제3장

모듈성

건축가와 개발자들은 모듈성이라는 개념을 이해하는 데 꽤 오랜 시간을 허비해 왔는데, 이는 『복합/구조 설계』(Van Nos! trand Reinhold, 1978)에 나오는 다음 인용문에서 분명히 드러납니다.

> 소프트웨어 아키텍처에 대해 쓰여진 글의 95%는 "모듈성"의 장점을 찬양하는 데 할애되고, 그것을 어떻게 달성하는지에 대해서는 거의 언급되지 않습니다.
>
> —글렌포드 J. 마이어스

플랫폼마다 코드 재사용 메커니즘은 다르지만, 모두 관련 코드를 모듈로 묶는 방식을 지원합니다. 이러한 모듈화 개념은 소프트웨어 아키텍처에서 보편적이지만, 명확하게 정의하기는 어렵습니다. 인터넷 검색만 해봐도 수십 가지 정의가 나오지만, 일관성이 없고 심지어 모순되는 경우도 있습니다. 이는 새로운 문제가 아닙니다. 하지만 공식적으로 인정된 정의가 없기 때문에, 이 책 전체의 일관성을 유지하기 위해 우리 스스로 정의를 내리고자 합니다.

아키텍트에게 있어 모듈성과 개발 플랫폼에 따른 다양한 구현 방식을 이해하는 것은 매우 중요합니다. 아키텍처 분석 도구(메트릭, 적합도 함수, 시각화 등)의 상당수는 모듈성 및 관련 개념에 기반합니다. 모듈성은 핵심적인 구성 원칙입니다. 아키텍트가 시스템 구성 요소들이 어떻게 연결되는지 고려하지 않고 시스템을 설계한다면, 그 시스템은 수많은 문제점을 드러낼 것입니다. 물리학에 비유하자면, 소프트웨어 시스템은 복잡한 시스템을 모델링하는데, 이러한 시스템은 무질서(엔트로피)로 치닫는 경향이 있습니다. 물리 시스템에서는 질서를 유지하기 위해 에너지를 지속적으로 공급해야 합니다. 소프트웨어 시스템도 마찬가지입니다. 아키텍트는 구조적 안정성을 확보하기 위해 끊임없이 노력해야 하며, 이는 우연히 이루어지는 것이 아닙니다.

모듈성을 잘 유지하는 것은 암묵적인 아키텍처 특성에 대한 우리의 정의를 잘 보여주는 예입니다. 사실상 프로젝트 요구사항 중 어느 것도 아키텍트에게 모듈성을 보장하도록 명시적으로 요구하는 경우는 없습니다.

모듈별 구분과 소통은 훌륭하지만, 지속 가능한 코드베이스를 위해서는 이러한 구분과 소통이 가져다주는 질서와 일관성이 필수적입니다.

모듈성 대 세분화

개발자와 아키텍트는 모듈성(modularity)과 세분성(granularity)이라는 용어를 종종 혼용하지만, 그 의미는 매우 다릅니다. 모듈성은 시스템을 더 작은 조각으로 나누는 것을 의미하며, 예를 들어 전통적인 n계층 아키텍처와 같은 모놀리식 아키텍처에서 마이크로서비스와 같은 고도로 분산된 아키텍처로 전환하는 것을 말합니다. 반면 세분성은 이러한 조각들의 크기, 즉 시스템(또는 서비스)의 특정 부분이 얼마나 커야 하는지를 나타냅니다. 그러나 다음은 필자 중 한 명이 언급한 것처럼, 아키텍트와 개발자가 어려움을 겪는 부분이 바로 세분성입니다.

> 모듈화를 수용하되, 세부적인 사항에는 주의해야 합니다.
>
> —마크 리처즈

세분화된 구조는 서비스나 컴포넌트 간의 결합을 심화시켜 스파게티 아키텍처, 분산 모놀리스, 그리고 악명 높은 '빅 볼 오브 디스트리뷰티드 머드'와 같은 복잡하고 유지보수가 어려운 아키텍처 안티패턴을 초래합니다. 이러한 아키텍처 안티패턴을 피하는 비결은 세분화 정도와 서비스 및 컴포넌트 간의 전반적인 결합 수준에 주의를 기울이는 것입니다.

모듈성 정의

메리엄-웹스터 사전은 모듈을 "더 복잡한 구조를 구성하는 데 사용할 수 있는 표준화된 부분 또는 독립적인 단위들의 집합 각각"이라고 정의합니다. 이와는 대조적으로, 이 책에서는 모듈성이라는 용어를 관련 코드들의 논리적 그룹화를 의미하는 것으로 사용합니다. 이는 객체 지향 언어의 클래스 그룹이나 구조적 또는 함수형 언어의 함수 그룹을 예로 들 수 있습니다. 개발자들은 일반적으로 관련 코드들을 함께 묶는 방법으로 모듈을 사용합니다. 예를 들어, Java의 ` com.mycompany.customer` 패키지에는 고객과 관련된 내용들이 포함되어야 합니다. 대부분의 언어는 모듈화를 위한 메커니즘 (Java의 패키지 , .NET의 네임스페이스 등) 을 제공합니다 .

현대 프로그래밍 언어는 매우 다양한 패키징 메커니즘을 제공하며, 많은 개발자들이 그 중에서 어떤 것을 선택해야 할지 어려움을 느낍니다. 예를 들어, 많은 현대 언어에서 개발자는 함수/메서드, 클래스 또는 패키지/네임스페이스에 동작을 정의할 수 있으며, 각각 다른 가시성과 스코프 규칙을 적용할 수 있습니다. 일부 언어는 메타 객체 프로토콜 과 같은 프로그래밍 구문을 추가하여 더욱 다양한 확장 메커니즘을 제공함으로써 이러한 복잡성을 더욱 가중시키기도 합니다.

아키텍트는 개발자가 패키지를 어떻게 구성하는지 잘 알고 있어야 합니다. 패키지 구성은 아키텍처에 중요한 영향을 미치기 때문입니다. 예를 들어, 여러 패키지가 긴밀하게 결합되어 있으면 관련 작업에 특정 패키지를 재사용하기가 더 어려워집니다.

수업 전 모듈 재사용

객체 지향 언어가 등장하기 이전 시대에 개발 교육을 받은 사람들은 왜 이렇게 다양한 분리 방식이 존재하는지 의아해할 수 있습니다. 그 이유의 상당 부분은 하위 호환성과 관련이 있는데, 이는 코드의 호환성이 아니라 개발자들이 사물을 바라보는 방식의 차이에서 비롯됩니다.

1968년 3월, 컴퓨터 과학자 에드거 다익스트라는 학술지 'ACM(Communications of the Association for Computing Machinery)'에 "go-to 문은 해롭다"라는 제목의 논문을 발표했습니다. 그는 당시 프로그래밍 언어에서 흔히 사용되던 go-to 문의 사용을 비판하며 , 이 문이 코드 내에서 비선형적인 이동을 허용하여 추론과 디버깅을 어렵게 만든다고 주장했습니다.

다익스트라의 논문은 1970년대 중반, 파스칼과 C를 비롯한 구조적 프로그래밍 언어 시대를 여는 데 기여했으며, 이러한 언어들은 구성 요소들이 어떻게 서로 연결되는지에 대한 심층적인 사고를 장려했습니다. 개발자들은 대부분의 프로그래밍 언어가 유사한 요소들을 논리적으로 묶는 효과적인 방법을 제공하지 않는다는 사실을 곧 깨달았습니다. 그리하여 1980년대 중반, 모듈라(파스칼 창시자 니클라우스 비르트의 차기 언어)와 에이다와 같은 모듈형 언어 시대가 잠시 도래했습니다. 이 언어들은 오늘날 우리가 패키지나 네임스페이스라고 생각하는 것과 유사한 모듈이라는 프로그래밍 구조를 채택했습니다(다만 클래스는 없습니다).

하지만 1980년대 중반의 모듈형 프로그래밍 시대는 객체 지향 언어가 인기를 얻고 코드를 캡슐화하고 재사용하는 새로운 방식을 제공하면서 오래가지 못했습니다. 그럼에도 불구하고 언어 설계자들은 모듈의 유용성을 인식하고 패키지와 네임스페이스 형태로 이를 유지했습니다. 오늘날에는 다소 이상하게 보일 수 있는 호환성 기능들이 여전히 많이 남아 있는데, 이는 당시 다양한 패러다임을 지원하기 위해 도입된 것입니다. 예를 들어, 자바는 모듈형 패러다임(패키지와 정적 초기화자를 사용한 패키지 수준 초기화를 통해)뿐만 아니라 객체 지향 및 함수형 패러다임도 지원하며, 각 패러다임은 고유한 스코핑 규칙과 특징을 가지고 있습니다.

이 책에서 아키텍처에 대해 논의할 때, 모듈성은 코드의 관련 그룹화(클래스, 함수 또는 기타 그룹)를 나타내는 일반적인 용어로 사용합니다. 이는 물리적 분리가 아니라 논리적 분리를 의미합니다. (이 둘의 차이는 때때로 중요합니다.) 예를 들어, 많은 클래스를 하나의 모놀리식 애플리케이션에 묶어두는 것은 편리할 수 있지만, 아키텍처를 재구성해야 할 때 느슨한 파티셔닝으로 인해 발생하는 결합도가 모놀리식 애플리케이션을 분리하는 데 방해가 될 수 있습니다. 따라서 모듈성을 특정 플랫폼이 강제하거나 암시하는 물리적 분리와는 별개로 개념적으로 논의하는 것이 유용합니다.

네임스페이스라는 일반적인 개념에 대해 논의해 볼 가치가 있습니다. 이는 .NET 플랫폼에서 기술적으로 구현된 네임스페이스와는 별개입니다.
개발자는 종종 다양한 소프트웨어 자산(컴포넌트, 클래스 등)을 서로 구분하기 위해 정확하고 완전한 명칭이 필요합니다. 가장 명백한 예는 다음과 같습니다.

사람들이 매일 사용하는 예로는 인터넷이 있는데, 인터넷은 IP 주소와 연결된 고유한 글로벌 식별자에 의존합니다.

대부분의 언어에는 변수, 함수 또는 메서드와 같은 것들을 구성하기 위한 네임스페이스 역할을 겸하는 모듈화 메커니즘이 있습니다. 때로는 모듈 구조가 물리적인 구조를 반영하기도 합니다. 예를 들어, Java의 패키지 구조는 실제 클래스 파일의 디렉터리 구조를 반영해야 합니다.

이름 충돌이 없는 언어: Java 1.0 Java의 초기 설계자들은 당

시 프로그래밍 플랫폼에서 흔히 발생하던 이름 충돌 문제를 해결하는 데 풍부한 경험을 가지고 있었습니다. Java 1.0은 두 클래스가 같은 이름을 가질 때 발생하는 모호성을 피하기 위해 영리한 해결책을 사용했습니다. 예를 들어 카탈로그 주문과 설치 주문처럼 둘 다 'order'라는 이름을 사용하지만 의미(및 클래스)는 매우 다릅니다. Java 설계자들은 패키지 네임스페이스 메커니즘을 만들고 물리적 디렉터리 구조가 패키지 이름과 일치해야 한다는 요구 사항을 도입했습니다. 파일 시스템은 같은 디렉터리에 같은 이름의 파일이 두 개 존재하는 것을 허용하지 않기 때문에, 이 방법은 운영 체제의 고유한 기능을 활용하여 모호성과 이름 충돌을 해결했습니다. 따라서 Java의 초기 클래스 경로는 디렉터리만으로 구성되었습니다.

하지만 언어 설계자들이 발견했듯이, 모든 프로젝트에 완전한 디렉토리 구조를 갖추도록 강제하는 것은 특히 프로젝트 규모가 커질수록 번거로운 일이었습니다.
게다가 재사용 가능한 자산을 구축하는 것도 어려웠습니다. 프레임워크와 라이브러리를 디렉토리 구조에 "압축 해제"해야 했기 때문입니다. 자바의 두 번째 주요 릴리스(1.2 버전이지만 자바 2라고 불림)에서 설계자들은 JAR 메커니즘을 추가했는데, 이를 통해 아카이브 파일이 클래스패스에서 디렉토리 구조 역할을 할 수 있게 되었습니다. 이후 10년 동안 자바 개발자들은 디렉토리와 JAR 파일의 조합으로 클래스패스를 정확하게 설정하는 데 어려움을 겪었습니다. 원래 의도와는 달리, 두 개의 JAR 파일이 클래스패스에서 충돌하는 이름을 생성할 수 있게 된 것입니다. 이것이 바로 그 시대의 자바 개발자들이 클래스 로더 디버깅에 대한 수많은 경험담을 가지고 있는 이유입니다.

모듈성 측정

모듈성은 매우 중요하기 때문에 아키텍트는 이를 더 잘 이해하는 데 도움이 되는 도구가 필요합니다. 다행히 연구자들은 이러한 목적을 위해 다양한 언어에 구애받지 않는 측정 지표를 개발했습니다. 여기서는 응집도, 결합도, 모듈성이라는 세 가지 핵심 개념에 초점을 맞추겠습니다.
결합.

응집력

응집성은 모듈의 구성 요소들이 동일한 모듈 내에 얼마나 잘 포함되어야 하는지를 나타냅니다. 다시 말해, 구성 요소들이 서로 얼마나 밀접하게 관련되어 있는지를 측정하는 척도입니다.
이상적인 응집력 있는 모듈은 모든 구성 요소가 하나로 통합된 형태입니다. 구성 요소를 더 작은 조각으로 나누면 유용한 결과를 얻기 위해 모듈 간 호출을 통해 각 조각을 다시 연결해야 합니다. 모듈화와 응집력 사이의 관계에 대한 주의 사항은 『구조적 설계(Structured Design)』(Pearson, 2008)라는 책에 나오는 다음 인용문에서 잘 드러납니다.

> 응집력 있는 모듈을 분할하려는 시도는 결합도 증가와 가독성 저하로 이어질 뿐입니다.
>
> —래리 콘스탄틴

컴퓨터 과학자들은 응집도를 최상에서 최악까지 측정하여 다양한 범위로 정의했습니다.

기능적 응집력: 모듈의 모든
: 부분은 서로 연관되어 있으며, 모듈은 기능을 수행하는 데 필요한 모든 필수적인 요소를 포함하고 있습니다.

순차적 응집성: 두 모듈
: 이 상호 작용합니다. 한 모듈이 출력하는 데이터는 다른 모듈의 입력이 됩니다.

의사소통 응집력
: 두 개의 모듈은 서로 정보를 주고받거나 특정 결과물을 생성하는 통신 체인을 형성합니다. 예를 들어, 한 모듈은 데이터베이스에 레코드를 추가하고 다른 모듈은 해당 정보를 기반으로 이메일을 생성합니다.

절차적 응집성: 두 모듈은
: 특정 순서대로 코드를 실행해야 합니다.

시간적 응집성 모듈은
: 시간적 종속성을 기반으로 서로 관련됩니다. 예를 들어, 많은 시스템에는 시스템 시작 시 초기화해야 하는 겉보기에는 관련 없어 보이는 여러 작업들이 있는데, 이러한 다양한 작업들은 시간적으로 응집되어 있습니다.

논리적 응집성이란 모
: 듈 내의 데이터가 논리적으로는 관련되어 있지만 기능적으로는 관련되어 있지 않다는 것을 의미합니다. 예를 들어, 텍스트, 직렬화된 객체 또는 스트림에서 다른 형식으로 정보를 변환하는 모듈을 생각해 보세요. 이 모듈의 연산은 서로 관련되어 있지만 기능은 상당히 다릅니다. 이러한 유형의 응집성을 보여주는 대표적인 예는 거의 모든 Java 프로젝트에서 찾아볼 수 있는 StringUtils 패키지입니다. 이 패키지는 문자열 에 대한 연산을 수행하는 정적 메서드들의 모음이지만 , 그 외에는 서로 관련이 없습니다.

우연한 응집

모듈 내의 요소들은 같은 소스 파일에 있다는 점 외에는 서로 관련이 없습니다.
이는 결속력의 가장 부정적인 형태를 나타냅니다.

다양한 변형이 있지만, 응집도는 결합도보다 정확도가 떨어지는 지표입니다. 특정 모듈의 응집도 수준은 종종 특정 아키텍트의 재량에 따라 결정됩니다. 다음 모듈 정의를 예로 들어 보겠습니다.

고객 유지보수

- 고객 추가
- 고객 정보 업데이트
- 고객 확보
- 고객에게 알림
- 고객 주문 받기
- 고객 주문 취소

마지막 두 항목은 이 모듈에 포함되어야 할까요? 아니면 개발자가 별도의 모듈 두 개를 만들어야 할까요? 두 개의 별도 모듈을 만들 경우 다음과 같은 모습이 될 것입니다.

고객 유지보수

- 고객 추가
- 고객 정보 업데이트
- 고객 확보
- 고객에게 알림

주문 관리

- 고객 주문 받기
- 고객 주문 취소

어떤 구조가 올바른 것일까요? 언제나 그렇듯이 상황에 따라 다릅니다.

- 주문 관리 에는 이 두 가지 작업만 필요한가요 ? 그렇다면 해당 작업을 고객 관리로 다시 통합하는 것이 좋을 수 있습니다.
- 고객 유지 관리 규모가 훨씬 더 커질 것으로 예상되나요? 그렇다면 개발자는 특정 동작을 다른 (또는 새로운) 모듈로 분리할 기회를 찾아야 합니다.
- 주문 관리 기능에는 고객 정보 에 대한 방대한 지식이 필요해서 두 모듈을 분리하면 기능 구현을 위해 높은 수준의 결합이 불가피한가요? (이 질문은 앞서 언급한 래리 콘스탄틴의 인용문과 관련이 있습니다.)

이러한 질문들은 소프트웨어 아키텍트의 업무 핵심에 있는 절충 분석의 유형을 나타냅니다.

컴퓨터 과학자들은 응집도를 판단하는 데 유용한 구조적 지표, 특히 응집도 부족(Lack of Cohesion)을 개발했습니다. (이 특성이 얼마나 주관적인지를 고려하면 다소 놀라운 사실입니다.) 치담버(Chidamber)와 케머러(Kemerer)의 객체 지향 메트릭 스위트(Object-Oriented Metrics Suite) 는 객체 지향 소프트웨어 시스템의 특정 측면을 측정하는 잘 알려진 메트릭 세트입니다. 여기에는 순환 복잡도(Cyclomatic Complexity, 84페이지의 "순환 복잡도" 참조) 와 같은 일반적인 코드 메트릭과 44페이지의 "결합도(Coupling)" 에서 논의되는 여러 중요한 결합도 메트릭이 포함됩니다.

Chidamber와 Kemerer는 모듈의 구조적 응집력을 측정하는 방법론의 응집력 부족(LCOM) 지표도 개발했습니다. 초기 버전은 방정식 3-1에 나와 있습니다.

방정식 3-1. LCOM, 버전 1

$$LCOM = \begin{cases} P \text{ 'Q, 만약 } P > Q \text{이면 } Q, \text{ 그렇지} \\ \text{않으면} \end{cases}$$

이 방정식에서 특정 공유 필드에 접근하지 않는 메서드의 경우 P는 1씩 증가하고, 특정 공유 필드를 공유하는 메서드의 경우 Q는 1씩 감소합니다. 이 공식이 혼란스럽게 느껴지신다면 충분히 이해합니다. 이 공식은 점차 더 복잡해지고 있습니다. 1996년에 도입된 두 번째 변형(그래서 LCOM96B라는 이름이 붙었습니다)은 방정식 3-2에 나타납니다.

방정식 3-2. LCOM96B

$$LCOM96b = \frac{1}{\text{중}} \sum \frac{m \text{ "}\mu Aj \# a}{(=1)}$$

방정식 3-2 의 변수와 연산자를 일일이 분석하는 것보다는 아래의 설명이 더 명확하므로 생략하겠습니다 . 기본적으로 LCOM 지표는 클래스 내의 우연한 결합도를 보여줍니다. LCOM을 더 정확하게 정의하자면 "공유 필드를 통해 공유되지 않는 메서드 집합의 합"이라고 할 수 있습니다.

개인 필드 a 와 b 를 가진 클래스를 생각해 봅시다 . 많은 메서드가 a에만 접근하고, 또 다른 많은 메서드는 b에만 접근합니다 . 공유 필드 (a 와 b) 를 통해 공유되지 않는 메서드들의 집합의 합이 크기 때문에, 이 클래스는 LCOM 점수가 높게 나타나며, 이는 메서드 응집도가 상당히 낮다는 것을 의미합니다.

그림 3-1 에 나타난 세 가지 클래스를 생각해 보세요 . 여기서 분야는 팔각형 안에 한 글자로, 방법은 블록으로 표시됩니다. 클래스 X는 LCOM 점수가 낮아 구조적 응집력이 우수함을 나타냅니다. 그러나 클래스 Y는 응집력이 부족합니다. 각 클래스는

클래스 Y의 필드/메서드 쌍은 시스템 동작에 영향을 주지 않고 별도의 클래스로 분리될 수 있습니다. 클래스 Z는 응집도가 혼합되어 있으며, 마지막 필드/메서드 조합은 별도의 클래스로 리팩토링될 수 있습니다.

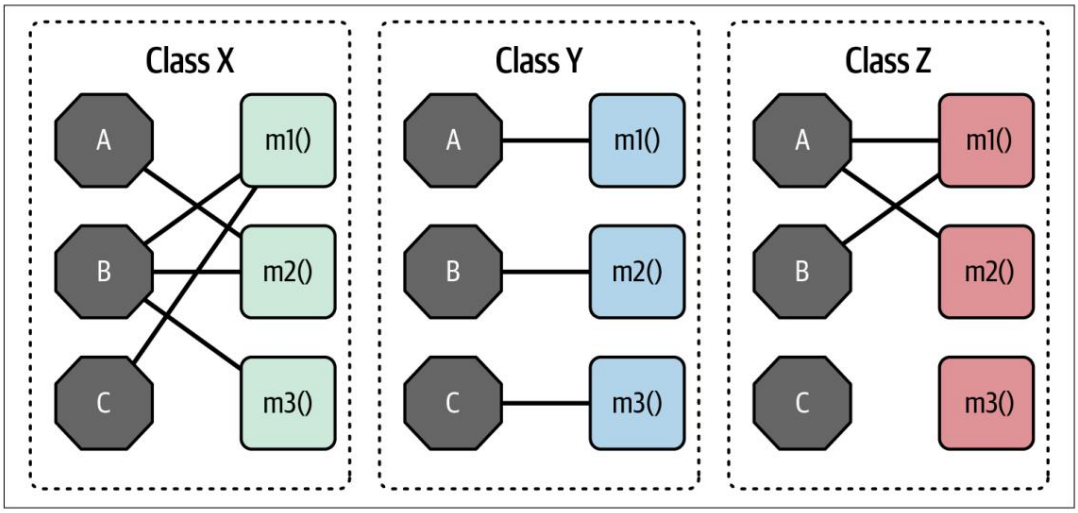

그림 3-1. e LCOM 메트릭. 여기서 필드는 팔각형이고, 방법은 사각형입니다.

LCOM 지표는 코드베이스를 분석하여 재구조화, 마이그레이션 또는 코드베이스 이해를 돕는 아키텍트에게 유용합니다. 공유 유틸리티 클래스는 아키텍처를 이전할 때 흔히 발생하는 골칫거리입니다. LCOM 지표를 사용하면 아키텍트는 의도치 않게 결합되어 애초에 단일 클래스로 존재해서는 안 되었던 클래스들을 찾아낼 수 있습니다.

많은 소프트웨어 메트릭에는 심각한 결함이 있으며, LCOM도 예외는 아닙니다. 이 메트릭은 구조적 응집력 부족만 찾아낼 수 있을 뿐, 특정 구성 요소들이 논리적으로 잘 어우러지는지 판단할 방법이 없습니다. 이는 소프트웨어 아키텍처의 제2법칙, 즉 '어떻게'보다 '왜'가 더 중요하다는 점을 다시금 상기시켜 줍니다.

연결

다행히 코드베이스의 결합도를 분석하는 데 더 나은 도구들이 있습니다. 이러한 도구들은 부분적으로 그래프 이론에 기반을 두고 있는데, 메서드 호출과 반환값이 호출 그래프를 형성하기 때문에 수학적으로 분석할 수 있기 때문입니다. 에드워드 요든과 래리 콘스탄틴의 저서 "구조적 설계: 컴퓨터 프로그램 및 시스템 설계 분야의 기초"(프렌티스-홀, 1979)는 상호 연결 결합도와 상호 연결 결합도와 같은 핵심 개념들을 정의했습니다. 상호 연결 결합도는 코드 아티팩트(컴포넌트, 클래스, 함수 등)에 들어오는 연결 수를 측정합니다.

결합도는 다른 코드 아티팩트로 나가는 연결을 측정합니다. 거의 모든 플랫폼에서 아키텍트가 코드의 결합 특성을 분석할 수 있는 도구가 있습니다.

결합도 측정 지표에 왜 이렇게 비슷한 이름이 붙었을까요?

건축계에서 정반대의 개념을 나타내는 두 가지 중요한 지표가 왜 발음이 가장 비슷한 모음만 다르게 붙여져 거의 같은 이름으로 불리는 걸까요? 이 용어들은 『구조적 설계(Structured Design)』라는 책에서 유래했습니다. 요든과 콘스탄틴은 수학 개념을 차용하여 현재는 흔히 사용되는 '에어런트(aerent)'와 '이런트(eerent)' 결합이라는 용어를 만들었습니다. 사실 '입력 결합(incoming)'과 '출력 결합(outgoing coupling)'이라고 부르는 것이 더 적절했겠지만, 저자들은 명확성보다는 수학적 대칭성을 선호했던 것 같습니다.

개발자들은 기억을 돕기 위해 여러 가지 연상 기호를 고안했습니다. 예를 들어, 영어 알파벳에서 a는 e보다 먼저 나오는데, 이는 incoming이 outgoing보다 먼저 나오는 것과 같습니다. eerent의 e는 exit의 첫 글자와 일치하므로, 이것이 나가는 연결을 나타낸다는 것을 기억하는 데 도움이 됩니다.

핵심 지표

컴포넌트 결합도는 아키텍트에게 기본적인 가치를 제공하지만, 그 외 여러 파생 지표를 통해 더욱 심층적인 평가가 가능합니다. 이 섹션에서 다루는 지표들은 소프트웨어 엔지니어 로버트 C. 마틴이 개발했으며, 대부분의 객체 지향 언어에 폭넓게 적용될 수 있습니다.

추상성은 추상적인 요소(추상 클래스, 인터페이스 등)와 구체적인 요소(구현체)의 비율입니다. 추상성 지표는 코드베이스의 추상성 수준과 구현 수준을 측정합니다. 예를 들어, 한쪽 극단에는 추상화가 전혀 없고 하나의 거대한 기능(예: 단일 main() 메서드)으로 이루어진 코드베이스가 있습니다. 다른 극단에는 추상화가 너무 많아 개발자가 구성 요소들이 어떻게 연결되는지 이해하기 어려운 코드베이스가 있습니다. (예를 들어, AbstractSingletonProxyFactoryBean 이라는 추상 클래스는 여러 겹의 추상화와 모호한 이름 때문에 개발자가 어떻게 사용해야 할지 파악하는 데 시간이 걸립니다 .)

추상성의 공식은 방정식 3-3에 나타납니다.

방정식 3-3. 추상성

$$A = \frac{\#\text{엄마}}{\#mc + \#ma}$$

아키텍트는 추상성 지표를 계산할 때 추상적인 요소의 합과 구체적인 요소 및 추상적인 요소의 합을 비교하는 비율을 사용합니다. 이 공식에서 ma는 모듈 내의 추상 요소(인터페이스 또는 추상 클래스)를 나타내고, mc는 구체적인 요소(추상적이지 않은 클래스)를 나타냅니다. 이 지표는 동일한 기준을 적용합니다. 이 지표를 가장 쉽게 이해하는 방법은 main() 메서드 하나에 5,000줄의 코드가 모두 포함된 애플리케이션을 생각해 보는 것입니다. 이 경우 추상성 지표의 분자는 1이고 분모는 5,000이 되어 추상성 점수는 거의 0에 가까워집니다. 이처럼 이 지표는 코드 내 추상화 비율을 측정합니다.

또 다른 파생 지표인 불안정성은 식 3-4 에 나타낸 바와 같이 원심성 결합과 구심성 결합의 합에 대한 원심성 결합의 비율로 정의됩니다 .

방정식 3-4. 불안정성

$$나 = \frac{세}{세륨 + 칼슘}$$

이 방정식에서 ce는 원심성(또는 나가는) 연결을 나타내고, ca는 구심성(또는 들어오는) 연결을 나타냅니다.

불안정성 지표는 코드베이스의 변동성을 측정합니다. 불안정성이 높은 코드베이스는 결합도가 높아 변경될 때 더 쉽게 오류가 발생합니다. 예를 들어, 한 클래스가 작업을 위임하기 위해 너무 많은 다른 클래스를 호출하는 경우, 호출하는 클래스는 호출된 메서드 중 하나 이상이 변경될 때 오류가 발생할 가능성이 높습니다.

주계열과의 거리

건축가가 아키텍처 구조를 평가할 때 사용하는 몇 안 되는 전체적인 지표 중 하나는 불안정성과 추상성을 기반으로 도출된 지표인 주계열과의 거리이며, 이는 방정식 3-5에 나타나 있습니다.

방정식 3-5. 주계열과의 거리

$$D = A + I - 1$$

이 방정식에서 A는 추상성을, I는 불안정성을 나타냅니다.

추상성과 불안정성은 모두 분수이며, 그 결과는 (일부 극단적인 경우를 제외하고) 항상 0과 1 사이에 있다는 점에 유의하십시오. 따라서 이러한 관계를 그래프로 나타내면 그림 3-2와 같은 그래프가 됩니다 .

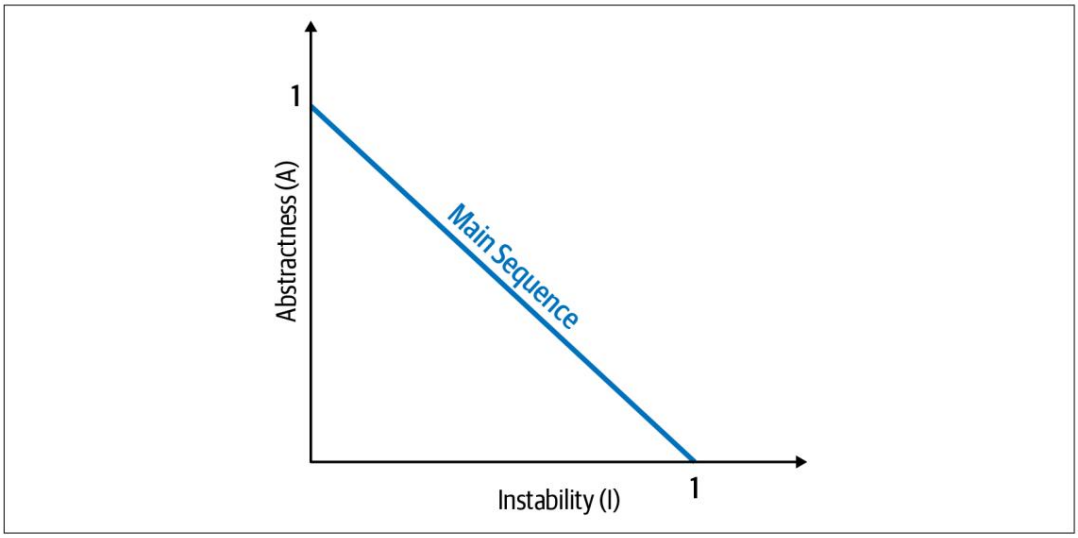

그림 3-2. 주계열은 추상성과 불안정성 사이의 이상적인 관계를 정의합니다.

거리 측정 기준은 추상성과 즉시성 사이의 이상적인 관계를 가정합니다. 이 이상적인 선에 가까운 클래스는 이 두 가지 상충되는 요소가 적절히 혼합되어 있음을 보여줍니다. 예를 들어, 특정 클래스의 그래프를 그리면 개발자는 그림 3-3에 나타낸 주계열 측정 기준에서 거리를 계산할 수 있습니다.

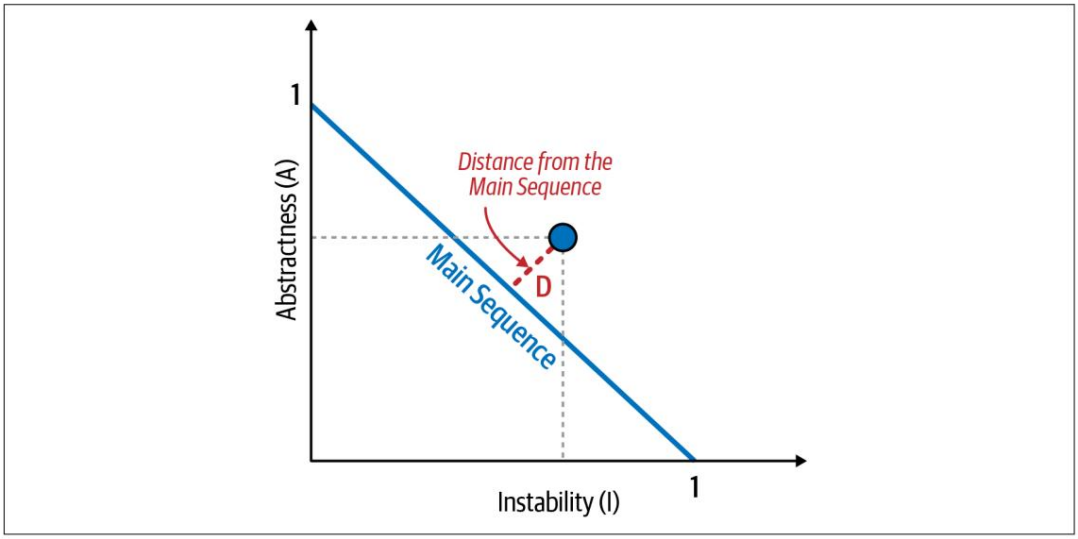

그림 3-3. 특정 클래스에 대한 주계열로부터의 정규화된 거리

그림 3-3은 후보 클래스를 그래프로 나타낸 다음 이상적인 선과의 거리를 측정합니다. 선에 가까울수록 클래스의 균형이 더 잘 잡혀 있습니다. 오른쪽 상단 모서리에 너무 치우친 클래스는 아키텍트들이 '무용성 영역(Zone of Uselessness)'이라고 부르는 영역에 들어갑니다. 지나치게 추상적인 코드는 사용하기 어려워집니다. 반대로 그림 3-4 에서처럼 왼쪽 하단 모서리에 있는 코드는 '고통 영역(Zone of Pain)'에 들어갑니다. 구현은 너무 많고 추상화는 부족하여 코드가 취약해지고 사용하기 어려워집니다.

유지하다.

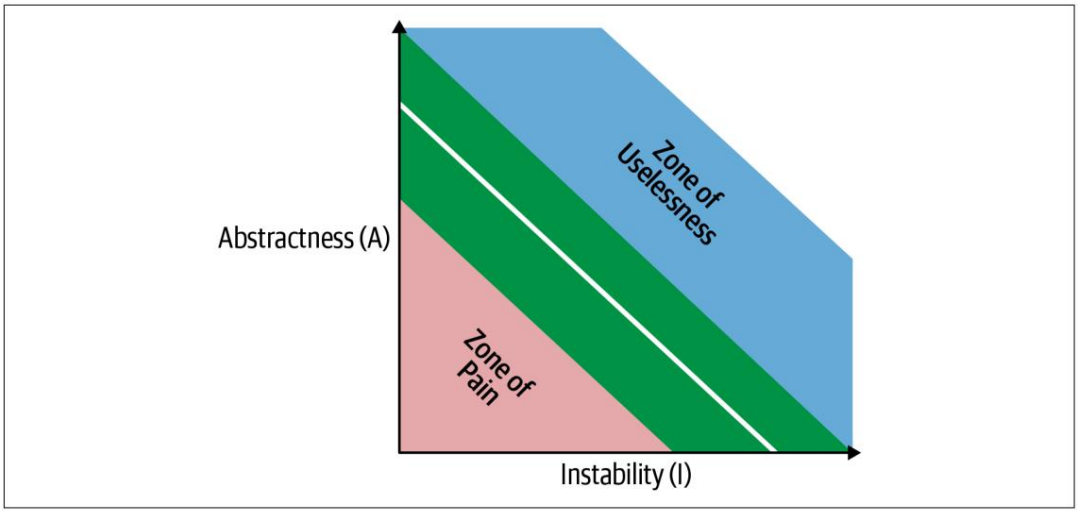

그림 3-4. 무용지물 영역 및 고통 영역

많은 플랫폼에서 이러한 측정값을 계산하는 도구를 제공하며, 이는 아키텍트가 코드베이스를 분석하여 익숙해지거나, 마이그레이션을 준비하거나, 기술 부채를 평가하는 데 도움이 됩니다.

측정 기준의 한계

업계에는 유용한 통찰력을 제공하는 몇 가지 코드 수준 지표가 있지만, 우리의 도구는 다른 엔지니어링 분야의 분석 도구와 비교하면 매우 투박합니다.
코드 구조에서 직접 도출된 메트릭조차도 해석이 필요합니다. 예를 들어, 순환 복잡도(84페이지의 "순환 복잡도" 참조) 는 코드베이스의 복잡도를 측정하지만, 본질적 복잡도(기저 문제가 복잡하기 때문에 코드가 복잡한 경우)와 우발적 복잡도(코드가 필요 이상으로 복잡한 경우)를 구분할 수 없습니다. 거의 모든 코드 수준 메트릭은 해석을 필요로 하지만, 아키텍트가 코드베이스의 유형을 평가할 수 있도록 순환 복잡도와 같은 중요한 메트릭에 대한 기준선을 설정하는 것은 여전히 유용합니다. 이러한 테스트 설정에 대해서는 86페이지의 "거버넌스 및 적합도 함수"에서 논의합니다.

1979년에 출판된 Yourdon과 Constantine의 『구조적 설계(Structured Design)』는 객체 지향 언어가 대중화되기 이전에 나온 책입니다. 이 책은 메서드가 아닌 함수와 같은 구조적 프로그래밍 구성 요소에 초점을 맞추고 있습니다. 또한 현대 프로그래밍 언어 설계로 인해 시대에 뒤떨어진 다른 유형의 결합 개념을 정의하기도 합니다. 객체 지향 프로그래밍은 입력 결합과 출력 결합을 아우르는 추가적인 개념들을 도입했는데, 여기에는 결합을 설명하는 더욱 정교한 용어인 '연결성(connascence)'이 포함됩니다.

결합

메일러 페이지-존스의 저서 《모든 프로그래머가 객체 지향 설계에 대해 알아야 할 것》(도셋 하우스, 1996)은 객체 지향 언어에서 다양한 유형의 결합을 보다 정확하게 설명하는 언어를 제시했습니다. 결합(Connascence)은 구심적 결합과 원심적 결합과 같은 결합 미터가 아니라, 아키텍트가 다양한 유형의 결합을 보다 정확하게 설명하고 (그리고 이러한 결합 유형의 공통적인 결과를 이해하는 데) 도움을 주는 언어입니다.

두 구성 요소가 동일하다는 것은 한 구성 요소의 변경으로 인해 시스템의 전체적인 정확성을 유지하기 위해 다른 구성 요소도 수정해야 하는 경우를 말합니다. 페이지-존스는 동일성을 정적 동일성과 동적 동일성의 두 가지 유형으로 구분합니다.

정적 연결

정적 연결성은 소스 코드 수준의 결합을 의미합니다(실행 시간 결합과는 반대되는 개념으로, 50 페이지의 "동적 연결성" 에서 다룹니다). 아키텍트는 정적 연결성을 입력 또는 출력 결합을 통해 어떤 요소가 얼마나 강하게 결합되어 있는지를 나타내는 정도로 간주합니다. 정적 연결성에는 여러 유형이 있습니다.

이름의 일치성 여러 구성
: 요소는 엔티티의 이름에 대해 일치해야 합니다.

  메서드 이름과 메서드 매개변수는 코드베이스를 결합하는 가장 일반적인 방식이며, 특히 시스템 전체의 이름 변경을 간편하게 구현할 수 있는 최신 리팩토링 도구를 고려할 때 가장 바람직한 방식이기도 합니다. 예를 들어, 개발자는 더 이상 활성화된 코드베이스에서 메서드 이름을 직접 변경하는 대신 최신 도구를 사용하여 메서드 이름을 리팩토링함으로써 코드베이스 전체에 변경 사항을 적용할 수 있습니다.

타입 일치성 여러 구성
: 요소는 엔티티의 타입에 대해 일치해야 합니다.

  이러한 유형의 유사성은 많은 정적 타입 언어에서 변수와 매개변수를 특정 타입으로 제한하는 일반적인 경향을 나타냅니다. 그러나 이러한 기능은 정적 타입 언어에만 있는 것은 아닙니다. Clojure와 Clojure Spec을 비롯한 일부 동적 타입 언어에서도 선택적 타입 지정을 제공합니다.

의미의 일치란 여러 구성 요
소가 특정 값의 의미에 대해 동의해야 함을 의미합니다. 이를 관습의 일치라고도 합니다.

코드베이스에서 이러한 유형의 연관성을 가장 흔하게 볼 수 있는 경우는 상수가 아닌 하드코딩된 숫자입니다. 예를 들어, 일부 언어에서는 int TRUE = 1; int FALSE = 0과 같이 어딘가에 정의하는 것이 일반적입니다.
누군가가 그 값들을 뒤집어 놓는다면 어떤 문제가 발생할지 상상해 보세요.

위치 일치성 여러 구성 요소
는 값의 순서에 대해 일치해야 합니다.

이는 정적 타입 기능을 지원하는 언어에서도 메서드 및 함수 호출 시 매개변수 값과 관련된 문제입니다. 예를 들어, 개발자가 ` void updateSeat(String name, String seatLocation)` 이라는 메서드를 만들고 `updateSeat("14D", "Ford", "N")` 과 같은 값으로 호출하면 , 타입은 맞더라도 의미론적으로 올바르지 않습니다.

알고리즘의 합의 여러 구성 요
소가 특정 알고리즘에 동의해야 합니다.

알고리즘 일치성의 일반적인 예는 개발자가 사용자를 인증하기 위해 서버와 클라이언트 모두에서 실행되어 동일한 결과를 생성해야 하는 보안 해싱 알고리즘을 정의할 때 발생합니다. 이는 명백히 높은 수준의 결합도를 나타냅니다. 알고리즘의 세부 사항 중 하나라도 변경되면 인증 과정이 더 이상 작동하지 않게 됩니다.

페이지-존스가 정의한

또 다른 유형의 연결은 런타임에 호출을 분석하는 동적 연결입니다. 동적 연결 유형에는 다음이 포함됩니다.

실행 순서의 일치 여러 구성
요소의 실행 순서는 중요합니다.

다음 코드를 살펴보세요:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

특정 속성들은 정해진 순서대로 설정되어야 하기 때문에 제대로 작동하지 않을 것입니다.

타이밍의 일치 여러 구성
요소의 실행 타이밍은 중요합니다.

이러한 유형의 충돌이 발생하는 일반적인 경우는 두 스레드가 동시에 실행되어 공동 작업 결과에 영향을 미치는 경쟁 조건입니다.

가치의 상호 연관성 여러
가치는 서로에게 의존하며 함께 변화해야 합니다.

개발자가 네 모서리를 나타내는 네 개의 점을 정의하여 직사각형을 만들었다고 가정해 보겠습니다. 데이터 구조의 무결성을 유지하려면 개발자는 직사각형의 모양을 유지하기 위해 다른 점에 미치는 영향을 고려하지 않고 임의로 한 점을 변경할 수 없습니다.

더 흔하고 문제가 되는 경우는 특히 분산 시스템에서 트랜잭션과 관련된 경우입니다. 여러 개의 데이터베이스로 구성된 시스템에서 누군가가 모든 데이터베이스에 걸쳐 단일 값을 업데이트해야 할 때, 모든 값이 동시에 변경되거나 전혀 변경되지 않아야 합니다.

동일성의 일치: 여러 구성
요소는 동일한 엔티티를 참조해야 합니다.

동일성 인식의 일반적인 예로는 분산 큐와 같이 공통 데이터 구조를 공유하고 업데이트해야 하는 두 개의 독립적인 구성 요소가 있습니다.

컨나센스 속성 컨나센스

는 건축가와 개발자를 위한 분석 프레임워크이며, 컨나센스의 몇 가지 속성은 이를 현명하게 사용하는 데 도움이 됩니다. 이러한 컨나센스 속성은 다음과 같습니다.

스트렝스 아

키텍트는 개발자가 결합도를 얼마나 쉽게 리팩토링할 수 있는지를 기준으로 시스템의 연결 강도를 판단합니다. 그림 3-5에서 볼 수 있듯이, 일부 유형의 연결성은 다른 유형보다 훨씬 더 바람직합니다. 더 나은 유형의 연결성을 향해 리팩토링하면 코드베이스의 결합 특성을 개선할 수 있습니다.

아키텍트는 동적 연결성보다 정적 연결성을 선호해야 합니다. 개발자가 간단한 소스 코드 분석을 통해 연결성을 파악할 수 있고, 최신 도구를 사용하면 정적 연결성을 개선하는 것이 매우 쉽기 때문입니다. 예를 들어, 의미 연결성은 이름 연결성으로 리팩토링하여 임의의 값 대신 명명된 상수를 생성함으로써 개선할 수 있습니다.

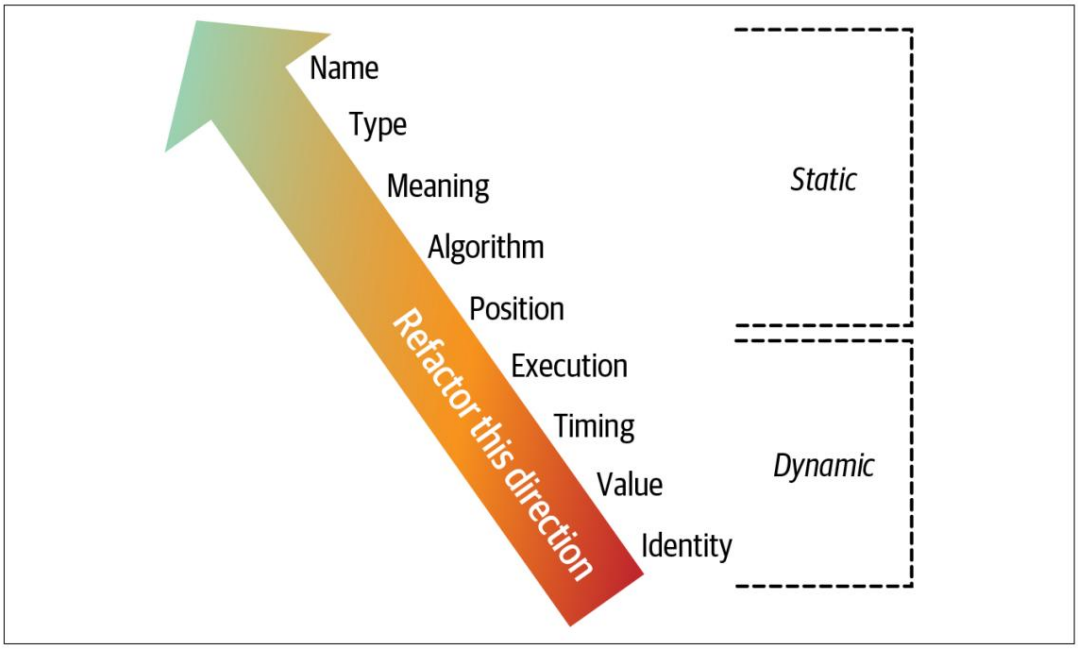

그림 3-5. 연결 강도는 좋은 리팩토링 지침이 될 수 있습니다.

지역성

(Locality)은 시스템의 연결성 지표로서, 코드베이스 내에서 모듈들이 서로 얼마나 근접해 있는지를 나타냅니다. 가까운 코드(동일 모듈 내의 코드)는 일반적으로 분리된 코드(별도의 모듈이나 코드베이스에 있는 코드)보다 더 다양하고 높은 수준의 연결성을 보입니다. 다시 말해, 구성 요소들이 멀리 떨어져 있을 때는 낮은 결합도를 나타내는 연결성 형태도, 가까이 있을 때는 문제가 되지 않습니다. 예를 들어, 동일한 모듈에 있는 두 클래스가 의미적 연결성(Connascence of Meaning)을 가지고 있다면, 이는 해당 클래스들이 서로 다른 모듈에 있는 경우보다 코드베이스에 미치는 악영향이 적습니다.

저자가 이 내용을 처음 발표했을 당시에는 건축가들이 이 관찰의 중요성을 제대로 인식하지 못했습니다. 현대적인 관점에서 보면, 저자는 건축가들이 구현 세부 사항(높은 결합도)의 범위를 가능한 한 좁게 제한해야 한다고 주장하는데, 이는 도메인 주도 설계(DDD)의 경계 컨텍스트 개념에서 파생된 조언과 동일합니다. 핵심적인 아키텍처적 관찰은 구현 결합도를 제한하는 것입니다. 메일리르-페이지는 DDD를 통해 더욱 완전하게 재조명된 훌륭한 설계 원칙을 설명했습니다(7 장 97페이지 의 "도메인 주도 설계의 경계 컨텍스트" 참조).

강점과 지역성을 함께 고려하는 것이 좋습니다. 동일한 모듈 내에서 강한 연관성을 보이는 것이 모듈 간에 분산된 동일한 연관성보다 코드 스멜(code smell)이 적습니다.

동적 연

결성의 정도는 특정 모듈 내 클래스를 변경했을 때 미치는 영향의 크기, 즉 해당 변경이 몇몇 클래스에 영향을 미치는지 아니면 많은 클래스에 영향을 미치는지와 관련이 있습니다. 동적 연결성이 낮을수록 다른 클래스와 모듈에 대한 변경 필요성이 줄어들고, 결과적으로 코드베이스 손상도 적어집니다. 다시 말해, 아키텍트가 다루는 모듈이 몇 개 되지 않는다면 동적 연결성이 높더라도 큰 문제가 되지 않습니다. 하지만 코드베이스는 시간이 지남에 따라 규모가 커지기 때문에 작은 문제도 그에 따라 변경해야 하는 범위가 커지게 됩니다.

Page-Jones는 그의 저서 "객체 지향 설계에 대해 모든 프로그래머가 알아야 할 것"에서 시스템 모듈성을 향상시키기 위해 connascence를 사용하는 세 가지 지침을 제시합니다.

- 시스템을 캡슐화된 부분으로 나누어 전체적인 연결성을 최소화합니다.
  강요.
- 캡슐화 경계를 넘어서는 잔류 연결을 최소화합니다. • 캡슐화 경계 내의 연결을 최대화합니다.

소프트웨어 아키텍처 분야의 전설적인 혁신가이자 '연결성(Connescence)'이라는 개념을 재조명한 짐 웨이리치는 2012년 '엔터프라이즈를 위한 신기술' 컨퍼런스에서 '연결성 검증!' 이라는 주제로 발표한 강연에서 두 가지 중요한 규칙을 제시했습니다.

정도의 법칙: 강한 형태의 친밀감을 약한 형태로 변환한다.
결합.

지역성 원칙: 소프트웨어 요소 간의 거리가 멀어질수록 약한 형태의 연결을 사용하십시오.

아키텍트가 코나센스(connascence)를 배우는 것은 디자인 패턴을 배우는 것과 같은 이유로 유익합니다. 코나센스는 다양한 유형의 결합을 설명하는 데 더 정확한 언어를 제공하기 때문입니다. 예를 들어, 아키텍트는 누군가에게 "서비스가 필요한데 인스턴스는 하나만 존재해야 합니다."라고 말하거나 "싱글턴 서비스가 필요합니다."라고 말할 수 있습니다. 싱글턴 디자인 패턴은 일반적인 문제에 대한 맥락과 해결책을 간단한 이름으로 깔끔하게 캡슐화합니다.

마찬가지로, 코드 리뷰를 수행할 때 아키텍트는 개발자에게 "메서드 선언 중간에 특수 문자열 상수를 넣지 마세요. 대신 상수로 추출하세요."라고 지시할 수 있습니다. 또는 "의미의 일치(Connascence of Meaning)를 사용하고 있으니, 이름의 일치(Connascence of Name)로 리팩토링하세요."라고 말할 수도 있습니다.

모듈에서 컴포넌트로

이 책에서는 모듈이라는 용어를 관련 코드 묶음을 나타내는 일반적인 이름으로 사용합니다. 그러나 대부분의 아키텍트는 모듈을 소프트웨어 아키텍처의 핵심 구성 요소인 컴포넌트라고 부릅니다. 이러한 컴포넌트 개념과 그에 따른 논리적 또는 물리적 분리 분석은 컴퓨터 과학 초창기부터 존재해 왔지만, 개발자와 아키텍트는 여전히 좋은 컴포넌트를 구현하는 데 어려움을 겪고 있습니다.
결과.

8 장 에서 문제 영역에서 구성 요소를 도출하는 방법에 대해 논의하겠지만 , 그 전에 소프트웨어 아키텍처의 또 다른 근본적인 측면인 아키텍처 특성과 그 범위에 대해 먼저 논의해야 합니다.