스템 계층화.
- 하위 시스템 간 재사용을 처리하는 방법 및 재사용 대상.
- 모든 하위 시스템에서 사용될 수 있는 일반적인 키 메커니즘 및 이의 구현화. 예를 들어, 모든 하위 시스템은 의사소통, 오류 보고 및 결함관리에 대한 공통적인 메커니즘을 사용해야 합니다. 그렇지 않으면 상위 시스템이 동종 시스템으로 작동하지 않습니다.

하위 시스템의 구조 문서에서는 다음에 대해 논의되어야 합니다.

- 상호 연결된 시스템의 시스템 내에 있는 하위 시스템의 역할.
- 하위 시스템에 대한 주요 유스 케이스 또는 시나리오.
- 하위 시스템이 상호 연결된 시스템의 시스템에 대해 정의된 계층 구조를 사용하는 방법. 이에 대해 언급하는 또 다른 방법은 하위 시스템이 계층화된 상위 시스템 구조에서 이에 대해 정의된 역할을 이행하는 방법을 정의해야 한다는 것입니다.
- 어떤 일반 키 메커니즘이 사용되는지, 어떤 어플리케이션 특정 키 메커니즘이 어떻게 추가되는지.

- 재사용이 적용되는 방법. 특히, 어떤 서브시스템이 두 개 이상의 하위 시스템에 걸쳐 공통적인지, 하위 시스템이 통신하도록 허용하기 위해 어떤 메커니즘이 빌드되는지.

시스템 간의 관계

보통 시스템 개발 활동이 상호 연결된 시스템의 시스템에서 구현된 시스템에서도 적용될 수 있는지 알아봤습니다. 이것은 다른 시스템에 사용되는 방법과 상당히 다른 방법으로 이런 시스템을 처리할 필요가 없다는 것을 의미하므로 장점을 가지고 있습니다. 또한 다른 하위 시스템의 양식으로 구현에서 상위 시스템의 적당한 분할을 얻을 수 있습니다. 상호 연결된 시스템의 시스템에서 각각의 시스템은 자체 라이프사이클을 가집니다. 각 시스템이 다른 특성을 가질 수 있으므로 변형된 개발 프로세스를 사용하여 이를 생성할 수 있습니다. Rational Unified Process [2]에 따라 각 시스템에 대해 다른 개발 케이스를 가질 수 있습니다.

상호 연결된 시스템의 시스템에 포함된 시스템 간의 독립성에 대한 최종 노트.

먼저, 하위 시스템을 살펴보십시오. 이런 각 시스템은 상위 시스템의 설계 모델에서 하나의 서브시스템을 구현합니다. 서브시스템은 서로의 인터페이스에 종속되며 서로에 명시적으로 종속되지 않습니다. 그림 12 를 참조하십시오. 따라서 새 서브시스템이 동일한 인터페이스를 따르는 한 다른 시스템에 영향을 주지 않고 하나의 서브시스템을 새 버전의 서브시스템으로 교환할 수 있습니다. 하위 시스템 간에 정확하게 동일한 관계를 얻습니다. 각 하위 시스템은 그 환경을 하나의 인터페이스 세트로 봅니다. 이것은 새 시스템이 다른 시스템과 동일한 역할을 수행하는 한, 예를 들어 동일한 인터페이스 세트로 표시될 수 있는 한, 시스템을 다른 시스템으로 교환할 수 있습니다. 시스템은 상위 모델의 서브시스템과 인터페이스 간 관계를 대응시켜 지정된 대로 서로의 인터페이스를 참조합니다.

하위 시스템의 유스 케이스 모델에서, 상호 작용하는 다른 하위 시스템의 인터페이스는 액터로 표시됩니다. 대응하는 액터에 의해 제공된 대로 하위 시스템이 다른 시스템의 인터페이스를 관찰하므로 다른 시스템을 직접 참조할 필요가 없다고 말할 수 있습니다. 그림 16 을 참조하십시오. 인터페이스 B 가 그림 12 의 여러 곳에 존재하여 이것이 실제로 상위 시스템의 서브시스템 및 대응하는 하위 시스템에서 참조되는 동일한 인터페이스임을 표시하는 것에 주의하십시오.

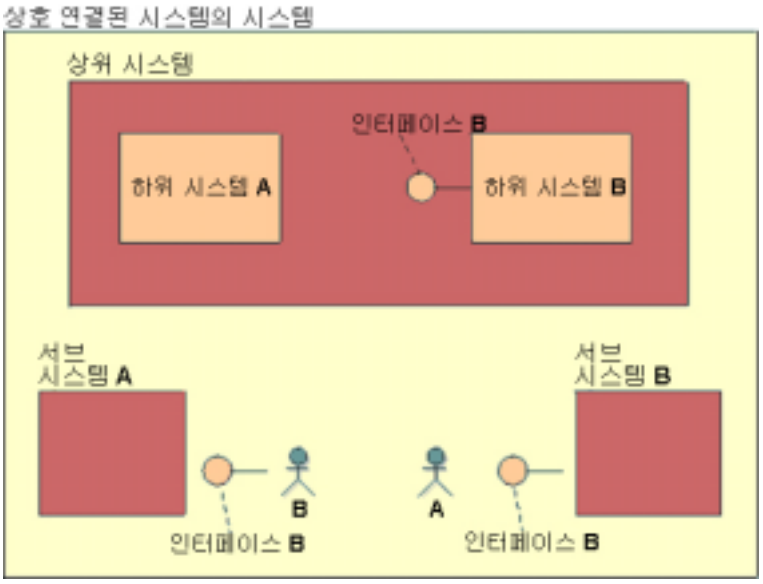

그림 16. 상위 시스템의 서브시스템은 이의 인터페이스를 통해서만 서로 종속됩니다. 따라서 하위 시스템을 구현하는 것은 같은 유형의 독립을 얻는 것입니다. 상위 시스템의 모델에서 서브시스템 B 는 인터페이스 B 를 다른 서브시스템에 제공합니다. 따라서 대응하는 하위 시스템 B 는 동일한 인터페이스 B 를 다른 하위 시스템에 제공해야 합니다.

상위 시스템은 어떻습니까, 하위 시스템과의 관계는 무엇입니까? 다음과 의미에서 시스템 구현은 독립적입니다. 이런 각 시스템은 상위 시스템의 모델에 지정한 것만을 구현한 것입니다. 이것은 스펙의 일부가 아닙니다. 실용적인 이유 때문에 여러 레벨에 있는 시스템 간의 추적성 링크를 정의하여 요구사항을 추적해야 하며 이를 수행하기에 가장 "좋은" 방법은 인터페이스 간의 링크만 정의하는 것입니다. 사실, 하위 시스템이 상위 모델에 정의된 인터페이스를 제공하는 구현 이상의 아무 것도 아니라고까지 말할 수 있습니다.

그러나 이것은 충분하지 않은 단순한 예 이상인 시스템에 대한 것입니다. 인터페이스는 특정 상호 작용 지점에서 계속되는 것 이외의 것을 지정하지 않습니다. 하위 시스템은 수백 개의 인터페이스를 포함할 수 있으며 각 인터페이스는 수십 개의 조작을 포함할 수 있습니다. 하나의 인터페이스에서의 입력을 다른 인터페이스에서의 하나 또는 다수의 출력에 연관시키는 것은 인터페이스 설명에서 수행하는 데 있어서 비실용적입니다. 이것은 하위 시스템의 의미를 설명하기 위해 유스 케이스가 필요한 이유입니다.

시스템이 상호 연결된 시스템의 시스템으로 구현될 때 연관된 각 시스템이 다른 시스템에 독립적이지만 서로의 인터페이스에는 강하게 종속되어 있다고 결론지을 수 있습니다 . 이것은 하위 시스템의 병렬 개발용으로 좋은 플랫폼을 제공합니다.

어플리케이션 영역

상호 연결된 시스템의 시스템에 대한 구조 및 모델링 설명은 다음과 같은 여러 시스템 유형에 사용될 수 있습니다.

- 분산 시스템
- 크거나 복잡한 시스템
- 여러 비즈니스 분야를 결합하는 시스템.
- 다른 시스템을 재사용하는 시스템
- 시스템의 분산 개발

상황이 반대로 될 수도 있습니다. 이미 존재하는 시스템에서, 시스템을 어셈블하여 상호 연결된 시스템의 시스템을 정의할 수도 있습니다. 사실, 일부 경우에 이것은 대규모 시스템이 더욱 초기 반복 단계에서 전개되는 방법입니다. 상호 연결될 수 있는 시스템이 있음을 알게 되고 이를 수행하여 두 개 이상의 개별 시스템을 추가하는 "대규모 시스템"을 작성합니다.

사실, 시스템의 여러 파트를 자체 시스템으로 볼 수 있는 모든 시스템의 경우 시스템을 상호 연결된 시스템의 시스템으로 정의하는 것이 타당합니다. 현재 시스템이 단일 시스템이더라도, 몇 가지 예에서 언급된 분산 개발, 재사용 이유 또는 시스템의 파트만을 구매하고자 하는 고객의 요구사항으로 인해 나중에 시스템을 여러 개별 제품으로 분리할 필요가 있는 것으로 판명될 수 있습니다.

결론적으로, 상호 연결된 시스템의 시스템에 대한 구조가 사용될 수 있는 몇 가지 케이스를 자세하게 검토합니다. 각각의 예에서 문제의 시스템이 단일 시스템 *및* 개별 시스템 세트 모두로 고려되어야 한다고 표시합니다. 이것은 해당 시스템이 상호 연결된 시스템의 시스템에서 구현되는 상위 시스템으로 간주되어야 함을 가리킵니다.

대규모 시스템

전화망이 세계에서 가장 큰 상호 연결된 시스템의 시스템일 것입니다. 이것은 복잡도를 관리하기 위해 둘 이상의 시스템 레벨이 필요한 곳의 아주 훌륭한 예입니다. 최상위 레벨 상위 시스템이 표준화 단체에 의해 소유되고 다른 경쟁 회사가 이 표준을 준수해야 하는 하나 이상의 하위 시스템을 개발하는 케이스의 예이기도 합니다. 여기서는 대규모 시스템을 상호 연결된 시스템의 시스템으로 구현하는 데서 오는 장점을 표시하기 위해 이동 전화 네트워크 GSM(Global System of Mobile Telephony)을 설명합니다.

일반적으로 큰 시스템의 기능은 여러 비즈니스 분야를 결합합니다. 예를 들어, GSM 표준은 전화를 거는 가입자에서 전화를 받는 가입자까지 전체 시스템에 적용됩니다. 다시 말해, 이동 전화 및 네트워크 노드의 작동 모두를 포함합니다. 여러 시스템 파트가 별도로 구매되는 독립된 제품이므로 고객 유형이 다르더라도 이런 파트는 독립적인 시스템으로 간주되어야 합니다. 예를 들어, 전체 GSM 시스템을 개발하는 회사는 가입자에게는

이동 전화를 판매하고 전화 조작원에게는 네트워크 노드를 판매합니다. 이것은 GMS 시스템의 다른 파트를 다른 하위 시스템으로 간주하는 한 가지 이유입니다. 또 다른 이유는 GSM 과 같은 대규모이거나 복잡한 시스템을 하나의 단일 시스템으로 개발하기에는 너무 많은 시간이 걸린다는 것입니다. 여러 파트가 몇몇 개발 팀에 의해 병렬적으로 개발되어야 합니다.

다른 한편, GSM 표준이 전체 시스템에 적용되므로 시스템을 전체적으로, 다시 말해 상위 시스템으로 고려해야 하는 이유도 있습니다. 이것은 개발자가 문제점 도메인 및 여러 파트가 서로 연관되는 방법에 대해 이해하는 데 도움울 줍니다.

분산 시스템

여러 컴퓨터 시스템에 걸쳐 분산된 시스템의 경우, 상호 연결된 시스템의 시스템 구조가 가장 적합합니다. 정의에 의하면, 분산 시스템은 항상 최소한 두 개의 파트로 이루어집니다. 잘 정의된 인터페이스가 분산 시스템에 필요하므로 이런 시스템은 분산 양식으로, 즉 병렬로 작업하는 여러 자치 개발 팀에 의해 *개발되는* 것이 가장 좋습니다. 분산 시스템의 하위 시스템은 독립된 제품으로도 판매될 수 있습니다. 따라서 분산 시스템을 독립된 제품의 세트로 간주하는 것은 당연한 것입니다.

분산 시스템의 요구사항은 일반적으로 전체 시스템의 기능을 포함하며 경우에 따라 시전 정의되지 않은 다른 파트 간의 인터페이스를 포함합니다. 더욱이, 문제점 도메인이 개발자에게 생소한 것이면 시스템이 분산되는 방법에 관계없이 전체 시스템의 기능성을 먼저 고려해야 합니다. 시스템을 단일 시스템으로 보는 가장 중요한 두 가지 이유가 있습니다.

레거시 시스템의 재사용

거의 대부분, 대규모 시스템은 레거시 시스템를 재사용합니다. 레거시는 하위 시스템으로 설명될 수 있습니다. 더 큰 상위 시스템 컨텍스트에서 작업할 수 있는 방법을 알기 위해 레거시 시스템에 대한 유스 케이스 모델 및 분석 모델을 "재설계"합니다. 이와 같이 재설계된 모델이 반드시 완료될 필요는 없습니다. 최소한 상호 연결된 시스템 중 나머지 시스템의 기능에 직접적인 영향을 주거나 수정을 필요로 할 수 있는 레거시의 기능을 포함해야 합니다.

사전 작성된 패키지 사용

시스템은 두 개 이상의 사전 작성된 패키지가 통합 및 사용자 정의된 것일 수 있습니다. ERP(Enterprise Resource Planning) 시스템이 좋은 예입니다. 대부분의 ERP 시스템은 MRP(Material Resource Planning), 재고 관리, 공급 체인 관리 등과 같은 하위 시스템의 조합입니다. 인사 및 급여 어플리케이션과 같은 다른 영역에서 유사한 조합을 사용할 수 있습니다. 이것은 전체 시스템을 얻기 위해 다른 표준 패키지를 특수화하고 상호 연결해야 하는 사전 작성된 패키지와 같습니다. 패키지 세트가 함께 무엇을 수행하는지 이해하려면 상위 시스템이 필요합니다. 이 케이스는 오늘날 대부분의 금융 업계 고객이 직면한 것입니다.

요약

이 문서는 상호 연결된 시스템의 시스템에 대한 구조적 패턴을 소개합니다. 이 구조는 하나의 모델 내에서만 반복을 허용합니다. 각 서브시스템을 자체 소유의 시스템으로 간주하며, 반복은 각 시스템의 모든 결과물 세트 간에 이루어집니다. 도입된 구조는 통신 중인 여러 시스템에 의해 구현되는 시스템용으로 사용됩니다. 연관된 각 시스템은 다른 시스템의 모델에서 분리되어 자체 모델 세트로 설명됩니다.

이 설명 사용의 장점은 명백합니다. 보다 복잡한 문제점에 접근할 수 있고 "분할 통치" 기술을 사용하여 이런 문제점을 이해할 수 있습니다. 그러나 결점은 더 많은 오버헤드, 비동기화된 스케줄의 위험입니다. 또한 예를 통해 조직이 상위 시스템에 반복적 라이프사이클을 사용하는 것이 매우 어렵다는 것을 알았습니다. 그로 인해 위험이 상위 시스템 라이프사이클의 끝 쪽으로 밀려나는 위험에 직면합니다. 또한 "스토브 파이프(stove-pipe)" 시스템 세트를 개발하지 않도록 효율적이고 타당한 재사용 전략이 준수되고 있는지 점검해야 합니다.

제공된 예는 상호 연결된 시스템의 시스템을 모델링하는 구조가 여러 다른 어플리케이션 영역에 유용하다는 것을 나타냅니다. 사실, 제시된 구조를 여러 파트를 독립된 시스템으로 볼 수 있는 모든 시스템에 사용할 수도 있습니다.

참조서

[1] Jacobson, I.; Palmkvist, K.; 및 Dyrhage, S., *Systems of Interconnected Systems*, ROAD, 2(1), 1995.

[2] Rational Unified Process 버전 5.1.

[3] Rumbaugh, J.; Booch, G.; Jacobson,I., *UML Reference Manual*, Addison Wesley Longman, 1999.

[4] Herbert A. Simon, *The Sciences of the Artificial*, MIT Press, 1981.

[5] Jacobson, I.; Bylund, S.; Jonsson, P., *Using Contracts and Use Cases to Build Plugable Architectures*, Journal of Object-Oriented Programming, May/June, 1995.

[6] Jacobson, J.; Griss, M.; Jonsson, P., Software Reuse - Architecture, Process and Organization for Business Success, Addison Wesley Longman, 1997.

[7] Jacobson, I., *Use Cases in Large-Scale Systems*, ROAD, 1(6), 1995.

Rational
the software development company

본사:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
전화번호: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
전화번호: (781) 676-2400

무료 전화번호: (800) 728-1212
전자 우편: info@rational.com
웹: www.rational.com
월드와이드: www.rational.com/worldwide