(19)대한민국특허청(KR)
(12) 공개특허공보(A)

(51) 。Int. Cl.[7]
G06F 9/06

(11) 공개번호 10-2005-0039534
(43) 공개일자 2005년04월29일

(21) 출원번호 10-2004-0070423
(22) 출원일자 2004년09월03일

(30) 우선권주장 10/692,320 2003년10월23일 미국(US)

(71) 출원인 마이크로소프트 코포레이션
미국 워싱턴주 (우편번호 : 98052) 레드몬드 원 마이크로소프트 웨이

(72) 발명자 무어안토니제이.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이 마이크로소프트 코포레이션 내
아브람스브래들리엠.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이 마이크로소프트 코포레이션 내
앤더슨크리스토퍼엘.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이 마이크로소프트 코포레이션 내
크월리나크리지스토프
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이 마이크로소프트 코포레이션 내
피조마이클제이.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이 마이크로소프트 코포레이션 내
브리엄로버트에이.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이 마이크로소프트 코포레이션 내

(74) 대리인 주성민
백만기
이중희

심사청구 : 없음

# (54) 응용 프로그래밍 인터페이스의 설계

요약

API(application programming interface)를 설계하는 제 1의 예시적 방법 구현은 코어 시나리오에 대해 다수의 코드 샘플들을 준비하는 단계, 다수의 코드 샘플들 중의 개별적인 각 코드 샘플을 다수의 프로그래밍 언어들 중의 개별적인 프로그래밍 언어에 대응시키는 단계, 및 다수의 코드 샘플들에 응답하여 코어 시나리오로부터 API를 유도하는 단계를 포함한다. API를 설계하는 제 2의 예시적 방법은 특징 영역에 대해 코어 시나리오를 선택하는 단계, 코어 시나리오에 대해 하나 이상의 코드 샘플을 기재하는 단계, 및 하나 이상의 코드 샘플에 응답하여 코어 시나리오에 대한 API를 유도하는 단계를 포함한다. API를 설계하는 제 3의 예시적 방법은 시나리오에 대해 기재된 하나 이상의 코드 샘플에 응답하여 시나리오에 대한 API를 유도하는 단계, 다수의 개발자들을 활용하여, API에 대해 하나 이상의 가용성 연구를 수행하는 단계, 및 하나 이상의 가용성 연구에 기초하여 API를 수정하는 단계를 포함한다.

대표도

도 2

색인어

API, 코어 시나리오, 코드 샘플, 가용성 연구

# 명세서

## 도면의 간단한 설명

도 1은 2개의 상이한 프레임워크에 대한 통상적인 API 학습 곡선의 그래프를 도시한다.

도 2는 2개의 상이한 추상화 수준에 대한 예시적인 발전적 API 학습 곡선을 도시한다.

도 3은 API에 대한 예시적 설계 원칙 및 관행(design principles and practices)을 도시한다.

도 4는 특징 영역마다 API를 설계하는 예시적 기술을 도시하는 흐름도이다.

도 5는 코어 시나리오마다 API를 설계하는 예시적 방식을 도시하는 블록도이다.

도 6은 2개의 상이한 목적에 목표를 두고 있는 예시적 컴포넌트 타입들간의 잠재적 부등성(potential disparity)을 도시한다.

도 7은 2개의 상이한 목적을 커버하기 위해 확장가능하도록 그리고/또는 상호작용가능하도록 설계된 컴포넌트 타입들간의 예시적인 관계를 도시한다.

도 8은 2-계층 API로 2개의 상이한 목적을 핸들링하는 예시적 AC(aggregate component) 및 관련 FT들(factored types)을 도시한다.

도 9는 작성-설정-호출 사용 패턴(create-set-call usage pattern)을 지원할 수 있는 예시적 AC 및 관련 API들을 도시한다.

도 10은 여기에서 설명한 API를 설계 및/또는 사용하는 하나 이상의 태양을 (전체적으로 또는 부분적으로) 구현할 수 있는 예시적 컴퓨팅 (또는 범용 장치) 동작 환경을 도시한다.

<도면의 주요 부분에 대한 부호의 설명>

101 : 통상적 API 학습 곡선의 그래프

200 : 발전적 API 학습 곡선의 그래프

300 : API에 대한 예시적 설계 원칙 및 관행

602, 702 : 컴포넌트 타입

802 : AC 멤버

804 : FT 멤버

902 : 구성자

904 : 이벤트

1000 : 동작 환경

1018 : 자기 디스크 드라이브

1022 ; 광학 디스크 드라이브

1024 : 비휘발성 광학 디스크

## 발명의 상세한 설명

### 발명의 목적

#### 발명이 속하는 기술 및 그 분야의 종래기술

본 개시는 일반적으로 응용 프로그래밍 인터페이스(API)에 관한 것으로, 보다 구체적으로는, 한정이 아닌 일례로써, 컨트롤과 유연성을 동시에 제공하면서 사용이 용이한 API를 설계하는 것에 관한 것이다.

API는 개발자들에 의해 광범위한 애플리케이션 및 프로그램을 작성하는데 사용된다. 개발자들로는 매크로를 기록하는 사무원들로부터 저급 장치의 드라이버 제작자들을 들 수 있다. 이들 개발자들은, 상이한 기술 세트들로 그리고/또는 상이한 목적들을 위해 프로그래밍하면서, 상이한 언어들 및/또는 복잡도가 다른 상이한 프레임워크들에 의존한다. 통상적으로, 상이한 API는 개개인의 상이한 기술 수준 및 (예를 들어, 상이한 관련 시나리오에 기초한) 상이한 컨트롤 요구를 목표로 삼아 설계되었다.

이러한 접근이 특정 개발자에 대해 최적화된 API를 제공하는데는 성공적일 수 있지만, 상당한 단점들을 가진다. 예를 들어, 이러한 다수 프레임워크 접근은, 개발자들이 하나의 기술 수준 및 시나리오 타입으로부터 다른 것으로 지식을 전달하는데 어려움을 겪는 상황을 초래한다. 개발자들이 상이한 프레임워크를 사용해 시나리오을 구현할 필요가 있을 경우, 개발자들은 아주 가파른 학습 곡선(very steep learning curve)을 나타낸다. 학습 곡선이 아주 가파를 뿐만 아니라, 일반적으로, 제 1의 저급-기술-수준 프레임워크에 기입된 코드를 스크래치로부터 제 2의 고급-기술-수준 프레임워크로 재기입해야 한다. 또한, 상이한 기술 수준의 개발자에 대해 개별적인 프레임워크를 생성하는 것은 통상적으로, 한가지 수준의 개발자를 목표로 하거나 그에 의해 구현된 API가 다른 수준의 개발자에 의해서는 사용될 수 없는 상황을 초래한다.

도 1은 상이한 2개의 프레임워크에 대한 통상적 API 학습 곡선의 그래프(101)를 도시한다. 제 1 프레임워크는, 요구되는 기술 및/또는 어려움이 상대적으로 저급인 동시에 개발자에 의한 컨트롤 용량이 상대적으로 낮은 프레임워크에 대응된다. 한편, 제 2 프레임워크는 필요한 기술 및/또는 어려움이 상대적으로 고급인 동시에 개발자에 의한 컨트롤 용량이 상대적으로 높은 프레임워크에 대응된다. 이러한 제 1 프레임워크는 초심자 또는 무경험 개발자에 의해 사용될 수 있고, 이러한 제 2 프레임워크는 경험자 또는 전문 개발자에 의해 사용될 수 있다. 예를 들어, 제 1 프레임워크는 비주얼 베이직용으로 설계된 것에 대응될 수 있고, 제 2 프레임워크는 C++용으로 설계된 것에 대응될 수 있다.

이러한 통상적 접근에서는, 각 프레임워크의 일부로서, 상대적으로 개별적이고 공통점이 없는 API들이 설계되어 이용된다. 상대적으로 낮은 기술 수준과 컨트롤 용량의 제 1 프레임워크에 대한 API 사용을 가능하게 하기 위해서는 가파르지만 상대적으로 높이가 낮은 학습 곡선이 선회된다. API 프레임워크 2개의 개별적이고 공통점이 없는 특성으로 인해, 제 1 프레임워크에 대한 경험은 제 2 프레임워크의 제 2 API를 학습하고자 하는 어떤 지식에도 전혀 도움이 되지 않는다. 따라서, 제 2 프레임워크에 대한 API 사용을 가능하게 하기 위해서는 마찬가지로 가파르지만 훨씬 높이가 높은 학습 곡선이 선회된다.

다시 말해, 제 1 프레임워크의 API 학습은 제 2 프레임워크의 API 학습을 위한 발판을 제공하지 않는다. 이와 같은 흩어진 세트의 API 프레임워크들에 대한 퇴보적 특성을 연속성 갭(continuity gap)으로 표시한다. 제 1 프레임워크의 API를 학습한 개발자가 제 2 프레임워크의 API 학습에 더 근접한 것이 아니므로 제 2 프레임워크의 기초에서 시작해야 한다.

통상적 프레임워크들에 대한 또 하나의 문제는, 이들이 어떤 경우에서든 전반적으로 불량한 가용성(poor usability)을 갖는 경향이 있다는 것이다. 일반적으로, OOD(object oriented design/development) 방법(예를 들어, UML(unified modeling language))은 얻어진 프레임워크의 가용성에 대해서가 아니라 얻어진 설계의 유지보수성에 대해 "최적화"되어 있다. OOD 방법은 내부 아키텍처 설계에 더 적합하며 재사용가능한 거대 라이브러리의 API 계층 설계에는 덜 적합하다. 예를 들어, 최저 기능 블록화에만 초점을 맞추며 그리고/또는 API 설계를 통한 엄격한 승계 계층 구조에 확고하게 충실한 OOD 방법으로부터 불량한 가용성이 초래될 수 있다.

## 발명이 이루고자 하는 기술적 과제

따라서, 통상적인 API 학습 곡선의 퇴보적인 연속성 갭을 적어도 개선할 수 있으며 그리고/또는 전반적으로 보다 양호한 API 가용성을 전달할 수 있는 방식 및/또는 기술에 대한 필요성이 존재한다.

## 발명의 구성 및 작용

### 요약

제 1의 예시적 방법 구현에서, API를 설계하는 방법은, 코어 시나리오(core scenario)에 대해, 각각이 다수 프로그래밍 언어의 개별적인 프로그래밍 언어에 대응되는 다수 코드 샘플을 준비하는 단계; 및 다수 코어 샘플에 응답하여, 코어 시나리오로부터 API를 유도하는 단계를 포함한다. 제 2의 예시적 방법 구현에서, API를 설계하는 방법은, 특징 영역(feature area)에 대해 코어 시나리오를 선택하는 단계; 코어 시나리오에 대해 하나 이상의 코드 샘플을 기입하는 단계; 및 하나 이상의 코드 샘플에 응답하여, 코어 시나리오에 대한 API를 유도하는 단계를 포함한다. 제 3의 예시적 방법 구현에서, API를 설계하는 방법은, 시나리오에 대해 기입된 하나 이상의 코드 샘플에 응답하여, 시나리오에 대한 API를 유도하는 단계; 다수 개발자들을 활용하여, API에 대해 하나 이상의 가용성 연구를 수행하는 단계; 및 하나 이상의 가용성 연구에 기초하여 API를 수정하는 단계를 포함한다.

다른 방법, 시스템, 접근, 장비, 장치, 매체, API, 과정, 구성 등의 구현을 여기에서 설명한다.

유사하고 그리고/또는 대응되는 태양, 특징, 및 컴포넌트를 참조하기 위해, 도면 전체에 걸쳐 동일한 참조부호들이 사용된다.

### 상세한 설명

도 2는 2개의 상이한 추상화 수준에 대한 예시적인 발전적 API 학습 곡선의 그래프(200)를 도시한다. 도시된 2개의 상이한 추상화 수준은 상대적으로 높은 추상화 수준과 상대적으로 낮은 추상화 수준이다. 높은 추상화 수준은, 요구되는 기술 및/또는 어려움이 상대적으로 낮은 수준인 동시에 개발자에 의한 상대적으로 낮은 컨트롤 용량과 관련이 있는 개발 환경에 대응된다. 한편, 낮은 추상화 수준은, 요구되는 기술 및/또는 어려움이 상대적으로 높은 수준인 동시에 개발자에 의한 상대적으로 높은 컨트롤 용량과 관련이 있는 개발 환경에 대응된다.

발전적 API 학습 곡선은, 요구되는 기술 및 수반되는 컨트롤 용량이 낮은 점으로부터 요구되는 기술 및 수반되는 컨트롤 용량이 높은 점까지, 추상화 수준이 높고 낮은 영역들을 통해 비교적 완만한 방식으로 상승하는 것으로 도시되어 있다. 발전적 API 학습 곡선은 높은 추상화 수준의 영역과 낮은 추상화 수준의 영역 사이에 연속성 구역(continuity zone)을 나타낸다. 통합된 API 프레임워크로 인해 점진적 학습 곡선이 가능해진다. API 프레임워크의 통합된 특성으로 인해, 높은 추상화 수준의 경험이 낮은 추상화 수준에 대해서 뿐만 아니라 더 큰 컨트롤을 요하는 시나리오에 대한 API 이용을 학습하기 위한 지식에 도움이 된다.

다시 말해, 높은 추상화 수준에 대한 API를 학습하는 것이 이 API를 더 낮은 추상화 수준으로 학습하고 그리고/또는 확장하는 발판을 제공한다. 높은 추상화 수준의 영역과 낮은 추상화 수준의 영역 모두를 포함하는 2-계층 (API) 프레임워크 형태로써, 이를 표시한다. 후술하는 바와 같은, 소정 API의 발전적 특성으로 인해 개발자들은, 처음에는 간단한 API를 사용하고 점진적으로 (그리고 부분적으로) 좀더 복잡한 API 컴포넌트의 사용을 시작할 수 있다. 따라서, 높은 추상화 수준을 목표로 하는 API를 학습한 개발자들도, 그들의 경험이 허용하는 바에 따라 그리고/또는 그들이 직면하는 시나리오의 복잡성이 요구하는 바에 따라, 낮은 추상화 수준을 목표로 하는 API를 사용하는 것으로 옮겨갈 수 있다.

발전적 API는 (특히 초기 학습 단계 동안에) 쉽게 사용될 수 있으며 (특히 API가 시간에 걸쳐 조사될 경우) 상당히 강력하다. 사용가능한 API는, 다음과 같은, 시작하기 위해서는 소수의 개념들 및/또는 클래스들이 필요하고, 몇 줄의 코드가 간단한 시나리오들을 구현할 수 있으며, 클래스들/방법들은 직관적인 이름들(intuitive names)을 가지고, 자연스러운 그리고/또는 명백한 시작점이 식별가능하며, 부가적으로 필요한 그리고/또는 관련 개념들/클래스들로의 분명한(예를 들어, 발견가능한) 발전이 존재한다는 예시적 속성들 중에서 하나 이상을 포함할 수 있다.

발전적 API는 또한, 어려움 및 수반되는 컨트롤 용량이 낮은 점에서 개발하는 것으로부터 어려움 및 수반되는 컨트롤 용량이 높은 점으로의 점진적 진보를 가능하게 한다. 여기에서는, 일반적으로 상당히 유용한 API들 뿐만 아니라 발전적 API들을 설계하는 예시적 패러다임들을 후술한다.

도 3은 API들에 대한 예시적 설계 원칙 및 관행을 표(300)로 도시한다. 표(300)는 4개의 예시적 카테고리(302-308)에 대한 일반적 설계 원칙 및 관행을 나타낸다. 구체적으로, 다음의 4개 카테고리: 시나리오 주도 설계(302;scenario driven design), 컴포넌트 지향 설계(304), 개별화가능한 디폴트(306), 및 자체 문서화 오브젝트 모델(308;self documenting object model)이 소개된다.

소정 API를 설계할 경우, 표시된 카테고리들(302-308) 중 하나 이상에 대한 설계 원칙 및 관행이 채택될 수 있다. 또한, 임의의 소정 카테고리(302-308)내에서, 도시된 설계 원칙 및 관행들 중의 하나 이상이 구현될 수 있다. 다시 말해, 소정 API 설계를 위해, 모든 카테고리나 그에 따른 모든 설계 원칙 및 관행이 채택되거나 구현될 필요는 없다.

시나리오 주도 설계의 카테고리(302)는 4개의 예시적 설계 원칙 및 관행을 도시한다. 첫번째, 선택된 특징들 또는 기술 영역들(technological areas)에 대한 코어 시나리오가 정의된다. 두번째, 코어 시나리오에 대응되는 코드 샘플들이 먼저 기입되고, 그에 응답하여 다음으로 API가 설계된다. 세번째, 위에서 소개하고 여기에서 후술하는 발전적 API가 설계된다. 네번째, 다른 시나리오들의 이용이 가능해지면서 정의된 코어 시나리오들의 이용이 용이해진다. "시나리오 주도 설계"라는 제목의 섹션에서 시나리오 주도 설계(302)를 보다 상세히 후술한다.

컴포넌트 지향 설계의 카테고리(304)는 3개의 예시적 설계 원칙 및 관행을 도시한다. 첫번째, AC들(aggregate components)이 작성된다. 일반적으로, AC들은 코어 시나리오들을 위한 것이고, 사용이 상대적으로 간단하고 용이하며, FT들(factored types)의 상부에 구축된다. FT들은 더 기본적이고, 더 낮은 논리 수준으로 분해된다. 이로 인해 2-계층 API 설계가 초래된다. 두번째, 이 AC들은 FT들과 상관된다. 세번째, 특히 AC들에 대해, 작성-설정-호출 사용 패턴이 지원된다. "컴포넌트 지향 설계"라는 제목의 섹션에서 컴포넌트 지향 설계(304)를 보다 상세히 후술한다.

개별화가능한 디폴트 카테고리(306)는 2개의 예시적 설계 원칙 및 관행을 도시한다. 첫번째, 적어도 AC를 사용하기 위해 필요한 초기화가 감소된다. 디폴트는 필요한 초기화를 감소시키는데 사용된다. 두번째, 정의된 코어 시나리오에 적합한 디폴트가 선택된다. "개별화가능한 디폴트"라는 제목의 섹션에서 개별화가능한 디폴트(306)를 보다 상세히 후술한다.

자체 문서화 오브젝트 모델 카테고리(308)는 4개의 예시적 설계 원칙 및 관행을 도시한다. 첫번째, 코어 시나리오에 대해 간단하고 직관적인 이름이 지정된다. 두번째, 승계 계층 구조(inheritance hierarchy)를 완고하게 고수하는 대신에, 컴포넌트 타입의 예정된 사용 또는 목적에 기초하여 이름들이 선택된다. 세번째, 조치가능한 예외들이 취해져 개발자가 예외 메시지로부터의 오류를 수정할 수 있는 방법을 지시하는 명령을 수신할 수 있게 한다. 네번째, 주된 이름 공간(main namespaces)이 클러터링(cluttering)되는 것을 방지하기 위해, 거의 사용되지 않는 타입들을 하부-이름 공간에 배치함으로써, 깨끗한 이름 공간을 생성한다. "자체 문서화 오브젝트 모델"이라는 제목의 섹션에서 자체 문서화 오브젝트 모델(308)을 보다 상세히 후술한다.

## 시나리오 주도 설계

설명한 구현에서는, 시나리오에 의해 API 스펙들(API specifications)이 유도된다. 따라서, API 설계자는 먼저, API의 사용자들이 코어(예를 들어, 메인) 시나리오에 기입해야 할 코드를 기입한다. 그 다음, API 설계자들은 이들 코드 샘플을 지원하는 오브젝트 모델을 설계한다. 이러한 접근은 (다양한 설계 방법을 사용해) 오브젝트 모델의 설계를 시작한 다음, 얻어진 API에 기초해 코드 샘플을 기입하는 것과 대조된다.

다시 말해, 특히 공중 API 설계(public API design)의 경우, API 설계자들은 각각의 특징 또는 기술 영역에 대한 시나리오들과 그에 대한 코드 샘플들의 리스트로 시작하며 그에 기초하여 헤더-스타일의 오브젝트 모델 기술(header-style object model description)을 생성한다. 특징 영역들의 예로는 파일 I/O, 네트워킹, 메시징, 콘솔, 진단, 데이터베이스 액세스, 웹 페이지, GUI(graphical user interface) 프로그래밍 등을 들 수 있다.

도 4는 특징 영역마다 API를 설계하는 예시적 기술을 도시하는 흐름도(400)이다. 블록 402에서는, 소정의 특징 영역에 대해 코어 시나리오가 선택된다. 예를 들어, 소정 기술 영역에 대해, 상위 5-10개의 시나리오가 선택될 수 있다. 이들은 소정 기술 영역에 대해 가장 일반적으로 사용되는 함수(예를 들어, 가장 공통적인 태스크) 또는 가장 빈번하게 추구되는 목표에 기초하여 선택될 수 있다. 예를 들어, 파일 I/O 기술의 특징 영역에 대한 예시적 시나리오들이 파일로부터 판독되고 파일에 기록된다.

블록 404에서는, 코어 시나리오에 대한 코드 샘플들이 다수(예를 들어, 2 이상의) 언어로 기입된다. 예를 들어, 선택된 코어 시나리오와 관련된 코드 샘플들이 3개의 상이한 언어로 기입될 수 있다. 코드 샘플들은 선택된 현재의 코어 시나리오를 3개의 언어로 구현할 수 있다. 이러한 언어로는, 예를 들어, VB, C#, MC++, 마크업 언어 등을 들 수 있지만, 다른 언어들도 사용될 수 있다. 별표로 지시한 바와 같이, 하나의 언어에 대해 사용가능하며 강력한 API를 설계할 경우, 코어 시나리오에 대한 코드 샘플(또는 하나 이상의 코드 샘플)이 하나의 언어로 기입될 수 있다는 것을 알 수 있다.

상이한 언어로 기입된 코드가 때때로 아주 상이하기 때문에, 코드 샘플들을 다수 언어로 기입하는 것이 수행될 수 있다. 설명된 구현에서는, 선택된 현재의 코어 시나리오에 대한 코드 샘플들이, 특정한 코드 샘플이 기입되는 특정 언어의 사용자들 사이에서 일반적인 상이한 코딩 스타일(예를 들어, 언어-특정 특징들 또는 특색들, 개발자들의 관행들/습관들 등)을 사용해 기입된다. 예를 들어, 샘플들이 언어-특정 케이싱(language-specific casing)을 사용해 기입될 수 있다. 예를 들어, VB는 케이스-인센시티브(case-insensitive)이므로, VB로 기입된 코드 샘플들은 이러한 변화성(variability)을 반영한다. 한편, C#으로 기입된 코드 샘플들은 그에 대한 케이싱 표준을 따른다.

또 하나의 예는, C#이 지원하는 "사용(using)"이라는 문장에 관한 것이다. 예를 들어, "사용" 호출은 시도/최종 블록(try/finally block)을 인캡슐레이트(encapsulate)한다. 그러나, VB는 이러한 특징을 지원하지 않으며 코드 샘플의 기입은 이러한 특징을 시도/최종 문장에 이용하는 것이 VB 사용자들에게 불편하다는 것을 지시할 수 있다. 또 다른 예는, C#이 지원하는 조건절의 할당에 관한 것이다. 파일 I/O 인스턴스에서, "if((text=reder.ReadLine() !=null)"은 C#에서 동작한다. 그러나, 할당문은 VB의 "if"절내에서 사용될 수 없으며, 대신에, 이 코드는 다수 문장으로 분할된다. 또 다른 예는, 파라미터화된 구성자들을 이용하는 C# 개발자들의 경향에 관한 것으로, VB 개발자들은 일반적으로 그렇지 않다. 예를 들어, C# 코딩은 "MyClass x = new MyClass("value")"일 수 있는 한편, 대응되는 VB 코딩은 "Dim x As MyClass" 및 "x.Property = "value"이다.

블록 406에서는, 다수 언어로 기재된 코드 샘플에 응답하여, 현재의 코어 시나리오로부터 API가 유도된다. 예를 들어, 다수 언어 각각으로 기재된 코드 샘플로부터 수집된 팩터들이 API에 통합될 수 있다. 이러한 팩터로는 상이한 코드 샘플에 대한 유사점, 2 이상 코드 샘플들간의 차이점 등을 들 수 있다. 도 5를 참조하여, 이러한 팩터들 뿐만 아니라 블록 404 및 406의 다른 태양들을 보다 상세히 설명한다.

마찬가지로, 단일 언어에 대해 API를 설계할 경우, API는 단일 언어로 기재된 코드 샘플(들)에 응답하여 현재의 코어 시나리오로부터 유도된다. 따라서, 단일 언어로 기재된 코드 샘플(들)로부터 수집된 팩터들이 API에 통합될 수 있다. 단일 또는 다수 언어 상황에 대한 부가적 API 설계 팩터의 예로서, API 설계 팩터는 코드 샘플(들)이 기재된 언어 또는 언어들에 적당한 도구들과의 호환성(compatibility)을 포함할 수 있다.

블록 408에서는, API가 지나치게 복잡한지를 판정한다. 예를 들어, API가 지나치게 복잡한지의 여부를 판정하기 위해 API 설계자(들)이 API를 검토할 수 있다. 다시 말해, API가 다수의 다른 특정 API에 대한 상당한 이해, 과도한 실험 등이 없이 사용될 수 있는지를 고려하기 위해, 초기 검사가 수행될 수 있다. 이러한 초기 검사로, 유도된 API가 모든 관련 언어의 현재의 코어 시나리오에서 실제로 동작가능한지를 확인할 수도 있다. API가 지나치게 복잡하면, API는 현재의 코어 시나리오를 참조하여 그리고 블록 406에서 다수 언어로 기재된 코드 샘플에 응답하여 설계자들에 의해 세분된다.

한편, (블록 408에서) API가 지나치게 복잡한 것은 아니라고 판정되면, 블록 410에서는, 통상적 개발자들과의 가용성 연구가 수행된다. 예를 들어, 통상적 개발자가 보통 사용하는 것과 유사한 개발 환경을 사용해, 하나 이상의 가용성 연구가 수행될 수 있다. 이러한 보통의 개발 환경으로는 인텔리센스(intellisense), 에디터, 언어, 및 목표로 하는 개발자 그룹에 의해 가장 널리 사용되는 문서 세트를 들 수 있다.

가용성 연구

광범위한 개발자들을 목표로 하는 가용성 연구는, 특히 일반적 공중 API를 설계할 경우, 시나리오-주도 설계를 용이하게 한다. 코어 시나리오에 대해 API 설계자(들)에 의해 기재된 코어 샘플들이 그들에게는 간단해 보일 수 있지만, 이 코드 샘플들이 사실상 목표 대상이 아닌 소정 그룹의 개발자들(예를 들어, 특히 초심자 및/또는 무경험 개발자들)에게도 동일하게 간단한 것은 아닐 수도 있다. 또한, 가용성 연구를 통해 얻어진, 개발자들이 각각의 코어 시나리오에 접근하는 방식에 관한 이해는 API의 설계 및 API가 대상이 되는 모든 개발자들의 요구를 충족시키는 방법에 대한 강력한 통찰력을 제공할 수 있다.

일반적으로, 가용성 연구는 제품 주기의 초기에 그리고 오브젝트 모델에 대한 임의의 주된 재설계 후에 다시 수행될 수 있다. 이는 비용이 많이 드는 설계 관행이지만, 이것이 실제로는 리소스를 궁극적으로 절감할 수 있다. 파괴적인 변화를 도입하지 않으면서, 사용할 수 없거나 단지 불완전한 API를 수리하는 비용은 엄청나다.

블록 412에서는, 통상적인 개발자들이 큰 문제(들)없이 API를 사용할 수 있는지를 확인한다. 예를 들어, 대부분의 대상자들이 선택된 현재의 시나리오에 대한 코드를 큰 문제없이 기입할 수 있어야 한다. 그럴 수 없다면, API는 (블록 414를 참조하여 후술하는 바와 같이) 수정된다.

중요한/주된 문제의 해석은 목표로 하는 소정 개발자 그룹에 대한 소정 가용성 수준에 달려있다. 예를 들어, 테스트 대상자들에 의한, 현재 코어 시나리오에 대한 API 상세 문서로의 빈번한 그리고/또는 광범위한 참조는 중요한 문제를 구성할 수 있다. 일반적으로, 테스트 개발자들의 대다수가 현재의 코어 시나리오를 구현할 수 없거나 그들이 취하는 접근이 예상했던 바와 크게 다르면, API는 (완전 재설계에 이르며 완전 재설계를 포함하는) 가능한 수정을 위해 평가되어야 한다.

(블록 412에서) 통상적인 개발자들이 큰 문제없이 API를 사용하는 것이 불가능하다고 확인되면, 블록 414에서는, 가용성 연구로부터의 교훈에 기초하여 API가 수정된다. 예를 들어, 디폴트가 변경될 수 있고, 다른 특성이 부가될 수 있으며, 하나 이상의 속성이 인캡슐레이트되는 대신에 노출될 수 있는 등이다.

한편, (블록 412에서) 통상적인 개발자들이 큰 문제없이 API를 사용할 수 있다고 확인되면, 블록 416에서는, 각각의 코어 시나리오에 대해 이 프로세스가 반복된다. 예를 들어, 소정 특징에 대해 선택된 코어 시나리오들 중 또 하나의 코어 시나리오가, (블록 404에서) 코드 샘플들이 기입되는 현재의 코어 시나리오가 된다. 2-계층 이상의 API 설계에 초점을 맞춘, API를 설계하는 또 하나의 예시적 기술을, 도 8을 참조하여 보다 상세히 후술한다.

도 5는 코어 시나리오마다 API를 설계하는 예시적 방식을 도시하는 블록도(404/406)이다. 도시된 예시적 방식은 다수 언어 구현에 대한 도 4의 블록들(404 및 406)에 대응된다. 제 1 언어에 대한 코드 샘플(502(1)), 제 2 언어에 대한 코드 샘플(502(2)), 및 제 3 언어에 대한 코드 샘플(502(3))이 도시되어 있다. 3개의 코드 샘플들(502(1, 2, 3)) 각각은 현재의 소정 코어 시나리오에 대한 것이다. 3개의 언어에 대응되는 3개의 코드 샘플들(502(1, 2, 3))이 도시되어 있지만, 다른 방법으로, 임의 갯수의 목표 언어들에 대한 2 이상의 코드 샘플들(502)이 이러한 예시적 다수-언어 구현에 사용될 수 있다.

설명된 구현에서는, 팩터들(506)이 3개 언어들 각각으로 기재된 코드 샘플들(502(1, 2, 3))로부터 수집된다. 이들 팩터(506)는 API(504)에 통합된다. 특히, API(504)는 3개의 개별적인 대응 언어로 기재된 3개의 코드 샘플들(502(1, 2, 3))을 지원하도록 설계된다. 그러나, 팩터들(506)이 단일-언어 구현에도 적용될 수 있다는 것을 알 수 있다.

도 4의 블록들(404 및 406)을 참조하여 예시적 팩터들(506)의 일부를 설명하며, 다른 예시적 팩터들(506)은 도 5의 블록도(404/406)에 표시되어 있다. 이러한 팩터들(506)은, 코드 샘플들(502(1, 2, 3))의 검토에 의해 드러나는 언어-특정 명령들(language-specific mandates)을 포함한다. 다음의 예시적 코드 라인 "Foo f=new Foo();"에 대해, 언어-특정 제한의 예를 설명한다. 이러한 샘플 라인을 지원하도록 설계된 발전적 API는 디폴트 구성자를 포함해야 하고; 그렇지 않으면, 코드 샘플이 정확하게 컴파일되지 않는다.

또한, 팩터들(506)은 언어 특성들(language peculiarities) 및 자연스럽게 상이한 언어로 끌리는 통상적 개발자들의 상이한 기술/경험 수준 모두에 의해 고무되는 개발자 기대값들을 포함한다. 팩터들(506)은 코드 샘플들(502(1, 2, 3))의 검토에 의해 발견가능한, 상이한 언어에 대한 코드 및 코딩 관행의 공통점들을 더 포함한다.

상이한 언어에 직접적으로 관련되는 팩터들(506)을 고려하면서, 여기에서 설명하는 바와 같은 다른 팩터들(506)도 계속 고려해야 한다. 예를 들어, 다음 팩터들(506)도 단일-언어 환경 뿐만 아니라 다수-언어 환경을 목표로 하는 발전적 API에 관련된다. 첫번째, 시나리오를 완성하기 위해 필요한 상이한 컴포넌트 타입의 갯수가 팩터이다. 일반적으로, 더 많은 컴포넌트 타입이 필요할수록, 학습하기는 더 어렵다. 두번째 팩터는 연속적인 코드 라인들간의 접속이다. 한 컴포넌트 타입의 사용이 필요한 후속 컴포넌트 타입의 사용에 대해 개발자를 선행하는 정도까지, API의 사용이 더 쉬워진다.

세번째, 식별자들의 명명 일관성이 또 하나의 팩터이다. 네번째 팩터는 속성, 방법 및 이벤트의 적절한 사용에 관한 것이다. 다섯번째 팩터는 하나 이상의 기존 API들에 대한 가능한 유사점들에 관한 것이다. 여섯번째, 또 하나의 팩터가 API에 대한 전반적 설계 가이드라인과의 적합성에 관련된다. 일곱번째 팩터는, API가 프레임워크의 다른 컴포넌트 타입과 중복되는지의 여부에 관한 것이다. 여덟번째, 특정 언어를 위한 도구와의 호환성이 또 하나의 팩터이다. 예를 들어, VB 개발자들은 통상적으로 파라미터가 적은 구성자 및 속성 설정자를 원한다. 다른 방법으로, 다른 팩터들(506)이 고려될 수 있다.

특히, 도 8을 참조하여, AC와 FT의 상호관계에 관한 또 다른 팩터들(506)을 후술한다. 도 4 및 도 5의 방법과 방식이 일반적인 API의 설계에 적용될 수 있지만, 이들은 특히 2-계층 API를 설계하는데 이용될 수 있다. "컴포넌트 지향 설계"라는 제목의 섹션에서, 도 6 내지 도 8을 참조하여, (예를 들어, AC 및 FT를 가진) 2-계층 API 패러다임을 후술한다.

**컴포넌트 지향 설계**

도 6은 연속체(600;continuum)를 따라 2개의 상이한 목적을 목표로 하는 예시적 컴포넌트 타입들(602)간의 잠재적 불일치를 도시한다. 연속체(600)는 좌측의 높은 가용성 범위로부터 우측의 높은 제어성 범위로 연장한다. 다수의 컴포넌트 타입들(602)이 연속체(600)에 걸쳐 흩어져 있다.

상대적으로 더 크게 도시되어 있는 컴포넌트 타입들(602)은 간단하여 사용이 보다 용이한 타입들을 나타낸다. 반대로, 상대적으로 더 작게 도시되어 있는 컴포넌트 타입들(602)은 복잡하여 사용이 더 어려운 타입들을 나타낸다. 이 문맥에서의 간단과 복잡은, 구체적인 시나리오를 구현할 때 특정 컴포넌트 타입(602)의 사용이 얼마나 용이한지 또는 어려운지를 나타낸다.

상대적으로 더 작게 도시되어 있는 컴포넌트 타입들(602)은, 다음과 같은 몇가지 예시적 이유로 인해, 일반적으로 사용이 더 어렵다. 첫번째, 개발자들은 그들이 사용할 컴포넌트 타입들(602)에 대해 더 많은 선택권을 가진다. 도시된 예에는, 더 큰 컴포넌트 타입(602)에 대한 3개의 "선택권"에 비해 더 작은 컴포넌트 타입들(602)에 대해서는 14개의 "선택권"이 존재한다. 보다 구체적으로, 개발자는 다양한 컴포넌트 타입들(602(HC)) 중에서 어떤 컴포넌트 타입 또는 타입들을 사용할 것인지를 인지하거나 분별하고 있어야 한다. 이는 다수(예를 들어, 14)의 컴포넌트 타입들(602(HC)) 각각을 이해하는 것 뿐만 아니라, 보다 적은(예를 들어, 3) 컴포넌트 타입들(602(HU)) 중 하나의 컴포넌트 타입(602(HU))으로 시작하는 것과 대조적으로, 이들이 어떻게 상관되는지를 이해하는 것과 관련이 있다. 컴포넌트 타입들(602(HC))과 컴포넌트 타입들(602(HU))간의 차이를 후술한다.

예시적인 유사점으로써, 더 작은 컴포넌트 타입들(602)은 스테레오 시스템의개별적인 컴포넌트들과 유사하므로, 사용자는, 어떤 컴포넌트들이 필요한지 그리고 이들을 함께 후킹(hooking)하는 방법을 인지하고 있어야 한다. 이들을 함께 후킹하지 않으면, 이들은 일반적으로 소용이 없다. 더 큰 컴포넌트 타입들(602)은, 용이하게 사용할 수 있지만 덜 강력할 뿐만 아니라 덜 호환적일 수 있는 올 인 원 스테레오(all-in-one stereos)와 유사하다. 더 작은 컴포넌트 타입들(602)을 사용하기가 더 어려운 두번째 이유는 잠재적으로 더 많은 "시작점들"이 존재하기 때문이다. 세번째, 일반적으로 이해해야 할 개념이 더 많다. 네번째, 개발자는 개별적인 컴포넌트 타입들(602)이 다른 컴포넌트 타입들(602)과 어떻게 관련되는지를 이해해야 한다.

설명된 구현에서, 컴포넌트 타입들(602)은 높은 가용성 목적을 가진 것들(602(HU))과 높은 가제어성 목적을 가진 것들(602(HC))로 나누어진다. 높은 가용성 컴포넌트 타입들(602(HU))은 좀더 간단하고 사용이 용이하지만, 호환성이 떨어지고, 한정적이며, 그리고/또는 제한적인 경향이 있다. 이들은 일반적으로 API 전반에 관한 광범위한 지식없이도 사용될 수 있다. 높은 가용성 컴포넌트 타입들(602(HU))은 일반적으로 제한된 수의 시나리오 또는 기껏해야 관심있는 시나리오 각각에 대한 한정된 수의 태양들을 구현할 수 있다.

한편, 높은 가제어성 컴포넌트 타입들(602(HC))은 사용이 복잡하지만, 개발자들에게 더 큰 제어도(degree of control)를 제공한다. 이들은 상대적으로 강력하며 개발자들로 하여금 낮은-수준의 미세 조정 및 세부 조정을 달성할 수 있게 한다. 그러나, 높은 가제어성 컴포넌트 타입들(602(HC))을 이용하는 개발은, 상대적으로 훨씬 직접적인 시나리오를 구현하기 위해 정확하게 상호연결되어 있는 다수의 높은 가제어성 컴포넌트 타입들(602(HC))에 대한 실현(instantiation)을 가능하게 하는 많은 컴포넌트 타입들(602)에 대한 보다 완전한 이해를 수반한다.

통상적으로, 높은 가용성 컴포넌트 타입들(602(HU))은 VB와 같은 입문 언어에 존재하고, 높은 가제어성 컴포넌트 타입들(602(HC))은 C++과 같은 전문-프로그래머-타입의 고급 언어에 존재한다. 도 6에 도시되어 있는 높은 가용성 컴포넌트 타입들(602(HU))과 높은 가제어성 컴포넌트 타입들(602(HC))간의 잠재적 부등성은 도 7의 컴포넌트 타입들(702)에 의해 적어도 부분적으로 완화된다. 구체적으로, 높은 가용성 컴포넌트 타입들(602(HU))과 높은 가제어성 컴포넌트 타입들(602(HC))간의 상호관계가 발전적 API에 의해 확립된다.

도 7은 연속체(700)를 따라 2 이상의 상이한 목적을 커버하기 위해 확장가능하도록 그리고/또는 상호작용가능하도록 설계되는 컴포넌트 타입들(702)간의 예시적 관계를 도시한다. 설명된 구현에서, 높은 가용성 목적을 가진 컴포넌트 타입들은 AC(702(AC))로서 실현되고, 높은 가제어성 목적을 가진 컴포넌트 타입들은 FT(702(FT))로서 실현된다. 컴포넌트 타입들(702)이 2가지 목적으로만 나누어 지지만, 다른 방법으로, 이들은 3 이상의 목적(또는 다른 카테고리)으로 분리될 수도 있다.

키(704)는, 실선이 노출형 FT(exposed factored types)에 대한 관계를 나타내고 점선이 인캡슐레이트형 FT(encapsulated factored types)에 대한 관계를 나타낸다는 것을 표시한다. 도시된 바와 같이, AC(702(AC))는 3개의 FT(702(FT))와 관계를 가진다. 특히, FT(702(FT)(1))는 AC(702(AC)(1))와 노출형 FT 관계를 가지고, FT(702(FT)(2) 및 FT(702(FT)(3)))는 AC(702(AC)(1))와 인캡슐레이트형 FT 관계를 가진다.

그렇게 도시되어 있지는 않지만, 2 이상의 AC(702(AC))가 동일한 FT(702(FT))와 인캡슐레이트된 그리고/또는 노출된 관계를 가질 수 있다. 노출되고 인캡슐레이트된 FT(702(FT))와 AC(702(AC)) 뿐만 아니라 그들간의 관계를 도 8을 참조하여 보다 상세히 후술한다.

컴포넌트 지향 설계는, 논리적 개념마다 다수의 오브젝트를 필요로 하는 것과 대조적으로, 사용자 개념마다 하나의 오브젝트를 제공하는 것과 관련이 있다. 따라서, AC는 대개 사용자 개념에 대응되며 가용성 관점에서 좀더 간단하다. AC는 FT의 상부에 적층된다. 예시적 비교로써, AC는 파일과 같은 것을 모델링할 수 있고, FT는 파일에 관한 뷰와 같은 것의 상태를 모델링할 수 있다. AC와 FT는 함께, 특히 소정의 특정 API에 대한 새로운 개발자들에게 발전적이고 점진적인 학습 곡선을 제공한다.

AC에 대한 컴포넌트 지향 설계

많은 특징 영역들이 특징 영역 API들의 좀더 복잡하지만 잘-팩터링된 나머지에 대해 간략화된 뷰로서 동작하는 외관 타입들(facade types)로부터 이점을 취할 수 있다. 설명된 구현에서, 외관은 소정 특징 영역의 상부 5-10개 시나리오 및 선택적으로 다른 고급 동작들을 커버한다. AC(702(AC))는 이러한 외관 타입으로 기능할 수 있으며, FT(702(FT))는 나머지의 잘-팩터링된 복잡한 API 가로 방향(landscape)을 제공할 수 있다.

각각의 AC는 팩터링된 다수의 저급 클래스를 탑 코어 시나리오를 지원하는 고급 컴포넌트로 묶는다. 예를 들어, 메일 AC는 SMTP 프로토콜, 소켓, 인코딩 등을 함께 묶을 수 있다. 일반적으로, 각각의 AC는 단지 상이한 방법으로 그 일을 하는 것보다 높은 추상화 수준을 제공한다. 간략화된 고급 동작을 제공하는 것이, 특징 영역에 의해 제공된 기능성 범위 전체의 학습을 원하지 않거나 상당한 연구 또는 API 개발없이 단지 대개는 아주 간단한 그들의 태스크들을 실현하고자 하는 개발자들에게 도움이 된다.

일반적으로, 컴포넌트 지향 설계는 구성자, 속성, 방법, 및 이벤트에 기초하는 설계이다. AC를 사용하는 것은 컴포넌트 지향 설계에 대한 비교적 최후의 애플리케이션이다. AC의 컴포넌트 지향 설계를 위한 파라미터들의 예시적 세트가 다음에 제공된다.

구성자: AC는 디폴트 (파라미터가 없는;parameter-less) 구성자를 가진

다.

구성자: 선택적인 구성자 파라미터는 속성에 대응된다.

속성: 대부분의 속성들은 수집자와 설정자(getters and setters)를 가

진다.

속성: 속성들은 감지가능한 디폴트를 가진다.

방법: 파라미터들이 (선택된 코어 시나리오에서) 방법 호출들을 통해

일정하게 유지되는 옵션을 특정하지 않으면 방법은 파라미터를 취하지 않는

다. 이러한 옵션은 속성을 사용해 특정될 수 있다.

이벤트: 방법은 델리게이트(delegates)를 파라미터로 취하지 않는다.

이벤트의 관점에서 콜백이 구현된다.

컴포넌트 지향 설계는 방법, 속성, 및 이벤트의 오브젝트 모델로의 간단한 포함에 초점을 맞추는 대신에 API가 어떻게 사용되는지에 대한 고려를 수반한다. 컴포넌트 지향 설계를 위한 예시적 사용 모델은 디폴트 또는 상대적으로 간단한 구성자를 가진 타입을 실현하고, 인스턴스상에 동일한 속성들을 설정한 다음, 간단한 방법을 호출하는 패턴과 관련이 있다. 이 패턴을 작성-설정-호출 사용 패턴이라 한다. 일반적인 예는 다음과 같다.

'VB

'Instantiate

Dim T As New T()

'Set properties/options.

T.P1 = V1

T.P2 = V2

T.P3 = V3

'Call methods; optionally change options between calls.

T.M1()

T.P3 = V4

T.M2()

AC가 이러한 작성-설정-호출 사용 패턴을 지원할 경우, AC는 AC의 주 사용자들에 대한 기대치를 따른다. 또한, 인텔리센스 및 설계자와 같은 도구는 이러한 사용 패턴을 위해 최적화된다. 작성-설정-호출 사용 패턴을 나타내는 구체적인 코드 예는 다음과 같다.

'VB

'Instantiate

Dim File As New FileObject()

```
'Set properties.

File.Filename = "c:\foo.txt"

File.Encoding = Encoding.Ascii

'Call methods.

File.Open(OpenMode.Write)

File.WriteLine("Hello World")

File.Close()
```

발전적 API의 일부인 예시적 AC의 경우, "File.Encoding" 속성을 설정하는 것은 옵션이다. API는, 사전-선택된 파일 인코딩이 특정되지 않으면, 그에 대한 디폴트를 가진다. 마찬가지로, "File.Open()"의 경우, "OpenMode.Write"를 특정하는 것은 옵션이다. 이것이 특정되지 않으면, API에 의해 사전-선택된 디폴트 "OpenMode"가 이용된다.

컴포넌트 지향 설계의 쟁점은, 그로 인해 모드 및 무효 상태를 가질 수 있는 타입을 초래된다는 것이다. 예를 들어, 디폴트 구성자로 인해 사용자들은 "FileName"을 특정하지 않고도 "FileObject"를 실현할 수 있다. "FileObject"는 개방되는 것(예를 들어, 파일명이 아직 특정되지 않음)에 대해 무효 상태에 있으므로, "FileName"을 먼저 설정하지 않고 Open()의 호출을 시도하는 것은 예외를 초래한다. 또 하나의 쟁점은, 선택적으로 그리고 독자적으로 설정될 수 있는 속성들이 오브젝트의 상태에 대해 일관적이고 원자적인 변화(consistent and atomic changes)를 강제하지 않는다는 것이다. 또한, 이러한 "모드" 속성("modal" properties)은, 제 2 사용자가 그 값을 임시로 변경시켜 놓은 경우 제 1 사용자는 그것을 재사용하기 전에 이전-설정값을 체크해야 하므로, 소비자들간의 오브젝트 인스턴스 공유를 금지한다. 그러나, AC의 가용성은 굉장히 많은 개발자들에 대한 이 쟁점들을 능가한다.

사용자들이 오브젝트의 현재 상태에서 유효하지 않은 방법을 호출할 경우, "InvalidOperationException"이 취해진다. 이러한 예외 메시지는, 오브젝트를 유효 상태화하기 위해 어떤 속성들이 변경되어야 하는지를 분명하게 설명할 수 있다. 이러한 분명한 예외 메시지들이 무효 상태 문제를 부분적으로 극복하여 보다 자체-문서적인 오브젝트 모델을 초래한다.

API 설계자들은 종종, 오브젝트가 무효 상태로 존재할 수 없는 타입을 설계하고자 시도한다. 예를 들어, 요구되는 모든 설정들을 구성자에 대한 파라미터로서 가지고, 실현후에는 변경될 수 없는 설정들에 대해서만 파라미터를 취하며, 기능을 속성들 및 방법들이 중복되지 않는 개별적인 타입으로 분해하는 것에 의해, 이를 실현한다. 설명된 구현에서는, 이러한 접근이 팩터링된 타입에는 이용되지만, AC에는 이용되지 않는다. AC의 경우, 개발자들에는 무효 상태를 그들에게 통신하는 명백한 예외가 제공된다. 이들 명백한 예외는, 컴포넌트가 초기화될 때가 아니라, 동작이 수행 중인 때에 취해질 수 있으므로, 임시적인 무효 상태가 후속적인 코드 라인에서 "고정되는" 상황을 방지한다.

<u>팩터링된</u> <u>타입(FTs)</u>

상술한 바와 같이, AC는 공통적인 고급 동작 대부분에 대한 쇼트컷을 제공하며 일반적으로, FT(factored types)라 하는, 보다 복잡하지만 동시에 보다 풍부한 타입들의 세트에 대한 외관으로서 구현된다. 설명된 구현에서, FT는 모드를 갖지 않으며 아주 분명한 수명을 가진다.

AC는 몇가지 속성 및/또는 방법을 통해 그 내부 FT로의 액세스를 제공할 수 있다. 사용자는 상대적으로 진보된 시나리오의 또는 시스템의 상이한 부분들과의 통합이 요구되는 시나리오의 내부 FT들에 액세스한다. AC에 의해 사용 중인 FT(들)에 액세스하는 기능으로 인해 AC를 사용해 기재된 코드가 진보된 시나리오에 대한 복잡성을 점진적으로 부가할 수 있거나, FT들을 사용하는 것에 초점을 맞추는 것으로 시작하여 코드를 재기입할 필요없이 다른 컴포넌트 타입들과 통합할 수 있다.

다음의 예는 그 내부의 예시적인 FT("StreamWriter")를 노출하는 예시적 AC("FileObject")를 나타낸다.

```
'VB

Dim File As New FileObject("c:/foo.txt")

File.Open(OpenMode.Write)

File.WriteLine("Hello World")

AppendMessageToTheWorld(File.StreamWriter)

File.Cose()

...
```

Public Sub AppendMessageToTheWorld(Byval Writer As StreamWriter)

...

End Sub

고급 동작들

설명된 구현에서, AC는, (예를 들어, 추상화 수준의 관점에서) 상부 또는 고급 API로서, 사용자가 그 아래에서 간혹 발생하는 복잡한 일들을 인식하지 못한 상태에서 "마술적으로" 동작하는 것처럼 보이도록 구현된다. 예를 들어, "EventLog" AC는, 로그가 사용을 위해 개방되는 판독 핸들 및 기입 핸들 모두를 가진다는 사실을 감춘다. 개발자가 관여할 수 있는 만큼, AC는 실현될 수 있고, 속성은 설정될 수 있으며, 로그 이벤트는 비밀 기능에 대한 관심없이 기재될 수 있다.

일부 상황에서는, 약간의 투명성이 개발자와의 어떤 태스크를 용이하게 할 수 있다. 일례는, 사용자가 동작(operation)의 결과로서 명시적인 액션(action)을 취하는 동작이다. 예를 들어, 파일을 암시적으로 개방한 다음 사용자가 명시적으로 그것을 폐쇄하기를 요구하는 것은 "마술적으로" 동작한다는 원칙을 과도하게 고수하는 것이다. 그럼에도 불구하고, 근면한 API 설계자는 종종 이러한 복잡성 조차도 숨기는 현명한 해결책을 설계할 수 있다. 예를 들어, 파일의 판독이, 파일을 개방하여, 그 내용을 판독하고 그것을 폐쇄하는 하나의 동작으로 구현될 수 있으므로, 사용자는 파일 핸들을 개방하고 폐쇄하는 것에 관한 복잡성으로부터 보호된다.

또한, AC를 사용하는 것이 어떠한 인터페이스 구현, 임의의 구성 파일 변경 등과도 관련되지 않는다. 대신에, 라이브러리 설계자들은 선언된 인터페이스에 대한 디폴트 구현을 제거할 수 있다. 또한, 구성 설정은 옵션으로 감지가능한 디폴트에 의해 뒷받침된다.

도 8은 2-계층 API(800)로 2개의 상이한 목적을 핸들링하기 위한 예시적 AC(702(AC)) 및 관련 FT(702(FT))를 도시한다. AC(702(AC))는 제 1 또는 더 높은 계층을 나타내고, FT(702(FT))는 제 2 또는 더 낮은 계층을 나타낸다. 제 1 계층은 주문형 인터페이스를 가진 제 2 계층상에 효과적으로 구축된다.

도시된 바와 같이, AC(702(AC))는 다수의 AC 멤버들(802)을 포함한다. 구체적으로, AC 멤버들(802(1), 802(2), 802(P)(1), 802(P)(2), 802(M)(1), 802(M)(2), 및 802(M)(3))이 도시되어 있다. AC(702(AC))는 또한 노출형 FT(702(FT-Ex)) 및 인캡슐레이트형 FT(702(FT-En))를 포함한다. 구체적으로, 노출형 FT(702(FT-Ex)(1) 및 702(FT-Ex)(2))와 인캡슐레이트형 FT(702(FT-En)(1) 및 702(FT-En)(2))가 도시되어 있다. FT(702(FT)) 또한 FT 멤버들(804)을 포함한다.

설명된 구현에서, AC(702(AC))는, 예를 들어, 방법 또는 속성일 수 있는 하나 이상의 AC 멤버(802)를 포함한다. 따라서, AC 멤버들(802)은 AC 방법들(802(M)) 및 AC 속성들(802(P))을 포함한다. AC 멤버들(802(1) 및 802(2))과 같은 이 AC 멤버들(802)은 AC(702(AC))에 대해 고유할 수 있다. 다시 말해, AC(702(AC))상에 존재하는 AC 멤버들(802(1) 및 802(2))과 유사한 일부 AC 멤버들(802)은 임의의 FT(702(FT))에 의존하지 않을 수 있다. 다른 방법으로, 일부 AC 멤버들(802)은 하부의 FT(702(FT))에 링크될 수 있다.

FT(702(FT))는 노출형 FT(702(FT-Ex)) 또는 인캡슐레이트형 FT(702(FT-En))일 수 있다. 노출형 FT(702(FT-Ex))는, 소정 AC(702(AC))의 개별적인 AC 멤버들(802)을 사용하거나 통과하지 않으면서, 다른 일반적 컴포넌트 타입들(702(FT 또는 AC))에 의해 또는 다른 일반적 컴포넌트 타입들(702(FT 또는 AC))로 액세스가능한 소정 AC(702(AC))의 FT(702(FT))이다. FT(702(FT-Ex/En))가 (방법 또는 속성인) AC 멤버(802)에 의해 리턴되면, FT(702(FT-Ex))는 노출된다. 그렇지 않으면, FT(702(FT-En))는 인캡슐레이트된다.

다시 말해, AC 멤버(802)는 FT 멤버를 노출시키거나, FT 인스턴스를 리턴할 수 있다. 후자는 노출형 FT(702(FT-Ex))로 발생할 수 있고, 전자는 인캡슐레이트형 FT(702(FT-En))로 발생할 수 있다. 인캡슐레이트형 FT(702(FT-En))는 소정 AC(702(AC))내에 포함되거나 소정 AC(702(AC)) 내부에 위치하는 소정 AC(702(AC))의 FT(702(FT))이다. 각각의 FT(702(FT))는 방법들 및/또는 속성들인 (그 일부가 도 8에 구체적으로 표시되어 있는) 하나 이상의 멤버(804)를 포함할 수 있다.

도시된 바와 같이, 인캡슐레이트형 FT(702(FT-En)(1))의 2개의 방법 멤버들(804)이 AC(702(AC))에 의해 방법 멤버(802(M)(1)) 및 방법 멤버(802(M)(2))로서 노출된다. 인캡슐레이트형 FT(702(FT-En)(2))의 1개의 방법 멤버(804)가 AC(702(AC))에 의해 방법 멤버(802(M)(3))로 노출된다.

노출형 FT(702(FT-Ex)(1)) 자체가 AC(702(AC))의 속성 멤버(802(P)(1))로 노출된다. 마찬가지로, 노출형 FT(702(FT-Ex)(2)) 또한 AC(702(AC))의 속성 멤버(802(P)(2))로 노출된다. 표시된 바와 같이, 노출형 FT(702(FT-Ex)(1))의 FT 멤버(804)는 개별적으로 액세스가능하도록(즉, AC(702(AC))의 개별적인 멤버(802)를 직접적으로 사용하지 않으면서 액세스가능하도록) 노출된다. 따라서, 노출형 FT(702(FT-Ex))의 FT 멤버(804)는, 그것이 AC(702(AC))의 AC 멤버(802)에 의해 개별적으로 노출되지 않은 경우에도, 여전히 액세스가능하다.

따라서, 노출형 FT(702(FT-Ex)(1))의 표시된 멤버(804)는 그에 대한 멤버(802)를 사용하지 않으면서 AC(702(AC))의 외부에 위치하는 컴포넌트 타입들(702)에 의해 액세스가능하도록 노출된다. 노출형 FT(702(FT-Ex)(1))로부터 발산되는 점선으로 표시된 바와 같이, 노출형 FT(702(FT-Ex))는, AC(702(AC))와 상호작용할 수 없거나 핸드오프된 노출형 FT(702(FT-Ex))만을 사용해 그들이 의도하는 목적을 양호하게 실현할 수 있는 다른 컴포넌트 타입들(702; 특히 다른 FT

들(702(FT)))에 의한 사용을 위해 "핸드오프"될 수 있다. "핸드오프"된 오브젝트(노출형 FT(702(FT-Ex))가 노출 중인 AC(702(AC))의 복사본이 아니라 실제 부분임에 주목해야 한다. 따라서, 핸드오프된 오브젝트상의 동작들이 AC(702(AC))에 영향을 미친다.

일반적으로, FT(702(FT))가 인캡슐레이트되면, 이는 소비자에게 노출되지 않는 대신, AC(702(AC))상에서 속성(802(P))을 설정하거나 방법(802(M))을 호출하는 것을 통해 하부 FT(702(FT))상에 FT(702(FT))가 생성되거나 속성(804)이 설정되거나 방법(804)이 호출될 수 있다. 이들 멤버(802 및 804)는 일-대-일 대응을 갖지 않을 수도 있는데, 예를 들어, AC(702(AC))상에서 수개의 속성(802(P))을 설정하는 것이 AC(702(AC))에 캐시될 수 있다. 후속적으로 AC(702(AC))상에서 방법(802(M))을 호출하는 것으로 인해 수개의 속성(802(P))에 대해 미리-특정된 값들을 사용해 FT(702(FT))가 FT(702(FT))에 대한 구성자 인수(constructor arguments)로서 작성될 수 있다.

설명된 구현에서, AC(702(AC))는 노출형 FT(702(FT-Ex))의 노출 이외에 적어도 2가지 측면에서 보다-전통적인 오브젝트-지향형 컴포넌트와 상이하다. 첫번째, AC(702(AC))가 반드시 그의 모든 FT(702(FT))에 대한 모든 멤버(804)를 노출하는 것은 아니다. 다시 말해, AC(702(AC))가 반드시 승계 계층 구조에만 할애되는 것은 아니다. 두번째, AC(702(AC))는 모드를 가질 수 있으므로 주기적으로 무효 동작을 초래하는 상태를 가질 수 있다.

(도 5의) 팩터(506)에 대한 컴포넌트 지향 설계의 가이드라인으로서, FT(702(FT))가 AC(702(AC))내에서 노출형인지 인캡슐레이트형인지의 여부는 다수의 팩터들 중 하나 이상에 기초할 수 있다. 첫번째, 특정 FT(702(FT))가 소정 AC(702(AC))에 의해 노출되지 않는 기능을 포함하면, 특정 FT(702(FT))는 소정 AC(702(AC))의 속성 멤버(802(P))로서 노출된다. 두번째, 프레임워크의 다른 일반적 컴포넌트 타입들(702)이 특정 FT(702(FT))의 직접적 소모를 위한 핸드오프로부터 이점을 취할 수 있다면, 특정 FT(702(FT))는 소정 AC(702(AC))의 속성 멤버(802(P))로서 노출된다. 한편, 특정 FT(702(FT))의 기능이 소정 AC(702(AC))에 의해 완전히 노출되고 특정 FT(702(FT))가 다른 컴포넌트 타입들(702)로의 핸드오프에 유용하지 않으면, 특정 FT(702(FT))는 노출되지 않는다(따라서 인캡슐레이트된다).

개발자는, 특히 보다 간단한 그리고/또는 코너 시나리오를 구현하기 위해, AC(702(AC))로 시작할 수 있다. 개발자가 좀더 복잡한 시나리오를 구현하고자 하거나 구현할 필요가 있으면, 개발자는 시간을 두고, 그에 대한 저급 속성을 포함하는 노출형 FT(702(FT-Ex))에 직접적으로 액세스하고 사용하는 것을 점진적으로 그리고 점차적으로 시작할 수 있다. 좀더 간단한 AC(702(AC))에 의존했던 원래의 코드를 폐기하고 FT(702(FT))에만 의존하는 좀더 복잡한 코딩으로 대체할 필요는 없다. API 프레임워크의 2-계층은 달라지는 비율로 사용될 수 있으며 동시에 존재할 수 있다.

2-계층 API 프레임워크의 설계는 10 단계로 설명된 다음의 예시적 기술을 사용해 실현될 수 있다. 첫번째, 특정한 특징 영역에 대한 한 세트의 코어 시나리오들이 선택된다. 두번째, 선택된 코어 시나리오에 대한 바람직한 코드 라인들을 나타내는 샘플 코드가 기재된다. 세번째, 코드 라인들로부터의 코드 샘플을 지원하기에 적절한 방법, 디폴트, 추상화, 명명 등을 가진 AC가 유도된다.

네번째, 제 2 단계로부터의 코드 샘플들이, 유도된 AC에 따라 적절하게 세분된다. 다섯번째, 세분된 코드 샘플들이 충분히 간단한지의 여부를 위해 이들을 평가한다. 그렇지 않다면, 이 기술은 세번째 단계에서 다시 계속된다. 그렇다면, 여섯번째 단계에서는, 부가적 시나리오, 목적, 다른 컴포넌트와의 상호작용, 및/또는 다른 요구사항의 존재 여부가 판정된다. 일곱번째, API 설계자는 여섯번째 단계에서 발견된 임의의 부가적 요구사항이, 선택된 코어 시나리오에 지나친 복잡성을 부가하지 않으면서, AC에 부가될 수 있는지를 판단한다.

여덟번째, 부가적 요구사항이 AC에 부가될 수 없으면, 일곱번째 단계에 기초하여, FT에 대한 풀 세트 기능의 (예를 들어, 오브젝트-지향 또는 다른 분석 방법에 기초한) 이상적 팩터링(ideal factoring)이 정의된다. 아홉번째, AC가 여덟번째 단계에서 정의된 FT로부터의 기능을 어떻게 그리고 인캡슐레이트하는지 노출하는지의 여부를 판정한다. 열번째, FT가 AC뿐만 아니라 부가적 요구사항을 지원하기에 적절하도록 정의된다. 이러한 예시적 기술을 사용하여, AC(702(AC)) 및 FT(702(FT))를 가진 2-계층 API 프레임워크가 설계될 수 있다.

도 9는 작성-설정-호출 사용 패턴을 지원할 수 있는 예시적 AC(702(AC)) 및 관련 API들(902, 802(P), 802(M), 및 904)을 도시한다. 구성자(902), 속성(802(P)), 방법(802(M)), 및 이벤트(904)의 예시적 API 그룹이 도시되어 있다. 생성, 설정, 호 이용 패턴을 사용하면, 디폴트(예를 들어, 파라미터가 없는) 구성자(902)에 의존하는 개발자에 의해 AC의 인스턴스가 초기에 작성된다.

두번째, 개발자는 디폴트값이 부적절한 그리고/또는 오브젝트의 의도된 사용에 바람직하지 못한 임의의 속성들(802(P))을 설정한다. 세번째, 개발자에 의해 소정 방법(802(M))이 호출된다. 그 다음, 이벤트(904)의 관점에서 콜백이 구현된다.

**개별화가능한 디폴트**

개별화가능한 디폴트들은 적어도 AC에 사용가능할 때마다 디폴트를 갖는 것과 관련이 있다. 다수 언어에 대응되는 다수의 코드 샘플을 가진 예에 대한 API를 설계할 경우, 각각의 코드 샘플에 전달된 동일한 값은 AC에 대한 디폴트로서 대신 설정될 수 있다. 개별화가능한 디폴트는 AC에 관한 하나 이상의 속성들을 설정하는 것에 의해 변경될 수 있다.

많은 개발자들은 문서를 판독하여 프로젝트를 시작하기 전에 특징 영역을 완전히 이해하는데 시간을 쓰기보다는 시행착오에 의한 코딩을 선호한다. 이는 특히 VB로 코딩하는 개발자들과 같은, 초심자 및 무경험 개발자들의 경우가 그러하다. 이들 개발자는 종종 무엇이 어떻게 작용하는지를 발견하기 위해 API로 실험한 다음, API 구현이 그들의 목표를 실현할 때까지 그들의 코드를 천천히 그리고 점차적으로 조정한다. 개발에 대한 편집과 지속적 접근의 대중성은 이러한 선호의 표현이다.

일부 API 설계는 "실험에 의한 코딩"에 적합하고 일부는 그렇지 않다. 실험에 의한 코딩 접근을 사용할 경우, 개발자가 실현하고자 하는 성공 수준에 영향을 미치는 다양한 측면이 존재한다. 이들 측면에는, ( i ) 태스크에 적합한 API를 가까이에

배치하기가 용이한가; (ii) 개발자가 API를 통해 수행하고자 하는 바를 (초기에) 수행하는지의 여부에 상관없이, API를 사용해 시작하기가 용이한가; (iii) API에 대한 개별화의 요점이 무엇인지 발견하기가 용이한가; (iv) 소정 시나리오에 대해 정확한 개별화를 발견하기가 용이한가 등이 포함된다.

설명된 구현에서, API는 초기화를 요구한다 하더라도 적은 초기화(예를 들어, 최소량의 초기화)를 요구하도록 설계된다. 예를 들어, API는, 디폴트 구성자 또는 하나의 간단한 파라미터를 가진 구성자가 타입과의 동작을 시작하기에 충분하도록 설계될 수 있다. 초기화가 필요할 경우, 초기화를 수행하지 않는 것으로부터 초래되는 예외는 이 예외를 제거하거나 방지하기 위해 수행되어야 할 그리고/또는 변경되어야 할 것이 무엇인지를 분명하게 설명한다. 예를 들어, 예외는 무엇이 또는 어떤 속성이 설정되어야 하는지를 조건으로서 요구할 수 있다.

한정이 아닌 일례로써, 가장 간단한 구성자가 (5의 상한을 가지며) 3개 미만의 파라미터를 가진다는 것이 경험칙이다. 또한, 가장 간단한 구성자는 파라미터들 중의 어떤 것으로서 복잡한 타입을 피해야 하는데, 이 경우, 복잡한 타입은 다른 FT 또는 AC일 수 있다. 가장 간단한 구성자는 목록, 스트링, 정수 등과 같은 프리미티브 타입(primitive types)에 의존한다는 것이 또 하나의 경험칙이다. 타입들은 또한 좀더 복잡한 시나리오를 지원하기 위한 좀더 복잡한 구성자 과부하(constructor overloads)를 구현할 수 있다.

간단히 말해, API의 개별화가능성은, 모든 개별화 포인트에 대해 양호한 디폴트를 가진 속성을 제공하는 것에 의해, 간략화될 수 있다. (그러나, 개발자들은 일반적으로, 그들의 시나리오를 개별화할 때, 기존 코드에 새로운 코드를 부가할 수 있어야 하고; 상이한 API를 사용해 스크래치로부터 전체 코드를 재기입하는 것은 옵션이어야 한다.) 예를 들어, 시스템 메시징 큐 AC는, 경로 스트링을 구성자에 전달하고 송신 방법을 호출한 후, 메시지의 송신을 가능하게 한다. 메시지 속성 및 암호화 알고리즘과 같은 메시지 속성들은 코어 시나리오에 코드를 부가하는 것에 의해 개별화될 수 있다.

**자체 문서화 오브젝트 모델**

자체 문서화 오브젝트 모델은, 개발자가 오브젝트와 그것을 학습하고자 하는 멤버들을 볼 수 있을 뿐만 아니라 이들을 사용할 수 있는 API 프레임워크의 설계에 관한 것이다. 예를 들어, 이름은, 타입이, 많은 개발자들이 연구하고 싶어하지 않는 승계 계층 구조를 위한 충당을 대신해 사용될 수 있을 것으로 기대되는 방법에 기초할 수 있다. 간단히 말해, 자체 문서화 오브젝트 모델은 지망 개발자들(would-be developers)에 의한 발견가능성을 용이하게 한다.

상기한 바와 같이, 일부 개발자들은 시행착오에 의한 코딩을 선호하며 그들의 직관이 그들이 의도하는 시나리오를 구현하는데 실패한 경우에만 문서 판독에 의지한다. 따라서, 자체 문서화 오브젝트 모델은, 개발자들이 아주 간단한 태스크를 수행하고자 할 때마다 문서 판독을 요구하는 것을 피해야 한다. 설명된 구현에 대해 상대적으로 자체 문서적인 직관적 API를 형성하는데 도움이 되는 원칙과 관행의 예시적 세트가 아래에 제시되어 있다. 이들 중 하나 이상이 소정 API 및/또는 API 설계 구현에 이용될 수 있다.

명명

제 1 가이드 원칙은, 사용자들이 가장 공통적 시나리오에 사용(예를 들어, 실현)해야 하는 타입들에 대해 간단하고 직관적인 이름을 지정하는 것이다. 설계자들은 종종, 대부분의 사용자들이 관심을 가질 필요가 없는 추상화에 대해 최선의 이름을 소진한다. 예를 들어, 모든 사용자들이 API를 사용해 시작할 수 있기 전에 승계 계층 구조를 이해해야 한다고 기대한다면, 추상화 베이스 클래스를 "File"이라 명명한 다음 구체적인 타입 "XYZFile"을 제공하는 것으로 충분하다. 그러나, 사용자들이 계층 구조를 이해하고 있지 않다면, 이들이 아주 흔하게 잘못 사용할 수 있는 첫번째가 "File" 타입이다. 보다 구체적으로, 가장 공통적이거나 기대되는 이름들이 개념들 및 추상화에 덜 공통적이거나 덜 익숙한 이름들이 사용되는 탑 코어 시나리오를 목표로 하는 AC에 대해 지정된다.

제 2 가이드 원칙은, 각각의 방법이 무엇을 수행하고 각각의 타입 및 파라미터가 무엇을 나타내는지 명백하게 기술하는 서술적인 식별자 이름(descriptive identifier names)을 사용하는 것이다. 예를 들어, API 설계자들은 식별자 이름을 선택할 때 다소 다변이 되는 것을 망설이지 말아야 한다. 예를 들어, "EventLog.DeletEventSource(string source, string machineName)"이 다소 다변적으로 보일 수 있지만, 틀림없이 긍정적인 순 가용성값(net positive usability value)을 가진다. 또한, 타입 및 파라미터 이름은, 타입 또는 파라미터가 무엇을 하는지가 아니라 무엇을 나타내는지 기술해야 한다. 방법 이름은, 그 방법이 무엇을 하는지 기술해야 한다. 물론, 간단하고 분명한 의미를 가진 방법의 경우, 정확하게 다변적인 방법 이름이 좀더 용이한데, 이것이, 따라야 할 양호한 일반적 설계 원칙이 복잡한 의미를 피해야 하는 또 하나의 이유이다.

설계 가이드 관행은 API 스펙 리뷰 및/또는 테스트의 주요한 일부로서 명명 선택에 관한 논의를 포함하는 것이다. 예시적 고려 및 질문은 다음과 같다. 대부분의 시나리오가 시작에 사용하는 타입은 무엇인가? 소정 시나리오의 구현을 시도할 때, 대부분의 사람들이 가장 먼저 생각하는 이름은 무엇인가? 공통적 타입들의 이름들이 사용자들이 가장 먼저 떠올리는 것인가? 예를 들어, 파일 I/O 시나리오를 다룰 때, 대부분의 사람들이 생각하는 이름이 "File"이므로, 파일을 액세싱하는 AC는 "File"로 명명될 수 있다. 또한, 가장 일반적으로 사용되는 타입들 및 그 파라미터들에 대해 가장 일반적으로 사용되는 방법을 리뷰하고 테스트한다. 예를 들어, 고려 중인 특정 API 설계가 아니라 그 기술에 익숙한 사람은 누구든지 그 방법들을 빨리, 정확하게, 그리고 용이하게 인식하고 호출할 수 있는가?

예외들

상기한 바와 같이, 예외는 자체-문서화 API를 용이하게 할 수 있다. 다시 말해, API는 사용자가 다음으로 수행해야 할 것을 안내해야 하고, 예외들은 다음으로 필요한 것을 전달할 수 있으며 전달하기에 적합하다. 예를 들어, 다음의 샘플 코드는 "The 'FileName' property needs to be set before attempting to open the 'FileObject'."의 메시지를 가진 예외를 표시한다.

```
'VB

'Instantiate

Dim File As New FileObject()

'The file name is not set.

File.Open()
```

강 자료형(strong typing)

직관적 API를 용이하게 하는 또 하나의 가이드 원칙은 강 자료형이다. 예를 들어, "Customer.Name"을 호출하는 것이 "Customer.Properties['Name']"을 호출하는 것보다 용이하다. 또한, 이러한 "Name" 속성이 이름을 스트링으로 리턴하게 하는 것은, 속성이 오브젝트로 리턴될 경우보다 훨씬 유용하다.

스트링 기반 접근자(string based accessor), 늦은 바인드 호출(late bind calls) 등을 가진 속성 주머니(property bags)가 API를 강 자료형화하지 않는 것이 바람직한 경우가 존재하지만, 이들은 규칙이 아니라 희귀한 것으로 분류된다. 또한, API 설계자들은, 사용자가 강 자료형이 아닌 API 계층에 대해 수행하는 좀더 일반적인 동작을 위해 강 자료형의 헬퍼(strongly typed helpers)를 제공할 수 있다. 예를 들어, 고객 타입은 속성 주머니를 가질 수 있지만, "이름", "주소" 등과 같이 좀더 일반적인 속성을 위해 강 자료형의 API를 추가적으로 제공할 수 있다.

간단화를 위한 벡터링

또 하나의 가이드 원칙은 간략화, 특히 코어 시나리오에 대한 간략화를 위해 노력하는 것이다. 표준 설계 방법은, 추상화를 사용함으로써와 같이, 유지보수성을 위해 최적화된 설계를 생성하는 것이 목표이다. 따라서, 현대의 설계 방법은 많은 추상화를 생성한다. 쟁점은, 이러한 설계 방법이, 아주 간단한 시나리오의 구현이 시작되기도 전에, 결과적인 설계의 사용자들이 그 설계의 전문가들이 될 것이라는 가정하에서 운영된다는 점이다. 그러나, 실제로 이는 흔한 경우가 아니다.

설명된 구현에서, 적어도 간단한 시나리오의 경우, API 설계자들은 오브젝트 모델의 계층 구조가 충분히 간단하여 이들이, 전체 특징 영역이 어떻게 합치되거나 상호작용하는지를 이해할 필요없이, 사용될 수 있다는 것을 보증한다. 얻어진 잘-설계된 API는, 개발자가 구현 중인 코어 시나리오를 이해할 것을 요구할 수 있지만, 이것이 구현에 사용 중인 라이브러리 설계에 대한 완벽한 이해를 요하는 것은 아니다.

일반적으로, 코어 시나리오 API는 추상화를 대신하여 시스템의 물리적 또는 널리-공지된 논리적 부분에 대한 것이거나 그에 대응된다. 추상화에 대응되는 타입들은 일반적으로 특징 영역의 모든 부분들이 어떻게 합치되어 상호작용하는지에 대한 이해없이 사용하기가 어려우므로, 이들은 교차-특징 통합(cross-feature integration)이 필요할 때 더 크게 관련된다.

또 하나의 가이드 관행은, 코어 또는 공통 시나리오에 대한 API(예를 들어, AC를 가진 것들)를 설계할 때는 아니지만, 내부 아키텍처 및 일부의 FT를 설계할 때 표준 설계 방법(예를 들어, UML)을 사용하는 것이다. 코어 시나리오에 대한 AC를 설계할 때는, 대신에, (여기에서 상술한 바와 같은) 프로토타이핑(prototyping), 가용성 연구, 및 반복과 함께 시나리오 주도 설계가 이용된다.

순수한 이름 공간(clean namespaces)

또 하나의 가이드 원칙은, 주된 이름 공간의 클러터링을 방지하기 위해, (아주) 드물게 사용되는 타입들을 하부-이름 공간에 배치하는 것이다. 예를 들어, 허가 타입들 및 설계 타입들의 2개 그룹의 타입들이 그들의 주된 이름 공간으로부터 분리될 수 있다. 예를 들어, 허가 타입들은 ".Permissions"의 하부-이름 공간에 상주할 수 있고, 설계-시의 타입들은 ".Design"의 하부-이름 공간에 상주할 수 있다.

도 1 내지 도 9의 액션, 측면, 특징, 컴포넌트 등이 다수 블록으로 분할된 도면에 도시되어 있다. 그러나, 도 1 내지 도 9에 설명되고 그리고/또는 도시되어 있는 순서, 상호접속, 상관관계, 레이아웃 등이 한정으로 해석되어서는 안되며, 임의 갯수의 블록이, API를 설계하기 하기 위한 하나 이상의 시스템, 방법, 장치, 과정, 매체, API, 장치, 구성 등을 구현하기 위한 임의의 방식으로 수정, 조합, 재배열, 확대, 생략 등이 될 수 있다. 또한, 여기에서의 설명이 특정 구현(및 도 10의 예시적 동작 환경)에 대한 참조를 포함하긴 하지만, 도시되고 그리고/또는 설명된 구현들이 적당한 임의의 하드웨어, 소프트웨어, 펌웨어 또는 이들의 조합으로 그리고 적당한 임의의 소프트웨어 아키텍처(들), 코딩 언어(들), 시나리오 정의(들), 가용성 연구 포맷(들) 등을 사용해 구현될 수 있다.

**컴퓨터 또는 다른 장치를 위한 예시적 동작 환경**

도 10은, 여기에서 설명된 바와 같은 API를 설계하기 위한 하나 이상의 시스템, 장치, 장비, 컴포넌트, 구성, 프로토콜, 접근, 방법, 과정, 매체, API, 그들의 어떤 조합 등을 (완전히 또는 부분적으로) 구현할 수 있는 예시적 컴퓨팅 (또는 범용 장치) 동작 환경(1000)을 도시한다. 동작 환경(1000)은 후술하는 컴퓨터 및 네트워크 아키텍처에 이용될 수 있다.

예시적 동작 환경(1000)은 환경의 일례일 뿐이며 (컴퓨터, 네트워크 노드, 오락 장치, 모바일 어플라이언스, 범용 전자 장치 등을 포함하는) 응용가능 장치 아키텍처의 사용 또는 기능 범위를 제한하려는 것이 아니다. 동작 환경(1000; 또는 그 장치들)은 도 10에 도시된 바와 같은 컴포넌트들 중 어느 하나 또는 그들의 임의 조합에 관한 어떤 의존이나 요구사항을 갖는 것으로 해석되어서는 안된다.

또한, API의 설계 및/또는 그로부터 얻어지는 API는 (컴퓨팅 시스템을 포함하는) 다수의 다른 범용 또는 특수 목적 장치 환경 또는 구성으로 구현될 수도 있다. 사용하기에 적합할 수 있는, 널리 공지되어 있는 장치, 시스템, 환경, 및/또는 구성의 예로는 퍼스널 컴퓨터, 서버 컴퓨터, 씬 클라이언트(thin clients), 씩 클라이언트(thick clients), PDA(personal digital assistants) 또는 휴대 전화, 워치, 핸드-헬드 또는 랩탑 장치, 멀티프로세서 시스템, 마이크로프로세서-기반 시스템, 셋톱 박스, 프로그램가능한 상용 전자제품, 비디오 게임 머신, 게임 콘솔, 포터블 또는 핸드헬드 게임 유닛, 네트워크 PC, 미니 컴퓨터, 메인프레임 컴퓨터, 네트워크 노드, 상기 시스템 또는 장치들 중 임의의 것을 포함하는 분산형 또는 멀티프로세서 컴퓨팅 환경, 이들의 어떤 조합 등을 들 수 있지만, 이에 한정되는 것은 아니다.

API의 설계 및/또는 그로부터 얻어지는 API에 대한 구현을 프로세서-실행가능 명령의 일반적인 문맥으로 설명할 수 있다. 일반적으로, 프로세서-실행가능 명령은, 특정한 태스크를 수행하고 그리고/또는 가능하게 하는 그리고/또는 특정한 추상적 데이터형을 구현하는 루틴, 프로그램, 모듈, 프로토콜, 오브젝트, 인터페이스, 컴포넌트, 데이터 구조 등을 포함한다. 여기에서 소정 구현으로 설명된 바와 같은, API의 설계 및/또는 그로부터 얻어지는 API는, 통신 링크 및/또는 네트워크를 통해 접속되어 있는 원격적으로-링크된 프로세싱 장치들에 의해 태스크가 수행되는 분산형 컴퓨팅 환경에서 실행되고 그리고/또는 존재할 수 있다. 한정이 아닌 일례로써, 분산형 컴퓨팅 환경에서, 프로세서-실행가능 명령들은 상이한 프로세서에 의해 실행되는 개별적 저장 매체에 위치하거나 그리고/또는 전송 매체를 통해 전파될 수 있다.

예시적 동작 환경(1000)은, 컴퓨팅/프로세싱 기능을 가진 임의(예를 들어, 전자) 장치를 구비할 수 있는 컴퓨터(1002) 형태의 범용 컴퓨팅 장치를 포함한다. 컴퓨터(1002)의 컴포넌트로는 하나 이상의 프로세서 또는 프로세싱 유닛(1004), 시스템 메모리(1006), 및 프로세서(1004)를 포함하는 다양한 시스템 컴포넌트들을 시스템 메모리(1006)에 결합하는 시스템 버스(1008)를 들 수 있지만, 이에 한정되는 것은 아니다.

프로세서(1004)는, 프로세서를 형성하고 있는 재료 또는 그에 채택된 프로세싱 메커니즘에 의해 한정되지 않는다. 예를 들어, 프로세서(1004)는 반도체(들) 및/또는 트랜지스터(예를 들어, 전자 IC(ntegrated circuits))로 구성될 수 있다. 이러한 문맥에서, 프로세서-실행가능 명령들은 전자적으로-실행가능한 명령들일 수 있다. 다른 방법으로, 프로세서(1004)의 또는 프로세서(1004)에 대한, 그리고 그에 따른 컴퓨터(1002)의 또는 컴퓨터(1002)에 대한 메커니즘은 양자 컴퓨팅, 광학 컴퓨팅, (예를 들어, 나노 기술을 사용하는) 기계적 컴퓨팅 등을 포함할 수 있지만, 이에 한정되는 것은 아니다.

시스템 버스(1008)는 메모리 버스 또는 메모리 컨트롤러, 점-대-점 접속, 스위칭 파이버(switching fabric), 주변장치 버스, 가속 그래픽 포트, 및 다양한 버스 아키텍처들 중 하나를 사용하는 프로세서 또는 로컬 버스를 포함하는 많은 타입의 유선 또는 무선 버스 구조들 중 하나 이상을 나타낸다. 일례로써, 이러한 버스 아키텍처로는 ISA(Industry Standard Architecture) 버스, MCA(Micro Channel Architecture) 버스, EISA(Enhanced ISA) 버스, VESA(Video Electronics Standards Association) 로컬 버스, 및 Mezzanine 버스라고도 하는 PCI(peripheral Component Interconnects) 버스, 이들의 일부 조합 등을 들 수 있다.

컴퓨터(1002)는 통상적으로 다양한 프로세서-액세스가능 매체를 포함한다. 이러한 매체는 컴퓨터(1002) 또는 다른(예를 들어, 전자) 장치에 의해 액세스될 수 있는 임의의 가용 매체일 수 있으며, 휘발성 및 비휘발성 매체, 분리형 및 비분리형 매체, 그리고 저장 및 전송 매체 모두를 포함한다.

시스템 메모리(1006)는 RAM(random access memory;1040)과 같은 휘발성 메모리 및/또는 ROM(read only memory;1012)과 같은 비휘발성 메모리 형태의 프로세서-액세스가능 저장 매체를 포함한다. 스타트-업(start-up) 동안과 같은 때에, 컴퓨터(1002)내의 요소들 사이에서의 정보 전달을 돕는 기본적 루틴을 포함하는 BIOS(basic input/output system;1014)는 통상적으로 ROM(1012)에 저장된다. RAM(1010)은 통상적으로, 프로세싱 유닛(1004)으로 즉시 액세스될 수 있거나 그리고/또는 프로세싱 유닛(1004)에 의해 현재 연산되고 있는 데이터 및/또는 프로그램 모듈들/명령들을 포함한다.

컴퓨터(1002)는 다른 분리형/비분리형 및/또는 휘발성/비휘발성 저장 매체를 포함할 수도 있다. 일례로써, 도 10은 (별도로 나타내지 않은) (통상적인) 비분리형, 비휘발성 자기 매체로부터 판독하고 그에 기입하는 하드 디스크 드라이브 또는 디스크 드라이브 어레이(1016); (통상적인) 분리형, 비휘발성 자기 디스크(1020;예를 들어, "플로피 디스크")로부터 판독하고 그에 기입하는 자기 디스크 드라이브(1018); 및 CD, DVD 또는 다른 광학 매체와 같은 (통상적인) 분리형, 비휘발성 광학 디스크(1024)로부터 판독하고 그리고/또는 그에 기입하는 광학 디스크 드라이브(1022)을 도시한다. 하드 디스크 드라이브(1016), 자기 디스크 드라이브(1018) 및 광학 디스크 드라이브(1022)는 하나 이상의 저장 매체 인터페이스(1026)에 의해 시스템 버스(1008)에 각각 접속되어 있다. 다른 방법으로, 하드 디스크 드라이브(1016), 자기 디스크 드라이브(1018) 및 광학 디스크 드라이브(1022)는 (나타내지 않은) 하나 이상의 개별적이거나 통합된 다른 인터페이스에 의해 시스템 버스(1008)에 접속될 수도 있다.

디스크 드라이브 및 그와 관련된 프로세서-액세스가능 매체는 데이터 구조, 프로그램 모듈, 및 컴퓨터(1002)에 대한 다른 데이터와 같은, 프로세서-실행가능 명령의 비휘발성 저장을 제공한다. 예시적 컴퓨터(1002)는 하드 디스크(1016), 분리형 자기 디스크(1020) 및 분리형 광학 디스크(1024)를 도시하지만, 다른 타입의 프로세서-액세스가능 매체가 자기 카세트 또는 다른 자기 저장 장치, 플래시 메모리, 컴팩트 디스크(CD), DVD(digital versatile disks) 또는 다른 광학 저장 장치, RAM, ROM, EEPROM(electrically-erasable programmable read-only memories) 등과 같은 장치에 의해 액세스될 수 있는 명령들을 저장할 수 있다는 것을 알 수 있다. 이러한 매체는 또한, 소위 특수 목적 또는 하드-배선형 IC 칩을 포함할 수 있다. 다시 말해, 임의의 프로세스-액세스가능 매체가 예시적 동작 환경(1000)의 저장 매체를 실현하는데 이용될 수 있다.

일반적인 예로써, 오퍼레이팅 시스템(1028), 하나 이상의 애플리케이션 프로그램(1030), 다른 프로그램 모듈(1032), 및 프로그램 데이터(1034)를 포함하는 임의 갯수의 프로그램 모듈(또는 API 프레임워크 및/또는 그에 기초하는 오브젝트를 포함하는 명령들/코드의 다른 유닛 또는 세트)이 하드 디스크(1016), 자기 디스크(1020), 광학 디스크(1024), ROM(1012), 및/또는 RAM(1040)에 저장될 수 있다.

사용자는 키보드(1036) 및 포인팅 장치(1038; 예를 들어, "마우스")와 같은 입력 장치를 통해 명령 및/또는 정보를 컴퓨터(1002)에 입력할 수 있다. (구체적으로 나타내지 않은) 다른 입력 장치(1040)로는 마이크로폰, 조이스틱, 게임 패드, 위성 안테나, 직렬 포트, 스캐너 등을 들 수 있다. 이들 및 다른 입력 장치들은 시스템 버스(1008)에 결합되어 있는 입출력 인터페이스(1042)를 통해 프로세싱 유닛(1004)에 접속되어 있다. 그러나, 입력 장치 및/또는 출력 장치들은 대신에, 병렬 포트, 게임 포트, USB(universal serial bus) 포트, 적외선 포트, IEEE 1394 ("Firewire") 인터페이스, IEEE 802.11 무선 인터페이스, Bluetooth?? 무선 인터페이스 등과 같은, 다른 인터페이스 및 버스 구조에 의해 접속될 수도 있다.

모니터/뷰 스크린(1044) 또는 다른 타입의 디스플레이 장치 또한, 비디오 어댑터(1046)와 같은 인터페이스를 통해 시스템 버스(1008)에 접속될 수 있다. 비디오 어댑터(1046)(또는 다른 컴포넌트)는 그래픽-인센시티브 계산을 프로세싱하고 디스플레이 요구사항을 처리하는 그래픽 카드이거나 그래픽 카드를 포함할 수 있다. 통상적으로, 그래픽 카드는 그래픽의 신속한 디스플레이 및 그래픽 동작의 성능을 용이하게 하는 GPU(graphics processing unit), 비디오 RAM(VRAM) 등을 포함한다. 모니터(1044) 이외에, 입출력 인터페이스(1042)를 통해 컴퓨터(1002)에 접속될 수 있는 다른 출력 주변 장치로는 (나타내지 않은) 스피커 및 프린터(1048)와 같은 컴포넌트를 들 수 있다.

컴퓨터(1002)는, 원격 컴퓨팅 장치(1050)와 같은, 하나 이상의 원격 컴퓨터로의 논리적 접속을 사용하는 네트워크 환경에서 동작할 수도 있다. 일례로써, 원격 컴퓨팅 장치(1050)는 퍼스널 컴퓨터, 휴대용 컴퓨터(예를 들어, 랩탑 컴퓨터, 태블릿 컴퓨터, PDA, 이동국 등), 팜 또는 포켓-사이즈형 컴퓨터, 워치, 게임 장치, 서버, 라우터, 네트워크 컴퓨터, 피어 장치, 다른 네트워크 노드, 또는 위에서 열거한 바와 같은 다른 장치 타입 등일 수 있다. 그러나, 원격 컴퓨팅 장치(1050)는 컴퓨터(1002)와 관련하여 여기에서 설명한 요소들 및 사양들 중의 많은 것을 또는 그 전부를 포함할 수 있는 휴대용 컴퓨터로서 도시되어 있다.

컴퓨터(1002)와 원격 컴퓨터(1050)간의 논리적 접속은 LAN(local area network;1052) 및 일반적인 WAN(wide area network;1054)으로 도시되어 있다. 이러한 네트워킹 환경은 사무실, 기업-범위의 컴퓨터 네트워크, 인트라넷, 인터넷, 고정 또는 이동 전화 네트워크, 게임 네트워크, 이들의 일부 조합 등에서 흔히 볼 수 있다. 이러한 네트워크 및 통신 접속은 전송 매체의 예들이다.

LAN 네트워킹 환경에 구현될 경우, 컴퓨터(1002)는 일반적으로, 네트워크 인터페이스 또는 어댑터(1056)를 통해 LAN(1052)에 접속된다. WAN 네트워킹 환경에 구현될 경우, 컴퓨터(1002)는 통상적으로 모뎀(1058) 또는 WAN(1054)에 대해 통신을 확립하는 다른 컴포넌트를 포함한다. 컴퓨터(1002) 내장형이거나 외장형일 수 있는 모뎀(1058)은 입출력 인터페이스(1042) 또는 임의의 적절한 타메커니즘(들)을 통해 시스템 버스(1008)에 접속될 수 있다. 도시된 네트워크 접속은 예시적인 것이며 컴퓨터들(1002 및 1050)간에 통신 링크(들)를 확립하는 다른 방식이 이용될 수 있다는 것을 알 수 있다.

동작 환경(1000)으로 도시된 바와 같은 네트워크 환경에서, 컴퓨터(1002)와 관련하여 도시된 프로그램 모듈이나 다른 명령들, 또는 그 일부는 원격 매체 저장 장치에 완전히 또는 부분적으로 저장될 수 있다. 일례로써, 원격 애플리케이션 프로그램(1060)은 원격 컴퓨터(1050)의 메모리 컴포넌트에 상주하지만 컴퓨터(1002)에 의해 이용가능하거나 액세스가능할 수 있다. 또한, 예시적인 목적을 위해, 애플리케이션 프로그램(1030) 및 오퍼레이팅 시스템(1028)과 같은 다른 프로세서-실행가능 명령을 여기에서는 개별적인 블록으로 도시하지만, 이러한 프로그램, 컴포넌트, 및 다른 명령들이 다양한 시기에 컴퓨팅 장치(1002; 및/또는 원격 컴퓨팅 장치(1050))의 상이한 저장 컴포넌트에 상주하며 컴퓨터(1002)의 프로세서(들)(1004; 및/또는 원격 컴퓨팅 장치(1050)의 프로세스들)에 의해 실행된다는 것을 알 수 있다.

시스템, 매체, 장치, 방법, 과정, 장비, 기술, API, 방식, 접근, 구성, 및 다른 구현들이 구성적, 논리적, 알고리즘적, 기능적, 및 액션-기반적 사양 및/또는 도면에 대해 특징적인 언어로 설명되었지만, 첨부된 청구항에 정의된 바와 같은 본 발명이 반드시, 설명된 특정 사양 또는 도면에 한정되는 것은 아니라는 것을 알 수 있다. 오히려, 특정 사양 및 도면들은 청구된 발명을 구현하는 예시적인 형태로서 개시되어 있다.

### 발명의 효과

따라서, 본 발명에 따르면, 통상적인 API 학습 곡선의 퇴보적인 연속성 갭을 적어도 개선할 수 있으며 그리고/또는 전반적으로 보다 양호한 API 가용성을 전달할 수 있는 방식 및/또는 기술이 제공된다.

### (57) 청구의 범위

## 청구항 1.

코어 시나리오에 대해, 각각이 복수개의 프로그래밍 언어들 중의 개별적인 프로그래밍 언어에 대응되는 복수개의 코드 샘플들을 준비하는 단계; 및

상기 복수개의 코드 샘플들에 응답하여, 상기 코어 시나리오로부터 API(application programming interface)를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 2.

제 1 항에 있어서,

API 설계자에 의해, 상기 유도된 API가 지나치게 복잡한지를 판정하는 단계를 더 포함하는 API 설계 방법.

## 청구항 3.

제 2 항에 있어서,

상기 유도된 API가 지나치게 복잡한 것으로 판정되면,

상기 API 설계자에 의해, 세분된 API를 생성하기 위해 상기 유도된 API를 세분하는 단계를 더 포함하는 API 설계 방법.

## 청구항 4.

제 3 항에 있어서,

상기 API 설계자에 의해, 상기 세분된 API가 지나치게 복잡한지를 판정하는 단계를 더 포함하는 API 설계 방법.

## 청구항 5.

제 1 항에 있어서,

복수의 개발자들을 활용하여, 상기 API에 대해 하나 이상의 가용성 연구를 수행하는 단계를 더 포함하는 API 설계 방법.

## 청구항 6.

제 5 항에 있어서,

상기 수행하는 단계는,

상기 복수개의 프로그래밍 언어에 익숙한 상기 복수의 개발자들을 활용하여, 상기 API에 대해 상기 하나 이상의 가용성 연구를 수행하는 단계를 포함하는 API 설계 방법.

## 청구항 7.

제 5 항에 있어서,

상기 복수의 개발자들이 큰 문제없이 상기 API를 사용할 수 있는지의 여부를 확인하는 단계를 더 포함하는 API 설계 방법.

## 청구항 8.

제 7 항에 있어서,

상기 복수의 개발자들이 큰 문제없이 상기 API를 사용할 수 있는 것으로 확인되지 않으면, 상기 API를 수정하는 단계를 더 포함하는 API 설계 방법.

## 청구항 9.

제 8 항에 있어서,

상기 수정하는 단계는,

상기 하나 이상의 가용성 연구로부터의 하나 이상의 교훈에 기초하여, 상기 API를 수정하는 단계를 포함하는 API 설계 방법.

## 청구항 10.

제 1 항에 있어서,

상기 유도하는 단계는,

상기 복수개의 프로그래밍 언어들에 개별적으로 대응되는 상기 복수개의 코드 샘플들을 지원하도록, 상기 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 11.

제 1 항에 있어서,

상기 유도하는 단계는,

상기 복수개의 코드 샘플들로부터 언어-특정 명령들을 수집하는 단계; 및

상기 언어-특정 명령들을 상기 API에 통합하는 단계를 포함하는 API 설계 방법.

## 청구항 12.

제 1 항에 있어서,

상기 유도하는 단계는,

상기 복수개의 코드 샘플들로부터 언어-고무형 개발자 기대값들(language-inspired developer expectations)을 수집하는 단계; 및

상기 언어-고무형 개발자 기대값들을 상기 API에 통합하는 단계를 포함하는 API 설계 방법.

## 청구항 13.

제 1 항에 있어서,

상기 유도하는 단계는,

상기 복수개의 코드 샘플들로부터 공통점들을 수집하는 단계; 및

상기 공통점들을 상기 API에 통합하는 단계를 포함하는 API 설계 방법.

## 청구항 14.

제 1 항에 있어서,

상기 유도하는 단계는,

복수개의 저급 FT들(factored types)을 상기 코어 시나리오를 지원하도록 함께 묶는 AC(aggregate component)를 갖도록 상기 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 15.

특징 영역에 대해 코어 시나리오를 선택하는 단계;

상기 코어 시나리오에 대해 하나 이상의 코드 샘플을 기재하는 단계; 및

상기 하나 이상의 코드 샘플에 응답하여, 상기 코어 시나리오에 대한 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 16.

제 15 항에 있어서,

상기 선택하는 단계는,

상기 특징 영역에 대해 복수개의 코어 시나리오들을 선택하는 단계를 포함하는 API 설계 방법.

## 청구항 17.

제 16 항에 있어서,

상기 특징 영역에 대해 선택된 상기 복수개의 코어 시나리오들 중의 각 코어 시나리오에 대해, 상기 기입하는 단계 및 상기 유도하는 단계를 반복하는 단계를 더 포함하는 API 설계 방법.

## 청구항 18.

제 15 항에 있어서,

상기 기재하는 단계는,

상기 코어 시나리오에 대해, 각각이 복수개의 프로그래밍 언어들 중의 개별적인 프로그래밍 언어에 대응되는 복수개의 코드 샘플들을 기재하는 단계를 포함하는 API 설계 방법.

## 청구항 19.

제 18 항에 있어서,

상기 유도하는 단계는,

상기 복수개의 코드 샘플들에 응답하여, 상기 코어 시나리오에 대한 상기 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 20.

제 15 항에 있어서,

복수의 개발자들을 활용하여, 상기 API에 대해 하나 이상의 가용성 연구를 수행하는 단계;

상기 복수의 개발자들이 큰 문제없이 상기 API를 사용할 수 있는지의 여부를 확인하는 단계; 및

상기 복수의 개발자들이 큰 문제없이 상기 API를 사용할 수 있는 것으로 확인되지 않으면, 상기 API를 수정하는 더 포함하는 API 설계 방법.

## 청구항 21.

제 20 항에 있어서,

상기 수정하는 단계는,

수정된 API를 생성하기 위해, 상기 하나 이상의 가용성 연구로부터의 하나 이상의 교훈에 기초하여, 상기 API를 수정하는 단계를 포함하는 API 설계 방법.

## 청구항 22.

제 21 항에 있어서,

상기 수행하는 단계 및 상기 확인하는 단계를 상기 수정된 API에 대해 반복하는 단계를 더 포함하는 API 설계 방법.

## 청구항 23.

제 15 항에 있어서,

상기 유도하는 단계는,

AC 및 복수개의 하부 FT들을 포함하는 2-계층 API를 생성함으로써, 상기 코어 시나리오에 대해 기재된 상기 하나 이상의 코드 샘플을 지원하도록 상기 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 24.

제 15 항에 있어서,

상기 유도하는 단계는,

상기 하나 이상의 코드 샘플로부터 하나 이상의 언어-특정 명령들을 수집하는 단계; 및

상기 하나 이상의 언어-특정 명령들을 상기 API에 통합하는 단계를 포함하는 API 설계 방법.

## 청구항 25.

제 15 항에 있어서,

상기 유도하는 단계는,

특정 FT의 모든 멤버들이 AC에 의해 노출되면, 상기 특정 FT를 상기 코어 시나리오와 관련되어 있는 상기 AC로 인캡슐레이트하는 단계를 포함하는 API 설계 방법.

## 청구항 26.

제 15 항에 있어서,

상기 유도하는 단계는,

특정 FT가 다른 컴포넌트 타입들에 대해 독자적으로 무관하면, 상기 특정 FT를 상기 코어 시나리오와 관련되어 있는 AC로 인캡슐레이트하는 단계를 포함하는 API 설계 방법.

## 청구항 27.

제 15 항에 있어서,

상기 유도하는 단계는,

특정 FT의 하나 이상의 멤버가 AC에 의해 노출되지 않으면, 상기 코어 시나리오와 관련되어 있는 상기 AC로부터 상기 특정 FT를 노출시키는 단계를 포함하는 API 설계 방법.

## 청구항 28.

제 15 항에 있어서,

상기 유도하는 단계는,

특정 FT가 AC와 무관하게 또 하나의 컴포넌트 타입에 의해 유용하게 사용될 수 있다면, 상기 코어 시나리오와 관련되어 있는 상기 AC로부터 상기 특정 FT를 노출시키는 단계를 포함하는 API 설계 방법.

## 청구항 29.

제 15 항에 있어서,

상기 유도하는 단계는,

상대적으로 더 높은 수준의 추상화를 목표로 하는 컴포넌트 타입들 및 상대적으로 더 낮은 수준의 추상화를 목표로 하는 컴포넌트 타입들을 포함하는 2-계층 프레임워크를 생성하는 단계를 포함하는 API 설계 방법.

## 청구항 30.

제 29 항에 있어서,

상기 상대적으로 더 높은 수준의 추상화를 목표로 하는 상기 컴포넌트 타입들은 코어 시나리오들에 배당되는 API 설계 방법.

## 청구항 31.

제 29 항에 있어서,

상기 상대적으로 더 낮은 추상화를 목표로 하는 상기 컴포넌트 타입들은 상기 상대적으로 더 높은 추상화를 목표로 하는 상기 컴포넌트 타입들에 비해 개발자들에게 상대적으로 더 많은 양의 컨트롤을 제공하는 API 설계 방법.

## 청구항 32.

제 15 항에 있어서,

상기 유도하는 단계는,

개발자가 상기 코어 시나리오에 대해 작성-설정-호출 사용 패턴(create-set-call usage pattern)을 구현할 수 있도록 하기 위해 상기 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 33.

제 32 항에 있어서,

상기 유도하는 단계는,

상기 코어 시나리오에 적합한 사전-선택된 파라미터로 상기 API를 생성하는 단계를 포함하는 API 설계 방법.

## 청구항 34.

시나리오에 대해 기재된 하나 이상의 코드 샘플에 응답하여, 상기 시나리오에 대한 API를 유도하는 단계;

복수의 개발자들을 활용하여, 상기 API에 대해 하나 이상의 가용성 연구를 수행하는 단계; 및

상기 하나 이상의 가용성 연구에 기초하여, 상기 API를 수정하는 단계를 포함하는 API 설계 방법.

## 청구항 35.

제 34 항에 있어서,

상기 시나리오에 대해, 각각이 복수개의 프로그래밍 언어들 중의 개개의 프로그래밍 언어에 대응되는 복수개의 코드 샘플들을 기재하는 단계를 더 포함하고,

상기 유도하는 단계는,

상기 복수개의 코드 샘플들에 응답하여, 상기 시나리오에 대한 상기 API를 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 36.

제 34 항에 있어서,

상기 API에 대해 상기 하나 이상의 가용성 연구를 수행하는 단계 이전에,

API 설계자에 의해, 상기 유도된 API가 지나치게 복잡한지를 판정하는 단계;

상기 유도된 API가 지나치게 복잡한 것으로 판정되면,

세분된 API를 생성하기 위해, 상기 API 설계자에 의해, 상기 유도된

API를 세분하는 단계; 및

상기 API 설계자에 의해, 상기 세분된 API가 지나치게 복잡한지를 판정하는 단계를 더 포함하는 API 설계 방법.

## 청구항 37.

제 34 항에 있어서,

상기 복수의 개발자들이 상기 API를 큰 문제없이 사용할 수 있는지의 여부를 확인하는 단계; 및

상기 복수의 개발자들이 상기 API를 큰 문제없이 사용할 수 있는 것으로 확인되지 않으면, 상기 수정하는 단계를 구현하는 단계를 더 포함하고,

상기 수정하는 단계는,

수정된 API를 생성하기 위해, 상기 하나 이상의 가용성 연구로부터의

하나 이상의 교훈에 기초하여, 상기 API를 수정하는 단계를 포함하는 API 설계 방법.

## 청구항 38.

제 37 항에 있어서,

적어도 상기 수행하는 단계 및 상기 확인하는 단계를 상기 수정된 API에 대해 반복하는 단계를 더 포함하는 API 설계 방법.

## 청구항 39.

제 37 항에 있어서,

상기 확인하는 단계는,

하나 이상의 목표 개발자 그룹에 대한 소정 가용성 수준에 관해, 상기 복수의 개발자들이 상기 API를 큰 문제없이 사용할 수 있는지의 여부를 확인하는 단계를 포함하고,

상기 소정 가용성 수준은, (ⅰ) 상기 복수의 개발자들에 의한, 상세한 API 문서에 대한 빈번한 그리고/또는 광범위한 참조, (ii) 상기 복수의 개발자들 대다수에 의한 상기 시나리오 구현 실패, 및 (iii) 상기 복수의 개발자들이 API 설계자에 의해 예상되는 것과 상당히 상이한 접근을 취하는지의 여부에 대한 고려들을 포함하는 API 설계 방법.

## 청구항 40.

제 34 항에 있어서,

특징 영역에 대해 복수개의 코어 시나리오들을 선택하는 단계; 및

상기 유도하는 단계, 상기 수행하는 단계, 및 상기 수정하는 단계를 상기 복수개의 코어 시나리오들 중의 각 코어 시나리오에 대해 반복하는 단계를 더 포함하는 API 설계 방법.

## 청구항 41.

제 40 항에 있어서,

상기 유도하는 단계는,

상기 복수개의 코어 시나리오들 중의 각 코어 시나리오에 대해 AC를 생성하는 단계를 포함하는 API 설계 방법.

## 청구항 42.

제 34 항에 있어서,

상기 유도하는 단계는,

복수개의 FT들 중의 개별적인 각 FT와 개별적인 관계를 가진 AC를 생성하는 단계를 포함하는 API 설계 방법.

## 청구항 43.

제 42 항에 있어서,

상기 생성하는 단계는,

상기 하나 이상의 코드 샘플이 기재된 상기 시나리오를 지원하는 상기 AC를 생성하는 단계를 포함하는 API 설계 방법.

## 청구항 44.

제 42 항에 있어서,

상기 생성하는 단계는,

상기 복수개의 FT들 중의 하나 이상의 FT와의 노출된 관계 및 상기 복수개의 FT들 중의 하나 이상의 다른 FT와의 인캡슐레이트된 관계를 가진 상기 AC를 생성하는 단계를 포함하는 API 설계 방법.

## 청구항 45.

제 44 항에 있어서,

상기 AC와 상기 인캡슐레이트된 관계를 가진 상기 복수개의 FT들 중의 상기 하나 이상의 다른 FT는 상기 AC에 의해 또 하나의 컴포넌트 타입과의 직접적 상호작용을 위해 핸드오프될 수 있는 API 설계 방법.

## 청구항 46.

제 42 항에 있어서,

상기 복수개의 FT들은 오브젝트-지향 방법을 사용해 설계된 API 설계 방법.

## 청구항 47.

코어 시나리오에 대해, 각각이 복수개의 프로그래밍 언어들 중의 각 프로그래밍 언어에 대응되는 복수개의 코드 샘플들을 준비하는 단계;

상기 복수개의 코드 샘플들에 응답하여, 상기 코어 시나리오에 대한 API를 유도하는 단계;

복수의 개발자들을 활용하여, 상기 API에 대해 하나 이상의 가용성 연구를 수행하는 단계; 및

상기 하나 이상의 가용성 연구에 기초하여, 상기 API를 수정하는 단계를 포함하는 API 설계 방법.

## 청구항 48.

시나리오에 대해 하나 이상의 코드 샘플을 기재하는 단계; 및

상기 하나 이상의 코드 샘플에 응답하여, 상기 시나리오에 대한 API를 유도하는 단계를 포함하고,

상기 API는 (ⅰ) 상기 시나리오의 구현을 용이하게 하기 위한 AC 및 (ii) 상기 AC에 대해 하부 기능을 제공하는 복수개의 FT들을 포함하며,

상기 API는 상기 AC를 보다 간단한 상황에 사용하는 것으로부터 상기 복수개의 FT들 중의 증가하는 부분을 상당히 복잡한 상황에 사용하는 것으로의 점진적 발전을 가능하게 하는 API 설계 방법.

## 청구항 49.

하나 이상의 시나리오에 대해 하나 이상의 코드 샘플을 지원하는 하나 이상의 AC를 유도하는 단계;

상기 하나 이상의 시나리오에 대해 부가적인 요구사항들을 판정하는 단계;

상기 하나 이상의 시나리오에 과도한 복잡성을 부가하지 않으면서, 상기 부가적인 요구사항들이 상기 하나 이상의 AC에 부가될 수 있는지를 판단하는 단계; 및

그렇지 않다면, 상기 판단하는 단계에 응답하여, 복수개의 FT들을 정의하는 단계를 포함하는 API 설계 방법.

## 청구항 50.

제 49 항에 있어서,

그렇다면, 상기 부가적인 요구사항들을 통합하기 위해 상기 하나 이상의 AC를 세분하는 단계를 더 포함하는 API 설계 방법.

## 청구항 51.

제 49 항에 있어서,

특징 영역에 대해, 하나 이상의 시나리오를 포함하는 복수개의 코어 시나리오들을 선택하는 단계; 및

상기 복수개의 코어 시나리오들에 대한 바람직한 코드 라인들을 나타내는, 하나 이상의 코드 샘플을 포함하는 복수개의 코드 샘플들을 기재하는 단계를 더 포함하고,

상기 유도하는 단계는,

상기 복수개의 코어 시나리오들에 대한 상기 복수개의 코드 샘플들을

지원하기 위해, 상기 하나 이상의 AC를 포함하는 복수개의 AC들을 유도하는 단계를 포함하는 API 설계 방법.

## 청구항 52.

제 49 항에 있어서,

상기 유도하는 단계는,

상기 하나 이상의 시나리오에 대한 상기 하나 이상의 코드 샘플을 지원

하기 위해, 적절한 방법들, 디폴트들, 및 추상화들로 상기 하나 이상의 AC를

유도하는 단계를 포함하는 API 설계 방법.

## 청구항 53.

제 49 항에 있어서,

상기 하나 이상의 유도된 AC에 따라, 상기 하나 이상의 코드 샘플을 세분하는 단계;

상기 세분된 하나 이상의 코드 샘플을 단순성에 관해 평가하는 단계; 및

상기 유도된 하나 이상의 코드 샘플이 상기 평가하는 단계에서 충분히 단순하지 않다면, 상기 유도하는 단계를 반복하는 단계를 더 포함하는 API 설계 방법.

## 청구항 54.

제 49 항에 있어서,

상기 판정하는 단계는,

상기 하나 이상의 시나리오에 대해 부가적인 요구사항들을 판정하는 단

계를 포함하고,

상기 부가적인 요구사항들은 부가적인 시나리오들, 부가적인 목적들,

및 다른 컴포넌트 타입들과의 부가적인 상호작용을 포함하는 API 설계 방법.

## 청구항 55.

제 49 항에 있어서,

상기 판단하는 단계는,

상기 부가적인 요구사항들을 상기 하나 이상의 AC에 부가하는 것이 작

성-설정-호출 사용 패턴에 장애가 될 것인지의 여부를 고려하는 단계를 포함

하는 API 설계 방법.

## 청구항 56.

제 49 항에 있어서,

상기 정의하는 단계는,

상기 판단하는 단계에 응답하여, 풀 세트 기능의 이상적 팩토링으로 상

기 복수개의 FT들을 정의하는 단계를 포함하는 API 설계 방법.

## 청구항 57.

제 49 항에 있어서,

상기 정의하는 단계는,

상기 판단하는 단계에 응답하여, 하나 이상의 오브젝트-지향 방법들을

사용해 상기 복수개의 FT들을 정의하는 단계를 포함하는 API 설계 방법.

## 청구항 58.

제 49 항에 있어서,

상기 하나 이상의 AC가 상기 복수개의 FT들 중의 각 FT에 대한 기능을 인캡슐레이트할 것인지 노출할 것인지 판정하는 단계를 더 포함하는 API 설계 방법.

## 청구항 59.

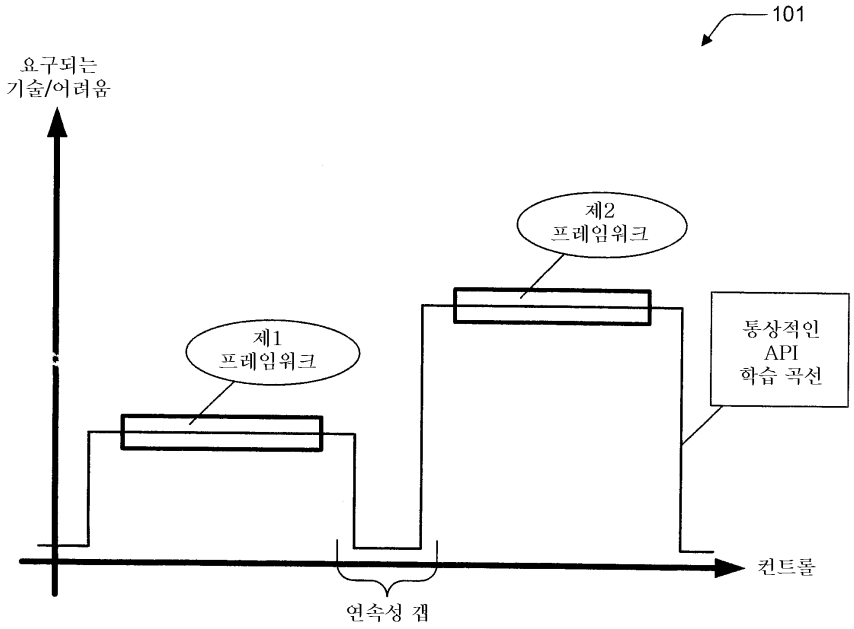
제 49 항에 있어서,

상기 하나 이상의 AC 및 상기 부가적인 요구사항들을 지원하기 위해, 상기 복수개의 FT들을 세분하는 단계를 더 포함하는 API 설계 방법.

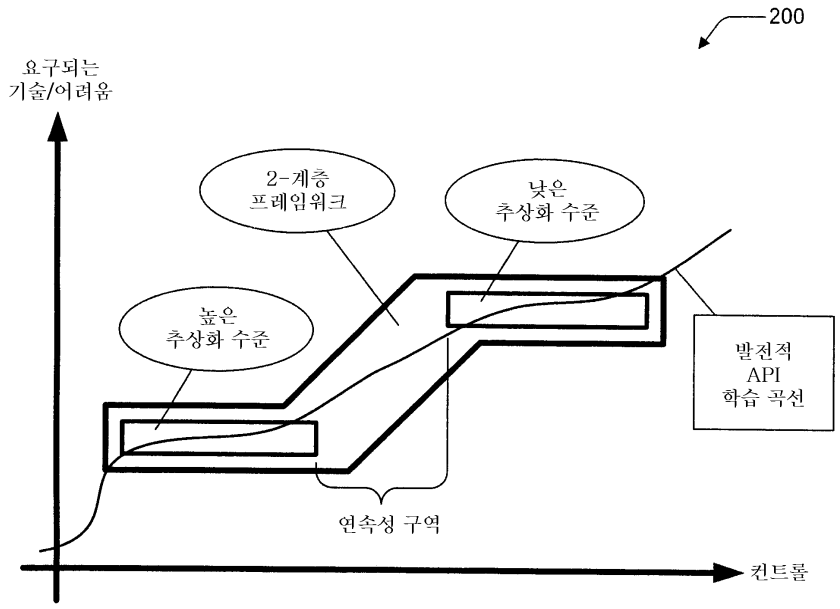
도면

도면1

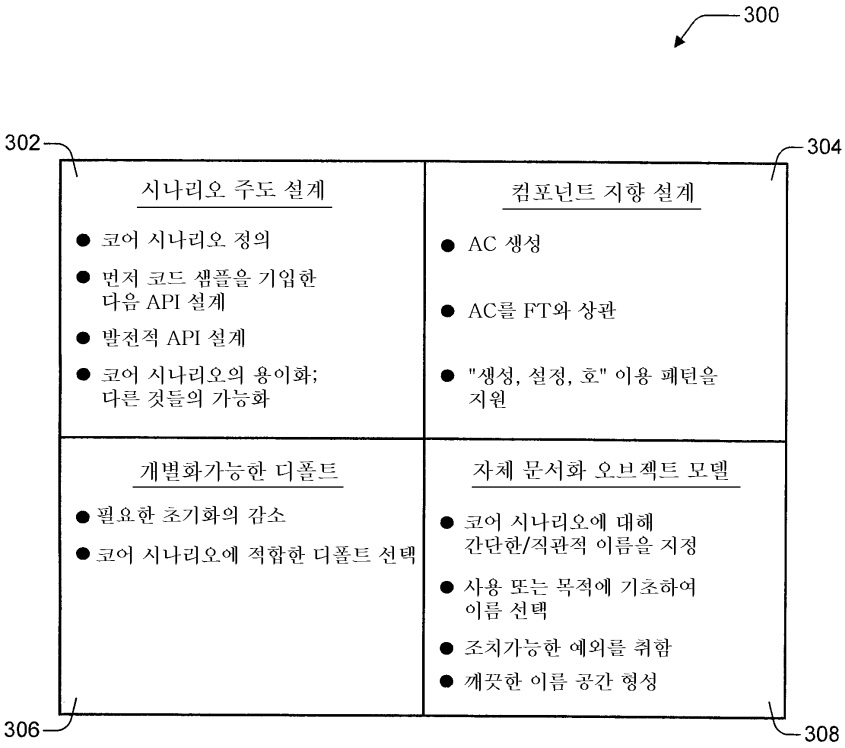
(종래 기술)


101
요구되는
기술/어려움
제2
프레임워크
제1
프레임워크
통상적인
API
학습 곡선
컨트롤
연속성 갭

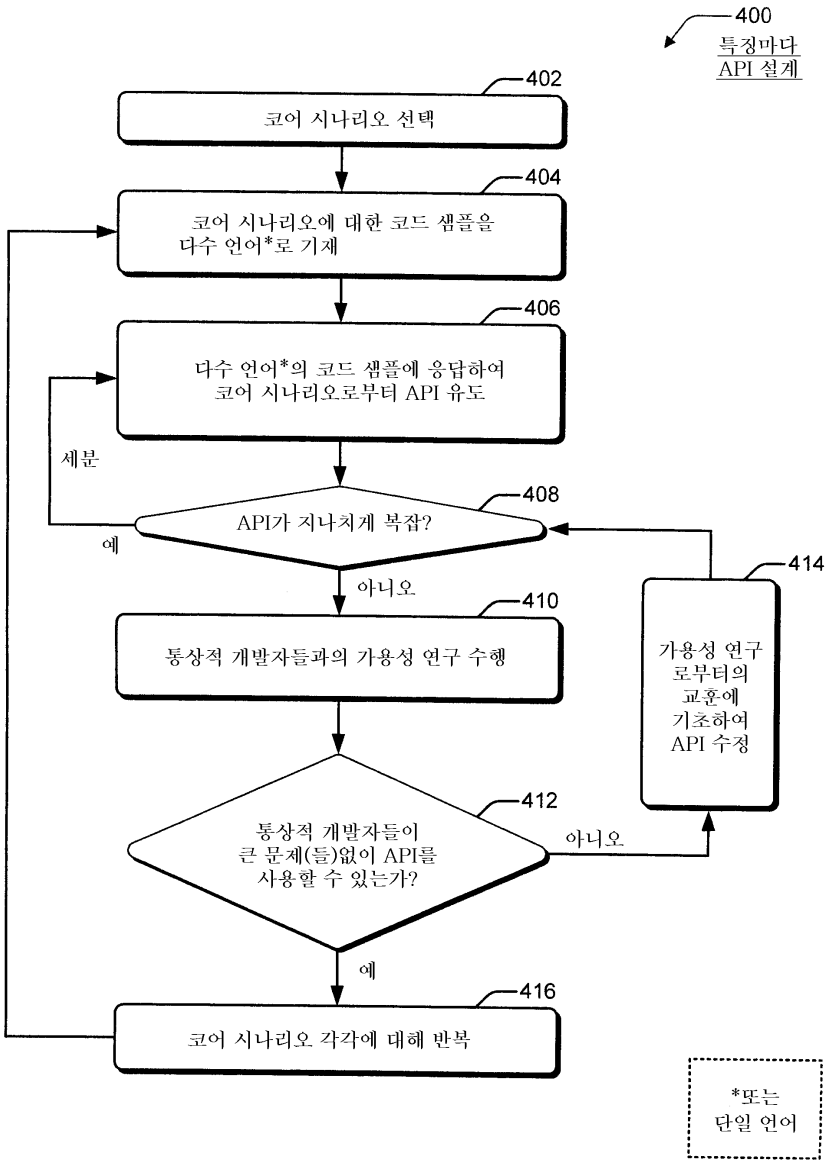
도면2


200
요구되는
기술/어려움
2-계층
프레임워크
낮은
추상화 수준
높은
추상화 수준
발전적
API
학습 곡선
연속성 구역
컨트롤

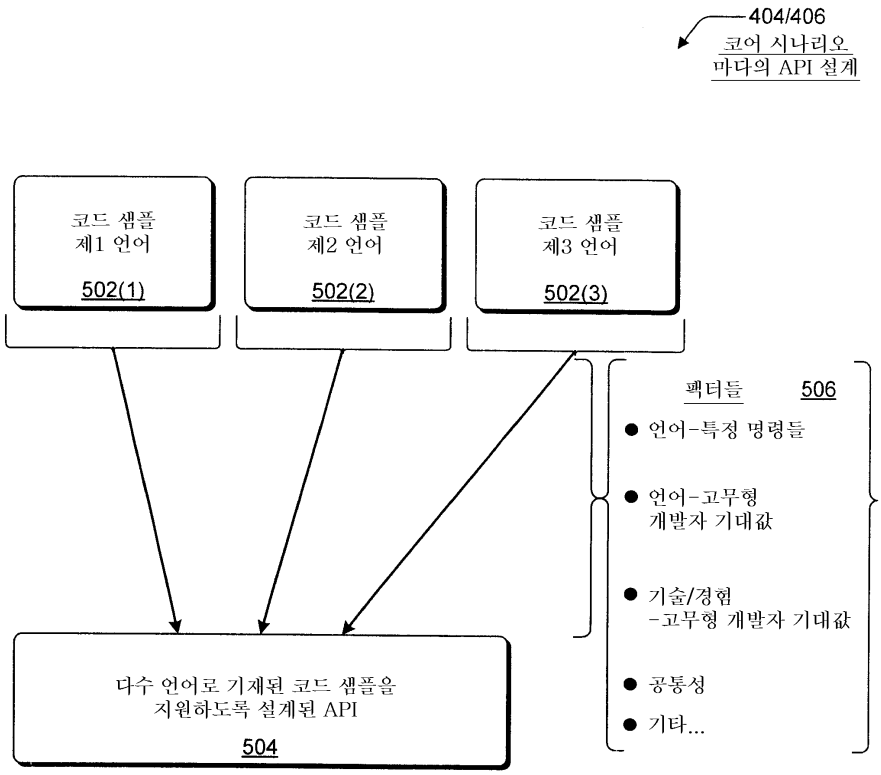
도면3


300
302
304
시나리오 주도 설계
● 코어 시나리오 정의
● 먼저 코드 샘플을 기입한 다음 API 설계
● 발전적 API 설계
● 코어 시나리오의 용이화; 다른 것들의 가능화
컴포넌트 지향 설계
● AC 생성
● AC를 FT와 상관
● "생성, 설정, 호" 이용 패턴을 지원
개별화가능한 디폴트
● 필요한 초기화의 감소
● 코어 시나리오에 적합한 디폴트 선택
자체 문서화 오브젝트 모델
● 코어 시나리오에 대해 간단한/직관적 이름을 지정
● 사용 또는 목적에 기초하여 이름 선택
● 조치가능한 예외를 취함
● 깨끗한 이름 공간 형성
306
308

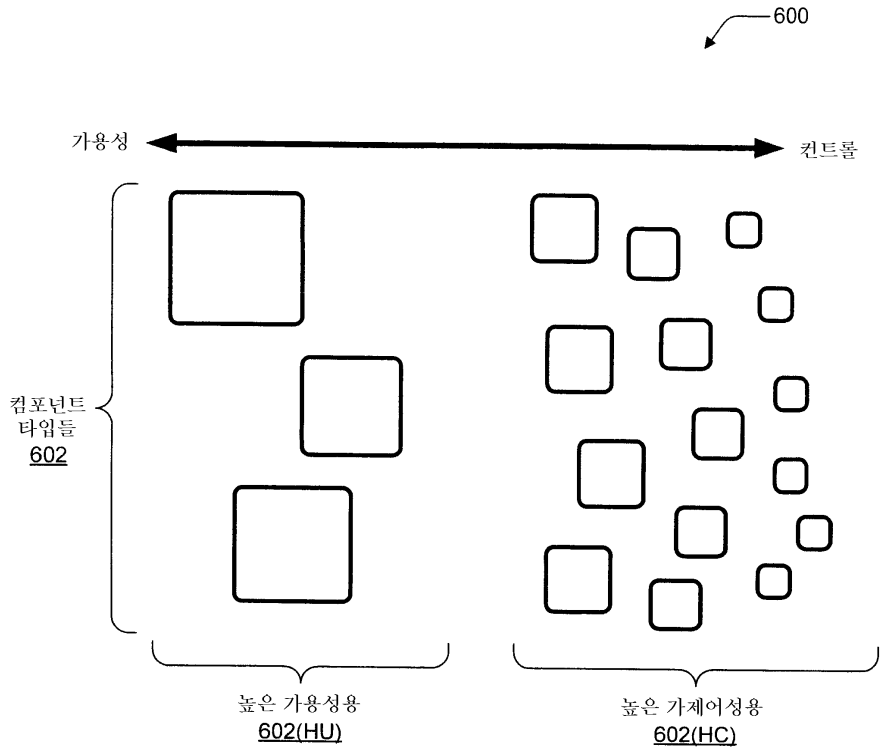
도면4


400
특징마다
API 설계
402
코어 시나리오 선택
404
코어 시나리오에 대한 코드 샘플을
다수 언어*로 기재
406
다수 언어*의 코드 샘플에 응답하여
코어 시나리오로부터 API 유도
세분
408
API가 지나치게 복잡?
예
아니오
410
통상적 개발자들과의 가용성 연구 수행
414
가용성 연구
로부터의
교훈에
기초하여
API 수정
412
통상적 개발자들이
큰 문제(들)없이 API를
사용할 수 있는가?
아니오
예
416
코어 시나리오 각각에 대해 반복
*또는
단일 언어

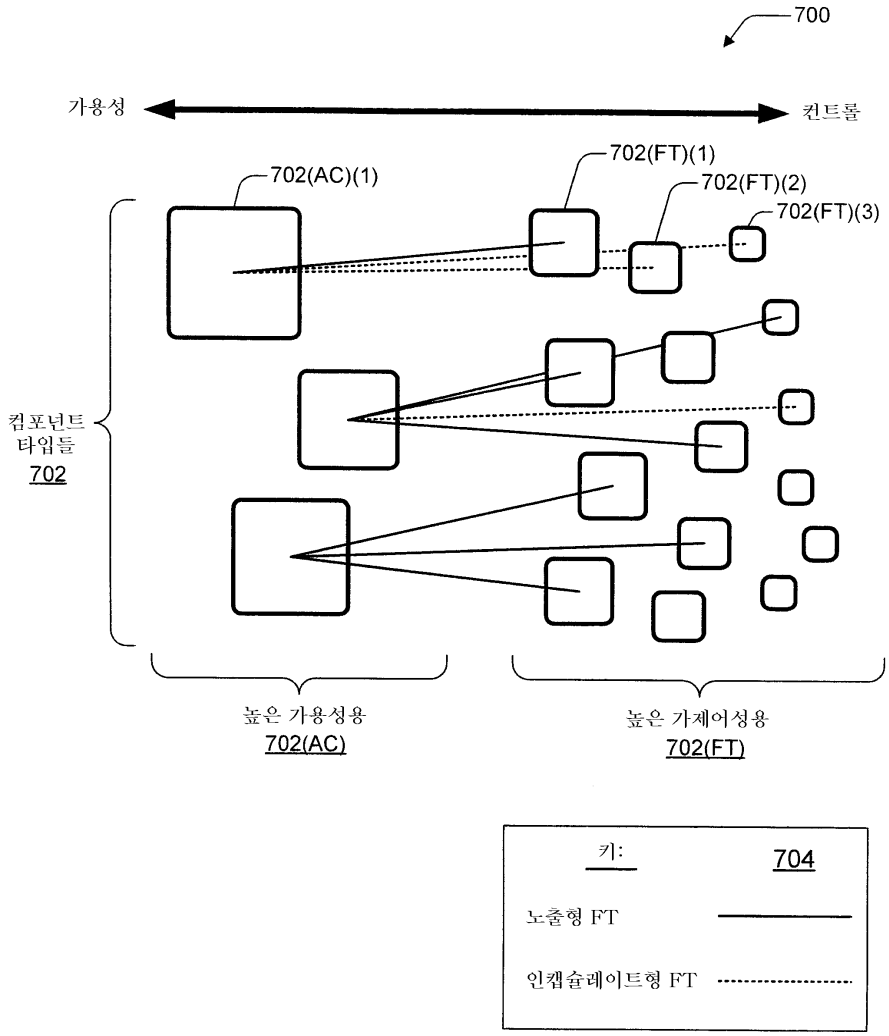
도면5


404/406
코어 시나리오
마다의 API 설계
코드 샘플
제1 언어
502(1)
코드 샘플
제2 언어
502(2)
코드 샘플
제3 언어
502(3)
팩터들 506
● 언어-특정 명령들
● 언어-고무형
개발자 기대값
● 기술/경험
-고무형 개발자 기대값
● 공통성
● 기타...
다수 언어로 기재된 코드 샘플을
지원하도록 설계된 API
504

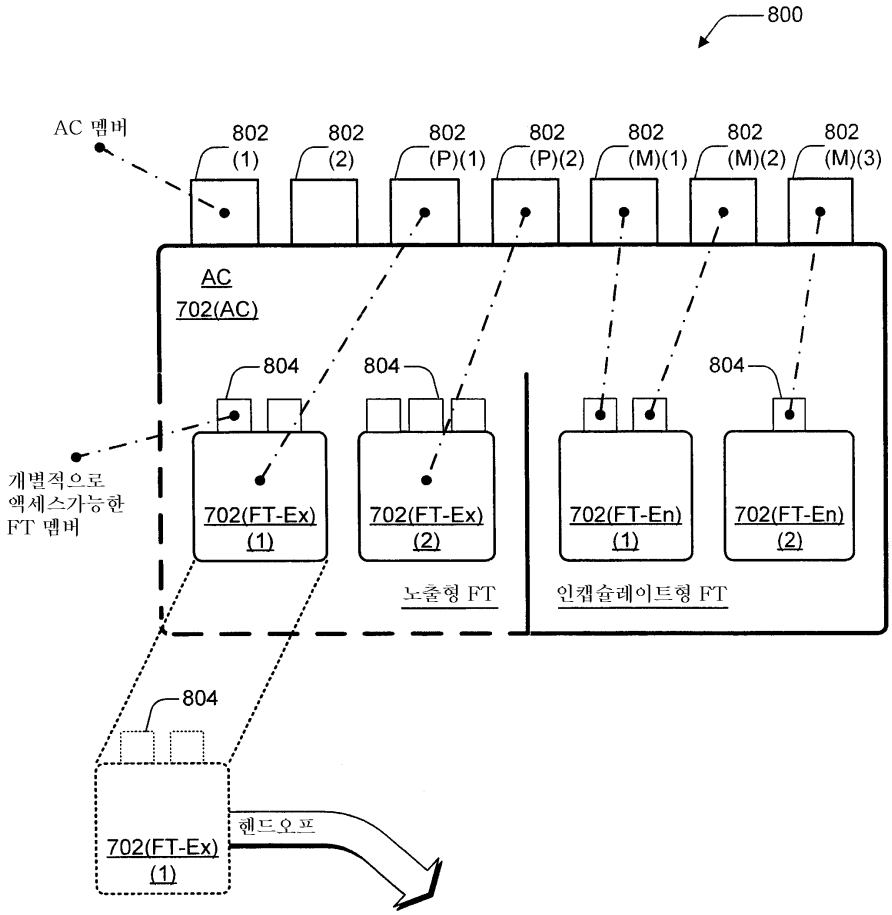
도면6


600
가용성
컨트롤
컴포넌트
타입들
602
높은 가용성용
602(HU)
높은 가제어성용
602(HC)

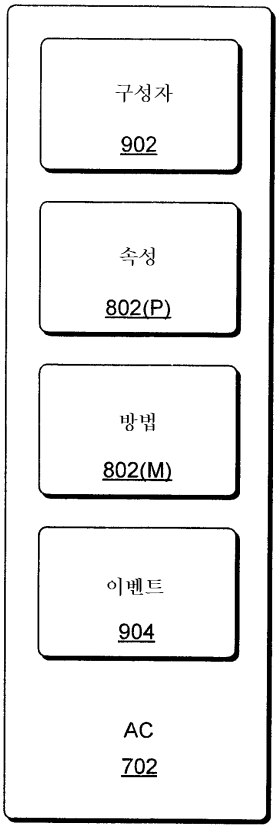
도면7


700
가용성
컨트롤
702(AC)(1)
702(FT)(1)
702(FT)(2)
702(FT)(3)
컴포넌트 타입들
702
높은 가용성용
702(AC)
높은 가제어성용
702(FT)
키:
704
노출형 FT
인캡슐레이트형 FT

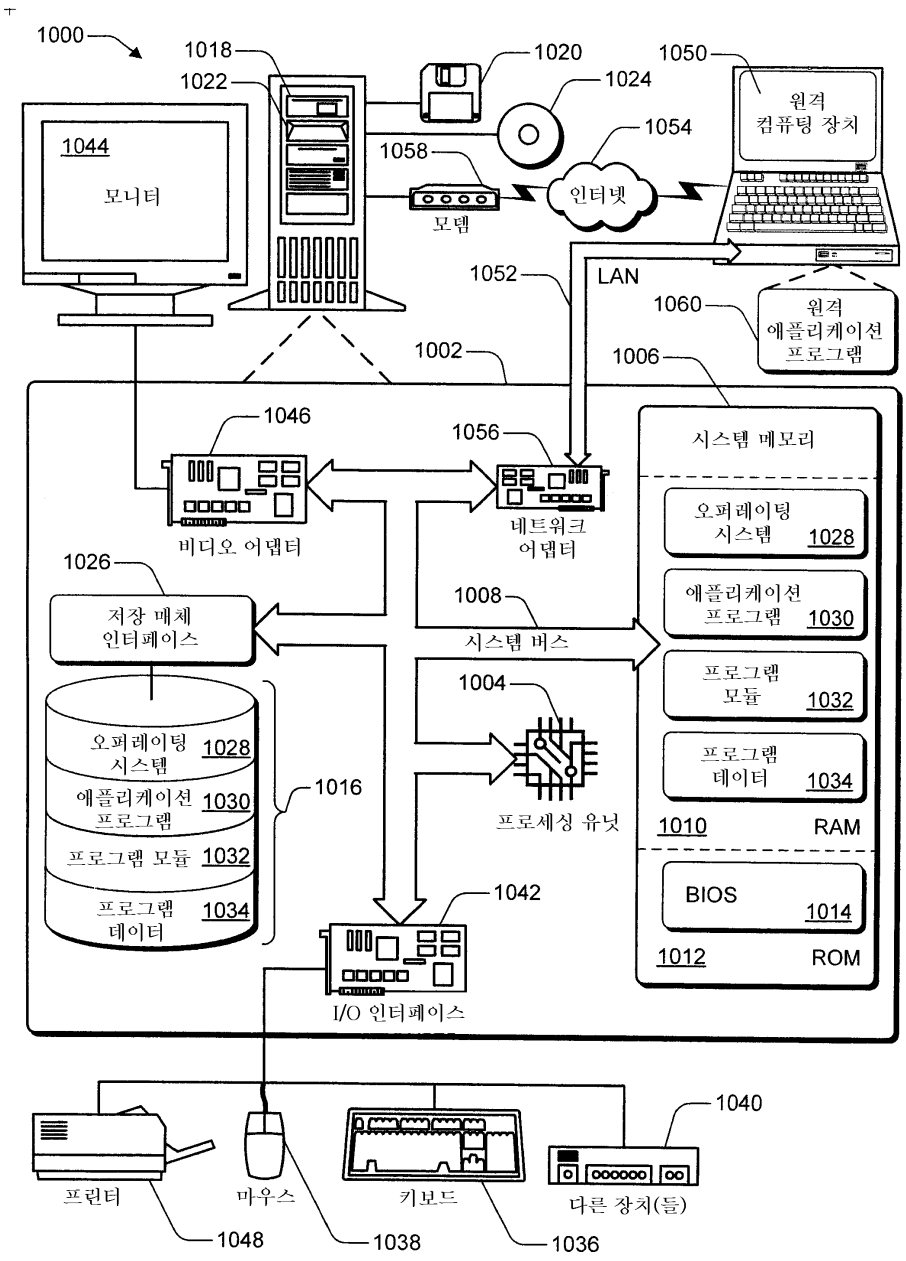
도면8


800
AC 멤버
802 (1)
802 (2)
802 (P)(1)
802 (P)(2)
802 (M)(1)
802 (M)(2)
802 (M)(3)
AC
702(AC)
804
804
804
개별적으로
액세스가능한
FT 멤버
702(FT-Ex) (1)
702(FT-Ex) (2)
702(FT-En) (1)
702(FT-En) (2)
노출형 FT
인캡슐레이트형 FT
804
702(FT-Ex) (1)
핸드오프

도면9



도면10


1000
1018
1022
1020
1024
1050
원격
컴퓨팅 장치
1044
모니터
1058
모뎀
1054
인터넷
1052
LAN
1060
원격
애플리케이션
프로그램
1002
1006
1046
1056
시스템 메모리
비디오 어댑터
네트워크
어댑터
오퍼레이팅
시스템 1028
1026
저장 매체
인터페이스
1008
애플리케이션
프로그램 1030
시스템 버스
프로그램
모듈 1032
1004
오퍼레이팅
시스템 1028
애플리케이션 1030
프로그램 모듈 1032
프로그램 1034
데이터
1016
프로세싱 유닛
프로그램
데이터 1034
1010
RAM
BIOS
1014
1042
1012
ROM
I/O 인터페이스
1040
프린터
마우스
키보드
다른 장치(들)
1048
1038
1036