Design Pattern

Design Pattern

- Design Pattern
  - ✓ 비슷한 목적으로 사용되는 클래스의 모범 사례를 정리한 것이 Design Pattern
  - ✓ 프로그램의 세계에는 다양한 Design Pattern이 존재하는데 유명한 것이 GoF Design Pattern 인데 GoF(고프)란 The Gang of Four의 약자이며 에리히 감마, 리처드 헬름, 랄프 존슨, 존 블리시데스의 4명이 인스턴스 지향 프로그래밍에 도움이 되는 Design Pattern을 도입해 Design Patterns: Elements of Reusable Object Oriented Software 라는 제목의 책에 정리
  - ✓ Design Pattern의 분류
    - 인스턴스의 생성에 관한 Pattern
    - 프로그램의 구조에 관한 Pattern
    - 인스턴스의 행동에 관한 Pattern
  - ✓ Design Pattern은 정교한 클래스 구조의 Pattern 집합이며 잘 이해해서 올바르게 적용하면 충실하고 알기 쉬운 클래스 구성을 생성할 수 있음

Design Pattern

- Design Pattern
  - ✓ 패턴: 어떤 상황의 문제에 대한 해법
  - ✓ 패턴에 포함되어야 하는 요소
    - 패턴 이름(pattern name)
      - 한두 단어로 설계 문제와 해법을 서술
      - 패턴에 이름을 부여하는 것은 설계 어휘를 늘리는 일이며 높은 수준의 추상화된 설계를 할 수 있도록 해 줌
      - 패턴의 이름을 정의해 두면 문서에서 이 이름을 사용하여 설계의 의도를 표현할 수 있게 되고 이름을 갖게 되면 설계에 대한 생각을 더욱 쉽게 할 수 있고 개발자들 간의 의사소통 이 원활해지기 때문에 좋은 이름을 생각해 내는 것은 카탈로그를 설정하는 데 있어서 가장 힘든 부분 중의 하나이기도 함
    - 문제(problem)
      - 언제 패턴을 사용하는가를 서술하며 해결할 문제와 그 배경을 설명
      - 어떤 알고리즘을 객체로 만들까 와 같은 설계의 세밀한 문제를 설명할 수 있음
      - 때론 유연성 없는 설계가 될 징조를 보이는 클래스나 객체의 구조를 제시하는데 문제를 제시함으로써 패턴을 적용하는 것이 의미 있는 사례들을 정의하기도 함

Design Pattern

- Design Pattern
  - ✓ 패턴에 포함되어야 하는 요소
    - 해법(solution)
      - 설계를 구성하는 요소들과 그 요소들 간의 관계, 책임 그리고 협력 관계를 서술
      - 해법이 어떤 구체적인 설계나 구현을 설명 하지는 않는데 왜냐하면 패턴은 다양한 경우에 적용할 수 있는 템플릿(template)이기 때문
      - 구체적인 부분 대신 디자인 패턴은 문제에 대한 추상적인 설명을 제공하고 문제를 해결하기 위해서 클래스나 객체들의 나열 방법을 제공
    - 결과(consequence)
      - 디자인 패턴을 적용해서 얻는 결과와 장단점을 서술
      - 어떤 설계를 결정할 때 그 설계의 결과를 고려하지 않기가 쉬운데 어떻게 보면 선택하는 과정에서 또는 비용과 효과를 측정하는 과정에서 설계의 결과는 가장 중요한 부분
      - 소프트웨어에서 결과란 가끔 시간이나 공간 사이의 균형일 수도 있는데 즉 시간을 중요한 요소로 볼 것인지 아니면 저장 공간의 효율을 중요한 요소로 볼 것인지에 따라 다른 설계 방법을 선택해야 한다는 것
      - 언어에 따라서도 차이가 있는데 재사용은 객체지향 설계의 주요 요소이므로 패턴의 결과는 시스템의 유연성, 확장성, 이식성 등에 커다란 영향을 주기 때문에 이런 설계의 결과들을 잘 정리해 두면 나중에 패턴들을 이해하거나 평가하는 데 도움을 받을 수 있음

Design Pattern

- Design Pattern
  - ✓ 패턴에 대한 시각은 패턴을 구분하고 분석하는 데 영향을 주는데 어떤 사람에게는 그것이 패턴이지만 다른 사람에게는 지극히 기초적인 조립용 부품으로 보일 수도 있음
  - ✓ 디자인 패턴은 연결 리스트와 해시 테이블 등을 클래스로 표현하고 그것 자체로 다시 쓸 수 있도록 설계하는 문제를 어떻게 푸느냐에 관한 것이 아니며 응용 프로그램 전체나 서브 시스템을 지원하는 복잡한 설계에 대한 것은 더더욱 아니고 디자인 패턴은 특정한 전후 관계에서 일반적 설계 문제를 해결하기 위해 상호교류하는 수정 가능한 객체와 클래스들에 대한 설명
  - ✓ 하나의 디자인 패턴은 재사용 가능한 객체지향 설계를 만들기 위해 유용한 공통의 설계 구조에서 주요 요소들을 식별하여 이들에게 적당한 이름을 주고 추상화 하고 패턴에 참여하는 클래스와 그들의 인스턴스를 식별하여 역할과 그들 간의 협력 관계를 정의하고 책임을 할당
  - ✓ 각 디자인 패턴은 각자 맡은 객체지향 설계 문제에 집중하는데 언제 패턴을 적용할지, 다른 설계 제약을 고려하여 패턴을 적용할 수 있는지, 패턴을 사용하면 어떤 결과가 발생하는지도 친절히 설명

Design Pattern

- Design Pattern
  - ✓ 기술
    - 패턴 이름과 분류(Pattern Name and Classification)
    - 의도(Intent)
      - 이 디자인 패턴은 무엇을 하는 것일까요?
      - 의도와 논리적인 근거가 무엇일까요?
      - 어떤 특정한 문제나 이슈를 해결하기 위한 것일까요?
    - 다른 이름(Also Known As)
    - 동기(Motivation): 설계 문제를 제시하고, 패턴 안에서 클래스나 객체 구조가 어떻게 문제를 해결하는지 설명해 주는 일종의 시나리오
    - 활용성(Applicability): 해당 패턴을 어떤 상황에 적용
    - 구조(Structure)
      - 객체 모델링 기법(Object Modeling Technique: OMT) 에 기반을 둔 표기법을 이용하여 해당 패턴에서 쓰는 클래스들을 시각적으로 나타냄
      - 객체 사이에 오가는 요청과 협력 관계의 순차를 표현하기 위해서 상호 작용 다이어그램도 이용
    - 참여자(Participant): 주어진 패턴을 구성하고 책임을 수행하는 클래스나 객체들을 설명
    - 협력 방법(Collaboration): 참여자들이 작업을 수행하기 위한 참여자들 간의 협력 관계를 정의

Design Pattern

- Design Pattern
  - ✓ 기술
    - 결과(Consequence)
      - 이 패턴이 자신의 목표를 어떻게 지원할까요?
      - 이 패턴을 이용한 결과는 무엇이고 장단점은 무엇일까요?
      - 이 패턴을 사용하면 시스템 구조의 어떤 면을 독립적으로 다양화 시킬 수 있을까요?
    - 구현(Implementation)
      - 패턴을 구현할 때 주의해야 할 함정, 힌트, 기법 등은 무엇일까요?
      - 특정 언어에 국한된 특이 사항은 무엇일까요?
    - 예제 코드(Sample Code)
    - 잘 알려진 사용 예(Known Use): 실제 시스템에서 찾아볼 수 있는 패턴들의 예
    - 관련 패턴(Related Pattern)
      - 이 패턴과 밀접하게 관련된 다른 패턴들은 무엇일까요?
      - 이들의 중요한 차이점은 무엇일까요?
      - 어떤 다른 패턴에 이 패턴이 사용되어야 할까요?

Design Pattern

- Design Pattern의 장점
  - ✓ 재사용성이 높고 유연성 있는 설계가 가능
    - 프로그램을 처음부터 설계하면 만들어진 성과물의 품질이 설계자의 직관이나 경험 등에 따라 달라지지만 Design Pattern을 도입하면 초보자도 고수들의 갈고 닦은 '지혜'를 이용한 설계가 가능
  - ✓ 의사 소통이 원할
    - Design Pattern을 습득하지 않은 기술자에게 설계를 설명할 경우 이런 클래스를 만들고, 이 클래스는 이런 역할을 갖고 있고..... 라고 끝없이 설명해야 하지만 Design Pattern을 습득하고 있는 기술자라면 ○○Pattern으로 만들어!라는 한마디로 끝남
    - Design Pattern을 습득하고 있는 기술자끼리라면 설계에 대해 상담할 때도 Design Pattern의 이름으로 설계 개요에 대해 동의를 얻는것이 가능해 의사 소통이 원활
  - ✓ Design Pattern을 이해하고 개념을 익혀두면 자신이 직접 설계하는 경우에도 적용할 수 있어 설계의 수준을 높일 수 있음

Design Pattern

- Design Pattern Catalog
  - ✓ 추상 팩토리(Abstract Factory): 구체적인 클래스를 지정하지 않고 관련성을 갖는 객체들의 집합을 생성하거나 서로 독립적인 객체들의 집합을 생성할 수 있는 인터페이스를 제공하는 패턴
  - ✓ 적응자(Adapter): 클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환하는 패턴으로 호환성이 없는 인터페이스 때문에 함께 동작할 수 없는 클래스들이 함께 작동하도록 함
  - ✓ 가교(Bridge): 구현부에서 추상층을 분리하여 각자 독립적으로 변형할 수 있게 하는 패턴
  - ✓ 빌더(Builder): 복합 객체의 생성 과정과 표현 방법을 분리하여 동일한 생성 절차에서 서로 다른 표현 결과를 만들 수 있게 하는 패턴
  - ✓ 책임 연쇄(Chain of Responsibility): 요청을 처리할 수 있는 기회를 하나 이상의 객체에게 부여하여 요청을 보내는 객체와 그 요청을 받는 객체 사이의 결합을 피하는 패턴으로 요청을 받을 수 있는 객체를 연쇄적으로 묶고 실제 요청을 처리할 객체를 만날 때까지 객체 고리를 따라서 요청을 전달
  - ✓ 명령(Command): 요청을 객체의 형태로 캡슐화하여 서로 요청이 다른 사용자의 매개변수화, 요청 저장 또는 로깅, 그리고 연산의 취소를 지원하게 만드는 패턴
  - ✓ 복합체(Composite): 객체들의 관계를 트리 구조로 구성하여 부분-전체 계층을 표현하는 패턴으로 사용자가 단일 객체와 복합 객체 모두 동일하게 다루도록 하는 패턴
  - ✓ 템플릿 메서드(Template Method): 객체의 연산에는 알고리즘의 뼈대만을 정의하고 각 단계에서 수행할 구체적 처리는 서브 클래스 쪽으로 미루는 패턴으로 알고리즘의 구조 자체는 그대로 놔둔 채 알고리즘 각 단계의 처리를 서브 클래스에서 재정의할 수 있게 함

Design Pattern

- Design Pattern Catalog
  - ✓ 장식자{Decorator): 주어진 상황 및 용도에 따라 어떤 객체에 책임을 덧붙이는 패턴으로 기능 확장이 필요할 때 서브 클래싱 대신 쓸 수 있는 유연한 대안이 될 수 있는 패턴
  - ✓ 퍼사드(Facade): 서브 시스템에 있는 인터페이스 집합에 대해서 하나의 통합 된 인터페이스를 제공하는 패턴으로 서브 시스템을 좀더 사용하기 편하게 만드 는 상위 수준의 인터페이스를 정의
  - ✓ 팩토리 메서드(Factory Method): 객체를 생성하는 인터페이스는 미리 정의하되 인스턴스를 만들 클래스의 결정은 서브 클래스 쪽에서 내리는 패턴으로 팩토리 메서드 패턴에서는 클래스의 인스턴스를 만드는 시점을 서브 클래스로 미룸
  - ✓ 플라이급(Flyweight): 크기가 작은 객체가 여러 개 있을 때 공유를 통해 이들을 효율적으로 지원하는 패턴
  - ✓ 해석자(Interpreter): 주어진 언어에 대해 그 언어의 문법을 위한 표현 수단을 정의하고 이와 아울러 그 표현 수단을 사용하여 해당 언어로 작성된 문장을 해석하는 해석기를 정의하는 패턴
  - ✓ 반복자(Iterator): 내부 표현부를 노출하지 않고 어떤 객체 집합에 속한 원소들을 순차적으로 접근할 수 있는 방법을 제공하는 패턴
  - ✓ 중재자(Mediator): 한 집합에 속해있는 객체들의 상호작용을 캡슐화하는 객체를 정의하는 패턴으로 객체들이 직접 서로를 참조하지 않도록 함으로써 객체들 사이의 소결합(loose coupling)을 촉진시키며 개발자가 객체들의 상호작용을 독립적으로 다양화 시킬 수 있는 패턴

Design Pattern

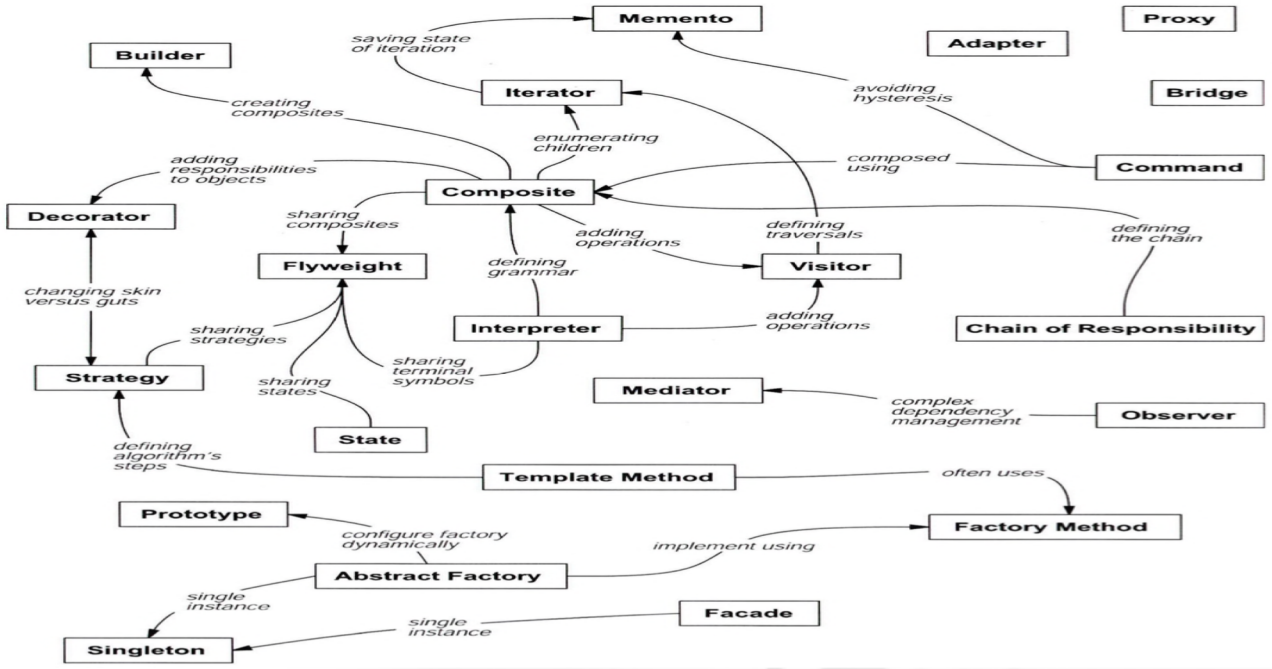
- Design Pattern Catalog
  - ✓ 메멘토(Memento): 캡슐화를 위배하지 않은 채 어떤 객체의 내부 상태를 잡아 내고 실체화시켜 이후에 해당 객체가 그 상태로 다시 되돌아올 수 있도록 하는 패턴
  - ✓ 감시자(Observer): 객체 사이에 일 대 다의 의존 관계를 정의해 두어 어떤 객체의 상태가 변할 때 그 객체에 의존성을 가진 다른 객체들이 그 변화를 통지받고 자동으로 갱신될 수 있게 만드는 패턴
  - ✓ 원형(Prototype): 생성할 객체의 종류를 명세화하는 데에 원형이 되는 예시물을 이용하고 그 원형을 복사함으로써 새로운 객체를 생성하는 패턴
  - ✓ 프록시(Proxy): 어떤 다른 객체로 접근하는 것을 통제하기 위해서 그 객체의 대리자(surrogate) 또는 자리 채움자(placeholder)를 제공하는 패턴
  - ✓ 단일체(Singleton): 어떤 클래스의 인스턴스는 오직 하나임을 보장하며 이 인스턴스에 접근할 수 있는 전역적인 접촉점을 제공하는 패턴
  - ✓ 상태(State): 객체의 내부 상태에 따라 스스로 행동을 변경할 수 있게끔 허가하는 패턴으로 이렇게 하면 객체는 마치 자신의 클래스를 바꾸는 것처럼 보임
  - ✓ 전략(Strategy): 동일 계열의 알고리즘군을 정의하고 각각의 알고리즘을 캡슐화하며 이들을 상호 교환이 가능하도록 만드는 패턴으로 알고리즘을 사용하는 사용자와 상관없이 독립적으로 알고리즘을 다양하게 변경할 수 있게 함
  - ✓ 방문자(Visitor): 객체 구조를 이루는 원소에 대해 수행할 연산을 표현하는 패턴으로 연산을 적용할 원소의 클래스를 변경하지 않고도 새로운 연산을 정의할 수 있게 함

Design Pattern

- Design Pattern Catalog 조직화
  - ✓ 디자인 패턴은 저마다 추상화 수준 과 입도가 천차만별인데다가 그 수효가 23개나 되기 때문에 이들을 조직화하지 않으면 안 됨
  - ✓ 패턴을 분류하는 기준
    - 첫 번째 분류 기준은 목적(purpose) – 패턴이 무엇을 하는지 정의하는 것으로 패턴은 생성, 구조, 행동 중의 한 가지 목적을 갖는 것으로 생성 패턴은 객체의 생성 과정에 관여하는 것이고 구조 패턴은 클래스나 객체의 합성에 관한 패턴들이며 행동 패턴은 클래스나 객체들이 상호 작용하는 방법과 책임을 분산하는 방법을 정의
    - 두 번째 분류 기준은 범위(scope) – 패턴을 주로 클래스에 적용하는지 아니면 객체에 적용하는지를 구분하는 것으로 클래스 패턴은 클래스와 서브 클래스 간의 관련성을 다루는 패턴이며 관련성은 주로 상속이며 컴파일 타임에 정적으로 결정되며 객체 패턴은 객체 관련성을 다루는 패턴으로서 런타임에 변경할 수 있으며 더 동적인 성격을 가지고 대부분의 패턴들은 어느 정도 상속을 이용하는데 클래스 패턴으로 정의한 패턴만 클래스 관련성을 이용하는데 일부만 클래스 패턴이고 대부분의 패턴은 객체 영역에 속하는데 생성(creational) 클래스 패턴은 객체를 생성하는 책임의 일부를 서브 클래스가 담당하도록 넘기지만 생성 객체 패턴은 이를 다른 객체에게 위임하며 구조(structural) 클래스 패턴은 상속을 이용해서 클래스를 복합하고 구조 객체 패턴은 객체를 합성하는 방법을 정의하며 행동(behavioral) 클래스 패턴은 상속을 이용해서 알고리즘과 제어 흐름을 기술하고, 행동 객체 패턴은 하나의 작업을 수행하기 위해 객체 집합이 어떻게 협력하는지를 기술

Design Pattern

- Design Pattern Catalog 조직화
  - ✓ 패턴을 분류하는 기준
    - 패턴을 조직하는 또 다른 방법은 일부 패턴은 함께 사용해야 할 때도 있는데 예를 들어 복합체 패턴은 반복자 패턴과 방문자 패턴을 함께 사용해야 할 때가 많고 어떤 패턴은 다른 패턴의 대안이 되기도 하는데 원형 패턴은 추상 팩토리 패턴의 대안 패턴이고 패턴 간의 의도는 서로 다르지만 결과적으로 유사한 설계 구조를 만드는 패턴도 있는데 복합체 패턴과 장식자 패턴의 의도는 다르지만 구조는 매우 비슷
    - 패턴 간의 참조 관계에 따라 관리



Design Pattern

- Design Pattern Catalog 조직화
  - ✓ 디자인 패턴 영역

		목적		
		생성	구조	행동
범위	클래스	팩토리 메서드	적응자	해석자 템플릿 메서드
	객체	추상 팩토리 빌더 원형 단일체	적응자 가교 복합체 장식자 퍼사드 플라이급 프록시	책임 연쇄 명령 해석자 중재자 메멘토 감시자 상태 방문자

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 적당한 객체 찾기
    - 객체지향 프로그램은 객체(object)로 작업을 수행하는데 객체는 데이터와 이 데이터에 연산을 가하는 프로시저(procedure)를 함께 묶은 단위
    - 프로시저를 일반적으로 메서드(method) 또는 연산(operation)이라고 하는데 객체는 요청(request) 또는 메시지 (message)를 사용자에게 받으면 연산을 수행
    - 요청은 객체가 연산을 실행하게 하는 유일한 방법이고 연산은 객체의 내부 데이터의 상태를 변경하는 유일한 방법인데 이러한 접근의 제약 사항으로 객체의 내부 상태는 캡슐화{encapsulate)된다고 함
    - 객체 외부에서는 객체의 내부 데이터에 직접 접근할 수 없고 객체의 내부 데이터 표현 방법(데이터 타입 등)을 알 수 없음
    - 객체지향 설계의 가장 어려운 부분은 시스템을 구성할 객체의 분할을 결정하는 것으로 여러 요인을 고려해야 하기 때문에 매우 어려운 작업
    - 고려해야 할 요인에는 캡슐화, 크기 정하기, 종속성, 유연성, 성능, 진화, 재사용성 등이 있는데 이 모두를 어떻게 고려하는가에 따라서 서로 다른 방법으로 분할이 가능
    - 이 문제에 대해 객체지향 설계 방법론들은 서로 다른 방법으로 접근하는데 문제 기술서를 작성하고 명사와 동사를 추출해서 각각을 클래스와 연산으로 만드는 방법도 있으며 시스템의 협력 관계나 책임성을 중심으로 설계하는 방법도 있고 실세계를 모델로 만들고 이를 분석해 설계로 전이하는 과정에서 객체로 바꾸는 방법을 사용 할 수도 있는데 어느 방법이 가장 좋은 방법이라고 말할 수는 없음

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 적당한 객체 찾기
    - 설계 단계의 객체 대부분은 분석 모델에서부터 만들어진 것이지만 객체지향 설계는 실세계 와 대응 관계를 갖지 못할 때가 많은데 분석 모델의 객체는 실세계 객체들이지만 설계 모델의 객체에는 배열, 리스트처럼 구현에 가까운 클래스들도 있음
    - 어떤 설계 클래스들은 높은 수준의 추상화를 보일 수도 있는데 예를 들어 복합체 패턴은 분석 모델과 물리적 대응 관계를 갖지는 않지만 객체들을 동일하게 다루게 해주는 새로운 추상적 개념
    - 실세계를 그대로 반영하는 모델링만 강조하면 현재의 실세계는 반영할 수 있지만 미래의 실세계는 반영할 수 없음
    - 설계 단계 동안 만들어야 하는 새로운 추상화는 설계의 유연성을 증진하기 위한 중요한 노력 중 하나
    - 디자인 패턴은 덜 명확한 추상적 개념과 이것을 잡아낸 객체를 알아보는 데에 도움을 줌
    - 처음 객체지향 프로그래밍을 하는 개발자들에게 어떤 공정이나 알고리즘을 객체로 꾸미는 것은 자연스러운 일이 아니지만 유연한 설계를 만드는 데는 반드시 필요한 일
    - 전략 패턴은 상호 교환이 가능한 알고리즘 군을 어떻게 구현할지를 설명하는데 상태 패턴은 대상들의 각 상태를 객체로 표현하고 이 상태 객체들은 분석 단계에서는 거의 식별하기 어려우며 설계의 초기 단계에서조차 찾아내기가 쉽지 않고 이 객체는 이후에 설계를 좀더 유연 하고 재사용한 것으로 만들려는 과정을 통해서 천천히 모습을 드러냄

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체의 크기 결정
    - 객체는 크기나 개수가 딱 정해져 있지 않음
    - 하드웨어 하나 하나를 모두 객체로 표현할 수도 있지만 응용 프로그램 전체를 하나의 객체로 만들 수도 있음
    - 퍼사드 패턴은 서브 시스템을 어떻게 객체로 표현할 수 있는지 설명하고 플라이급 패턴은 규모는 작지만 개수는 많은 객체를 다루는 방법을 설명하며 또 어떤 패턴들은 객체를 좀더 작은 규모의 객체로 분할하는 구체적인 방법을 다루기도 함
    - 추상 팩토리 패턴과 빌더 패턴은 다른 객체를 생성하는 책임만 있는 객체를 만들어 내고 방문자 패턴과 명령 패턴은 요청을 자신이 처리하는 것이 아니라 다른 객체나 객체 집합이 요청을 처리하여 구현하도록 책임지는 객체를 만들어 냄

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 인터페이스의 명세
    - 객체가 선언하는 모든 연산은 연산의 이름, 매개변수로 받아들이는 객체들, 연산의 반환 값을 명세하는데 이를 연산의 시그너처(signature)라고 함
    - 인터페이스(interface)는 객체가 정의하는 연산의 모든 시그너처들을 일컫는 말로 객체의 인터페이스는 객체가 받아서 처리할 수 있는 연산의 집합
    - 객체 인터페이스에 정의 된 시그너처 와 일치하는 어떤 요청이 객체에 전달되면 객체는 연산을 수행하여 그 요청을 처리
    - 타입(type)은 특정 인터페이스를 나타낼 때 사용하는 이름으로 객체가 Window 타입을 갖는다는 것은 Window 인터페이스에 정의한 연산들을 모두 처리할 수 있다는 것을 의미
    - 객체는 여러 타입을 가질 수 있고 서로 다른 객체가 하나의 타입을 공유할 수도 있음
    - 객체의 인터페이스에 정의된 연산들 중 일부는 A 타입이 정의하는 연산이고 다른 일부는 B 타입이 정의한 연산일 수 있음
    - 같은 타입의 두 객체는 인터페이스의 일부를 공유해야 하는데 인터페이스가 다른 인터페이스를 부분집합으로 포함할 때도 있는데 다른 인터페이스를 포함하는 인터페이스를 서브 타입(subtype)이라고 하고 다른 인터페이스가 포함하는 인터페이스를 슈퍼 타입(supertype)이라 하는데 서브 타입은 슈퍼 타입의 인터페이스를 상속한다고 이야기하는데 서브 타입이 슈퍼 타입을 상속하면 서브 타입은 슈퍼 타입에 정의된 연산을 포함

Design Pattern

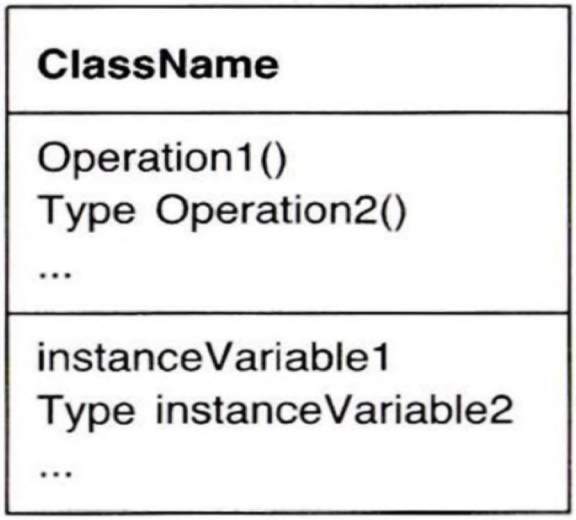
- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 인터페이스의 명세
    - 인터페이스 개념은 객체지향 시스템에서 가장 기본적인 것으로 객체는 인터페이스로 자신을 드러냄
    - 외부에서 객체를 알 수 있는 방법은 인터페이스 밖에 없기 때문에 인터페이스를 통해서만 처리를 요청할 수 있음
    - 객체의 인터페이스는 구현에 대해서는 전혀 알려주지 않기 때문에 서로 다른 객체는 인터페이스에 정의한 요청의 구현 방법을 자유롭게 선택할 수 있는데 이 의미는 동일한 인터페이스를 갖는 두 객체가 완전히 다른 구현을 가질 수 있다는 것
    - 객체에 요청이 전달되면 요청과 이를 받는 객체에 따라서 수행되는 처리 방식이 달라질 수 있는데 동일한 요청이라도 처리하는 객체들이 다른 객체라면 이 요청에 대한 구현을 어떻게 했는가에 따라서 다른 결과가 나올 수 있음
    - 어떤 요청과 그 요청을 처리할 객체를 프로그램 실행 중 즉 런타임에 연결 짓는 것을 동적 바인딩(dynamic binding)이라고 함
    - 동적 바인딩은 요청이 어떻게 구현되어 어떤 결과를 만들어 낼지를 런타임에 결정할 수 있음을 의미하는데 프로그램을 작성할 때 객체가 어떤 특정 인터페이스를 갖도록 작성하며 이 객체는 요청을 처리할 정확한 인터페이스를 갖고 있고 동적 바인딩은 프로그램이 기대하는 객체를 동일한 인터페이스를 갖는 다른 객체로 대체할 수 있게 하는데 이런 대체성을 우리는 다형성(polymorphism)이라고 하며 이는 객체지향 시스템의 핵심 개념

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 인터페이스의 명세
    - 다형성은 사용자의 정의를 단순화하고 객체 간의 결합도를 없애며 프로그램 실행 중에는 서로 간의 관련성을 다양화할 수 있게 해주는데 다시 말해 사용자는 어떤 특정 인터페이스를 제공하는 객체에게 요청을 보낸 것으로 프로그래밍하지만 런타임에 그 객체를 동일한 인터페이스를 제공하는 다른 객체로 대체할 수 있게 해서 런타임에 대체한 객체와 새로운 관련성이 수립되는 것
    - 디자인 패턴은 인터페이스에 정의해야 하는 중요 요소가 무엇이고 어떤 종류의 데이터를 주고받아야 하는지 식별하여 인터페이스를 정의하도록 도와주는데 가끔 디자인 패턴은 인터페이스에 넣지 말아야 할 것을 알려주기도 하는데 메멘토 패턴은 객체의 내부 상태를 어떻게 저장하고 캡슐화해야 하는지를 정의함으로써 객체가 나중에 그 상태로 복구할 수 있는 방법을 알려주고 이 패턴에서는 객체에 두 개의 인터페이스를 정의하도록 규정하며 이 두 가지는 사용자가 상태를 저장하고 복사할 수 있도록 해 주는 인터페이스와 원본 객체가 그 메멘토에서 상태를 저장하고 검색하기 위해 사용하는 인터페이스
    - 디자인 패턴은 인터페이스 간의 관련성도 정의하는데 특히 클래스 간에 유사한 인터페이스를 정의하도록 하거나 클래스의 인터페이스에 여러 가지 제약을 정의하는데 예를 들어 장식자 패턴과 프록시 패턴은 장식되고 중재되는 객체와 동일한 인터페이스를 갖도록 장식자 객체와 프록시 객체의 인터페이스를 요청하는데 이는 프록시 객체의 인터페이스는 자신이 대리하는 다른 객체의 인터페이스와 동일하다는 것
    - 방문자 패턴에서 방문자 인터페이스는 방문자 객체가 방문하는 객체들의 클래스 인터페이스를 그 방문자 인터페이스에 모두 반영하도록 함

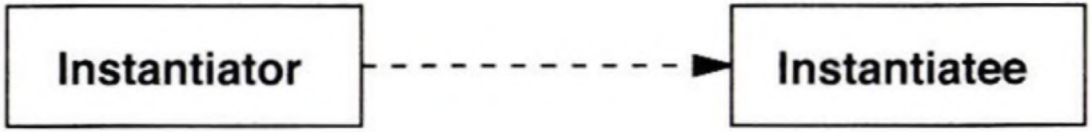
Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 구현 명세
    - 어떤 객체의 구현은 클래스(class)에서 정의(define)
    - 클래스는 객체의 내부 데이터와 표현 방법을 명세하고, 그 객체가 수행할 연산을 정의
    - OMT 기반의 표기법에서는 클래스를 표현하는 사각형에 진한 글자체로 클래스의 이름을 표현하고 연산의 이름은 클래스 이름 아래 줄에 나열하고 클래스가 정의하는 데이터는 연산 아래 줄에 표시
    - 클래스 이름과 연산 이름, 연산과 데이터를 구분하는 선을 그음



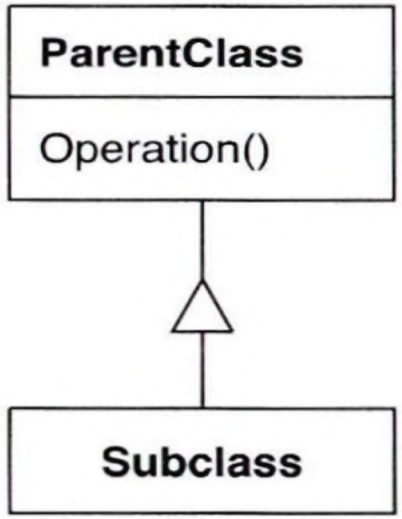
Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 구현 명세
    - 어떤 특정한 구현 언어를 가정하지 않았기 때문에 연산의 반환 타입과 인스턴스 변수 타입의 정의는 선택 사항
    - 객체는 클래스를 인스턴스로 만듦으로써 생성되므로 객체는 클래스의 인스턴스라고 할 수 있음
    - 클래스의 인스턴스화 과정은 객체의 내부 데이터(instance variable)에 대한 공간을 할당하고 이 데이터들을 연산과 관련짓는 것
    - 클래스의 인스턴스화 과정을 통해 객체의 인스턴스를 얻게 됨
    - 점선 화살표는 한 클래스(instantiator에 해당)가 다른 클래스(instantiatee에 해당)의 객체를 인스턴스화함을 의미하며 화살표의 방향은 생성할 객체의 클래스(instan tiatee)로 향함



Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 구현 명세
    - 새로운 클래스는 기존 클래스에 기반을 둔 클래스 상속을 사용하여 정의할 수 있는데 서브 클래스(subclass)가 부모 클래스(parent class)를 상속하면 부모 클래스가 갖는 모든 데이터와 연산을 서브 클래스가 갖게 됨
    - 서브 클래스의 인스턴스는 부모 클래스가 정의한 모든 데이터를 가지며 부모 클래스가 정의한 연산을 모두 수행할 수 있음
    - 서브 클래스 관계는 다음과 같이 수직선과 삼각형을 써서 나타내는데 삼각형의 밑변이 서브 클래스이며 꼭지점이 부모 클래스

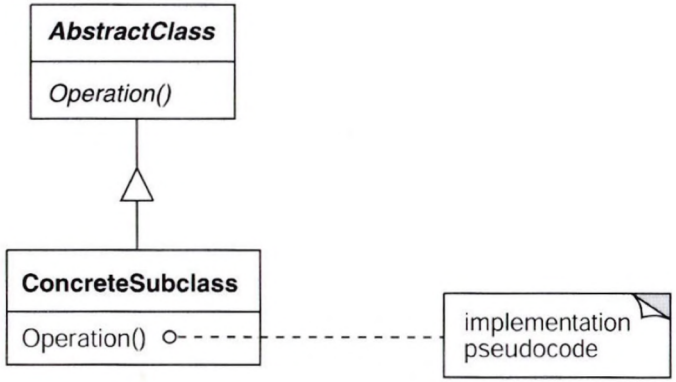


Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 구현 명세
    - 추상 클래스(abstract class)는 모든 서브 클래스 사이의 공통되는 인터페이스를 정의
    - 추상 클래스는 정의한 모든 연산이나 일부 연산의 구현을 서브 클래스에게 넘기는데 정의한 연산 모두가 추상 클래스로 구현된 것이 아니므로 추상 클래스는 인스턴스를 생성할 수 없음
    - 정의만 하고 구현하지 않는 연산을 추상 연산(abstract operation)이라 하고 추상 클래스가 아닌 클래스를 구체 클래스(concrete class)라고 함
    - 서브 클래스는 부모 클래스가 정의한 행동을 재정의하거나 정제할 수 있는데 서브 클래스는 부모 클래스에 정의한 연산의 구현을 바꿀 수 있는데 이를 오버라이드 (override) 하고 하고 서브 클래스는 부모 클래스에 정의된 처리 방식을 변경할 수 있음
    - 클래스 상속은 다른 클래스를 확장하여 새로운 클래스를 정의할 수 있게 하는데 이로써 비슷한 기능성을 갖는 객체 군을 정의할 수 있게 되는 것
    - 추상 클래스의 이름은 이탤릭체로 표기하여 다른 클래스와 구분
    - 추상 연산도 이탤릭체를 이용
    - 다이어그램에 연산 구현에 대한 사항을 언급할 수 있는데 이는 모서리가 접힌 노트 기호에 작성하고 연산과 실선으로 연결하면 됨

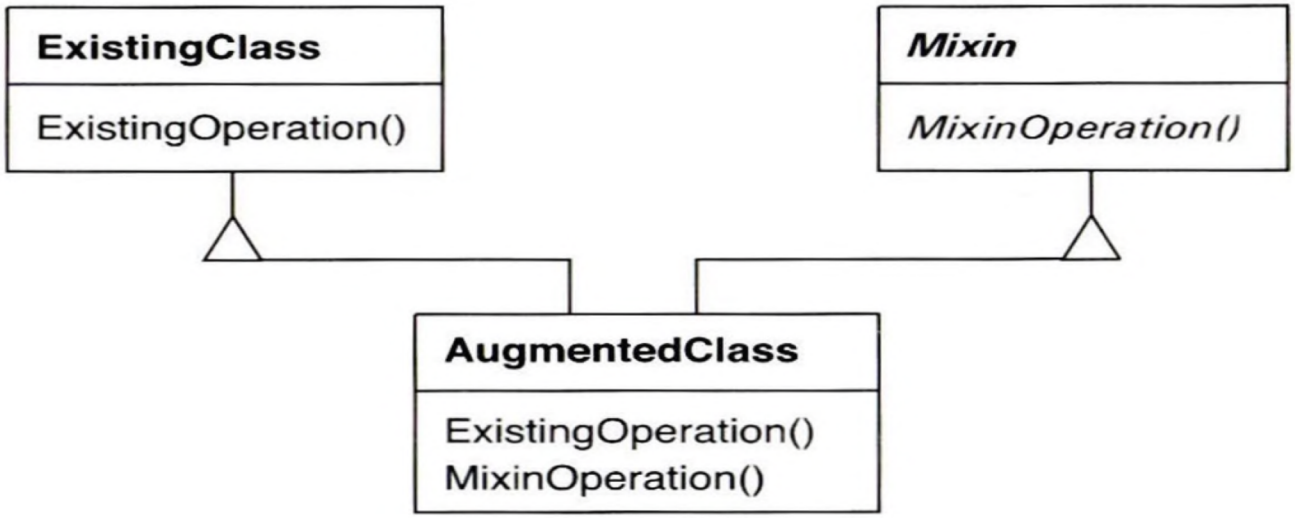
Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 구현 명세



Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 객체 구현 명세
    - 믹스인(mixin) 클래스는 다른 클래스들에게 선택적인 인터페이스 혹은 기능을 제공하려는 목적을 가진 클래스
    - 인스턴스로 만들 의도가 없다는 면에서 추상 클래스와 비슷한데 믹스인 클래스를 사용하기 위해서는 다중 상속이 필요
    - AugmentedClass는 ExistingClass를 통해 연산과 그 구현을 상속받고 Mixin 클래스를 통해서 인터페이스를 상속받지만 MixinOperation은 자신이 직접 구현해야 함



Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 클래스 상속 대 인터페이스 상속
    - 객체의 클래스는 그 객체가 어떻게 구현되느냐를 정의하는데 클래스는 객체의 내부 상태와 그 객체의 연산에 대한 구현 방법을 정의하는 반면 객체의 타입은 그 객체의 인터페이스 즉 그 객체가 응답할 수 있는 요청의 집합을 정의
    - 하나의 객체가 여러 타입을 가질 수 있고 서로 다른 클래스의 객체들이 동일한 타입을 가질 수 있음
    - 객체의 구현은 다를지라도 인터페이스는 같을 수 있다는 의미
    - 클래스와 타입 간에는 밀접한 관련이 있는데 클래스도 객체가 수행할 수 있는 연산을 정의하므로 객체의 타입을 정의하는 것이기도 하므로 어떤 객체가 어떤 클래스의 인스턴스라고 말할 때 그 객체는 그 클래스가 정의한 인터페이스를 지원한다는 뜻이 숨어있다고 보면 됨
    - C++와 Eiffel 같은 언어에서 클래스는 객체 타입과 구현 모두를 의미
    - 스몰토크 프로그램에서는 변수의 타입을 정의하지 않기 때문에 컴파일러는 변수에 할당된 객체의 타입이 변수 타입의 서브 클래스인지를 점검하지 않음
    - 어떤 메시지를 보내려면 수신 객체의 클래스가 이 메시지를 구현하고 있는지 우선 확인해야 하지만 그렇다고 꼭 수신측 객체가 어떤 특정 클래스의 인스턴스일 필요는 없고 단지 메시지를 처리할 수 있는지의 여부만을 확인하는 것

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 클래스 상속 대 인터페이스 상속
    - 클래스 상속은 객체의 구현을 정의할 때 이미 정의된 객체의 구현을 바탕으로 하는데 코드와 내부 표현 구조를 공유하는 메커니즘
    - 인터페이스 상속(서브 타이핑)은 어떤 객체가 다른 객체 대신에 사용될 수 있는 경우를 지정하는 메커니즘으로 인터페이스 상속 관계가 있다면 프로그램에는 슈퍼 타입으로 정의하지만 런타임에 서브 타입의 객체로 대체할 수 있음
    - 많은 언어가 이 두 개념을 구분하지 않기 때문에 두 개념을 혼동하기 쉬운데 C++와 Eiffel 언어에서 상속은 인터페이스 와 구현 상속 모두를 의미
    - C++에서 인터페이스를 상속하는 표준적인 방법은 (순수) 가상 함수를 갖는 클래스를 public으로 상속하는 것인데 public으로 상속되면 서브 클래스도 부모 클래스가 갖는 가상 함수를 상속받고 서브 클래스가 구현을 담당하며 상속받은 인터페이스가 서브 클래스의 사용자에게도 공개
    - C++에서 순수한 인터페이스 상속은 순수 가상 함수를 정의한 추상 클래스를 public으로 상속하면 비슷하게 구현할 수 있는데 순수 가상 함수는 전혀 구현을 정의할 수 없는 함수이기 때문에 이를 상속한다는 것은 진정한 의미의 인터페이스 만을 상속받는다는 의미
    - 구현이나 클래스의 상속은 private 상속으로 비슷하게 얻을 수 있는데 private로 상속하면 부모 클래스에 정의된 연산은 서브 클래스의 사용자에게는 공개되지 않기 때문에 상속의 목적은 인터페이스 확장이 아닌 부모 클래스 구현의 재사용

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 클래스 상속 대 인터페이스 상속
    - 스몰토크에서 상속은 단순히 구현 상속 만을 의미
    - 스몰토크에서는 특정한 변수(객체)의 값에 정의된 연산을 지원하기만 하면 어떤 클래스의 인스턴스라도 다른 변수에 대입(assign)할 수 있음
    - 대부분의 프로그래밍 언어는 인터페이스와 구현 상속을 구분하지 않지만 프로그래머들은 실제로 구분해서 사용
    - 스몰토크 프로그래머들은 서브 클래스를 서브 타입으로 사용하고 C++ 프로그래머들은 구체적인 클래스를 정의하고 요청을 보내기보다 추상 클래스의 객체에게 메시지를 보내도록 프로그래밍하는데 이렇게 하면 런타임에 구체 클래스의 인스턴스로 바꿀 수 있는데 추상 클래스를 상속한다는 것은 단순한 코드의 재사용을 위한 상속이 아니라 추상 클래스가 정의하는 인터페이스를 상속하겠다는 의미
    - 책임 연쇄 패턴에 나오는 객체들은 반드시 동일한 타입을 가져야 하지만 이들이 구현을 공유할 부분은 없고 복합체 패턴에서 Component 클래스는 공통의 인터페이스를 정의하고, Composite 클래스는 공통의 구현을 정의하며 명령, 감시자, 상태, 전략 패턴은 순수 인터페이스인 추상 클래스를 써서 구현될 때가 많음

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 구현에 따르지 않고 인터페이스에 따르는 프로그래밍
    - 클래스 상속은 기본적으로 부모 클래스에서 정의한 구현을 재사용하여 응용 프로그램의 기능성을 확장하려는 메커니즘
    - 이미 있는 것을 이용해서 새로운 객체를 빨리 정의해 보려는 것으로 기존의 클래스를 그대로 상속할 수 있다면 새로운 구현에 드는 비용은 공짜인데 구현의 재사용이 전부는 아니고 상속이 가진 다른 기능들 중에는 동일한 인터페이스를 갖는 객체 군을 정의하는 것이 있는데 매우 중요한 특징
    - 객체 군을 정의하는 것이 중요한 이유는 그것으로 다형성을 끌어낼 수 있기 때문
    - 상속을 적절하게 이용하면 모든 클래스는 추상 클래스를 상속하도록 하여 인터페이스를 공유할 수 있게 되는데 이것은 서브 클래스가 단순히 연산을 추가하거나 재정의할 뿐 부모 클래스의 연산을 감추지 않는다는 뜻
    - 모든 서브 클래스들은 추상 클래스에 정의한 인터페이스를 처리할 수 있는데 부모 클래스에 정의된 요청이 서브 클래스에게 전달되어도 서브 클래스는 이를 처리할 수 있다는 의미인데 이로써 모든 서브 클래스들은 부모 클래스의 서브 타입이 되는 것
    - 추상 클래스를 정의하고 인터페이스 개념으로 객체를 다룰 때 얻을 수 있는 이점
      - ◆ 사용자가 원하는 인터페이스를 그 객체가 만족하고 있는 한 사용자는 그들이 사용하는 특정 객체 타입에 대해 알아야 할 필요는 없음
      - ◆ 사용자는 이 객체들을 구현하는 클래스를 알 필요가 없고 단지 인터페이스를 정의하는 추상 클래스가 무엇 인지만 알면 됨

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 구현에 따르지 않고 인터페이스에 따르는 프로그래밍
    - 구현이 아닌 인터페이스에 따라 프로그래밍
      - 어떤 변수(객체)를 구체 클래스의 인스턴스로 선언하는 일은 피하는 대신 추상 클래스의 인터페이스를 따르는 인스턴스 변수를 정의
      - 추상 팩토리, 빌더, 팩토리 메서드, 원형 패턴 및 단일체 패턴에서는 구체 클래스에서 인스턴스를 생성하도록 하고 있는데 이들 패턴에서는 객체 생성의 과정을 추상화함으로써 인스턴스화할 때 인터페이스와 구현을 연결하는 다른 방법을 제시해서 생성 패턴 역시도 시스템이 구현의 관점이 아닌 인터페이스 관점으로 작성되도록 보장하는 것

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 재사용을 실현 가능한 것으로
    - 상속 대 합성
      - 객체지향 시스템에서 기능의 재사용을 위해 구사하는 가장 대표적인 기법은 클래스 상속, 그리고 객체 합성(object composition)
      - 클래스 상속은 서브 클래싱 즉 다른 부모 클래스에서 상속받아 한 클래스의 구현을 정의하는 것으로 서브 클래싱에 의한 재사용을 화이트박스 재사용(white-box reuse)이라고 함
      - 화이트박스는 내부를 볼 수 있다는 의미에서 나온 말인데 상속을 받으면 부모 클래스의 내부가 서브 클래스에 공개되기 때문에 화이트박스
      - 객체 합성은 클래스 상속에 대한 대안으로 다른 객체를 여러 개 붙여서 새로운 기능 혹은 객체를 구성하는 것으로 객체를 합성하려면 합성에 들어가는 객체들의 인터페이스를 명확하게 정의해 두어야 하는데 이런 스타일의 재사용을 블랙박스 재사용(black-box reuse)이라고 하는데 객체의 내부는 공개되지 않고 인터페이스를 통해서만 재사용되기 때문
      - 상속과 합성은 서로 장단점을 가지고 있는데 클래스 상속은 컴파일 시점에 정적으로 정의되고 프로그래밍 언어가 직접 지원하므로 그대로 사용하면 되므로 클래스 상속으로 부모 클래스의 구현을 쉽게 수정할 수도 있는데 서브 클래스는 모든 연산이 아닌 일부만 재정의할 수도 있음

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 재사용을 실현 가능한 것으로
    - ● 상속 대 합성
      - ◆ 클래스 상속의 단점
        - ▪ 런타임에 상속받은 부모 클래스의 구현을 변경할 수는 없다는 점인데 상속은 컴파일 시점에 결정되는 사항이기 때문
        - ▪ 부모 클래스는 서브클래스의 물리적 표현의 최소 부분만을 정의하기 때문에 서브 클래스는 부모 클래스가 정의한 물리적 표현들을 전부 또는 일부 상속받는다는 점으로 상속은 부모 클래스의 구현이 서브 클래스에 다 드러나는 것이기 때문에 상속은 캡슐화를 파괴한다고 주장하는 의견도 있음
        - ▪ 서브 클래스는 부모 클래스의 구현에 종속될 수밖에 없으므로 부모 클래스 구현에 변경이 생기면 서브 클래스도 변경해야 하므로 이 구현의 종속성이 걸림돌로 작용하면서 서브 클래스를 재사용하려고 할 때 문제가 발생
        - ▪ 상속한 구현이 새로운 문제에 맞지 않을 때 부모 클래스를 재작성해야 하거나 다른 것으로 대체하는 일이 생기게 되는데 이런 종속성은 유연성과 재사용성을 떨어뜨리는데 이를 해결하는 방법 한 가지는 추상 클래스에서만 상속받는 것으로 추상 클래스에는 구현이 거의 없거나 아예 없으므로 이미 추상 클래스를 상속했다는 것은 구현이 아닌 인터페이스를 상속한 것이므로 구현 자체는 서브 클래스가 정의하는 것으로 구현이 변경되면 서브 클래스만 변경하면 되고 상위 추상 클래스는 고려할 필요가 없음

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 재사용을 실현 가능한 것으로
    - ● 상속 대 합성
      - ◆ 객체 합성은 한 객체가 다른 객체에 대한 참조자를 얻는 방식으로 런타임에 동적으로 정의되는데 합성은 객체가 다른 객체의 인터페이스만을 바라보게 하기 때문에 인터페이스 정의에 더 많은 주의를 기울여야 함
      - ◆ 객체는 인터페이스에서만 접근하므로 캡슐화를 유지할 수 있으며 동일한 타입을 갖는다면 다른 객체로 런타임에 대체가 가능
      - ◆ 객체는 인터페이스에 맞춰 구현되므로 구현 사이의 종속성은 확실히 줄어듬
      - ◆ 객체 합성은 시스템 설계에 또 다른 영향을 끼치는데 클래스 상속보다 객체 합성을 더 선호하는 이유는 각 클래스의 캡슐화를 유지할 수 있고 각 클래스의 한 가지 작업에 집중할 수 있기 때문
      - ◆ 클래스와 클래스 계층이 소규모로 유지되면서 통제 불능의 괴물로 자랄 가능성은 적은데 객체 합성으로 설계되면 클래스의 수는 적어지고 객체의 수는 좀더 많아질 수 있지만 시스템의 행동은 클래스에 정의된 정적인 내용보다는 런타임에 드러나는 객체 합성에 의한 상호 관련성에 따라 달라질 수 있음

Design Pattern

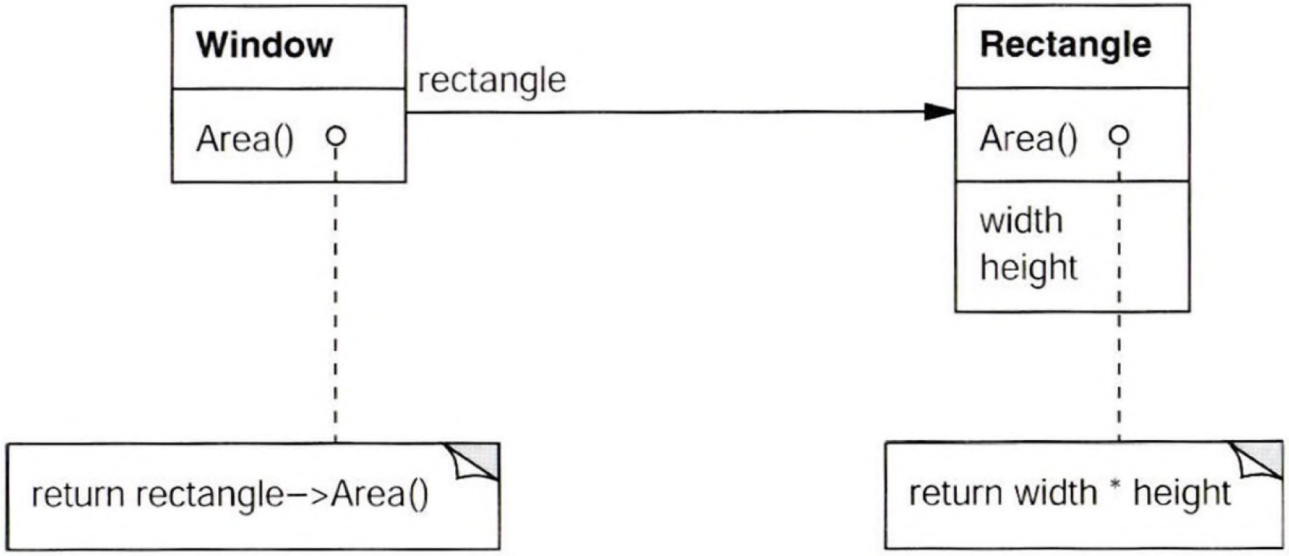
- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 재사용을 실현 가능한 것으로
    - 객체 합성이 클래스 합성보다 더 나은 방법
      - 재사용을 위해서 새로운 구성 요소를 생성할 필요없이 필요한 기존의 구성 요소를 조립해서 모든 새로운 기능을 얻어올 수 있으나 가능한 구성 요소의 집합이 실제로 사용할 수 있을 만큼 충분하지 않기 때문에 기존 구성 요소의 조합을 통한 재사용만으로 목적을 달성할 수 있는 경우는 드뭄
      - 상속에 의한 재사용은 기존 클래스들을 조합해서 새로운 구성 요소를 쉽게 만들 수 있도록 해주므로 상속과 객체 합성은 적절히 조합되어야 완벽한 재사용이 가능
      - 설계자들은 재사용 기법으로 상속을 많이 쓰지만 객체 합성으로 더욱 재사용 가능 한 설계를 만들 수 있음

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 위임
    - 위임(delegation)은 합성을 상속만큼 강력하게 만드는 방법
    - 위임에서는 두 객체가 하나의 요청을 처리하는데 수신 객체가 연산의 처리를 위임자(delegate)에게 보내는데 이는 서브 클래스가 부모 클래스에게 요청을 전달하는 것과 유사한 방식
    - 상속에서는 상속받은 연산이 늘 수신 객체를 참조하게 되는데 이때 C++에서는 this를 이용하고 스몰토크에서는 self를 이용해서 수신 객체를 참조
    - 위임과 동일한 효과를 얻으려면 수신 객체는 대리자에게 자신을 매개변수로 전달해서 위임된 연산이 수신자를 참조하게 함
    - Window 클래스를 Rectangle 클래스의 서브 클래스로 만드는 대신 Window 클래스는 Rectangle 클래스를 자신의 인스턴스 변수로 만들고 Rectangle 클래스에 정의된 행동이 필요할 때는 Rectangle 클래스에 위임함으로써 Rectangle의 행동을 재사용할 수도 있는데 상속에 의해 Window 인스턴스를 Rectangle 인스턴스로 간주하는 방식이 아닌 Window 인스턴스가 Rectangle 인스턴스를 포함하도록 하고 Window 인스턴스는 자신이 받은 요청을 Rectangle 인스턴스로 전달하는 것

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 위임



Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 위임
    - Window 클래스는 Area() 연산을 Rectangle 인스턴스에 전달
    - 실선 화살표는 한 클래스가 다른 클래스의 인스턴스에 대한 참조자를 갖고 있음을 보여줌
    - 참조는 이름을 선택적으로 정의할 수 있는데 이때는 rectangle로 정의
    - 위임의 가장 중요한 장점은 런타임에 행동의 복합을 가능하게 하고 복합하는 방식도 변경해 준다는 것
    - Window 객체가 런타임에 Rectangle 인스턴스를 Circle 인스턴스로 대체하면 원형의 윈도우가 될 것인데 물론 이를 위해서는 Rectangle 클래스 및 Circle 클래스가 동일한 타입이라는 가정이 필요
    - 위임이 갖는 단점은 객체 합성을 통해 소프트웨어 설계의 유연성을 보장하는 방법 과 동일하게 동적인데다가 고도로 매개변수 화된 소프트웨어는 정적인 소프트웨어 구조보다 이해하기가 더 어렵다는 것인데 그 이유는 클래스에 상호 작용이 다 정의되어 있는 것이 아니라 런타임 객체에 따라서 그 결과가 다르기 때문이며 또한 런타임에 비효율적일 수 있음
    - 이런 위임이 만들어 내는 복잡함보다 단순화의 효과를 더 크게 할 수 있다면 그 설계는 사용하기 좋은 설계이지만 이러한 유용성은 상황에 따라 다르고 얼마나 많은 경험을 갖고 있는가에 좌우되므로 위임은 고도로 표준화된 패턴에서 사용하는 것이 최상

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 위임
    - 몇 개의 디자인 패턴은 위임을 부분적으로 사용하는데 상태, 전략, 방문자 패턴에서 위임 방식을 사용
    - 상태 패턴에서 객체는 현재 상태를 표현하는 상태 객체에 요청의 처리를 위임하고 전략 패턴에서 객체는 요청을 수행하는 추상화 한 전략 객체에게 특정 요청을 위임하는데 이 두 패턴의 목적은 처리를 전달하는 객체를 변경하지 않고 객체의 행동을 변경할 수 있게 하자는 것
    - 방문자 패턴에서 객체 구조의 각 요소에 수행하는 연산은 언제나 방문자 객체에게 위임된 연산
    - 위임에 전적으로 의존하는 패턴들도 있는데 중재자 패턴은 객체 간의 교류를 중재하는 객체를 도입하여 중재자 객체가 다른 객체로 연산을 전달하도록 구현하며 이때 연산에 자신에 대한 참조자를 함께 보내고 위임받은 객체가 다시 자신에게 메시지를 보내서 자신이 정의한 데이터를 얻어가게 함으로써 진정한 위임을 구현
    - 책임 연쇄 패턴은 한 객체에서 다른 객체로 고리를 따라서 요청의 처리를 계속 위임하는데 이 요청에는 요청을 처음 받은 원본 객체에 대한 참조자를 포함
    - 가교 패턴은 구현과 추상적 개념을 분리하는 패턴으로 추상화와 특정 구현을 대응시키고 추상화는 단순히 자신의 연산을 구현에 전달
    - 위임은 객체 합성의 극단적인 예로서 코드 재사용을 위한 메커니즘으로 상속을 객체 합성으로 대체할 수 있음

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 상속 대 매개변수 화된 타입
    - 기능의 재사용에 이용할 수 있는 다른 방법이 매개변수화된 타입(parameterized type)
    - Ada와 Eiffel에서는 제네릭(generic)이라고 하며 C++에서는 템플릿(template)
    - 이 기법은 타입을 정의할 때 타입이 사용하는 다른 모든 타입을 다 지정하지 않은 채 정의
    - 미리 정의하지 않은 타입은 매개변수로 제공하는데 예를 들어 List 클래스는 내부에 포함할 원소들의 타입으로 매개변수 화 될 수 있는데 정수형의 리스트를 선언하고 싶으면 List 타입에 Integer 타입을 매개변수로 넘겨주면 되고 String 객체의 리스트를 만들고 싶다면 매개변수로 String 을 넘기면 됨
    - 매개변수화된 타입은 객체지향 시스템에서 행동을 복합할 수 있는 세 번째 방법으로 첫 번째가 클래스 상속이었고 두 번째가 객체 합성이었는데 어지간한 설계는 대부분 이 세 가지 기법 중 어느 하나를 써서 구현
    - 원소들을 비교하기 위한 정렬 루틴을 설계하는 세 가지 방법을 비교
      - 서브 클래스에 의해 연산을 구현하는 방법(템플릿 메서드 패턴의 응용): 상속
      - 정렬 루틴으로 전달된 객체(전략): 합성
      - C++ 템플릿이나 Ada의 제네릭으로 정의한 클래스의 인자로 원소를 비교할 함수 이름을 명시: 매개변수 화

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 상속 대 매개변수 화된 타입
    - 세 가지 기법에는 중요한 차이가 있는데 객체 합성은 런타임에 행동을 변경할 수 있지만 행동이 위임되기 때문에 비효율적일 수 있으며 상속이 연산에 대한 기본 행동을 부모 클래스가 제공하고 이를 서브 클래스에서 재정의하도록 하는 것이라면 매개변수 화된 타입은 클래스가 사용하는 타입을 변경하게 하는 것
    - 상속도 매개변수 화된 타입이라고 볼 수 있지만 런타임에 변경이 일어나지는 않은데 어떤 방법이 최적의 방법인가 하는 것은 설계와 구현 제약 사항에 따라 달라질 수 있음

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 런타임 및 컴파일 타임의 구조를 관계 짓기
    - 객체지향 프로그램의 실행 구조는 종종 코드 구조와 일치하지 않은데 코드 구조는 컴파일 시점에 확정되는 것이고 이 구조에는 고정된 상속 클래스 관계들을 포함하지만 프로그램의 런타임 구조는 교류하는 객체들에 따라서 달라질 수 있음
    - 이 두 구조는 전혀 다른 별개의 독립성을 갖는데 한쪽 구조를 가지고서 다른 하나를 이해하려는 것은 생태계의 동적인 성질을 식물과 동물과 같은 정적 분류 구조를 바탕으로 이해하려는 것과 동일
    - 객체 관계 중에는 집합(aggregation)과 인지(acquaintance)라는 것이 있는데 집합은 한 객체가 다른 객체를 소유하거나 그것에 책임을 진다는 의미인데 보통 우리는 한 객체가 다른 객체를 포함한다거나 다른 객체의 부분이라고 하는데 객체 통합에는 통합된 객체 및 그 객체를 소유한 객체의 생존 주기가 똑같다는 의미도 들어 있음
    - 객체 인지는 한 객체가 다른 객체에 대해 알고 있음(kmms of)을 의미하는데 이를 연관(association) 관계 또는 사용(using) 관계라고도 하는데 인지를 받는 객체는 서로의 연산을 요청할 수도 있지만 서로에 대해 책임은 지지 않음
    - 인지 관계는 통합 관계보다 관련성이 약해서 객체들 사이의 결합도가 약함
    - 다이어그램에서 실선 화살표는 인지 관계를 의미하는데 다이아몬드를 갖는 실선 화살표는 집합 관계를 의미

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 런타임 및 컴파일 타임의 구조를 관계 짓기
    - 사실 집합 관계와 인지 관계는 쉽게 구분하기가 까다로움
    - 두 관계를 구현하는 방법이 구현상으로 동일할 때가 잦기 때문
    - 스몰토크에서 모든 변수는 다른 객체에 대한 참조자이므로 집합 관계와 인지 관계 사이에 차이가 없음
    - C++에서는 멤버 변수를 다른 객체의 인스턴스로 정의하여 집합 관계를 구현하지만 집합 관계를 표현하는 더 일반적인 방법은 다른 인스턴스를 가리키는 포인터를 정의하는 것인데 인지 관계 역시 포인터로 구현
    - 인지 관계와 집합 관계는 언어의 처리 방식이 아닌 사용 목적에 따라 결정해야 함
    - 이러한 차이를 컴파일 시점에 발견하기는 힘들지만 중요한 의미를 갖는데 집합 관계는 인지 관계보다는 강력한 영속성의 개념을 갖는데 자전거에 바퀴가 있어야 한다는 것은 불변의 영속적 사실이며 이에 반해 인지 관계는 자주 바뀌게 되는데 사람과 회사 관계는 근무한다는 관련성이 있을 수도 있고 없어질 수도 있는데 인지 관계가 더 동적이라는 의미
    - 컴파일 시점의 구조와 런타임의 구조 간에 차이가 있기 때문에 코드 자체가 시스템 의 동작 방법을 모두 보여줄 수 없는데 시스템의 런타임 구조는 언어가 아닌 설계자가 만드는데 객체와 타입 사이의 관계는 대단히 세심하게 설계해야 하는데 어떻게 만들어져 있느냐에 따라 런타임 구조가 좋아지거나 망가지기 때문

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 런타임 및 컴파일 타임의 구조를 관계 짓기
    - 많은 디자인 패턴이 컴파일 시점과 런타임 구조를 명시적으로 구분하고 있는데 복합체 패턴과 장식자 패턴은 복잡한 실행 구조를 구축하는 데 유용한 패턴이며 감시자 패턴으로 만드는 런타임 구조는 이 패턴을 잘 알고 있지 않는 한 이해하기가 종종 까다로움
    - 책임 연쇄 패턴은 상속이 드러나지 않는 교류 패턴을 만들어 냄
    - 일반적으로 런타임 구조는 패턴을 잘 이해할 때까지는 코드만 보고는 명확하게 이해할 수 없음

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 변화에 대비한 설계
    - 재사용을 최대화하기 위해서는 새로운 요구 사항과 기존 요구 사항에 발생한 변경을 예측하여 앞으로의 시스템 설계가 진화할 수 있도록 해야 함
    - 변화에 잘 대응하기 위한 소프트웨어를 설계하기 위해서는 소프트웨어를 운영하는 동안 앞으로 일어날 변화를 어떻게 수용할 것인가를 미리 고려해야 함
    - 변화를 수용하지 못하는 설계는 앞으로 재설계가 필요하게 되는데 이런 변경들은 클래스의 재설계와 재구현, 사용자의 수정, 새로운 테스팅을 유발하는데, 재설계의 영향은 소프트웨어의 여러 부분에서 나타날 수 있으며 예측하지 못한 변경에 대해서는 엄청 나게 비싼 대가를 지불할 수밖에 없음
    - 디자인 패턴은 어떤 구체적인 원인으로 앞으로 시스템을 변경해야 한다는 것을 미리 보여줌으로써 이런 위험을 줄여줌
    - 디자인 패턴은 다른 부분에 독립적으로 시스템 구조를 변경할 수 있게 하여 시스템이 어떤 특정 변화에 순응할 수 있도록 함

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 변화에 대비한 설계
    - ● 디자인 패턴을 써서 재설계를 할 수밖에 없게 하는 이유
      - ◆ 특정 클래스에서 객체 생성
        - ▪ 객체를 생성할 때 클래스 이름을 명시하면 어떤 특정 인터페이스가 아닌 어떤 특정 구현에 종속됨
        - ▪ 이런 종속은 앞으로 의 변화를 수용하지 못하는데 이를 방지하려면 객체를 직접 생성해서는 안됨
        - ▪ 디자인 패턴: 추상 팩토리, 팩토리 메서드, 원형
      - ◆ 특정 연산에 대한 의존성
        - ▪ 특정한 연산을 사용하면 요청을 만족하는 한 가지 방법에만 매이게 됨
        - ▪ 요청의 처리 방법을 직접 코딩하는 방식을 피하면 컴파일 시점과 런타임 모두를 만족하면서 요청 처리 방법을 쉽게 변경할 수 있음
        - ▪ 디자인 패턴: 책임 연쇄, 명령

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 변화에 대비한 설계
    - ● 디자인 패턴을 써서 재설계를 할 수밖에 없게 하는 이유
      - ◆ 하드웨어와 소프트웨어 플랫폼에 대한 의존성
        - ▪ 기존에 존재하는 시스템 인터페이스 와 응용 프로그램 프로그래밍 인터페이스는 소프트웨어 및 하드웨어 플랫폼마다 모두 다름
        - ▪ 특정 플랫폼에 종속된 소프트웨어는 다른 플랫폼에 이식하기도 어렵고 또한 본래의 플랫폼에서도 버전의 변경을 따라가기 어려울 수도 있음
        - ▪ 플랫폼 종속성을 제거하는 것은 시스템 설계에 있어 매우 중요
        - ▪ 디자인 패턴: 추상 팩토리, 가교
      - ◆ 객체의 표현이나 구현에 대한 의존성
        - ▪ 사용자가 객체의 표현 방법, 저장 방법, 구현 방법, 존재의 위치에 대한 모든 방법을 알고 있다면 객체를 변경할 때 사용자도 함께 변경해야 하는데 이런 정보를 사용자에게 감춤으로써 변화의 파급을 막을 수 있음
        - ▪ 디자인 패턴: 추상 팩토리, 가교, 메멘토, 프록시

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 변화에 대비한 설계
    - ● 디자인 패턴을 써서 재설계를 할 수밖에 없게 하는 이유
      - ◆ 알고리즘 의존성
        - ▪ 알고리즘 자체를 확장할 수도 최적화할 수도 다른 것으로 대체할 수도 있는데 알고리즘에 종속된 객체라면 알고리즘이 변할 때마다 객체도 변경해야 하므로 변경이 가능한 알고리즘은 분리해 내는 것이 바람직
        - ▪ 디자인 패턴: 빌더, 반복자, 전략, 템플릿 메서드, 방문자
      - ◆ 높은 결합도
        - ▪ 높은 결합도를 갖는 클래스들은 독립적으로 재사용하기 어려움
        - ▪ 높은 결합도를 갖게 되면 하나의 커다란 시스템이 되어 버리고 이렇게 되면 클래스 하나를 수정하기 위해서 전체를 이해해야 하고 다른 많은 클래스도 변경해야 하며 또한 시스템은 배우기도 힘들고 이식은 커녕 유지 보수하기 조차도 어려운 공룡이 되어 버림
        - ▪ 약한 결합도는 클래스 자체의 재사용을 가능하게 하고 시스템의 이해 와 수정, 확장이 용이해서 이식성을 증대시킴
        - ▪ 추상 클래스 수준에서 결합도를 정의한다거나 계층화시키는 방법으로 디자인 패턴은 낮은 결합도의 시스템을 만들도록 함
        - ▪ 디자인 패턴: 추상 팩토리, 가교, 책임 연쇄, 명령, 퍼사드, 중재자, 감시자

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 변화에 대비한 설계
    - ● 디자인 패턴을 써서 재설계를 할 수밖에 없게 하는 이유
      - ◆ 서브 클래싱을 통한 기능 확장
        - ▪ 서브 클래싱으로 객체를 재정의하는 것은 쉬운 일이 아님
        - ▪ 새로운 클래스마다 매번 반드시 해야 하는 초기화, 소멸 등에 대한 구현 오버헤드를 늘 가지게 됨
        - ▪ 서브 클래스를 정의하려면 최상위 클래스부터 자신의 직속 부모 클래스까지 모든 것을 이해하고 있어야 하는데 예를 들어 하나의 연산을 재정의하려면 상속받은 연산을 호출해야 할 때가 있기 때문에 모든 부모 클래스를 다 이해하고 어떤 클래스에 정의된 연산을 호출할지 결정할 수 있어야 하고 또한 단순히 확장 만을 이유로 새로운 서브 클래스를 만든다면 서브 클래싱은 클래스의 수를 엄청나게 증가시킬 수도 있음
        - ▪ 일반적으로 객체 합성과 위임은 행동 조합을 위한 상속보다 훨씬 유연한 방법
        - ▪ 기존 객체들을 새로운 방식으로 조합함으로써 새로운 서브 클래스를 정의하지 않고도 응용 프로그램에 새로운 기능성을 추가할 수 있음
        - ▪ 객체 합성을 많이 사용한 시스템은 이해하기가 어려워지는데 많은 디자인 패턴에서는 그냥 서브 클래스를 정의하고 다른 인스턴스와 새로 정의한 클래스의 인스턴스를 합성해서 기능을 재정의하는 방법을 도입
        - ▪ 디자인 패턴: 가교, 책임 연쇄, 장식자, 감시자, 전략

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 변화에 대비한 설계
    - ● 디자인 패턴을 써서 재설계를 할 수밖에 없게 하는 이유
      - ◆ 클래스 변경이 편하지 못한 점
        - ▪ 가끔 클래스를 변경하는 작업이 그렇게 단순하지 않을 때가 많음
        - ▪ 소스 코드가 필요한데 없다고 가정하거나 또한 어떤 변경을 하면 기존 서브 클래스의 다수를 수정해야 한다고 가정
        - ▪ 디자인 패턴은 이런 환경에서 클래스를 수정하는 방법을 제시
        - ▪ 디자인 패턴: 적응자, 장식자, 방문자

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 응용 프로그램
    - 문서 편집기나 스프레드시트와 같은 응용 프로그램을 구축할 때는 내부 재사용(internal reuse), 유지보수성 및 확장성의 우선 순위가 높음
    - 내부 재사용이 가능해지면 불필요한 설계와 구현을 하지 않아도 되는데 종속성을 제거해 주는 디자인 패턴은 내부 재사용성을 증가시킴
    - 결합도가 더 낮으면 한 클래스가 다른 여러 클래스와 연동할 수 있는 가능성이 대폭 늘어나는데 예를 들어 각 연산을 캡슐화하여 종속성을 없애면 다른 상황에서 연산을 쉽게 재사용할 수 있으며 알고리즘과 표현의 종속성을 없애도 동일한 효과를 얻을 수 있음
    - 디자인 패턴은 플랫폼 종속성을 없앰으로써 시스템의 유지보수성을 좋게 하고 기존 클래스 계층의 확장과 객체 합성을 통해 시스템 확장성을 강화시킴
    - 결합도를 감소시키면 확장성이 증가하는데 그 이유는 다른 클래스 간의 종속성이 적으면 쉽게 한 개의 클래스만 따로 확장할 수 있기 때문

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ 툴킷
    - 응용 프로그램은 툴킷(toolkit)이라는 사전에 정의된 라이브러리나 이미 있는 클래스 들을 이용하기도 함
    - 툴킷이란 일반적인 목적의 유용한 기능을 제공하는 재사용 가능한 클래스들의 집합체
    - 툴킷의 예로 리스트나 테이블, 스택 등과 같은 컬렉션 클래스를 들 수 있음
    - C++ 입출력 스트림 라이브러리도 툴킷의 예가 될 수 있는데 툴킷은 응용 프로그램에 대해 특정한 설계를 강요하지는 않고 단지 응용 프로그램이 작업을 수행하는 데 도움이 되는 기능을 제공할 뿐
    - 툴킷 을 사용하면 공통으로 쓰는 기능을 재구현할 필요가 없음
    - 툴킷은 코드 재사용을 강조한 것으로 서브 루틴 라이브러리 와 동일한 객체지향 라이브러리
    - 툴킷을 여러 응용프로그램에서 유용하게 써야 하기 때문에 툴킷을 설계하는 일은 응용 프로그램 설계보다 어려울 수밖에 없고 더구나 툴킷 개발자가 툴킷을 개발하는 동안은 어떤 응용 프로그램이 어떤 목적으로 이 툴킷을 사용하게 될지 알 수 없으므로 어떤 가정을 두고 설계를 진행하면 툴킷의 유연성에 제한을 가하게 되므로 결과적으로 툴킷의 활용성과 효과가 줄게 됨

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ Framework
    - 프레임워크(framework)는 특정한 부류의 소프트웨어에 재사용성을 부여하여 개발할 수 있도록 만들어 주는 관련 클래스들의 집합
    - 그림 그리기, 음악 작곡, 기계 CAD 등은 서로 다른 분야에서 쓰지만 시각적인 효과를 넣는 편집기라는 부분에서는 똑같은 어떤 편집기 프레임워크를 구성할 수 있dma
    - 다른 프레임워크의 예로 다른 프로그래밍 언어를 해석하고 목적 기계어로 만들어 주는 컴파일러 클래스 생성을 지원하는 프레임워크도 있고 금융 모델링 응용 프로그램을 만들 수 있도록 지원하는 프레임워크도 있는데 이처럼 프레임워크는 특정 영역에서 재사용할 수 있도록 지원
    - 프레임워크의 재정의란 프레임워크에 정의한 클래스를 상속받아 특정 응용 프로그램을 지원하는 서브 클래스를 정의하는 것을 의미하는데 이러한 재정의 과정을 통해 해당 영역에서 새로운 응용 프로그램을 만들 수 있는데 이때 프레임워크가 정의한 추상 클래스를 상속하여 새로운 클래스를 정의하는 방법을 사용
    - 프레임워크는 응용프로그램에 대한 뼈대를 제공하는데 프레임워크는 클래스와 객체들의 분할, 전체 구조, 클래스와 객체들 간의 상호 작용, 객체와 클래스 조합 방법, 제어 흐름에 대해 미리 정의
    - 프레임워크는 설계의 가변성을 미리 정의 해 두고 만들었기 때문에 응용프로그램 설계자나 구현자는 응용프로그램에 종속된 부분에 대해서만 설계하면 됨
    - 프레임워크는 응용 프로그램 영역에 걸쳐 공통의 클래스들을 정의하여 일반적 설계 결정을 미리 내려두는데 이를 통해 프레임워크는 코드의 재사용보다는 설계의 재사용을 강조

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ Framework
    - 툴킷을 사용하여 응용프로그램을 작성할 때는 직접 응용프로그램의 주(main) 본문을 작성하고 재사용하고자 하는 코드를 호출하지만 프레임워크를 재사용할 때는 프레임워크가 제공하는 주 본문을 바로 재사용하고 이 부분에서 호출하는 코드를 작성하는 것
    - 특정 이름과 호출 방식이 결정된 연산을 작성해야 하지만 결정해야 하는 전체 설계 개념은 줄어들고 응용 프로그램 나름대로 구체적인 연산만 구현하면 남는 것이 없음
    - 프레임워크를 재사용하면 응용 프로그램을 신속하게 개발할 수 있으며 재정의에 의해 개발한 응용 프로그램들은 비슷한 구조를 갖게 되고 이렇게 되면 유지보수가 쉬워지고 사용자들에게는 좀더 일관된 모습을 보일 수 있는 반면 설계의 창의성은 줄어들게 됨
    - 응용 프로그램의 설계가 어렵다면 툴킷은 더 설계하기 어렵고 프레임워크는 모든 설계 중 가장 어려움
    - 프레임워크 설계자는 한 영역에서 모든 응용 프로그램을 지원할 뼈대, 즉 아키텍처를 하나로 만들어야 하는데 한 영역에서 새로운 응용 프로그램이 등장할 때마다 프레임워크에 상당한 변화가 가해진다면 프레임워크를 재사용함으로써 얻을 수 있는 장점이 줄어들게 되는데 왜냐하면 프레임워크가 정의한 아키텍처가 응용 프로그램에서 가장 중요한 장점인데 아키텍처가 변한다면 그것은 무의미해지기 때문이므로 프레임워크는 더 유연하고 확장 가능하도록 설계해야 함
    - 응용 프로그램의 설계는 프레임워크에 종속되어 있어서 프레임워크의 변경에 매우 민감할 수밖에 없는데 프레임워크가 진화하면 응용 프로그램도 따라서 진화해야 하기 대문에 응용프로그램과 프레임워크 사이의 결합도는 낮을수록 더 좋은데 그렇지 않으면 프레임워크에서 일어난 작은 변화가 응용프로그램을 크게 변화시키는 요인으로 작용

Design Pattern

- Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ Framework
    - 재사용 가능하고 확장 가능한 유연한 설계는 프레임워크 설계에서 매우 필수적
    - 디자인 패턴을 이용하는 프레임워크는 그렇지 않은 프레임워크보다 설계와 코드 재사용의 수준을 높일 수 있음
    - 완성도가 어느 정도 되는 프레임워크는 일반적으로 여러 개의 디자인 패턴을 사용하는데 이런 디자인 패턴은 프레임워크의 아키텍처를 재설계하지 않고도 다른 많은 응용 프로그램에 재사용할 수 있도록 요소 요소에서 활약하고 있음
    - 디자인 패턴은 프레임워크의 문서화에도 도움을 주는데 패턴을 아는 사람들은 프레임워크에 대한 직관을 더 빨리 가질 수 있고 패턴을 모르는 사람들이라도 프레임워크 문서화 구조를 통해 도움을 받을 수 있음
    - 문서화를 강화하는 것은 프레임워크 뿐만 아니라 모든 종류의 소프트웨어에서도 중요
    - 프레임워크를 배우는 데 는 많은 시간이 걸리는데 디자인 패턴이 프레임워크 습득을 혁신적으로 개선해 주지는 못하지만 설계 개념을 명확히 함으로써 학습 시간을 줄일 수는 있음

Design Pattern

- ❖ Design Pattern을 이용하여 문제를 푸는 방법
  - ✓ Framework
    - ● 패턴 과 프레임워크의 차이점
      - ◆ 디자인 패턴이 프레임워크보다는 더 추상적
        - ▪ 프레임워크는 구현을 정의하지만 패턴은 예만을 코드로 작성
        - ▪ 프레임워크의 강점은 프로그래밍 언어로 구현되어 있기 때문에 단순한 연구 대상만이 아니라 실행할 수 있고 바로 재사용할 수 있다는 점이지만 디자인 패턴은 패턴을 사용할 때마다 별도로 구현해야 하는데 디자인 패턴의 목적이 설계의 의도나 장단 점 등을 설명하려는 것이지 구현한 결과물을 제공하려는 것은 아니기 때문
      - ◆ 디자인 패턴은 프레임워크에 비해서 소규모의 아키텍처 요소: 일반적으로 프레임워크는 여러 디자인 패턴을 포함하지만 디자인 패턴이 프레임워크를 포함하는 일은 없음
      - ◆ 디자인 패턴은 프레임워크에 비해 덜 특수화되어 있음
        - ▪ 프레임워크는 어떤 특정 응용 프로그램 영역을 목표로 하는데 공장 시뮬레이션에 그래픽 편집기 프레임워크를 사용할 수도 있지만 시뮬레이션 프레임워크를 사용하는 것이 더 바람직하지만 디자인 패턴은 모든 종류의 응용 프로그램에도 다 적용할 수 있음

Design Pattern

- Design Pattern을 고르는 방법
  - ✓ 패턴이 어떻게 문제를 해결하는지 파악: 디자인 패턴이 어떻게 적합한 객체를 찾고 객체의 크기를 결정하며 어떻게 객체의 인터페이스를 명세화하는지에 대한 내용을 정리하면서 패턴이 설계 문제를 해결해 나가는 방법을 참조하면 적당한 패턴을 찾을 수 있을 것
  - ✓ 패턴의 의도 부분을 확인: 각 패턴의 의도를 잘 읽어보고 갖고 있는 문제와 비슷한 것을 찾음
  - ✓ 패턴들 간의 관련성을 파악: 관련성을 파악함으로써 올바른 패턴을 하나 또는 그룹으로 선택할 수 있음
  - ✓ 비슷한 목적의 패턴들을 모아서 학습
  - ✓ 재설계의 원인을 파악
  - ✓ 설계에서 가변성을 가져야 하는 부분이 무엇인지 파아