Ch. 6 재사용을 고려한 디자인

전문가를 위한 C++

오기환

재사용 철학

- 같은 기능 작성은 한 번, 사용은 여러 번
- 다른 프로그래머도 쉽게 재사용할 수 있게 디자인
- 용도나 분야가 약간 달라도 충분히 사용하도록 범용성을 갖춰야 함
- 사용하기 쉬워야 함

코드 재사용

- 소스 코드의 재사용
- 정적/동적 라이브러리의 재사용
- 인터페이스와 구현을 분리한 적절한 추상화

코드 추상화

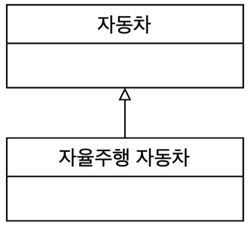
- 잘 정의된 인터페이스를 통해 내부 구현을 감출 수 있음
- 구현 코드를 변경하고자 할 때 코드 사용방식을 고칠 필요가 없음
- 재사용 코드의 내부에서 사용할 데이터는 핸들 (handle) 객체로 보관

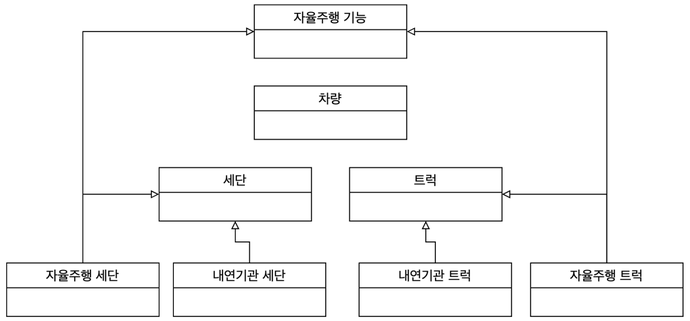
재사용에 최적화된 코드 구조화

- 응집도 높이기
  - Single Responsibility Principle (SRP)
- 결합도 낮추기
  - 추상화

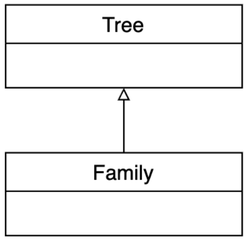
코드 레벨 재사용 성

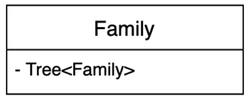
- 클래스 계층 구조 활용 (is-a)





- Has-a 관계 활용





다른 코드에 대한 의존성 줄이기

- 예) cout, cerr, cin, stdout 등과 같은 사용자 인터페이스에 대한 의존성 분리
- MVC 등과 같은 패턴을 활용

템플릿 사용하기

- C++ 에서는 템플릿, 다른 언어에서는 제네릭
- std::vector<type> 이 가장 이해하기 쉬운 예제
- 구현하고자 하는 데이터와 거기에 필요한 알고리즘의 정확한 구분이 필요
- void* 형태의 제네릭 구현은 캐스팅 오류 가능성이 매우 높음

템플릿과 상속

- 템플릿은 구현하기 불편함 - 문법이 복잡
- 동일한 기능을 다양한 타입에 대해 지원 -> 템플릿 (정렬, 트리 등)
- 타입이 다른 경우 동작이 달라짐 -> 상속 (큐 > 우선순위 큐)
- 둘 모두 당시에 사용도 가능

계약 (contract) 정의 후 구현

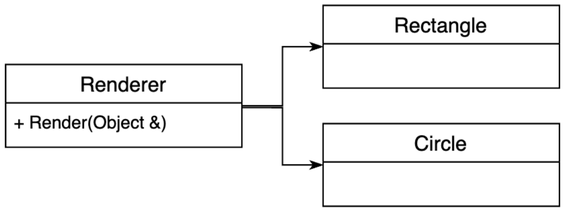
- 사전(선행) 조건 - 코드 사용 전 항상 해야 할 일
- 불변 조건 - 코드 실행 시 항상 해야 할 일
- 사후(후행) 조건 - 코드 실행 후 항상 해야 할 일
- std::vector 예시
  - [ ] 접근시경계값 (boundary) 검사는 사용자가 직접 해야함
    - at( ) 메소드는 경계값 검사를 해줌

최대한 안전하게 디자인하기

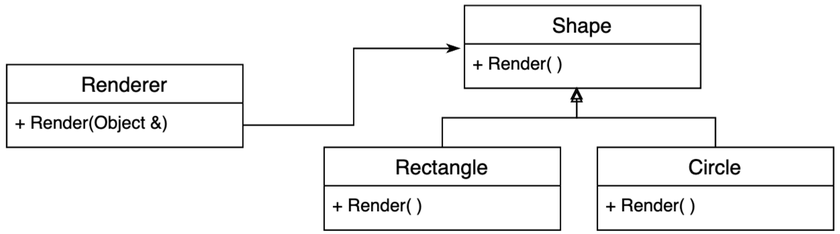
- 제공하는 코드 내에서 사용자가 입력한 값을 항상 확인하고 잘못된 입력은 에러처리
- 보통 false나 nullptr 리턴을 통해 에러를 표기할 수 있음
- Exception 이용 가능
- 메모리 동적할당과 같은 리소스 관련 코드는 스마트 포인터를 활용

확장성을 고려한 디자인

- Open/Closed Principle (개발 폐쇄 원칙)
  - 확장하는데는 open
  - 구현을 수정하는데는 closed



BAD



GOOD

사용성 높은 인터페이스 - 1/2

- 사용하기 쉬운 인터페이스
  - 간결하고 직관적으로 설계하기 - 기존의 익숙한 방식을 이용
    - init, cleanup 대신 생성자 소멸자 사용
  - 연산자 오버로딩을 적절하게 활용하기
- 필요한 기능 빼먹지 않기
  - 작성한 코드를 사용하는 가능한 한 모든 경우의 수를 따져보기
  - 최대한 많은 기능을 지원하기

사용성 높은 인터페이스 - 2/2

- 군더더기 없는 인터페이스
  - 쓸데없는 기능 제거
- 문서와 주석 제공하기
  - 코드 안에 주석으로 문서 제공하기, 추가적인 별도 문서 제공하기
  - 구현이 아닌 동작에 초점 맞추기
    - 퀵소트를 사용한다 x, 오름차순으로 정렬한다 o
  - 추가 문서로 내부 구현방식을 기술하기

범용 인터페이스 디자인

- 하나의 기능을 다양한 방식으로 실행
  - 자동차 문 열때 스마트 키, 지문인식, 수동 열쇠 등
  - std::vector의 at( ) 은 자동 경계값 검사, [ ] 는 사용자가 직접 검사
- 커스텀 지원
  - 에러 로그 켜기/끄기
  - 정렬 순서 지원하기

범용성과 사용성

- 범용성 ⬆ -> 사용성 ⬇ 경우가 많음
- 인터페이스를 다양하게 지원
  - Interface Segregation Principle (ISP) - 하나의 범용성 인터페이스보다 다수의 인터페이스가 낫다
- 자주 사용하는 기능 쉽게 만들기
  - 텍스트 출력 시 영어를 기본값으로 지원 - 옵션으로 다른 언어 지원

SOLID 원칙

- 위키 피디아 SOLID (객체 지향 설계) 참조

두문자	약어	개념
S	SRP	**단일 책임 원칙 (Single responsibility principle)** 한 클래스는 하나의 책임만 가져야 한다.
O	OCP	**개방-폐쇄 원칙 (Open/closed principle)** "소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다."
L	LSP	**리스코프 치환 원칙 (Liskov substitution principle)** "프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다." 계약에 의한 설계를 참고하라.
I	ISP	**인터페이스 분리 원칙 (Interface segregation principle)** "특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다."[4]
D	DIP	**의존관계 역전 원칙 (Dependency inversion principle)** 프로그래머는 "추상화에 의존해야지, 구체화에 의존하면 안된다."[4] 의존성 주입은 이 원칙을 따르는 방법 중 하나다.