유스 케이스에 기반한 노력 예상

John Smith

Rational Software 백서

TP 171, 10/99

Rational®
the software development company

목차

문제점

직관적으로, 개발 시 필요한 크기 및 노력은 유스 케이스 모델의 특성에 기반하여 예상할 수 있어야 하는 것처럼 보입니다. 그러나 유스 케이스 모델은 기능적 요구사항을 캡처하므로 유스 케이스가 기능 점수를 기반으로 해야 하는 것은 아닌지 검토해 보아야 합니다. 여기에는 다음과 같은 몇 가지 어려움이 있습니다.

- 다양한 유스 케이스 스펙 스타일과 형식성으로 인해 메트릭(예를 들어, 유스 케이스의 길이를 측정할 수 있는 메트릭)을 정의하기가 매우 어렵습니다.
- 유스 케이스는 외부 액터의 시스템 보기를 나타내야 하므로 500,000 개의 소프트웨어 코드 행(SLOC)으로 구성되는 시스템의 유스 케이스는 5,000 개의 SLOC 로 구성되는 서브시스템에 대해 작성된 유스 케이스와 ***레벨***이 상당히 다릅니다(레벨 및 목적에 대한 개념은 Cockburn97 참조).
- 유스 케이스는 서면으로 명시적으로 작성되었거나 필수 실현(realization)에 내재적이거나 복잡도가 모두 다를 수 있습니다.
- 유스 케이스는 액터의 관점에서 동작을 설명해야 하지만 특히 시스템에 상태가 포함되는 경우(일반적인 경우) 해당 설명은 매우 복잡할 수 있습니다. 따라서 이 동작을 설명하기 위해서는 실현이 수행되기 전 시스템 모델이 필요할 수 있습니다. 이러한 경우 동작의 핵심을 캡처하기 위해 기능적 분해 및 세부사항의 레벨이 너무 많아질 수 있습니다.

그렇다면 예상을 가능하게 하기 위해 특정 유형의 유스 케이스 실현(realization)이 필요한지 여부를 확인해 보아야 합니다. 일반적으로 유스 케이스로부터 직접 예상하는 경우 기대가 너무 높으므로, 유스 케이스 점수의 개념과 기능 점수를 비교하는 것은 올바르지 않습니다. 또한 기능 점수를 계산하려면 시스템 모델이 필요합니다. 유스 케이스 설명과 기능 점수가 다른 경우 유스 케이스 표현에 있어 레벨이 일정해야 하며, 실현(realization)이 막 시작되는 시점에서만 기능 점수에 확신을 가질 수 있습니다. Fetcke97 에서는 유스 케이스에서 기능 점수로의 맵핑에 대해 설명하지만, 올바른 맵핑을 위해서는 유스 케이스의 레벨 역시 적절해야 합니다. 다른 메소드는 클래스/오브젝트 기반 메트릭을 소스로 사용합니다(예: PRICE 오브젝트 점수(Minkiewicz96)).

기타 작업

유스 케이스에 대한 설명 및 형식화와 관련된 많은 작업이 있습니다(Hurlbut97 의 조사 참조). 유스 케이스에서 예상 메트릭을 도출하는 데 대해서는 훨씬 적습니다. Graham95 및 Graham98 에서는 유스 케이스를 심하게 비판하고 있으며, 다양한 길이 및 복잡도와 같은 유스 케이스 관련 문제점을 극복하기 위한 방법으로 '타스크 스크립트'라는 아이디어를 제안합니다(그러나 저자는 Graham 이 자신의 아이디어와 유스 케이스가 다르다고 생각하는 이유를 정확히 이해하지 못합니다). Graham 의 '원자 타스크 스크립트'는 '타스크 점수' 메트릭 수집을 위한 기반입니다. 원자 타스크 스크립트 관련 문제점은 하위 레벨이라는 데 있습니다. Graham 에 따르면 이상적인 원자 타스크 스크립트는 단일 문장으로서, 해당 분야 용어만을 사용하여 더 이상 분해할 수 없습니다. Graham 의 '루트 타스크'에는 하나 이상의 원자 타스크 스크립트가 포함되며, 각 루트 타스크는 계획을 시작하는 클래스에서 정확히 하나의 시스템 오퍼레이션과 일치합니다(Graham98). 이러한 루트 타스크는 하위 레벨 유스 케이스와 매우 유사하며 원자 타스크 스크립트는 해당 유스 케이스의 단계와 유사하다고 볼 수 있습니다. 그러나 레벨 문제점은 여전히 존재합니다.

Karner(Karner93), Major(Major98), Armour 및 Catherwood(Armour96)와 Thomson(Thomson94)은 다른 작업을 수행했습니다. Karner 문서에서는 유스 케이스 점수를 계산하는 방법을 가정하지만 역시 클래스로 실현 가능한 방법(예를 들어, 서브시스템보다 정교한 세부사항 레벨)으로 유스 케이스를 표현하는 것으로 가정합니다.

그렇다면 예상을 위해 유스 케이스를 사용하지 않고 대신 새 분석 및 디자인 실현(realization)에 의존해야 하는지 알아보아야 합니다. 이와 관련된 문제점은 예상을 작성하는 기능이 지연됨으로써 이 기술을 선택한 프로젝트 관리자를 만족시키지 못한다는 것입니다. 먼저 예상을 작성한 후 다른 방법을 사용해야 합니다. 프로젝트

관리자는 예상을 지연시키고 계획되지 않은 방식으로 진행하는 것보다 계획을 위해 빨리 예상을 얻은 다음 해당 예상을 반복 별로 ***정제***할 수 있어야 합니다.

이 문서에서 설명하는 내용은 모든 레벨의 유스 케이스를 사용하여 노력 예상을 작성할 수 있는 프레임워크입니다. 아이디어를 나타내기 위해 일부 간단한 표준 구조를 설명합니다. 이 구조에는 일부 경험을 기반으로 하는 연관된 차원 및 크기가 포함됩니다. 그러나 이 문서의 내용은 대부분 지나치거나 노골적인 추측 내용이 많습니다. 이 영역의 작업 및 데이터 부족을 고려할 때 발전적인 다른 방식을 발견할 수 없기 때문입니다. 저자는 공식화에서 '상호 연결된 복합 시스템' 아이디어를 제시했습니다.

다음은 이 내용의 일부 배경 지식을 규정하기 위해 약간 다른 주제에 대해 논의합니다.

기능 분해 방지

기능 분해 아이디어는 많은 소프트웨어 개발 종사자가 아주 싫어하는 것 같습니다. 또한 과도한 기능 분해에 대한 저자의 개인적인 경험에 의하면 어떠한 경우에도 낙관적인 생각을 가질 수 없습니다. 예를 들어, 깊이가 다섯 또는 여섯 레벨인 대규모 데이터 플로우 다이어그램에서 3,000 개의 기본요소 변환을 수행하는 경우 하부 구조 레벨을 제외하고는 아키텍처를 고려할 수 없었습니다. 이러한 경우의 문제점은 기능 분해뿐만 아니라, 기능 기본요소 레벨에 도달할 때까지 프로세스에 대해 설명하지 않는 아이디어와 관련된 문제점입니다. 기능 기본요소 레벨에서는 스펙의 길이가 1 페이지 미만이어야 합니다.

결과 또한 이해하기 어렵습니다. 상위 레벨에서 필요한 올바른 동작이 이러한 기본요소 변환에서 시작되는 과정을 식별하기 어렵습니다. 더불어, 기능 구조가 성능 및 기타 품질 요구사항을 충족시킬 실제 구조에 맵핑되는 방법도 명확하지 않습니다. 따라서 '문제점을 해결'할 수 있는 레벨(기본요소 레벨)에 도달할 때까지 계속 분해하면서도 관련 기본요소가 실제로 상위 레벨의 목적을 충족시켰는지 명확하지 않거나 이를 논증할 수 없는 모순이 발생합니다. 이 방법에서는 비기능적 요구사항을 고려할 수도 없습니다. 하부 구조(통신, 운영 체제 등)뿐만 아니라 전체 아키텍처가 분해 시 함께 발전해야 하며 각각은 서로 영향을 주어야 합니다.

Bauhaus 의 '형태가 기능을 따른다'는 경구도 고려해볼 수 있습니다. 디자인에 대한 기능 중심 접근 방식은 많은 장점이 있는 반면 동시에 항상 평평한 지붕을 사용하는 것과 같은 단점도 있습니다. 지붕의 기능만을 강조함으로써 그 집에 거주하는 사람들을 보호하기 위한 지붕의 기능에 디자인을 종속시키는 경우 적어도 특정 영역에 있어 불만족스러운 결과를 가져오게 됩니다. 이처럼 평평한 지붕은 눈이 그대로 쌓여 방수 기능도 떨어집니다.

이제 이러한 문제점을 해결할 수 있습니다. 그러나 다른 디자인을 선택한 경우보다 많은 비용을 지출해야 합니다. 따라서 진부하게 들릴 수도 있겠지만 모든 기능적, 비기능적 요구사항을 기반으로 양식을 작성해야 하며 비기능적 요구사항에 미적 요소가 포함될 수 있습니다. 설계자가 직면한 문제점은 일반적으로 비기능적 요구사항이 충분히 명시되지 않아 '올바른 방식'에 있어 설계자의 경험에 많이 의존할 수 밖에 없다는 점입니다. 따라서 아키텍처가 전적으로 기능적 분해에만 의존하는 경우 문제가 발생합니다. 분해가 여러 레벨을 진행하는 경우 기능 기본요소가 '모듈'과 일 대 일로 맵핑되며 해당 인터페이스를 정의합니다.

이러한 고려사항을 볼 때 아키텍처 작업 이전에 유스 케이스를 표준화된 레벨(클래스 협업으로 실현 가능)로 분해하는 방법은 적합하지 않다는 결론에 이를 수 있습니다. 이러한 분해는 특정 크기(Jacobson97 참조)의 시스템에서 분해를 위한 기준 및 엔지니어링 프로세스가 중요할 때 발생합니다. 이러한 경우 임시 기능 분해로는 충분하지 않습니다.

시스템 고려사항

시스템 엔지니어가 기능 분석, 분해 및 할당을 통해 디자인을 통합하지만 아키텍처에 있어 기능만이 유일한 구동 요소는 아닙니다. 또한 전문 엔지니어 팀이 대체 디자인 평가에 기여합니다. IEEE Std 1220(시스템 엔지니어링 프로세스의 적용 및 관리 표준)의 *6.3, Functional Analysis 섹션, 6.3.1 Functional Decomposition* 서브섹션과 *6.5 Synthesis* 섹션은 각각 기능 분해 사용과 시스템 제품 솔루션에 대해 설명합니다. 여기서 특히 *6.5.1 Group and Allocate Functions* 및 *6.5.2 Physical Solution Alternatives* 서브섹션의 내용에 관심을 기울일 필요가

있습니다. *6.3.1* 섹션에서는 시스템이 달성해야 하는 목적을 명확히 ***이해***하기 위해 분해를 수행하며 ***일반적으로 한 레벨의 분해만으로도 충분하다고 설명합니다.***

기능 분해의 목적은 통합과 같이 시스템을 구체화하는 것이 아닌 시스템이 수행해야 하는 기능을 이해하고 알리는 것입니다. 또한 이를 수행하기 위한 올바른 방법은 기능 모델입니다. 통합에서는 하위 기능이 솔루션 구조에 할당된 후 모든 기타 요구사항을 고려하여 솔루션을 평가합니다. 이 접근 방식과 멀티레벨 기능 분해의 차이점은 다음 레벨의 동작을 보다 세분화하여 정제하고 하위 레벨 컴포넌트에 할당해야 하는지 여부를 결정하기 전에 각 레벨에서 필수 동작을 설명하고 해당 동작을 구현하기 위한 솔루션을 모색한다는 것입니다.

이러한 경우 결과적으로 특정 레벨의 동작을 설명하기 위해 수백 개의 유스 케이스가 필요하지 않습니다. 설명한 시스템, 서브시스템, 클래스의 동작을 올바르게 나타내는 외부 유스 케이스(및 연관 시나리오)의 수 또한 매우 적습니다. 여기서 외부 유스 케이스에 대해 좀 더 설명하겠습니다. 서브시스템이 클래스로 구성되는 시스템을 예로 들 수 있습니다. 외부 유스 케이스는 시스템 및 해당 액터의 동작에 대해 설명하는 유스 케이스입니다. 서브시스템 또한 자체 유스 케이스를 가질 수 있으며 이러한 유스 케이스는 시스템에는 내부 유스 케이스이지만 서브시스템에는 외부 유스 케이스입니다. 궁극적으로 대규모 시스템(예를 들어, 1,000,000 개 이상의 코드 행으로 구성)을 구성하는 데 사용되는 ***총 유스 케이스(내부, 외부 모두 포함)의 수는 수백 개입니다. 이러한 크기의 시스템은 복합 시스템 또는 적어도 서브시스템의 복합 시스템으로 구성되기 때문입니다.***

구조 및 크기에 대한 가정

유스 케이스 수

Rational Software 에서는 일반적으로 유스 케이스의 수가 적어야 하며(예를 들어, 10 – 50 개) 유스 케이스가 많은 경우(예를 들어, 100 개 이상) 유스 케이스가 액터에 아무런 가치도 제공하지 않는 기능 분해가 필요한 상황이 전개될 수 있는 것으로 주장합니다. 그럼에도 불구하고 실제 프로젝트에 많은 유스 케이스가 사용되며 모든 유스 케이스가 '나쁜' 것도 아닙니다(혼합 레벨 처리). 예를 들어, Rational 내부 전자 우편의 경우 다음과 같은 Ericsson 의 예를 인용할 수 있습니다.

> 차세대 전화 스위치의 상당 부분을 모델링한 Ericsson 은 600 이상의 직원 시간(년)(최대 3 – 400 명의 개발자)과 200 개의 유스 케이스를 예상했습니다***(여러 유스 케이스 레벨을 사용하는 경우 "상호 연결된 복합 시스템"***을 참조하십시오).

직원 시간(년)이 600 이상인 시스템 1,500,000 개 이상의 C++ 코드 행으로 구성된 시스템의 경우서브시스템보다 한 레벨 상위 레벨에서 유스 케이스 분석이 중지될 수 있으며(즉, 서브시스템을 7,000 – 10,000 개의 코드 행으로 정의하는 경우) 그렇지 않은 경우 그 수가 더 많을 수 있습니다.

따라서 *외부* 유스 케이스 수가 적어야 한다는 주장을 고수할 수 있습니다. 이 문서에서 제안한 구조와 차원을 일치시키기 위해서는 각각 ***30 개의 연관 시나리오***[1] 를 갖는 ***열 개의 외부 유스 케이스가 동작을 설명하는 데 적합합니다.***[2] 실제 예제에서 유스 케이스 수가 열 개를 초과하고 유스 케이스가 해당 레벨에서 실제로 외부 유스 케이스인 경우, 설명하는 시스템은 해당 규정 양식보다 규모가 큽니다. 이러한 유스 케이스 수가 합당한 이유는 문서 후반부에서 설명합니다.

---

[1] UML1.3 에서의 **시나리오**는 동작을 나타내는 조치의 특정 시퀀스입니다. 시나리오는 유스 케이스 인스턴스의 실행 또는 상호작용을 나타내는 데 사용됩니다. 여기서는 두 번째 의미 즉, 유스 케이스 인스턴스의 실행을 나타내기 위해 사용됩니다.

[2] 이 시나리오 수는 유스 케이스의 복잡도를 나타내기 위한 것이며 개발자가 ***반드시*** 모든 유스 케이스에 30 개의 시나리오를 생성하고 작성해야 함을 나타내는 것은 아닙니다. 오히려 30 개의 시나리오는 유스 케이스를 통한 보다 많은 경로가 있는 경우라도 해당 유스 케이스의 관심 동작을 대부분 캡처합니다.

구조적 계층 구조

제안되는 구조적 계층 구조:

4 — 복합 시스템(system-of-systems)

3 — 시스템

2 — 시스템 그룹

1 — 서브시스템

0 — 클래스

클래스 및 서브시스템은 UML 로 정의됩니다. UML 에서는 더 큰 집계가 서브시스템(서브시스템 포함)입니다. 설명상 편의를 위해 이름을 다르게 지정했습니다. 2167 또는 498 과 같은 군사 표준 용어에 익숙한 경우 서브시스템 그룹 집계는 CSCI 와 크기가 같습니다. 이러한 용어에서는 서브시스템과 클래스를 각각 CSC 와 CSU 로 나타냅니다. 또한 Ada 구조가 맵핑되어야 하는 레벨에 대한 2167 일 간의 논쟁은 결국 Ada 패키지가 일반적으로 CSU 로 맵핑된 것으로 결론지어졌습니다. 시스템이 이 계층 구조를 엄격하게 준수해야 하는 것은 아니지만(레벨 간 혼합 가능) 이 계층 구조를 통해 유스 케이스당 노력에 대한 크기의 영향을 추론할 수 있습니다.

각 레벨마다 유스 케이스가 있지만(일반적으로 개별 클래스에 대한 유스 케이스는 아님) 정교한 단일 세부사항은 아니며 해당 레벨[3]의 각 컴포넌트(예: 서브시스템, 서브시스템 그룹 등)에 대한 유스 케이스입니다. 앞에서 각 레벨의 컴포넌트마다 열 개의 유스 케이스가 필요하다고 주장했습니다. 유스 케이스 설명이 평균 10 페이지인 경우 스펙의 길이는 잠재적으로 100 페이지가 됩니다. 여기에 비기능적 요구사항에 대해 이와 유사하거나 적은 분량이 더 추가됩니다. 이는 Stevens98 에서 선호하는 분량이며 Royce98 에서 제안되는 분량과도 유사합니다. 그렇다면 열 개의 유스 케이스를 선호하는 이유에 대해서도 생각해 보아야 합니다. 저자는 서브시스템당 클래스 수, 클래스 크기, 오퍼레이션 크기 등에 대한 올바른 값에 대한 저자 본인의 의견을 기반으로 상향식 추론 방법을 적용했습니다. 이러한 값은 다음 표에서 다른 가정과의 참조를 위해 표시됩니다.

오퍼레이션 크기	70SLOC
클래스당 오퍼레이션 수	12
서브시스템당 클래스 수	8
서브시스템 그룹당 서브시스템 수	8
시스템당 서브시스템 그룹 수	8
복합 시스템(system-of-systems)당 시스템 수	8
외부 유스 케이스 수(시스템, 서브시스템 등 기준)	10
유스 케이스당 시나리오 수	30
유스 케이스 설명당 페이지 수[4]	10

[3] 일부 검토자는 네 가지 레벨에서 유스 케이스 예상에 대해 경고를 나타냈지만 이는 일반적으로 규모가 매우 큰 복합 시스템에만 해당됩니다. 이러한 경우, 특히 주 계약자(복합 시스템), 하청업체(시스템) 및 심지어 하청업체의 하청업체(서브시스템)가 작업을 수행하는 경우 네 가지 레벨의 유스 케이스는 당연한 결과입니다.

[4] 문서 후반부에서는 다양한 시스템 클래스에 대해 정제됩니다.

경험적 데이터는 그다지 많지 않으며 텍스트 전체에 약간씩 분포되어 있습니다. Lorentz94 및 Henderson-Sellers96 에도 일부 데이터가 있고 저자의 경우 주로 오스트레일리아 군사 항공 우주 분야 프로젝트의 데이터를 갖고 있습니다. 어쨌든 이 단계에서는 올바른 프레임워크 위치를 지정하는 작업이 중요합니다.

계층 구조 내 컴포넌트 크기

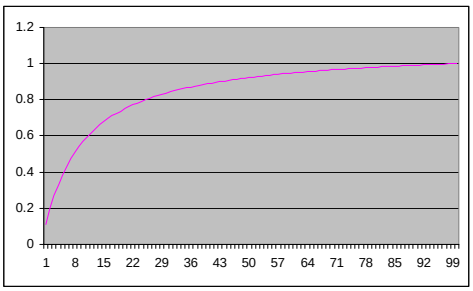
저자는 일부 사람들이 좋아하지 않는 방법인 코드 행을 사용했습니다. 즉, C++(또는 해당 레벨 언어) 코드 행을 사용하여 기능 점수에 역효과를 가져올 수 있습니다.

컨테이너의 클래스 수와 표현할 수 있는 동작의 다양성에는 특정 관계가 있습니다. 저자는 여덟 개의 클래스/서브시스템[5], 여덟 개의 서브시스템/서브시스템 그룹, 여덟 개의 서브시스템 그룹/시스템 등을 선택했습니다. 여기서 숫자 8 의 의미를 생각해 보아야 합니다.

- 7 을 기준으로 2 의 오차를 의미합니다.
- 클래스당 C++ 코드 행 수가 850 개인 경우(각각 코드 행 수가 70 개인 12 개 오퍼레이션) 서브시스템은 코드 행 수가 7000 개 이상이기 때문입니다. 이 크기는 4 – 9 개월 간 소규모 팀(예를 들어, 3 – 7 명의 인력)이 제공할 수 있는 기능성/코드의 양으로서, 코드 행 수가 300,000 – 1,000,000 개(RUP99)인 시스템의 반복 길이와 일치해야 합니다.[6]

다음은 여덟 개 클래스의 동작을 표현하는 외부 유스 케이스의 수와, 서로 결합되어 서브시스템에 함께 위치하는 유스 케이스를 확인해야 합니다. 이는 단순히 유스 케이스의 수뿐만 아니라 다양성을 결정하는 각 유스 케이스에 대한 시나리오 수이기도 합니다. 현재 시나리오/유스 케이스 확장과 관련한 가이드라인이 충분하지 않습니다. 이와 관련하여 Grady Booch 는 Booch98 에서 유스 케이스에서 시나리오에 이르는 확장 요소가 존재한다고 나타냅니다. 약간 복잡한 시스템의 경우 해당 동작을 캡처하는 여러 유스 케이스가 존재할 수 있으며 각 유스 케이스는 여러 시나리오로 확장될 수 있습니다. 이와 관련하여 Bruce Powel Douglass 는 Douglass99 에서 유스 케이스를 정확하게 설명하려면 많은 시나리오(일반적으로 12 개 이상)가 필요하다고 주장합니다. 저자는 중간 정도의 30 개 시나리오/유스 케이스를 선택했지만 Rechtin 은 Rechtin91 에서 엔지니어가 5 – 10 개의 상호 변수(저자의 경우 하나의 협업에서 5 – 10 개의 클래스로 해석)와 10 – 50 개의 상호작용(저자의 경우 시나리오로 해석)을 처리할 수 있다고 주장합니다. 이러한 방식의 해석에서는 복수 유스 케이스가 이 변수 공간의 복수 인스턴스입니다.

따라서 각각 30 개의 시나리오를 갖는 열 개의 유스 케이스(즉, 총 300 개의 시나리오)로도 여덟 개 클래스의 중요 동작을 충분히 다룰 수 있습니다. 또한 이러한 경우 나중에 300 개 이상의 테스트 케이스가 생성됩니다. 다른 곳에서도 이 숫자의 올바른 의미를 유추할 수 있습니다. 파레토의 80 대 20 규칙을 적용해보면 20%의 클래스가 80%의 기능을 제공하며 마찬가지로 80%의 기능이 각 클래스 내 20%의 오퍼레이션으로 제공됩니다. 여기서는 신중한 자세로 75%의 기능에 도달하기 위해 20%의 클래스가 필요한 것으로 가정하여 파레토 분포를 구성합니다(그림 1).



[5] 이 수는 분석을 나타냅니다. 즉, 디자 3 이상의 인수로 클래스 수가 증가하는 반면 오퍼레이션 크기 및 클래

[6] 반복 시간이 더 짧은 소규모 시스템의 경우 서브시스템 규모를 더 작게 계획하거나 각 반복에 대해 항상 부분 전달을 계획할 수 있습니다. 그러나 이러한 경우 신중한 제어가 필요하며 '스텁'을 전달해야 합니다.

그림 1: 유사 파레토 분포

전체 동작의 80%를 처리하려는 목표 아래 파레토 규칙을 클래스, 오퍼레이션 및 시나리오 수에 적용하는 경우, 각각 93%($0.93^3 = 0.8$)의 동작 범위가 필요합니다. 이러한 경우 각각 50%, 즉 네 개의 클래스 및 다섯 개의 오퍼레이션(= (12 – 2 생성자/소멸자)/2)이 필요합니다. 각각 다섯 개의 오퍼레이션을 갖는 네 개 클래스의 실행 패턴을 나타내기 위해 구성된 노드 트리의 다른 순회 수는 수천 개까지도 가능합니다. 저자는 최상위 오퍼레이션(인터페이스 오퍼레이션)이 열 개인 계층 구조를 가정하여 각 노드에서 최대 세 개의 링크를 포함하는 트리를 구성했습니다(세 가지 레벨의 트리 구성). 이러한 경우 약 1,000 개의 경로 또는 시나리오가 생성되며 500 개의 시나리오로 93%를 설명할 수 있습니다. 시나리오가 300 개인 경우(동일한 가정 사용) 약 73%를 설명할 수 있습니다. 중복되는 동작 스펙이 없도록 트리를 간단하게 작성하는 방법을 점검하는 과정에서, 선택한 알고리즘에 따라 더 적은 수도 가능한 것으로 제안합니다.

또 다른 접근 방식은 7000 개의 C++ 코드 행에 필요한 테스트 케이스(시나리오에서 파생) 수를 파악하는 것입니다. 이러한 테스트는 유닛 테스트 레벨 이상으로서, Jones91 및 Boeing 777 프로젝트(Pehrson96)에서 이 숫자가 안전하다는 일부 증거를 제공하며 사례를 제공합니다. 이 소스에서는 250 – 280 범위의[7] 시나리오를 제안합니다. 완전히 다른 레벨에서 볼 때, 캐나다 자동 항공 교통 시스템(CAATS) 프로젝트에서는 200 개의 시스템 테스트(개인용 커뮤니케이션)를 사용합니다.

유스 케이스 크기

올바른 유스 케이스 크기는 원하는 동작을 실현할 수 있을 정도의 세부사항을 나타낼 수 있어야 합니다. 이는 해당 복잡도, 내부 및 외부 유스 케이스에 따라 결정되며 시스템 유형과도 관련이 있습니다. 이와 관련하여 시스템 내부 조치를 어느 정도 설명해야 하는가에 대한 문제점을 해결해야 합니다. 외부 동작의 설명으로 시스템을 빌드하려면 출력과 입력이 서로 명확히 연관되어야 합니다. 예를 들어, 동작이 히스토리에 민감하거나 복잡한 경우 시스템 모델 내부에 대한 개념 모델과 해당 조치가 있어야 해당 동작을 설명할 수 있습니다. 그러나 이 방법으로 시스템 내부 구성을 설명하지 못할 수도 있으며 비기능적 요구사항을 충족시키고 모델 동작과 일치하는 디자인으로 설명할 수 있습니다.

UML1.3 에서 제공하는 **유스 케이스[클래스]**의 정의는 변형을 포함하여, 시스템 액터와의 상호작용을 통해 시스템(또는 기타 엔티티)이 수행할 수 있는 조치 시퀀스의 스펙입니다. 복잡한 동작의 경우, 일반 사용자와 관련이 적은 실현(realization) 단계까지 연기되지 않으면 이 정의에 내부 조치를 포함할 수 있습니다. 액터의 동작을 제한하는 비즈니스 규칙 또한 유스 케이스에 통합되어야 합니다. 예를 들어, ATM 시스템의 경우 은행에서는 계정 잔액에 관계 없이 1 회 거래 시 최대 인출 금액을 $500 이하로 제한하는 규칙이 있습니다.

이러한 유형의 해석을 적용하면 이벤트 설명의 유스 케이스 플로우는 2 – 20 페이지[8] 까지 가능합니다. 단순 동작으로 구성되는 단순 알고리즘의 시스템에는 긴 설명이 필요하지 않습니다. 단순 비즈니스 시스템의 길이는 일반적으로 2 – 10 페이지(평균 5 페이지)입니다. 복잡한 시스템의 경우, 비즈니스 및 과학 시스템은 6 –

[7] Rational 내부 검토자의 피드백에 따르면 대부분의 중요하지 않은 시스템에는 너무 많은 숫자라는 의견도 있습니다. 이러한 시스템의 경우 유스 케이스당 30 개 미만의 시나리오를 갖게 됩니다. 이 경우와, 사용 중에 발견된 결함 수와 테스트 케이스 수의 관계에 대한 추가 데이터를 확보하는 것도 흥미로운 일입니다.

[8] 이는 상한선을 엄격하게 적용하기 위한 것은 아닙니다. 유스 케이스 설명의 길이는 극단적인 경우가 발생할 가능성이 적은 통계 분포 유형을 따릅니다.

15 페이지(평균 9 페이지), 복잡한 명령 및 제어 시스템은 8 – 20 페이지(평균 12 페이지)입니다. 이 길이는 크기가 동일한 복합 시스템 유형과 노력의 비선형 관계를 반영한 것입니다. 그러나 길이의 근거가 되는 데이터는 없습니다. 보다 자세한 설명 양식, 상태 머신 또는 활동 다이어그램은 보다 적은 공간을 필요로 합니다. 아직까지는 텍스트를 강조하는 경향이 있으므로 현재로서 다른 방식은 무시합니다. 실제 해당 데이터도 거의 또는 전혀 없습니다.

이러한 크기와 구조적으로 다른 개발 프로젝트의 경우 이러한 방법에서 파생된 유스 케이스당 시간에 승수를 적용해야 합니다. 저자는 시스템 분류(단순 비즈니스, 복잡한 명령 및 제어 등)에 대한 필수 평균 크기/제안 평균 크기인 COCOMO 스타일 비용 구동 요소를 추가하도록 제안합니다.

유스 케이스 크기의 다른 측면은 시나리오 수입니다. 예를 들어, 5 페이지 길이의 유스 케이스의 경우 많은 경로를 허용하는 복잡한 구조를 가질 수 있습니다. 또한 시나리오 수를 예상해야 하며 30 개를 기준으로 한 시나리오 수 비율(유스 케이스당 시나리오 수)을 비용 구동 요소로 사용해야 합니다.

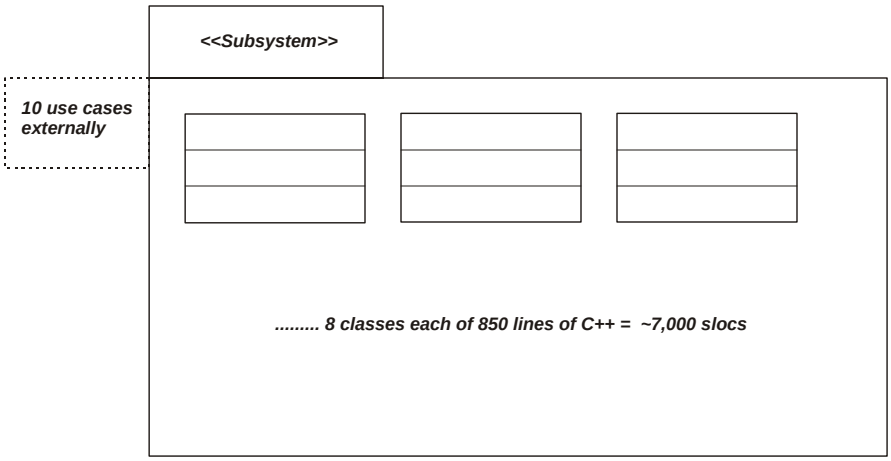
결론적으로, 보충 스펙 이외에, 해당 레벨에 관계 없이 외부 스펙에는 100 페이지의 유스 케이스 기반 스펙이 적합합니다. 해당 범위는 20 – 200 페이지입니다(이 한계는 명확하지 않습니다). 그러나 ***최하위 레벨 시스템(서브시스템 그룹)의 총계*** 는 3 – 15 페이지/KSLOC(단순 비즈니스 시스템) 또는 12 – 30 페이지/KSLOC(복잡한 명령 및 제어)입니다. 이는 아티팩트의 페이지 수가 매우 적은 Royce98 표 14 – 9 와, 특히 국방 분야에서 많은 문서를 생성한 실제 프로젝트 간의 명백한 대조를 설명하는 것처럼 보입니다. 이 문서는 반드시 서면(복잡한 대규모 시스템의 경우 200 페이지)으로 작성하지 않아도 되는 스펙 레벨에서 작성됩니다. Royce 의 주장대로 비전 설명과 같이 표에 표시된 순서가 중요합니다.

서브시스템 계층 구조

이제 서브시스템 계층 구조에 대해 생각해 봅니다. 먼저 저자가 사용해 본 단순 '표준' 양식을 살펴봅니다. 이러한 양식은 시스템을 실현하는 데 사용된 개념 양식입니다. 실제 시스템 경계는 이 양식 콜렉션의 범위에서 벗어납니다. 또한 각각에 대한 외부 유스 케이스의 총계는 시스템의 외부 유스 케이스의 총계입니다. 따라서 실제 시스템의 경우 외부 유스 케이스가 열 개보다 많지만 이 상한은 제한적이며 이 내용은 나중에 설명합니다. 여기서는 모든 개발 프로젝트에서 해당 설명에 네 가지 유스 케이스 레벨을 사용해야 하는 것으로 제안하지는 않습니다. SLOC 가 50,000 개 미만인 소규모 시스템의 경우 하나 또는 두 가지 레벨만 사용합니다.

레벨 1

레벨 1 에서는 하나 이상의 서브시스템의 클래스로 유스 케이스를 실현하거나 서브시스템이 없습니다.



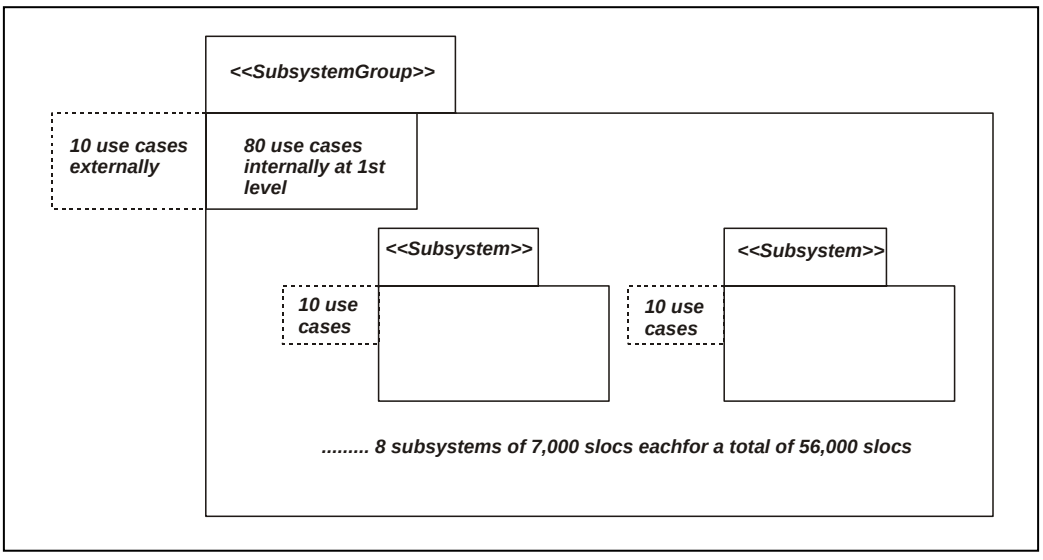
이 레벨의 시스템 크기 범위 예상(7 기준, 오차 범위 2 적용):

- 2 – 9 개 클래스(서브시스템을 구성하지 않음 —1700 – 8000 개 SLOC
- 다섯 개 클래스로 구성되는 하나의 서브시스템(총 4,000 개의 SLOC)
- 일곱 개 클래스로 구성되는 아홉 개의 서브시스템(총 53,550 개의 SLOC)

클래스 인스턴스로 실현 가능하도록 유스 케이스를 표현합니다. 유스 케이스 범위는 2 – 76 개입니다. 이 한계, 적어도 상한선은 명확하지 않습니다. 이 한계에서, 상위 레벨 양식으로 원하는 동작을 표현하지 않는 이러한 방식 또한 이러한 크기의 시스템 빌드 가능성은 0 으로 감소해야 합니다. 유스 케이스 수가 더 많은 경우 비정상적인 상태를 표시할 수 있습니다.

레벨 2

다음 레벨에서는 서브시스템 그룹이 여덟 개의 서브시스템으로 구성됩니다. 이는 방위 용어의 컴퓨터 시스템 형상 항목(CSCI)과 일치합니다. 이 레벨에서는 서브시스템의 협업으로 유스 케이스가 실현됩니다.



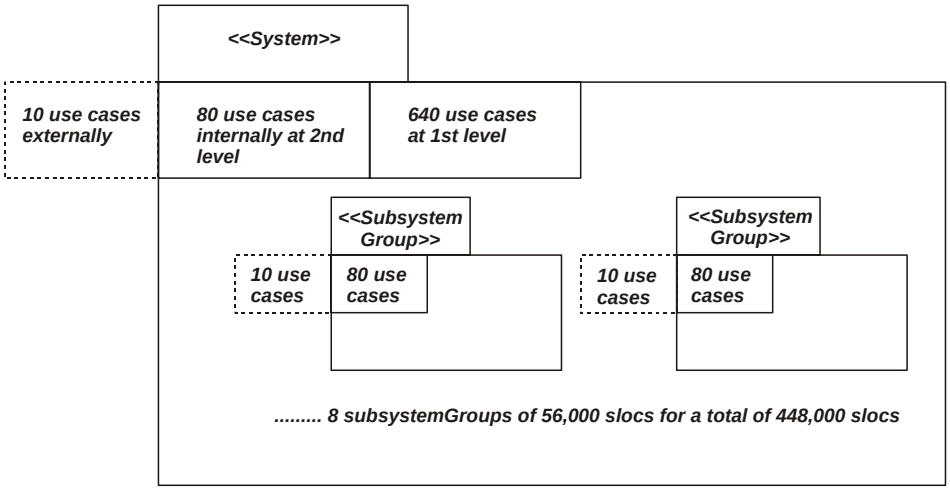
이 레벨의 시스템 크기 범위 예상(7 기준, 오차 범위 2 적용):

- 다섯 개의 클래스, 다섯 개의 서브시스템으로 구성되는 하나의 서브시스템 그룹(총 22,000 개의 SLOC)
- 각각 일곱 개의 클래스, 일곱 개의 서브시스템으로 구성되는 아홉 개의 서브시스템 그룹(총 370,000 개의 SLOC)

외부 유스 케이스 범위는 4 – 66 개입니다. 이 한계 역시 명확하지는 않습니다.

레벨 3

이 다음 레벨에서는 시스템이 서브시스템 그룹으로 구성됩니다. 레벨 3 에서는 서브시스템 그룹의 협업으로 유스 케이스가 실현됩니다.



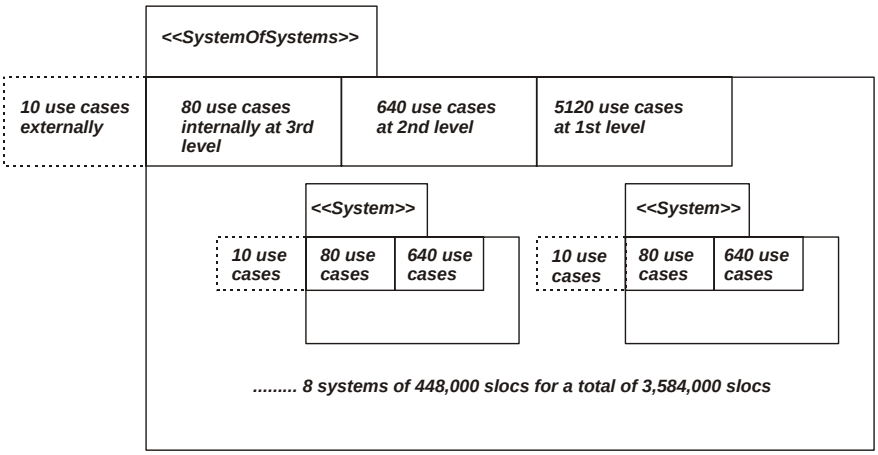
이 레벨의 시스템 크기 범위 예상(7 기준, 오차 범위 2 적용):

- 다섯 개의 클래스, 다섯 개의 서브시스템, 다섯 개의 서브시스템 그룹으로 구성되는 하나의 시스템
- 각각 일곱 개의 클래스, 일곱 개의 서브시스템, 일곱 개의 서브시스템 그룹으로 구성되는 아홉 개의 시스템(총 2,600,000 개의 SLOC)

외부 유스 케이스 범위는 3 – 58 개입니다. 이 한계 역시 명확하지는 않습니다.

레벨 4

다음 레벨은 복합 시스템입니다. 레벨 4 에서는 시스템의 협업으로 유스 케이스가 실현됩니다.



이 레벨의 시스템 크기 범위 예상(7 기준, 오차 범위 2 적용):

- 다섯 개의 클래스, 다섯 개의 서브시스템, 다섯 개의 서브시스템 그룹, 다섯 개의 시스템으로 구성되는 하나의 복합 시스템(총 540,000 개의 SLOC)
- 각각 일곱 개의 클래스, 일곱 개의 서브시스템, 일곱 개의 서브시스템 그룹, 일곱 개의 시스템으로 구성되는 아홉 개의 복합 시스템(총 18,000,000 개의 SLOC)

외부 유스 케이스 범위는 2 – 51 개입니다. 이 한계 역시 명확하지는 않습니다. 더 큰 집계도 가능하지만 여기서는 고려하지 않습니다.

유스 케이스당 노력

각 레벨에서 이러한 명목 크기에 대한 노력을 예상하여 유스 케이스당 노력을 파악할 수 있습니다. Estimate Professional 도구[9] (COCOMO 2[10] 및 Putnam 의 SLIM[11] 모델 참조)를 사용하여 언어를 C++(기타 명목으로 설정된 비용 구동 요소)로 설정하고 각 명목 크기 점수(열 개의 외부 유스 케이스 가정)에서 각 시스템 유형 예제에 대한 노력을 계산하면 표 1 의 결과가 산출됩니다.

크기(SLOC)	노력 시간/유스 케이스 단순 비즈니스 시스템	노력 시간/유스 케이스 과학 시스템	노력 시간/유스 케이스 복잡한 명령 및 제어 시스템
7000(L1)	55(범위 40 – 75)	120(범위 90 – 160)	260(범위 190 – 350)
56000(L2)	820(범위 710 – 950)	1700(범위 1500 – 2000)	3300(범위 2900 – 3900)
448000(L3)	12000	21000	38000
3584000(L4	148000	252000	432000

Boehm97 및 http://sunset.usc.edu/COCOMOII/cocomo.html을 참조하십시오.

[11] Putnam92 를 참조하십시오.

표 1: 다양한 샘플 유형에 대한 유스 케이스당 노력

표 1 에서 레벨 1(L1) 및 레벨 2(L2)에 대해 표시된 범위는 개별 유스 케이스의 복잡도(COCOMO 의 코드 복잡도 메트릭스에서 유추하여 예상함)를 고려합니다. L2 에서는 다양한 복잡도가 시스템 유형에 따른 특성에 반영되기 시작하여 상위 레벨의 복잡한 명령 및 제어 시스템 유스 케이스에 하위 레벨의 복잡도가 함께 포함됩니다. 이러한 복잡도를 로그-로그 배율에 적용함으로써 그림 2 와 같은 패턴이 생성됩니다.

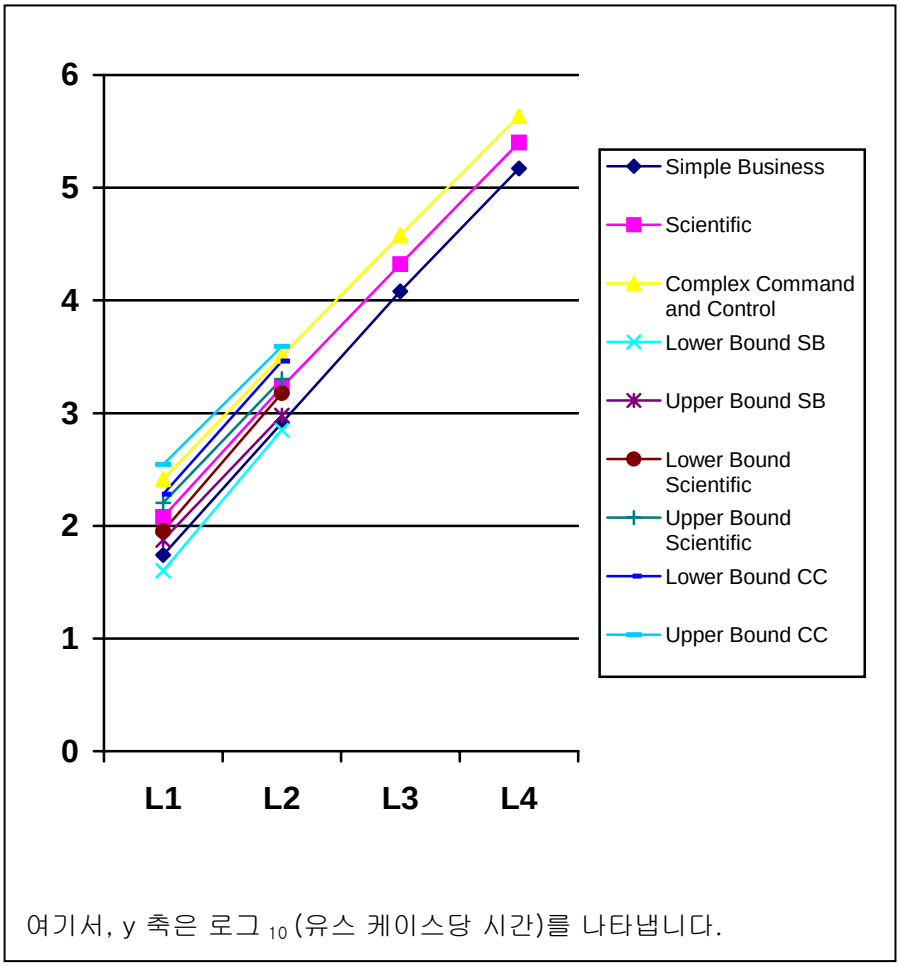


그림 2: 크기별 유스 케이스 노력

이 표를 통해, 이전 Objectory 수인 150 – 350 시간/유스 케이스($10^{2.17}$– $10^{2.54}$)가 L1 과 거의 일치함을 알 수 있습니다. 즉, 이 유스 케이스가 클래스 협업으로 실현할 수 있는 유스 케이스입니다. 따라서 이 수에 맞게 일부 조정됩니다. 그러나 분석 시 모든 프로젝트의 특성을 나타내는 데는 바람직하지 않습니다. 한 동료의 전자 우편 내용처럼 지나치게 '단조로운' 패턴이 됩니다.

노력 예상

실제 시스템이 이처럼 편리한 슬롯에 적용되지 않으므로, 시스템 특성을 나타내는 방법을 추론하기 위해 함께 도출된 불확실한 한계를 사용하여 그림 3 과 같이 나타낼 수 있습니다.

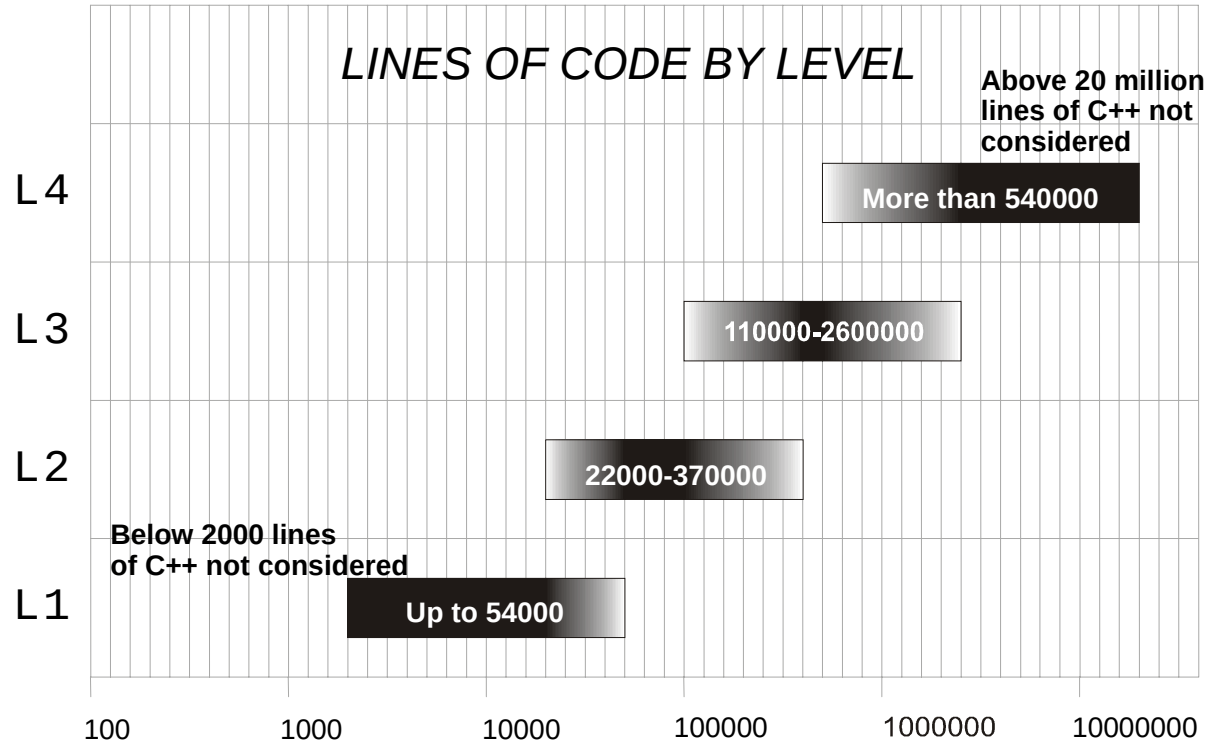


그림 3: 각 레벨별 크기 범위

그림 3 에서 볼 때, 최대 22,000 개 SLOC 로 구성된 시스템을 레벨 1 에서 설명할 수 있습니다(유스 케이스 수가 2 – 30 개). 이 크기에서 유스 케이스 수가 많은 경우 유스 케이스가 지나치게 세분화되어 있음을 나타냅니다.

22,000 – 54,000 SLOC 의 경우, 레벨 1 과 레벨 2 의 유스 케이스가 혼합되어 있을 수 있으며 유스 케이스 수의 범위는 4(모두 레벨 2) – 76(모두 레벨 1)개입니다. 차트에서 알 수 있듯이 이러한 극한 값은 발생 가능성이 낮습니다.

54000 – 110000 SLOC 의 경우, 올바른 구조의 시스템 전체를 레벨 2 에서 설명할 수 있습니다(10 – 20 개의 유스 케이스). 또한 L1/L2/L3 이 혼합되어 있을 수 있습니다. 유스 케이스 수가 160 개인 극단적인 경우가 발생할 확률은 ***매우*** 낮습니다.

110000 – 370000 SLOC 의 경우, 레벨 2 와 레벨 3 이 혼합될 수 있으며 유스 케이스 수는 3(모두 레벨 3) – 66(모두 레벨 2)개입니다.

370000 – 540000 SLOC 의 경우, 모두 레벨 3 에서 설명하는 경우 유스 케이스 수는 9 – 12 개입니다. 또한 L2/L3/L4 가 혼합되어 있을 수 있습니다. 유스 케이스 수가 100 개인 극단적인 경우가 발생할 확률은 ***매우*** 낮습니다.

540000 – 2600000 SLOC 의 경우, 레벨 3 과 레벨 4 가 혼합될 수 있으며 유스 케이스 수는 2(모두 레벨 4) – 50(모두 레벨 3)개입니다.

2600000 SLOC 이상인 경우 레벨 4 의 유스 케이스 수는 8 개 이상입니다.

충분한 유스 케이스 수

여기서는 일부 경험에 의한 규칙을 지원하는 흥미로운 관찰 내용을 확인할 수 있습니다. "어느 정도의 유스 케이스가 지나치게 많은 것입니까?"라는 질문이 자주 제기됩니다. 이 질문은 일반적으로 ***요구사항을 캡처할 때 어느 정도가 지나치게 많은 것인가를 의미합니다.*** 이에 대한 응답은 최대 규모의 시스템인 경우라도 70 개 이상인 경우이며 이러한 경우 디자인 이전에 너무 세분화된 상태임을 나타냅니다. 반면에 5 – 40 개 정도가 적합하지만 레벨을 고려하지 않은 숫자 자체로 크기 및 노력을 예상하는 데 사용할 수는 없습니다. 이 숫자는 특정 레벨에 적합한 ***초기*** 숫자입니다. 대규모 상위 시스템이 여러 시스템, 서브시스템 순으로 분해되는 경우 유스 케이스 수는 수백 개로 늘어납니다. 클래스 레벨에 도달할 때까지 유스 케이스가 개발된 경우, 최종 수는 수백 또는 수천 개까지 늘어날 수 있습니다(예를 들어, 직원 시간(년)이 140 인 프로젝트의 경우 유스 케이스 수는 600 개 이상이며 유스 케이스당 기능 점수는 약 15 점입니다). 그러나 이러한 경우는 디자인에 관계 없이 순수 유스 케이스 분해로서는 발생하지 않습니다. 이러한 유스 케이스는 Jacobson97 에서 설명하는 프로세스에서 발생합니다. 이 프로세스에서는 시스템 레벨의 유스 케이스가 서브시스템에 할당된 동작으로 분할되며 해당 서브시스템의 하위 레벨에 대해 유스 케이스를 작성할 수 있습니다(다른 서브시스템을 액터로 사용).

노력 예상 프로시저

예상을 작성하기 위해 수행하는 데는 몇 가지 전제조건이 필요합니다. 즉, 유스 케이스를 기반으로 한 예상은 문제 도메인을 이해하지 못하고, ***제안된 시스템 크기와 예상이 작성되는 단계에 적절한 아키텍처에 대해 알지 못하는 경우 작성할 수 없습니다.***

개략적인 최초 예상은 전문가의 의견을 이용하거나 Wideband Delphi 기법(1948 년 Rand Organization 에서 개발, 자세한 설명은 Boehm81 참조)을 통해 보다 공식적으로 수행할 수 있습니다. 이 기법에서는 평가자가 그림 3 의 크기 영역 중 하나에 시스템을 적용할 수 있으며 이를 통해 유스 케이스 수의 범위가 제안됩니다. 또한 표시 레벨(L1, L1/L2 등)을 나타냅니다. 평가자는 현재 아키텍처에 대한 지식, 해당 도메인 용어, 유스 케이스가 특정 레벨에 잘 맞는지, 정확하게 구분되는지 또는 레벨이 혼합되는지 여부를 기반으로 이벤트 플로우가 표현되는 방식으로 결정해야 합니다.

이러한 고려사항을 바탕으로, 데이터를 수정할 수 있는지 여부를 명확히 파악할 수 있어야 합니다. 예를 들어, Delphi 예상값이 600,000 SLOC(또는 해당 기능 점수)이고 아키텍처 작업이 거의 없는 경우, 시스템 구조에 대해 알려져 있는 사항이 많지 않으므로 그림 3 에서는 필요한 유스 케이스 수를 2(모두 레벨 4) – 14(모두 레벨 3)개로 제안합니다. 실제 유스 케이스 수가 100 개인 경우, 해당 유스 케이스가 너무 일찍 분해되었거나 Delphi 예상값이 틀린 경우입니다.

역시 이 예제에서, 실제 유스 케이스 수가 20 개이고 평가자가 이 유스 케이스를 모두 L3 으로 결정하고 유스 케이스의 평균 길이가 7 페이지이며 시스템이 복잡한 비즈니스 유형인 경우 유스 케이스당 시간(그림 2 참조)은 20,000 입니다. 이 값에 7/9 를 곱하면 유스 케이스 길이를 기반으로 명백히 낮은 복잡도를 설명할 수 있습니다. 따라서 이러한 경우 총 노력은 20 * 20000 * (7 / 9) = 약 310,000 직원 시간(시간) 또는 2050 직원 시간(월)이 됩니다. Estimate Professional 에 따르면 복잡한 비즈니스 시스템의 600,000 개의 C++ 코드 행 수를 처리하려면 1928 직원 시간(월)이 필요합니다. 따라서 이 가상 예제에서는 올바른 합의점을 찾을 수 있습니다.

실제 유스 케이스 수가 다섯 개이고 평가자가 유스 케이스를 L4 와 L3 에 각각 하나와 네 개씩 분할하고 L4 유스 케이스 길이를 12 페이지로, L3 유스 케이스의 평균 길이를 10 페이지로 결정하는 경우 해당 노력은 1 * 250,000

* 12 / 9 + 4 * 21000 * (10 / 9) = 약 2800 직원 시간(월)이 됩니다. 이는 주요 시스템 부분을 상위 레벨에서만 이해한다는 것을 가정하더라도 오류 범위가 너무 커 Delphi 예상값을 재검토해야 함을 의미합니다.

원래 Delphi 예상값이 100,000 개의 C++ 코드 행 수인 경우 그림 3 에서 알 수 있는 것은 유스 케이스 중 약 18 개가 L2 에 속해야 한다는 것입니다. 첫 번째 예제와 같이 실제 유스 케이스 수가 20 개인 경우 실제 유스 케이스 레벨을 고려하지 않고 메소드를 적용하고 Delphi 예상값이 크게 잘못된 경우 잘못된 결과가 나타납니다.

따라서 평가자는 유스 케이스가 실제로 제안 추상 레벨(L2)에 속하며 서브시스템 협업으로 실현할 수 있는지 또한 유스 케이스가 실제로 모두 L3 에 속하지 않는지 확인해야 합니다. 그러나 Wideband Delphi 기법이 항상 잘못된 결과를 제공하는 것은 아닙니다(예를 들어, 실제값이 600,000 에 가까운 경우에는 예상값 100,000 을 나타냄). 중요한 점은 유스 케이스 모델에 맞는 개념 아키텍처를 구현하지 않으면 이 예상 기법을 확실하게 적용할 수 없다는 것입니다. 해당 분야의 풍부한 경험을 갖고 있는 평가자의 경우 이 모델을 통해 레벨을 판단할 수 있지만 경험이 부족한 평가자와 팀의 경우 아키텍처 모델링을 통해 특정 레벨에서의 유스 케이스 실현 방법을 확인할 수 있습니다.

혼합 표현 유스 케이스의 수(즉, 레벨 N 과 레벨 N+1 의 혼합)는 하한 유스 케이스 유형의 $n=8^{(\text{두 레벨 간 차이})}$로 계산되어야 합니다. 따라서 L1 에서 50%, L2 에서 50%로 평가되는 유스 케이스는 $8^{0.5}$ = 3 L1 유스 케이스로 계산되어야 전체 수를 얻을 수 있습니다. L2 와 L3 사이에 30%로 평가되는 유스 케이스는 $8^{0.3}$ L2 유스 케이스 = 2 L2 유스 케이스로 계산되어야 합니다. L2 와 L3 사이에 90%로 평가되는 유스 케이스는 $8^{0.9}$ = 7 L2 유스 케이스로 계산되어야 합니다.

표 크기 조정

전체 크기를 고려하기 위해 개별 시간/유스 케이스 숫자를 실제로 더 수정해야 합니다. 노력 숫자는 ***해당 크기의 시스템 관점에서 각 레벨에 적절해야 합니다.*** 따라서 표 1 의 L1 에서 유스 케이스당 55 시간은 7000 SLOC 시스템을 빌드할 때 적용됩니다. 실제 수는 총 시스템 크기에 따라 결정됩니다. 따라서 빌드할 시스템의 크기가 예를 들어, 40,000 SLOC 이고 시스템을 설명하는 유스 케이스가 레벨 1 의 57 개인 경우, 해당 노력은 단순한 비즈니스 시스템에 대한 55 * 57 시간이 아닌 $(40/7)^{0.11}$ * 55 = 66 시간/유스 케이스입니다. 이는 크기와 노력에 대한 COCOMO 2 관계를 기반으로 합니다. COCOMO 모델에 따르면 노력 = A * $(\text{크기})^{1.11}$ 입니다. 여기서,

- 크기는 ksloc 입니다.
- A 에는 포함된 비용 구동 요소가 있습니다.
- 프로젝트 배율은 명목 배율입니다(지수는 1.11).

이러한 계산은 Estimate Professional 과 같은 도구에 적용하여 계산에 따른 부담을 줄일 수 있습니다. 여기서는 정확성을 위해 표시됩니다.

따라서 ksloc 당 또는 단위당 노력은 A* $(\text{크기})^{1.11}$/크기가 되며 A* $(\text{크기})^{0.11}$ 입니다. 또한 크기 S2 의 노력/단위에 대한 크기 S1 의 노력/단위의 비율은 $(S1/S2)^{0.11}$ 입니다.

Delphi 예상 이외에, 다양한 레벨의 유스 케이스 수에서 개략적으로 시스템 크기를 계산할 수 있습니다. 예를 들어, 레벨 1 의 유스 케이스가 N1 이고, 레벨 2 의 N2, 레벨 3 의 N3, 레벨 4 의 N4 인 경우, 총 크기는 [(N1 / 10) * 7 + (N2 / 10) * 56 + (N3 / 10) * 448 + (N4 / 10) * 3584]ksloc 입니다. 또한 이 총 크기를 표 1 의 열 1 에 표시된 각 레벨 크기(ksloc)로 나누어 표 1 의 유스 케이스 숫자당 각 노력에 대한 노력 승수를 계산할 수 있습니다.

- 따라서 레벨 1 의 결과는 다음과 같습니다. $(0.1 * N1 + 0.8 * N2 + 6.4 * N3 + 51.2 * N4)^{0.11}$
- 레벨 2 의 결과는 다음과 같습니다. $(0.0125 * N1 + 0.1 * N2 + 0.8 * N3 + 6.4 * N4)^{0.11}$

- 레벨 3 의 결과는 다음과 같습니다. $(0.00156 * N1 + 0.0125 * N2 + 0.1 * N3 + 0.8 * N4)^{0.11}$
- 레벨 4 의 결과는 다음과 같습니다. $(0.00002 * N1 + 0.00156 * N2 + 0.0125 * N3 + 0.1 * N4)^{0.11}$

예를 들어, 레벨 4 에서 레벨 1 유스 케이스의 수는 레벨 3 또는 레벨 4 와 비교하여 그 영향이 적습니다.

요약

유스 케이스를 기반으로 한 예상 프레임워크가 제시되었습니다. 보다 정확한 프리젠테이션을 위해 전반적으로 오류가 없는 것으로 알려져 있는 프레임워크 매개변수에 대한 값이 선택되었습니다. 또한 이러한 예상값은 실제값과 비교하여 테스트를 마쳐야 하며 데이터 수집에 따라 매개변수를 다시 평가해야 합니다. 이 프레임워크는 다양한 시스템 카테고리에 대한 유스 케이스 레벨, 크기 및 복잡도 아이디어를 고려하며 세분화된 기능 분해에 의존하지 않습니다. 또한 보다 간편한 계산을 위해, 유스 케이스를 기반으로 대체 크기 입력 방법을 제공하는 Estimate Professional 과 같은 도구의 프론트 엔드를 구성할 수 있습니다.

이 백서에 대한 의견 및 피드백은 John Smith(jsmith@rational.com)에게 보내주시기 바랍니다.

참조

1. Armour96: Experiences Measuring Object Oriented System Size with Use Cases, F. Armour, B. Catherwood, et al., Proc. ESCOM, Wilmslow, UK, 1996
2. Boehm81: Software Engineering Economics, Barry W. Boehm, Prentice-Hall, 1981
3. Booch98: The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, 1998
4. Cockburn97: Structuring Use Cases with Goals, Alistair Cockburn, Journal of Object-Oriented Programming, Sept-Oct 1997 and Nov-Dec 1997
5. Douglass99: Doing Hard Time, Bruce Powel Douglass, Addison Wesley, 1999
6. Fetcke97: Mapping the OO-Jacobson Approach into Function Point Analysis, T. Fetcke, A. Abran, et al., Proc. TOOLS USA 97, Santa Barbara, California, 1997
7. Graham95: Migrating to Object Technology, Ian Graham, Addison-Wesley, 1995
8. Graham98: Requirements Engineering and Rapid Development, Ian Graham, Addison-Wesley, 1998
9. Henderson-Sellers96: Object-Oriented Metrics, Brian Henderson-Sellers, Prentice Hall, 1996
10. Hurlbut97: A Survey of Approaches For Describing and Formalizing Use Cases, Russell R. Hurlbut, Technical Report: XPT-TR-97-03, http://www.iit.edu/~rhurlbut/xpt-tr-97-03.pdf
11. Jacobson97: Software Reuse - Architecture, Process and Organization for Business Success, Ivar Jacobson, Martin Griss, Patrik Jonsson, Addison-Wesley/ACM Press, 1997
12. Jones91: Applied Software Measurement, Capers Jones, McGraw-Hill, 1991
13. Karner93: Use Case Points - Resource Estimation for Objectory Projects, Gustav Karner, Objective Systems SF AB(Rational Software 저작권 소유), 1993
14. Lorentz94: Object-Oriented Software Metrics, Mark Lorentz, Jeff Kidd, Prentice Hall, 1994
15. Major98: A Qualitative Analysis of Two Requirements Capturing Techniques for Estimating the Size of Object-Oriented Software Projects, Melissa Major and John D. McGregor, Dept. of Computer Science Technical Report 98-002, Clemson University, 1998
16. Minkiewicz96: Estimating Size for Object-Oriented Software, Arlene F. Minkiewicz, http://www.pricesystems.com/foresight/arlepops.htm, 1996
17. Pehrson96: Software Development for the Boeing 777, Ron J. Pehrson, CrossTalk, January 1996
18. Putnam92: Measures for Excellence, Lawrence H. Putnam, Ware Myers, Yourdon Press, 1992
19. Rechtin91: Systems Architecting, Creating & Building Complex Systems, E. Rechtin, Prentice-Hall, 1991
20. Royce98: Software Project Management, Walker Royce, Addison Wesley, 1998
21. RUP99: Rational Unified Process, Rational Software, 1999
22. Stevens98: Systems Engineering - Coping with Complexity, R. Stevens, P. Brook, et al., Prentice Hall, 1998

23. Thomson94: Project Estimation Using an Adaptation of Function Points and Use Cases for OO Projects, N. Thomson, R. Johnson, et al., Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA ’94, 1994

Rational
the software development company

본사 안내:

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

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

수신자 부담 전화번호: (800) 728-1212
전자 우편: info@rational.com
웹: www.rational.com
전세계 지사 안내: www.rational.com/worldwide