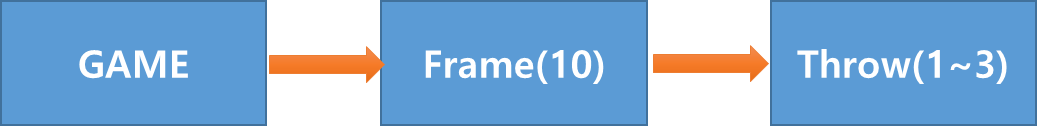
**볼링게임**

[**https://github.com/simjunbo/CleanCode**](https://github.com/simjunbo/CleanCode) **(커밋로그 참조)**

****

짝 프로그램으로 볼링게임을 제작

초기에 Game -> Frame(10번) -> Throw(Max3) 방식의 UML 다이어 그램을 생각했음.

1) Frame 추가

-throw, frame 테스트 클래스 추가

2) Frame add에 test에 대한 구현

-Frame add test에 대한 메소드 구현. (취약성 11로 호출하면 값이 넘어감)

3) Game 추가

-Frame은 해당 스코어를 계산하기 위해 다음 Frame 인스턴스 확인 필요. 그래서 Game이란 객체를 만들어서 Frame을 관리 (연결 리스트 형태로 구상)

4) Game에 frame별 점수 계산 테스트 추가

5) Game에 Frame별 점수 획득 세부 구현

잠재적 단일 책임원칙(SRP) 위반, 하지만 여기서 무시

itsThrows[21]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 4 | 7 | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

↑ itsCurrentThrow

itsCcore = 5 + 4 + 7 + 2 = 18 (스페어 시 문제 생김)

6) 스페어 처리 추가

1Frame 2Frame

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 7 | 3 | 5 |

계산 : 3 + 7 + 3 = 13

에 대한 처리 추가

7) 현재 프레임 계산

혀재 프레임을 모르기 때문에 getCurrentFrame() 추가해 주고 각 frame의 2번째 투구가 끝날 때 마다 증가 시켜 준다. (하지만 아직 스트라이크에 대한 미대응)

8) 현재 프레임의 총 Score 계산

현재 itsScore는 넘어뜨린 핀 수의 합으로 계산되고 있기 때문에 각각 frame별 합으로 변경해 준다. 그리고 테스트케이스는 목적에 부합하지 않으면 삭제 가능

Frame별 첫투구만 한경우에는 score관련 테스트 케이스를 하지 않는다.

score에서 getCurrentFrame() -1 을 해주는 이유는 완료 된, Frame까지 계산하기 위해

9) 스트라이크 처리

itsThrows[21]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 3 | 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

↑ itsCurrentThrow

계산 : 10 + 3 + 6 = 19

스트라이크일 경우 한번만 투구하고 종료되기 때문에 현재 itsCurrentFrame을 증가 시켜준다. 그리고 상태가 true 이기 때문에 다음 Frame 바로 시작

점수 계산 할때는 스페어인 경우 다음 첫번째 투구를 합해 준다

frameScore(firstThrow + secondThrow) + itsThrows[ball];

스트라이크인 경우는 다음 두번째 투구까지 합해 준다.

10 + itsThrows[ball] + tisThrows[ball+1];

10) 퍼펙트 게임 처리

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

↑ itsCurrentThrow

|  |
| --- |
| assertEquals(300, g.score());  public int score() {  return scoreForFrame(getCurrentFrame() - 1);  }  for(int c = 0; c < 10; c ++ ) {  …  } |

11)배열 마지막에서의 스트라이크

배열 마지막에서 스트라이크를 하면 배열 범위를 넘어가지 않을까 생각했지만

itsCurrentFrame 범위를 MAX11로 해놓았기 때문에 넘어가지 않는다.

**MAX11의 의미 score range (0 < 10(current() - 1)**

|  |
| --- |
| for(int c = 0; c < 10; c ++ ) {  …  } |

12) 기타 테스트 케이스 추가

**볼링게임 (리팩토링)**

13) scoreForFrame 리팩토링

scoreForFrame의 else 부분을 handleSecondThrow() 메서드로 빼기 위한 작업

(개인적으로 전역 변수들의 참고가 많아져서 별로인 것 같다.)

🡺사용하는 곳이 많이 질수록 이해하기 어려워짐

🡺여러 스레드에서 접근 시, 데이터 안정성이 떨어짐 (멀티스레드 환경)

1> 지역 변수 🡺 멤버 변수 변경

2> Boolean firstThrow 🡺 Boolean firstThrowFrame 변경해서 충돌피하고 의미 명확하게

3> else 부분을 scoreForFrame으로 변경

최종 목표 (의사코드)

|  |
| --- |
| If strike  score += 10 + nextTwoBalls();  else if spare  score += 10 + nextBall();  else  score += twoBallsInFrame() |

14) Plus Ball에 대한 리팩토링

strike = 10 + nextTwoBalls

spare = 10 + nextBall

twoBallsInFrame(first + second)

등을 이용하여 함수형태로 변경

🡺firstThrow, secondThrow, frameScore등을 삭제 (함수 내부의 itsThrows[ball] 방식으로 다 변경)

🡺ball을 전역으로 뺐기 때문에 가능

Ball 기존에 ++ 했던 부분을 첫번째는 +1, 두번째는 +2 식으로 명확하게 변경

Throw -> Game (위에서 아래가 아닌 ) 다음에는 위에서 아래로 내려오는 형태(하향식).

테스트 우선 설계

15) 의사코드 형태로 변경

|  |
| --- |
| If strike  score += 10 + nextTwoBalls();  else if spare  score += 10 + nextBall();  else  score += twoBallsInFrame() |

형태로 변경하면서 함수 이름은 좀더 명확하게

16) Scorer 추가 (단일책임의 원칙)

스코어 관련 기능은 Scorer 클래스 쪽에 기능 이관 시키고,

Game에서는 Scorer를 호출하는 구조로 변경한다.

17) adjustCurrentFrame 리팩토링

itsCurrentFrame++ 부분을 advanceFrame으로 변경해서 증가와 min 체크를 동시에 수행

18) itsCurrentFrame 기준 변경

미래 시점이 아닌, 현재 시점으로

19) 테스트코드 정리

**결론**

결국 Frame과 Throw는 사용 안 함.

Game => Frame => Scorer 버전으로 만든 사람도 있었는데 더 복잡해 짐

🡺이 프로그램에서는 UML 자체가 도움이 안됨

🡺UML을 만들었다고 가장 최적의 설계라고 가정해서는 안 된다.

**애자일 설계**

**서론**

지금 당장을 위해서 설계와 구현을 동시에 진행한다.

**잘못된 설계의 증상**

* 경직성(Rigidity) : 설계를 변경하기 어려움
* 취약성(Fragility) : 설계가 망가지기 쉬움
* 부동성(Immobility) : 설계를 재사용하기 어려움
* 점착성(Viscosity) : 제대로 동작하기 어려움
* 불필요한 복잡성(Needless Complexity) : 과도한 설계
* 불필요한 반복(Needless Repetition) : 마우스 남용
* 불투명성(Opacity) : 혼란스러운 표현

**원칙**

[SOLID](https://ko.wikipedia.org/wiki/SOLID)

* SRP : 단일책임원칙 : 클래스는 하나의 책임(기능)만 가지고 있는다.
* OCP : 개방 폐쇄 원칙 : 확장에는 열려 있고 변화에는 닫혀 있다.

🡺둘다 낮은 결합도

* LSP : 리스코프 치환 원칙 : 부모클래스는 자식클래스로 변경 가능해야 한다.

🡺인터페이스만 알면 구현체를 몰라도 사용 가능

* DIP : 의존 관계 역전 원칙 : 구체화 클래스가 아닌 추상이나 인터페이스에 의존
* ISP : 인터페이스 분리 원칙 : 하나의 인터페이스보다는 구체적인 여러 개의 인터페이스가 낫다.

**악취와 원칙**

대개 이런 악취는 하나 이상의 원칙을 위반했을 때 발생한다.

Ex) 경직성 🡺 OCP를 지키지 않아서…

**Chapter 7. 애자일 설계란 무엇인가?**

*설계는 우선적으로 소스 코드에 의해 문서화되며, 소스 코드를 표현하는 다이어그램은 설계에서 부수적인 것일 뿐, 설계 그 자체는 아니다*

* UML = 설계의 일부일뿐 설계 자체는 아니다.
* 설계 = 소스코드

**소프트웨어에서 어떤 것이 잘못되는가?**

소프트웨어는 상한고기 처럼 부패하기 시작해서 시간이 지남에 따라 부패 범위가 넓어지고 심해진다.

🡺 재설계(renewal)보다는 지속적으로 발전시켜야 한다.

**설계의 악취: 부패하고 있는 소프트웨어의 냄새**

* 경직성(Rigidity)

🡺한 군데의 변경이 의존적인 모듈에서 단계적으로 계속 변경을 불러일으킴

🡺높은 결합도로 인해 변경 어려움 (오래 걸림)

* 취약성(Fragility)

🡺한군데 변경했을 때 프로그램의 많은 부분이 잘못되는 경향

🡺설계가 망가지기 쉬움 (고치기 힘듬)

🡺취약성이 심해질수록 재설계의 필요성 인식, 버그 목록에 내려가지 않음. 고칠수록 나빠짐

* 부동성(Immobility)

🡺설계를 재사용하기 어려움

🡺원래 시스템에서 분리하는 수고와 위험성이 클 때 설계는 움직이게 할 수 없다.

* 점착성(Viscosity)

🡺소프트웨어의 점착성, 환경의 점착성

🡺프로그래머는 설계를 유지하기 쉬운 시스템과 프로젝트 환경을 만들고 싶어 한다.

* 불필요한 복잡성(Needless Complexity)

🡺요구 사항에 변경을 미리 예상하고 기능을 자주 집어 넣어서, 현재 필요 없는 요소까지 들어 있음.

* 불필요한 반복(Needless Repetition)

🡺Ctrl + C,V의 남용. 기능 개발하기 전 비슷한 코드가 있나 살펴 본다.

🡺같은 코드 반복 시, 추상화 개념을 잃어 버린다.

🡺반복된 부분의 버그는 모든곳을 수정해야 한다.

* 불투명성(Opacity)

🡺모듈을 이해하기 어려운 경향.

🡺명료하고 표현적으로 유지하려는 지속적 노력 필요 (지속적인 리팩토링)

**애자일 팀은 소프트웨어가 부패하도록 내버려두지 않는다**

시스템의 설계를 가능한 한 명료하고 단순하게 유지

🡺많은 단위 테스트와 인수 테스트로 뒷받침 (우리는 테스트 커버리지)

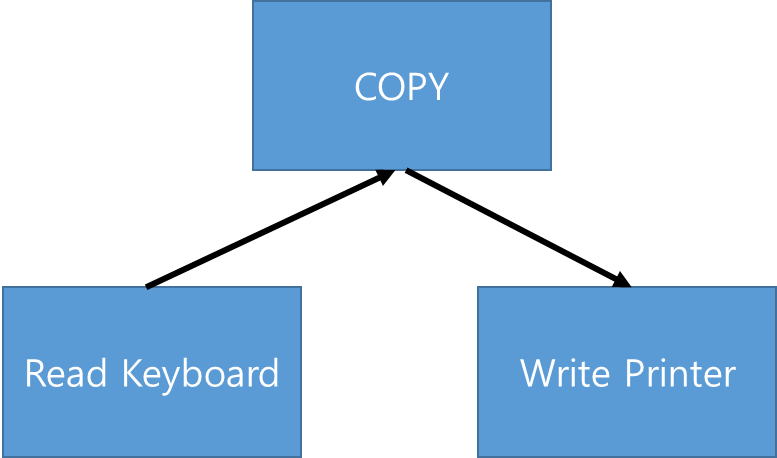
**애자일 개발자는 해야 할 일을 어떻게 알았는가?**

1. 그들은 다음과 같은 애자일 실천방법으로 문제를 찾아냈다. - 유연하지 않음
2. 그들은 설계 원칙을 적용해 문제를 진단했다. - OCP, DIP
3. 그리고 적절한 디자인 패턴을 적용해 문제를 해결했다. - 전략패턴

**COPY 프로그램**

초기 요구사항 : 키보드에서 프린터로 문자를 복사하는 프로그램

초기 설계 :



Copy 모듈은 Read Keyboard 모듈에서 문자를 가지고 와서 Write Printer 모듈에 보낸다.

|  |
| --- |
| **void copy () {**  **int c;**  **while((c = Rdkdb()) != EOF)**  **WrtPrt(c);**  **}** |

요구사항 변경 : Copy 프로그램이 종이테이프 판독기에서도 문자를 읽을 수 있으면 좋겠다.

|  |
| --- |
| **bool ptFlag = false // true 이면 종이테이프 판독기, false 이면 키보드**  **void copy () {**  **int c;**  **while((c = (ptflag ? RdPt() : RdKbd())) != EOF)**  **WrtPrt(c);**  **}** |

🡺 플래그를 재설정해야 함을 상기시킴

추가요구 사항 : Copy 프로그램으로 종이테이프 천공기에 출력

|  |
| --- |
| **bool ptFlag = false // true 이면 종이테이프 판독기, false 이면 키보드**  **bool punchFlg = false // true 이면 천공기, false이면 프린트**  **void copy () {**  **int c;**  **while((c = (ptflag ? RdPt() : RdKbd())) != EOF)**  **puchFlag ? WrtPuch(c) : WrtPrt(c);**  **}** |

🡺결국 주석을 통해 프로그램을 이해하게 된다.

단 두번의 변경 후에 경직성, 취약성, 부동성, 복잡성, 반복성, 불투명성의 신호를 보였다.

요구사항은 언제나 바뀐다를 명심!!

🡺우리는 변화 하는 요구사항의 세계에서 살고 있고 우리가 만든 소프트웨어가 이런 변화 속에서 살아남을 수 있게 만드는 것이 우리 할일

**애자일 개발자는 해야 할 일을 어떻게 알았는가?**

Copy 모듈이 KeyboardReader와 PrintWriter에 직접 의존하기 때문에 하위 수준의 세부 사항이 바뀔 때 마다, 정책이 영향을 받게 된다. (유연하지 못함)

애자일 개발자는 다음과 같은 이유 때문에 해야 할 일을 알고 있다.

1. 그들은 다음과 같은 애자일 실천 방법으로 문제를 찾아냈다.
2. 그들은 설계 원칙을 적용해 문제를 진단했다.
3. 그리고 적절한 디자인 패턴을 적용해 문제를 해결했다.

**가능한 좋은 상태로 설계 유지하기**

* 매일, 매 시간, 분마다 소프트웨어를 가능한 명료하고 간단하고 표현적인 상태로 유지
* 가능 한 좋은 상태로 유지

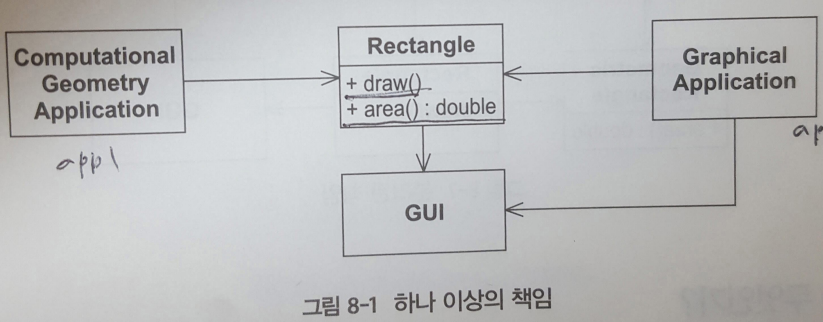
**결론**

* 애자일 설계는 과정이지, 결과가 아니다.
* 시스템을 좋은 상태로 유지하려는 노력이다.

**8. 단일책임 원칙 (Single-Responsibility Principle)**

**한 클래스는 단 한가지의 변경 이유만을 가져야 한다.**

한가지 객체에서 두 가지 일을 하면 귀찮은 문제들이 유발된다.

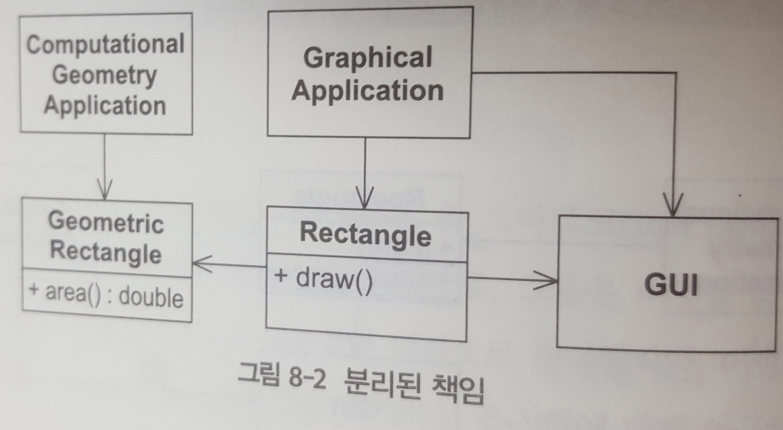


1) 필요없는 기능 부분도 포함되어 컴파일 시간, 메모리 영역 소비등이 나타난다.

ex) 계산 기하학 애플리케이션에 GUI 포함

2) 다른 한쪽 때문에 변경이 이뤄지면 나머지 한쪽은 잘못 동작할 수 있다.

ex) 그래픽 어플리케이션관련 기능 변경했는데 계산 기하학 애플리케이션이 잘못 동작할 수 있다.



완전히 다른 클래스로 분리하면 영향을 주지 않는다.

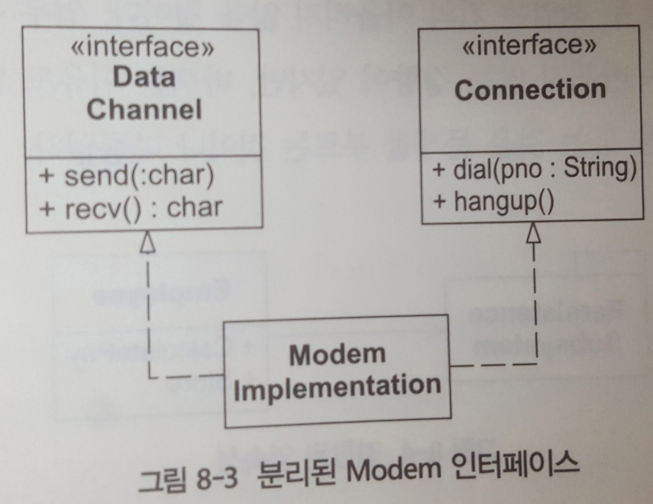
**책임이란 무엇인가?**

책임을 변경을 위한 이유로 정의한다.

|  |
| --- |
| interface Modem {  public void dial(String pno);  public void hangup();  public void send(char c);  public char recv();  } |

dial과 hangup 함수는 모뎀을 관리하는 반면, send와 recv 함수는 데이터를 주고받으며 통신

책임의 분리는 애플리케이션이 어떻게 바뀌느냐에 달려 있다.



1) send, recv를 자주 호출 시, 위와 같이 기능을 분리 (경직성)

2) 잘 변경이 유발되지 않는경우 분리하면 오히려 불필요한 복잡성이 생김

🡺변경의 축은 변경이 실제로 일어날 때만 변경의 축이다. (증상도 없는데 SRP 적용시키면 현명하지 못함)

**결론**

SRP는 가장 간단한 원칙 중 하나임과 동시에 제대로 적용하기 가장 어려운 원칙

**9. 개방 폐쇄 원칙 (Open-Closed Principle)**

**소프트웨어 개체(클래스, 모듈, 함수 등)는 확장에 대해 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다.**

**1) 확장에 대해 열려 있다.**

애플리케이션은 요구사항이 변경될 때, 이 변경에 맞게 새로운 행위를 추가해 모듈을 확장할 수 있다.

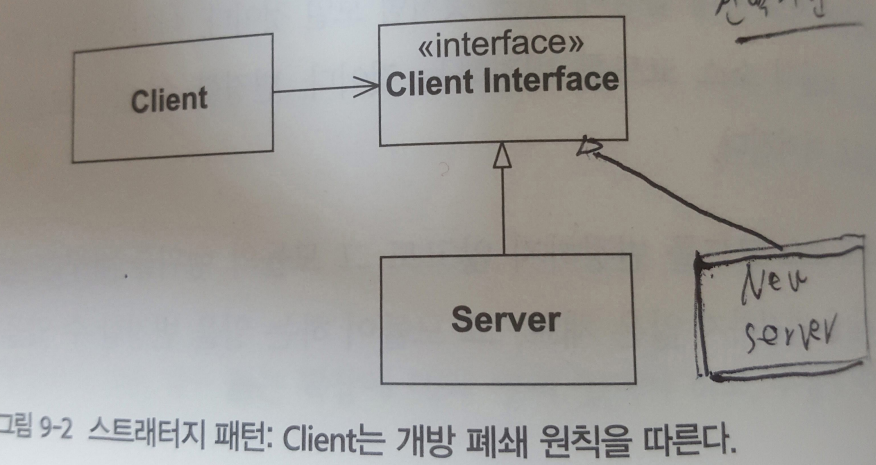
**2) 수정에 대해 닫혀 있다.**

어떤 모듈의 행위를 확장하는 것이 그 모듈의 소스 코드나 바이너리 코드의 변경을 초래하지 않는다.

ex) java 하위 버전 호환, FunctionalInterface

**해결책은 추상화다**

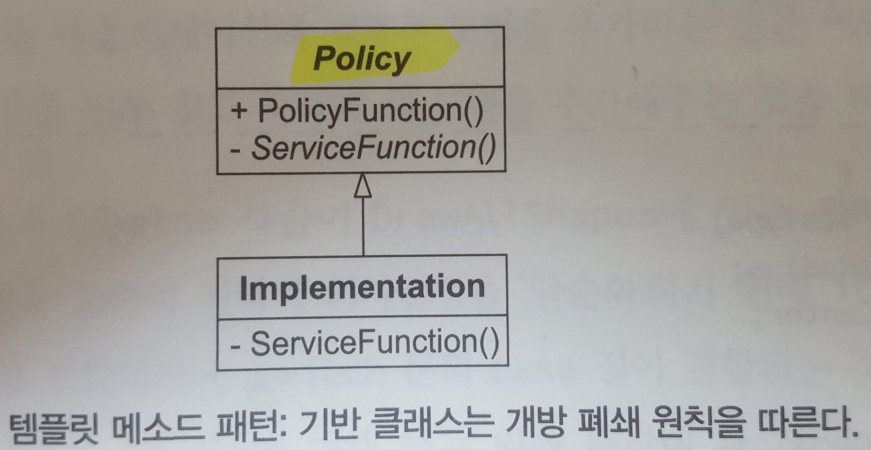
고정된 추상화에 의존하기 때문에 수정에 대해 닫혀있다. 하지만 새 파생 클래스들을 만들어 확장 가능하다.



전략 패턴은 OCP를 따른다.

여기서 AbstractServer가 아닌 Client Interface로 설계된 이유는 추상 클래스는 자신을 구현하는 클래스보다도 클라이언트에 더 밀접하게 관련되어 있기 때문이다.

정책 함수들은 어떤 추상 인터페이스의 형식을 통해 작업하려는 일을 설명한다. 그러나 이번에는 추상 인터페이스가 Policy 클래스 자체의 한 부분이다.



템플릿 메소드 패턴은 OCP를 따른다.

**이 2개의 패턴 (전략, 템플릿 메소드)은 OCP를 따르는 가장 흔한 수단이다.**

**기능의 구체적인 구현으로부터 일반적인 기능을 깔끔하게 분리해낸다.**

탬플릿 메소드 패턴(추상클래스)에서는 부모의 권한(책임)을 하위클래스로 위임시켜

하위클래스가 사용의 주체가 된다.

전략패턴에서는 권한(책임)을 객체로 위임시켜, 부모클래스 안에서, 전략객체가 작동하도록 하는 것.

즉, 권한(책임)의 위임(여기서는 알고리즘)을 하위클래스에서 담당하는 것이

탬플릿 메소드 패턴(추상클래스)이며, 권한의 위임(알고리즘)을 외부 객체에게

하는 것이 전략패턴.

**전략, 템플릿 메소드, 데코레이터 패턴은 OCP랑 연관이 있다**.

**OCP 위반**

public class DrawAllShapes {  
  
 public static void main(String[] args) {  
 List<Shape> list = new ArrayList<Shape>();  
 list.add(new Shape(ShapeType.*CIRCLE*));  
 list.add(new Shape(ShapeType.*SQUARE*));  
  
 *getShape*(list, 2);  
 }  
  
 public static void getShape(List<Shape> list, int n) {  
 for (int i = 0; i < n; i++) {  
 Shape shape = list.get(i);  
 switch (shape.itsType) {  
 case *CIRCLE*:  
 Circle circle = new Circle();  
 circle.drawCircle();  
 break;  
 case *SQUARE*:  
 Square square = new Square();  
 square.drawSquare();  
 break;  
 default:  
 System.*out*.print("No Matched");  
 }  
 }  
 }  
}  
  
enum ShapeType {  
 *CIRCLE*, *SQUARE*}  
  
class Shape {  
 ShapeType itsType;  
  
 Shape(ShapeType type) {  
 this.itsType = type;  
 }  
}  
  
class Square {  
 public void drawSquare() {  
 System.*out*.println("Square");  
 }  
}  
  
class Circle {  
 public void drawCircle() {  
 System.*out*.println("Circle");  
 }  
}

getShape 함수는 새로운 도형 종류에 대해 닫혀 있을 수 없기 때문에 OCP를 따르지 않는다. 도형이 추가되면 이 함수도 또한 수정해야 된다.

또한 ShapeType enum에 의존적이기 때문에, 이것들을 모두 다시 컴파일해야 한다.

**나쁜설계**

Triangel 추가 시, ShapeType, Square, Circle, DrawAllShapes의 재컴파일과 재배포를 불러일으킨다. 또한 switch/case 문이나 if/else 문이 많기 때문에 취약하다.

**OCP 따르기**

public class DrawAllShapes {  
 public static void main(String[] args) {  
 List<Shape> list = new ArrayList<Shape>();  
 list.add(new Circle());  
 list.add(new Square());  
  
 *getShape*(list);  
 }  
  
 public static void getShape(List<Shape> list) {  
 for (Shape shape : list) {  
 shape.draw();  
 }  
 }  
  
}  
  
abstract class Shape {  
 public abstract void draw();  
}  
  
class Square extends Shape {  
 public void draw() {  
 System.*out*.println("Square");  
 }  
}  
  
class Circle extends Shape {  
 public void draw() {  
 System.*out*.println("Circle");  
 }  
}

만약 도형이 추가된다면 Shape를 상속받는 파생클래스만 추가하면 된다.

Enum도 사용하지 않기 때문에 별도 추가 필요 없다.

즉 도형이 추가된다고 해도 어느 것에도 아무런 영향을 주지 않는다.

Shape에 더 많은 메소드를 가지고 있다고 해도 애플리케이션 전체를 뒤지면서 변경이 필요한 부분을 찾을 필요는 없다. (취약하지 않다.)

수정되어야 하는 소스 모듈도 없을뿐더러, 하나만 제외하면 재빌드되어야 하는 바이너리 모듈도 없다. (경직성도 없다.)

단 호출부는 수정되어야 한다, (main)

DrawAllShapes는 Square와 Circle의 편승 없이도 다른 애플리케이션에 재사용가능하다.

(부동성이 없다.)

**예상과 자연스러운 구조**

순서가 도형 종류보다 더 중요한 시스템에서는 이 모델이 자연스럽지 않다.

일반적으로 모듈이 얼마나 ‘닫혀’ 있든지 간에, 닫혀 있지 않은 것에 대한 변경은 항상 존재한다.

🡺모든 상황에서 자연스러운 모델은 없다.

OCP를 따르자면 비용이 많이 든다. 적절한 추상화를 만들기 위해서는 개발 시간과 노력이 들 뿐만 아니라, 이런 추상화는 소프트웨어 설계의 복잡성을 높이기도 한다.

모든 것을 한 후에, **변경이 일어날 때까지 기다린다.**

**올가미 놓기**

우리가 놓은 올가미는 종종 틀렸고 그것은 사용되지 않음에도 불구하고 유지 보수되어야 하는 **불필요한 복잡성의** 악취를 풍겼다.

**나를 한번 놀리면…** 소프트웨어를 불필요한 복잡성의 부하에서 구하려면, 우리 자신이 한번은 놀림당할 각오를 해야할지도 모른다.

즉 첫번째 총알은 그냥 맞고, 그 총에서 쏘는 다른 총알에 대해서는 확실히 보호한다는 것이다.

**변경 시뮬레이션하기** 첫 번째 총알을 맞기로 결정했다면, 총알이 빨리 그리고 자주 날아올수록 유리하다. 어떤 종류의 변경이 일어날 것이지 알기 위해 기다리는 시간이 길어질수록, 적절한 추상화를 만드는 일은 더 어려워진다.

-테스트를 먼저 작성한다.

-아주 짧은 주기로 개발한다.

-기능 요소를 먼저 개발하고, 이해당사자에게 보여준다(피드백)

-가장 중요한 기능 요소를 먼저 개발한다.

-빨리, 자주 릴리즈 한다. 가능한 자주 고객과 사용자 앞에서 그것을 시연한다.

**명시적 폐쇄를 위해 추상화 사용하기**

DrawAllShapes를 순서에 대해 닫으려면, **순서 추상화** 가 필요하다. 이 추상화는 정책을 통해 추상 인터페이스를 제공할 것이다.

**결 론**

OCP는 객체 지향 설계의 심장이라 할 수 있다. 이 원칙을 따르면 객체 지향 기술에서 당여하게 요구되는 최상의 효용을 낳는다. (유연성, 재사용성, 유지보수성)

하지만 객체지향 프로그래밍 언어를 사용하는 것만으로 이 원칙을 따른다고 할 수 없다. 또한 애플리케이션의 모든 부분에 마구 추상화를 적용하는 것도 좋은 생각이 아니다.

**그보다는 프로그램에서 자주 변경되는 부분에만 추상화를 적용하기 위한 개발자의 헌신이 필요하다.**

**🡺어설픈 추상화를 피하는 일은 추상화 자체만큼이나 중요하다.**

**10. 리스코프 치환 원칙 (Liskov Substitution Principle)**

서브타입은 그것의 기반 타입으로 치환 가능해야 한다.

|  |
| --- |
| Class B {}  Class D extends B {}  void F(B b) {}; |

형태의 클래스가 있을 경우

F(B); 는 정상이지만

F(D); 는 동작이 잘못된다면 LSP를 위반한다.

**LSP 위반의 간단한 예**

public class LSP {  
 public void drawShape(Shape s) {  
 if (s.itsType == Shape.ShapeType.*SQUARE*) {  
 Square square = (Square) s;  
 square.draw();  
 } else if (s.itsType == Shape.ShapeType.*CIRCLE*) {  
 Circle circle = (Circle) s;  
 circle.draw();  
 }  
 }  
}  
  
class Point {  
 double x, y;  
}  
  
abstract class Shape {  
 enum ShapeType {  
 *SQUARE*, *CIRCLE* }  
  
 ShapeType itsType;  
  
 Shape(ShapeType type) {  
 this.itsType = type;  
 }  
}  
  
class Circle extends Shape {  
 Point itsCenter;  
 double itsRadius;  
  
 Circle(ShapeType type) {  
 super(type);  
 }  
  
 public void draw() {  
 System.*out*.println("Circle");  
 }  
}  
  
class Square extends Shape {  
 Point itsTopLeft;  
 double itsSide;  
  
 Square(ShapeType type) {  
 super(type);  
 }  
  
 public void draw() {  
 System.*out*.println("Square");  
 }  
}

drawShape 함수는 OCP를 위반한다. 모든 파생 클래스를 알아야 하고, 새로운 파생 클래스가 생길 때마다 변경되야 한다.

**프로그래머는 왜 이와 같은 함수를 작성 하는 것일까?**

조는 객체 지향 기술을 공부하고 다형성의 부하가 지나치게 크다는 결론에 도달했다.

그래서 추상메서드를 포함하지 않은 Shape 클래스를 구현했다.

Circle과 Square는 Shape 클래스를 상속 받았지만 함수를 오버라이드 하지 않는다.

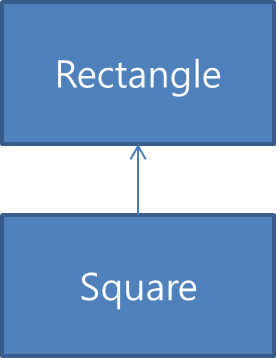
🡺Circle과 Square는 Shape를 대체할 수 없다는 것에 LSP 위반이다. 이 위반은 drawShape를 통해 OCP 위반을 유발했다.

즉 LSP 위반은 잠재적인 OCP 위반이다.

**정사각형과 직사각형, 좀 더 미묘한 위반**

public class Rectangle {  
 private Point itsTopLeft;  
 private double itsWidth;  
 private double itsHeight;  
  
 public void setWidth(double w) {  
 this.itsWidth = w;  
 }  
  
 public void setHeight(double h) {  
 this.itsHeight = h;  
 }  
  
 double getWidth() {  
 return itsWidth;  
 }  
  
 double getHeight() {  
 return itsHeight;  
 }  
}

잘동작하고 있는 애플리케이션이지만 추가 요구 사항이 들어 왔다. 정사각형도 조작할 수 있게 해달라고 한다.



모든 정사각형은 직사각형이기 때문에 Square 클래스는 Rectangle에서 파생된다고 보는 것이 합리적이다. 이런 식의 생각은 미묘하지만 심각한 문제를 낳을 수 있다.

첫번째 증거는 Square가 itsHeight와 itsWidth 멤버 변수를 필요로 하지 않는다.

(정사각형은 가로와 세로가 같기 때문이다.)

🡺그럼에도 불고하고 Square는 Rectangle에서 이 멤버 변수들을 상속받는데 소모적이다.

|  |
| --- |
| void setWidth(double w){  Rectangle r = new Rectangle();  r.setWidth(w);  r.setHeight(w);  }  void setHeight(double w){  Rectangle r = new Rectangle();  r.setHeight (w);  r.setWidth (w);  } |

요런식으로 설정하면 된다. 하지만 다음 함수에 대해 생각해보자

|  |
| --- |
| void f(Rectangle r) {  r.setWidth(32);  } |

Square 객체에 대한 참조값을 이 함수에 넘겨준다면 가로값에 맞춰 세로가 바뀌지 않을 것이다. (이부분 좀 이해 안감. Square 참조값 넘겨도 본인거 호출됨)

LSP위반이다. 왜냐면 추상 메서드로 선언되지 않아서, 다형적이지 않다.

하지만 이 문제는 간단하게 추상 메서드 선언해서 변경할 수 있다. 하지만 파생 클래스를 만드는 것이 기반 클래스의 변경으로 이어질 때, 대개는 설계에 결점이 있다.

public class Square extends Rectangle {

Rectangle r = new Rectangle();

@Override  
 public void setWidth(double w) {  
 r.setWidth(w);  
 r.setHeight(w);  
 }  
  
 @Override  
 public void setHeight(double h) {  
 r.setWidth(h);  
 r.setHeight(h);  
 }  
}

🡺Java에서는 virtual 키워드 사용안해도 오버라이딩 가능

**본질적인 문제**

public static void g(Rectangle r) {  
 r.setWidth(5);  
 r.setHeight(4);  
 assert (r.Area() == 20);  
}

Square의 객체를 넘겨 받을 때 다음과 같은 조건을 만족하지 못한다. (LSP 위반)

함수 g는 Square/Rectangle 계층 구조에 대해 취약하다.

**유효성은 본래 갖추어진 것이 아니다**

LSP는 ‘모델만 별개로 보고, 그 모델의 유효성을 충분히 검증할 수 없다’

특정 설계가 적절한지 아닌지 판단할 때는 단순히 별개로 봐선 해답을 찾을 수 없다. 그 설계를 사용자가 택한 합리적인 가정의 관점에서 봐야 한다.

🡺사실 이런 것을 모두 예상하려고 시도한다면, 시스템을 **불필요한 복잡성의** 악취로 찌들게 하는 결과를 낳을 것이다.

🡺관련된 취약성의 악취를 맡을 때까지 가장 명백한 LSP위반을 제외하고 연기한다.

**IS-A는 행위에 대한 것이다.**

행위야 말로 소프트웨어의 모든 것이다.

행위 측면에서 볼 때 Square는 Rectangle이 아니다

**계약에 의한 설계**

합리적인 추정을 명시적으로 만들어 LSP를 강제하는 테크닉이 있는데 이를

계약에 의한 설계(DBC : design by contract) 라고 한다.

마이어가 설명한 파생 클래스의 사전조건과 사후조건에 대한 규칙

**루틴 재선언(파생 클래스에서)은 오직 원래 사전조건과 같거나 더 약한 수준에서 그것을 대체할 수 있고, 원래 사후조건과 같거나 더 강한 수준에서 그것을 대체할 수 있다.**

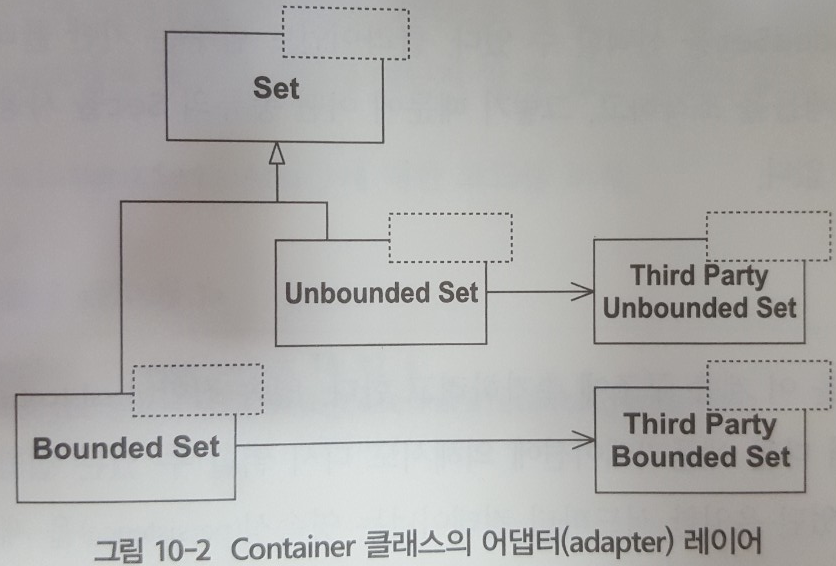
파생된 객체는 기반 클래스가 요구하는 것보다 더 강한 조건을 따를 것이라고 기대할 수 없다. 즉 파생된 객체는 기반 클래스가 받아들일 수 있는 것을 모두 받아들여야 한다.

또한 파생 클래스는 기반 클래스의 모든 사후조건을 따라야 한다.

🡺행위와 출력은 기반 클래스의 제약을 위반해서는 안 된다.

**단위 테스트에서의 계약사항 구체화하기**

계약은 또한 단위 테스트를 작성함으로써 구체화될 수 있다. 단위테스트는 그 클래스의 행위를 좀 더 분명하게 만들어 준다.



Set이란 추상 클래스를 만들고, Add, Delete, IsMember를 제공한다.

이 구조는 유계(Bounded Set)와 무계(Unbounded Set)를 통합한다.

public abstract class Set<T> {  
 public abstract void add(T t);  
  
 public abstract void delete(T t);  
  
 public abstract boolean isMember(T t);  
  
}

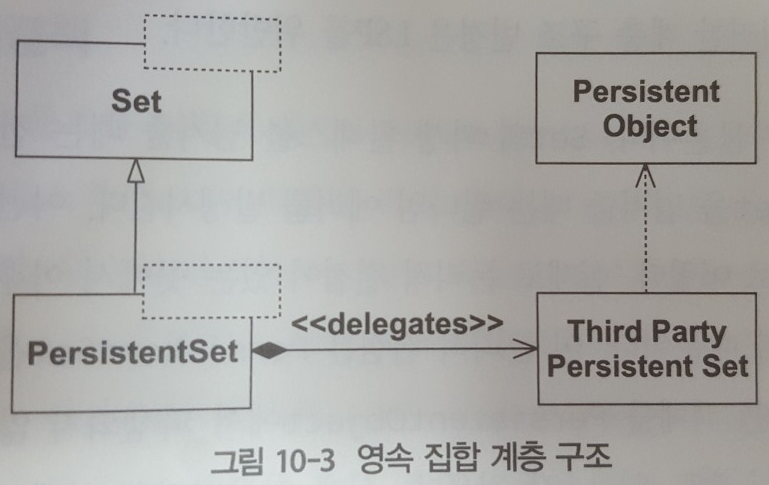
사용할 Set의 종류를 알고 그것에 신경 쓸 필요가 없다는 건 큰 장점이다.

🡺메모리가 빠듯하고 속도가 중요하지 않은 경우UnboundedSet,

메모리가 충분하고 속도가 중요한 경우에는 BoundedSet을 선택할 수 있다.

**문제**

여기에 PersistentSet을 추가하려고 한다.



여기에는 좋지 않은 의미가 숨어 있다.

서드파티 영속 집합에 추가되는 원소는 PersistentObject에서 파생되어야 한다.

PersistentSet은 단순히 서드파티 영속 집합에 위임하는 역할을 하기 때문에, PersistentSet에 추가되는 원소는 PersistentObject에서 파생되어야 하는 것이다.

public class PersistentSet extends Set {  
 public void add(Object o) {  
 if(o instanceof PersistentObject) {  
 PersistentObject p = (PersistentObject)o;  
 itsThirdPartyPersistentSet.Add(p);  
 }else {  
 throw new Exception();  
 }  
 }

}

이 코드는 PersistentObject 클래스에서 파생되지 않은 객체를

PersistentSet에 추가하려는 클라이언트가 있으면 에러를 발생할 것이다.

🡺추상 기반 클래스 Set의 클라이언트 중 어느 것도 add에서 예외가 발생할 것이라고 기대하지 않는다.(LSP 위반)

**LSP를 따르지 않는 해결책**

규정에 따라 해결한다.

1)PersistentSet과 PersistentObject가 전부 다 애플리케이션에 알려지지 않게 한다.

🡺단 하나의 특정한 모듈에만 알려진다. (모든 컨테이너를 읽고 쓰는 책임)

2)컨테이너 Write 시, 적절한 PersistentObject의 파생 객체에 복사되고 PersistentSet에

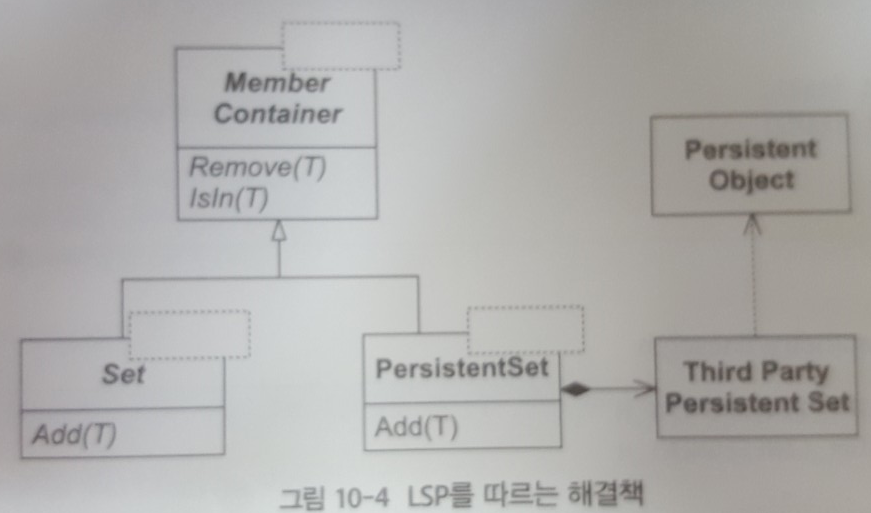
추가된다 (add)

3)컨테이너 Read 시, PersistentSet은 스트림에서 읽히며, PersistentObject는

PersistentSet에서 지워지고 일반 객체에 복사된다.

4)규정이 너무 제한적이지만 효과를 보지 못했다. 일부 개발자들이 이것을 위반했기 때문이다. (또한 각 개발자에게 끊임없이 전파 해야된다. 귀찮다)

**LSP를 따르는 해결책**



PersistentSet이 Set과 IS-A 관계에 있지 않기 때문에 파생이 아니라 따로 분리 한다.

이것은 PersistentSet 객체들을 순환 검색할 수 있고, 멤버 여부도 테스트할 수 있게 만들어줄 것이다. 그러나 PersistentObject에서 파생되지 않은 객체를 PersistentSet에 추가할 수 있게 만들 수는 없다.

**파생 대신 공통 인자 추출하기**

public class Line {  
 private Point itsP1;  
 private Point itsP2;  
  
 public Line(Point p1, Point p2) {  
 }  
  
 public double getSlope() {  
 return 0.0;  
 }  
  
 public double getIntercept() {  
 return 0.0;  
 }  
  
 Point getP1() {  
 return itsP1;  
 }  
  
 Point getP2() {  
 return itsP2;  
 }  
  
 boolean isOn(Point point) {  
 return true;  
 }  
}  
  
class LineSegment extends Line {  
 public LineSegment(Point p1, Point p2) {  
 super(p1, p2);  
 }  
  
 public double getLength() {  
 return 0.0;  
 }  
  
 @Override  
 boolean isOn(Point point) {  
 return true;  
 }  
}

Line은 IsOn(Intercept()) == true 이지만

LineSegment는 이 판정식에 실패 한다.

🡺판단(올바른 결정을 내리기 어려운 상황에서 내리는 결정)을 내려야 하는 문제다.

🡺설계를 고쳐서 완벽하게 LSP에 맞는 설계를 하기보다는 다형적인 행위에서 미묘한 결점은 놔두는 것이 좀더 적절한 대응인 경우도 있다. (드물게..)

🡺그러나 LSP를 가볍게 포기해서는 안 된다.(기반 클래스가 사용되는 곳에서 서브클래스가 항상 제대로 동작함을 보장하는 것은 복잡성을 다루는 강력한 방법이다.)

public class LinerObject {  
  
 private Point itsP1;  
 private Point itsP2;  
  
 public LinerObject(Point p1, Point p2) {  
  
 }  
  
 public double getSlope() {  
 return 0.0;  
 }  
  
 public double getIntercept() {  
 return 0.0;  
 }  
  
 public Point getP1() {  
 return itsP1;  
 }  
  
 public Point getP2() {  
 return itsP2;  
 }  
  
 public boolean isOn(Point point) {  
 return true;  
 }  
}  
  
class Line extends LinerObject {  
 public Line(Point p1, Point p2) {  
 super(p1, p2);  
 }  
   
 @Override  
 public boolean isOn(Point point) {  
 return true;  
 }  
}  
  
class LineSegment extends LinerObject {  
 public LineSegment(Point p1, Point p2) {  
 super(p1, p2);  
 }  
  
 public double getLength() {  
 return 0.0;  
 }  
  
 @Override  
 public boolean isOn(Point point) {  
 return true;  
 }  
}

두 서브클래스(Line과 LineSegment)에서 isOn 메서드만 제외하고, 대부부분 부모클래스에서 제공한다.

Line클래스에 수십 개의 클라이언트가 있다면 LinerObject 클래스를 추출하기 쉽지는 않을 것이다. 하지만 이건 강력한 수단이 된다.

🡺어떤 클래스 집합이 모두 같은 책임을 진다면, 공통 슈퍼클래스에서 그 책임을 상속 받는다. 이 새로운 슈퍼클래스는 아마 추상클래스가 될 것이다.

**파생 클래스에서의 퇴화 함수**

???

**파생 클래스에서의 예외 발생**

기반 클래스가 발생시키지 않는 예외를 파생 클래스의 메소드에 추가하는 것이다.

이들은 치환 가능하지 않다.

**결론**

LSP는 OCP를 가능하게 하는 주요 요인 중 하나다.

**서브타입의 진실된 정의는 ‘치환 가능성’ 이다.**

**11. 의존관계 역전 원칙(Dependency Inversion Principle)**

a. 상위 수준의 모듈은 하위 수준의 모듈에 의존해서는 안 된다. 둘 모두 추상화에 의존해야 한다.

b. 추상화는 구체적인 사항에 의존해서는 안 된다. 구체적인 사항은 추상화에 의존해야 한다.

왜 역전인가?

전통적인 소프트웨어 개발 방법에서는 상위 수준의 모듈이 하위 수준의 모듈에 의존하고

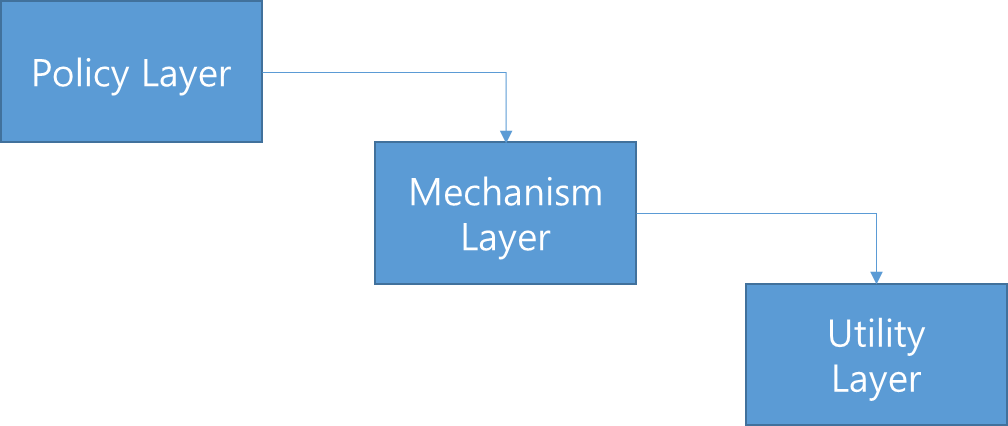
정책이 구체적인것에 의존하는 경향이 있었기 때문이다.

**🡺잘 설계된 객체 지향 프로그램의 의존성 구조는 전통적인 절차적 방법의 의존성 구조가 ‘역전’ 된 것이다.**

상위 수준의 모듈은 구체적인 구현을 포함한 모듈에 우선하면서 동시에 독립적이어야 한다.

상위 수준의 모듈은 어떤 식이로든 하위 수준의 모듈에 의존해서는 안 된다.

**레이어 나누기**

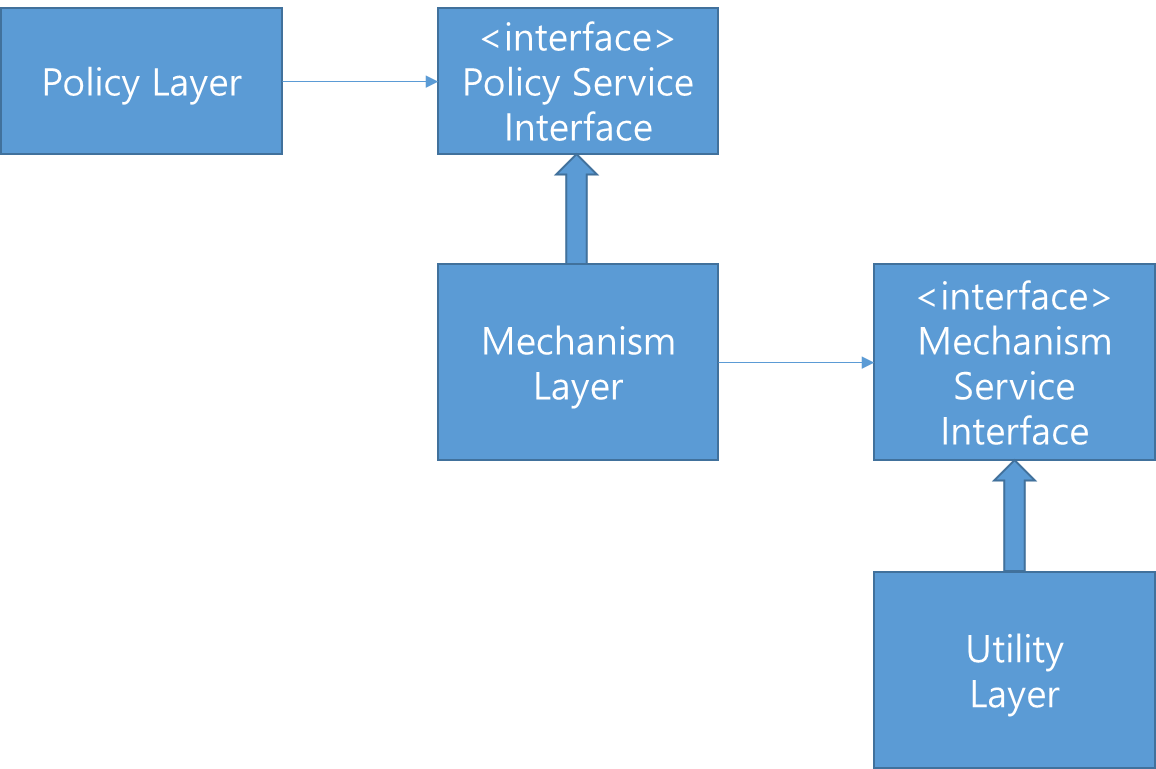


Policy 레이어가 아래 Utility 레이어의 모든 변화에 민감하다는 특성이 함적으로 숨어 있다.

이행적 종속 관계

A->B이고 B->C일 때 A->C를 만족하는 관계

Policy Layer는 Utiltiy Layer에 이행적이다. (불행하다)



각 상위 수준 레이어는 그것이 필요로 하는 서비스에 대한 추상 인터페이스를 선언한다.

하위 수준 레이어는 추상 인터페이스를 통해 실체화된다.

각 상위 클래스는 추상 인터페이스를 통해 하위 수준의 레이어를 사용한다.

🡺상위 레이어는 하위 레이어에 의존하지 않는다.

🡺반대로 하위 레이어는 상위 레이어에 의존한다.

**소유권의 역전**

역전은 의존성에 대해서만이 아니라, 인터페이스 소유권에 대한 것도 의미한다.

**추상화에 의존하자**

DIP에 대한 강력한 해석은 ‘**추상화에 의존하자**’ 라는 간단한 경험적 접근 방식이다.

**🡺구체 클래스에 의존해서는 안 되고 모든 관계는 어떤 추상 클래스나 인터페이스에서 맺어져야 한다.**

구체적이긴 하지만 비휘발적인 클래스에는 이 경험적 접근 방식을 적용할 이유가 없어 보인다.

구체 클래스가 너무 많이 변경되지 않고 파생클래스가 만들어 지지 않는다면 큰 해가 되지 않는다.

**간단한 예**

public class Button {  
 private Lamp itsLamp;  
  
 public void poll() {  
 itsLamp.turnOn();  
 }  
}  
  
class Lamp {  
 public void turnOn() {  
 System.*out*.println("On");  
 }  
  
 public void turnOff() {  
 System.*out*.println("Off");  
 }  
}

Button 클래스가 Lamp 클래스에 직접 의존하고 있다. 이런 의존성은 Button이 Lamp에 대한 변경에 영향을 받을 것임을 의미한다.

이 설계에서는 Button 객체는 오직 Lamp 객체만을 제어한다.

🡺DIP를 위반한다. (상위 수준 정책은 하위 수준 구현에서 분리되지 않았다.)

**내재하는 추상화를 찾아서**

public class Button {  
 private ButtonServer button;  
  
 public Button(ButtonServer button) {  
 this.button = button;  
 }  
 public void poll() {  
 button.turnOn();  
 }  
}  
  
interface ButtonServer {  
 public void turnOn();  
  
 public void turnOff();  
}  
  
class Lamp implements ButtonServer {  
 public void turnOn() {  
 System.*out*.println("On");  
 }  
  
 public void turnOff() {  
 System.*out*.println("Off");  
 }  
}

Lamp 객체의 의존성을 역전시킴으로써 개선했다.

Lamp는 이제 Button에 의해 의존당하는 것이 아니라ButtonServer를 의존하게 되었다.

Button이 ButtonServer 인터페이스를 구현하려는 어떤 장치든 제어할 수 있다.

🡺그러나 이 해결책은 Button에 의해 제어되길 원하는 객체에 제약을 걸게 된다. 이런 객체는 ButtonServer 인터페이스를 구현해야 되는데, 이는 Button이 아닌 Switch로 제어되길 원할수도 있다.

* + ButtonServer를 SwitchableDevice 같은 개념으로 변경함으로써 고칠 수 있다.

**동적 다형성과 정적 다형성**

동적 다형성(추상 클래스나 인터페이스)를 이용해서 의존성의 역전을 해결할 수 있다.

정적 다형성은 소스 코드의 의존성을 깔끔하게 끊어주지만, 동적 다형성만큼 많은 문제를 해결해주지는 않는다.

🡺그냥 동적 다형성 써라

**결론**

절차 지향 프로그래밍 방식은 정책이 구체적인 것에 의존하는 의존성 구조를 만든다.

🡺정책이 구체적인 사항의 변경에 따라 같이 변한다.

객체지향 프로그래밍은 이런 의존성 구조를 역전시켜 구체적인 사항과 정책이 모두 추상화에 의존하고, 클라이언트가 서비스 인터페이스를 소유하게 만든다.

추상화와 구체적 사항이 서로 분리되어 있기 때문에, 유지보수하기 훨씬 쉽다.

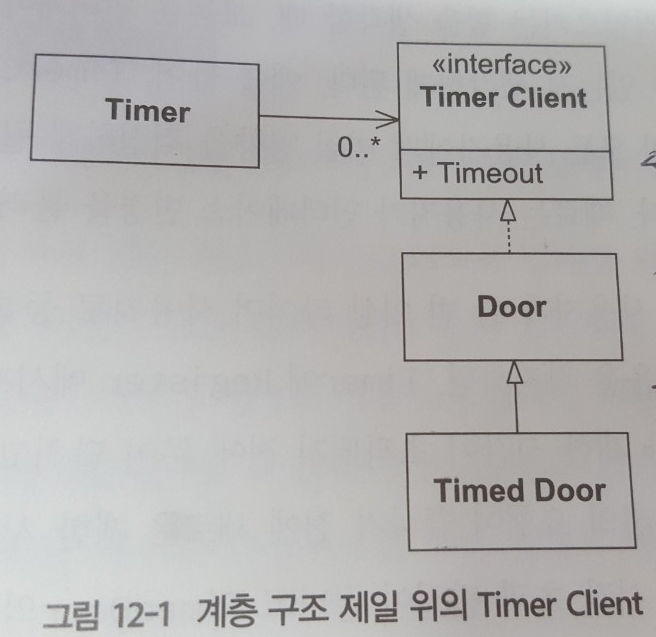
**인터페이스 분리 원칙(Interface-Segregation Principle)**

비대한 인터페이스를 가지는 클래스는 응집력이 없는 인터페이스를 가지는 클래스다.

**인터페이스의 오염**

public abstract class Door {  
 public abstract void lock();  
  
 public abstract void unLock();  
  
 public abstract boolean isDoorOpen();  
}  
  
class Timer {  
 public void register(int timeout, TimerClient client) {  
  
 }  
}  
  
abstract class TimerClient {  
 public abstract void timeOut();  
}

이 클래스는 추상 클래스이기 때문에 클라이언트는 Door 의 특정한 구현에 의존하지 않고,도 인터페이스를 따르는 객체를 사용할 수 있다.



TimerClient 클래스가 TimedDoor 클래스와 통신하여 TimedDoor의 코드에서 제한 시간 초과 여부를 받게 하기 위해 이런 구조를 선택했다.

🡺심각한 문제는 Door 클래스가 이제 TimerClient에 의존하게 되었다.

🡺Door의 모든 변형 클래스가 타이머 기능을 필요로 하는 것은 아니다.

🡺타이머 기능을 쓰지 않는 Door의 변형 클래스가 만들어진다면 잠재적 LSP위반이다.

결국 이 구조는 서브 클래스 중 하나의 이득 때문에 이 메소드를 포함시켜야 했다. (Timeout 사용하려고…)

🡺파생 클래스 추가 될때매다 더 오염시키고, 비대하게 만든다.

🡺더 심각한건 부모 클래스에 메서드 추가 시, 하위 클래스에서도 구현되어야 한다.

**클라이언트 분리는 인터페이스 분리를 의미하다.**

Timer는 TimerClient에 의해 사용되고,

문을 조작하는 클래스는 Door를 사용한다.

🡺클라이언트가 분리되어 있기 때문에 인터페이스도 분리된 상태로 있어야 한다.

**클라이언트가 인터페이스에 미치는 반대 작용**

보통 인터페이스 변경이 어떻게 그 사용자에게 영향을 미칠 수 있는지 생각하게 된다.

예를 들어 Timer의 몇몇 사용자가 한 번 이상 타이머 사용자로 등록하게 된다.

첫 번째 문이 열린 Timer(5초) 는 시간이 측정되는 사이 문이 닫힌다. (3초)

두 번째 문이 열린다 (2초 후 문이 열려 있으므로 기존에 5초 Timer가 울린다.)

class Timer {  
 public void register(int timeout, int timeOutId, TimerClient client) {  
  
 }  
}  
  
abstract class TimerClient {  
 public abstract void timeOut(int timeOutId);  
}

TimeOutId를 추가해서 어떤 타이머 사용 요청에 대한 응답을 받고 있는지 알 수 있게 해준다.

TimerClient가 수정되었으므로 Door와 Door의 모든 클라이언트가 이 수정에 영향을 받는다.

🡺 경직성과 점착성의 악취를 풍긴다.

🡺 왜 TimerClient 버그를 사용하지도 않는 Door 파생 클라이언트까지 영향을 주어야 하는가?

🡺 프로그램 한 부분의 변경이 전혀 관계 없는 부분에도 영향을 줄 때, 이 변경에 드는 비용과 그 영향은 예상할 수 없을 정도가 된다.

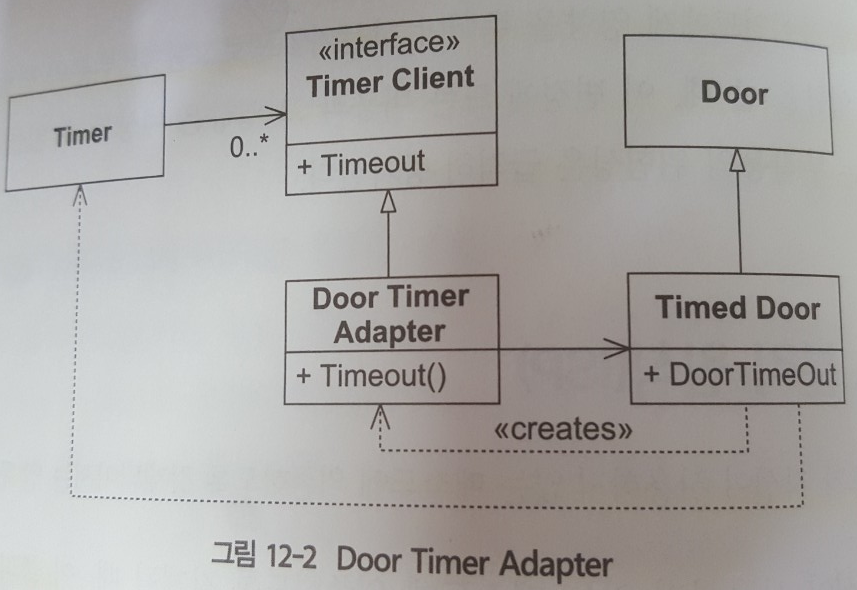
**인터페이스 분리 원칙(ISP)**

**클라이언트가 자신의 사용하지 않는 메소드에 의존하도록 강제되어서는 안 된다.**

**클래스 인터페이스와 객체 인터페이스**

위임이나 기반 클래스를 통해 접근할 수 있다.

**위임을 통한 분리**

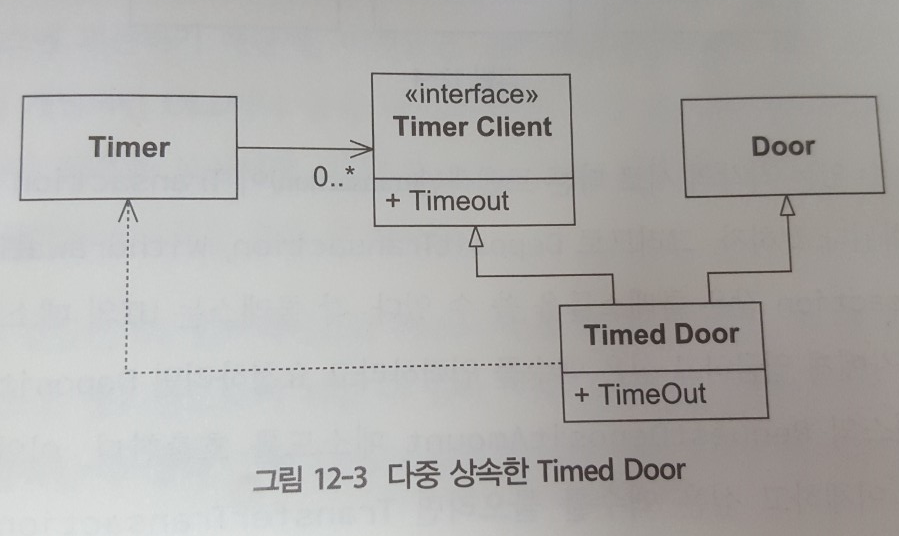


TimerClient에서 파생된 객체를 생성하고 그것의 일을 TimedDoor에 위임하는 것이다.

abstract class TimerClient {  
 public abstract void timeOut(int timeOutId);  
}  
  
class DoorTimerAdapter extends TimerClient {  
 private TimedDoor itsTimerdoor;  
  
 public DoorTimerAdapter(TimedDoor theDoor) {  
 this.itsTimerdoor = theDoor;  
 }  
  
 public void timeOut(int timeOutId) {  
 itsTimerdoor.doorTimeOut(timeOutId);  
 }  
}

🡺다소 새련되지 못하다. 타이머 등록할 때마다 새 객체를 생성.

**다중 상속을 통한 분리**



TimedDoor는 Door와 TimerClient에서 모두 상속을 받는다. 실제로 둘다 TimedDoor 클래스에 의존하지는 않는다. 따라서 이들은 분리된 인터페이스를 통해 같은 객체를 사용하게 된다.

class TimedDoor implements Door, TimerClient{  
 public void TimeOut(int timeOutId);  
}

결론

비대한 클래스는 클라이언트들 간에 기이하고 해가 되는 결합도를 유발한다.

이 비대한 클래스의 인터페이스를 클라이언트 고유의 인터페이스 여러 개로 분해해야 한다.

🡺클라이언트 고유의 각 인터페이스는 자신의 특정한 클라이언트나 클라이언트 그룹이 호출하는 함수만 선언한다.

**패턴은 관련 링크로 대체**

커맨드 패턴

http://huiyu.tistory.com/entry/7-Command-Pattern%EC%BB%A4%EB%A7%A8%EB%93%9C-%ED%8C%A8%ED%84%B4

액티브 오브젝트 패턴

http://hamait.tistory.com/215

상속의 목적은 재사용성이 아닌 확장성

수퍼 클래스가 변경되면 하위 클래스도 영향을 준다

http://jinwoojeon.tistory.com/2

퍼사드 (정면)

https://minwook-shin.github.io/facade-pattern/

미디에이터 (중재자)

http://leetaehoon.tistory.com/67

모노스태이트

널 오브젝트 패턴

http://kunoo.tistory.com/entry/%ED%96%89%EC%9C%84-%ED%8C%A8%ED%84%B4-Null-Object-pattern-%EB%84%90-%EC%98%A4%EB%B8%8C%EC%A0%9D%ED%8A%B8-%ED%8C%A8%ED%84%B4

**급여 관리 사례 연구**

**급여 관리 시스템의 기본 명세**

전제

-회사의 직원들 및 그들과 관련된 타임카드(출퇴근 시간 기록) 같은 데이터로 구성되어 있다.

-이 시스템은 각 직원에게 임금을 지급해야 하며, 직원들은 그들이 지정한 방식으로 정확한 시간에 정확한 액수를 지급받아야 한다.

-이들은 임금에서 다양한 공제가 가능해야 한다.

**직원타입**

1) 알바

-직원 레코드의 시급필드에 따라 임금을 받는다.

-매일 날짜와 일한 시간을 기록한 타임카드를 제출한다

-하루에 8시간 이상 일하면 초과 근무 시간에 대해 1.5배를 받는다.

-매주 금요일마다 임금을 받는다. (주급)

2) 일반직원

-매달 마지막 평일에 임금을 받는다. (월급)

-월급 액수는 직원 레코드의 한 필드가 된다.

3) 영업

-날짜와 판매량이 기록된 판매 영수증을 제출한다.

-수수료율은 직원 레코드의 한 필드가 된다.

-이들은 격주로 금요일마다 임금을 받는다. (2주)

**임금지급 방법**

1) 우편 주소로 급료 우송

2) 급여 담당자에게 맡겨놓았다가 찾아갈 수도 있다.

3) 자신이 선택한 은행 계좌로 직접 입금

**공 제**

1)몇몇 직원은 조합에 속해 있다.

-직원 레코드에는 주당 조합비 비율을 나타내는 필드가 있다. (임금에서 공제)

-가끔 조합원 개인에게 공제액을 부과할 수도 있다.

(주 단위로 조합에 의해 제출, 다음달 임금에서 공제)

**실행주기**

1)급여 관리 애플리케이션은 평일에 한 번씩 실행되고 해당 직원에게 그날 임금을 지급한다.

2)직원이 임금을 받을 날짜를 입력받아, 지정된 날짜 전에 마지막으로 임금을 받은 날부터 지정된 날까지의 임금을 계산한다.

**20. 패키지 설계의 원칙**

크기와 복잡성이 증가하면서 조직화가 요구됨

작은 애플리케이션 🡺 클래스

큰 애플리케이션 🡺 패키지

패키지 응집도(3가지) : 클래스를 패키지에 할당하는 일을 도와준다.

패키지 결합도(3가지) : 패키지 간의 관계를 결정하는 일을 도와준다.

**패키지를 이용한 설계?**

클래스를 패키지로 묶어놓으면 더 높은 추상화 차원에서 설계에 대한 논의를 할 수 있다.

패키지는 소프트웨어 개발이나 배포를 관리하기 위해 쓰이기도 한다.

**단위 크기:패키지 응집도의 원칙**

세가지 원칙은 개발자가 어떻게 클래스를 패키지에 분류해 넣을지 결정할 때 도움이 된다. (상향식)

**1) 재사용 릴리즈 등가 원칙(REP : Reuse-Release Equivalence Principle)**

먼저, 다른 사람의 코드를 재사용해서 유용하게 쓸 수 있도록 작성자가 그 코드를 유지 보수해주기를 바라게 된다.

다음으로, 작성자가 인터페이스나 기능을 바꾸기 전에 미리 여러분에게 통해주기 바라게 된다.

🡺둘 중 어떤 경우라도 여러분이 새 버전을 거부했다면 작성자는 일정 기간 옛 버전을 사용할 수 있도록 지원해주어야 한다.

작성자가 여러분과 관계를 끊고 지원을 거부해선 안 된다.

🡺이 문제는 근본적으로 정치적이다. 이러한 정치적이고 사무적인 문제가 소프트웨어의 패키지 구조에 중요한 영향을 미친다.

재사용 릴리즈 등가 원칙에 따르면 재사용 단위는 **릴리즈 단위보다 작을 수 없다**.

잠재적인 재사용자들에게 필요한 통보, 안정성, 지원에 대한 보장을 제공하는 추적 시스템이 먼저 있은 다음에야 재사용성이라는 말을 할 수 있다.

**패키지의 모든 클래스가 재사용 가능하든지, 모두 그렇지 않든지 해야 한다.**

**2)공통 재사용 원칙(CRP : Common-Reuse Principle)**

패키지 안의 클래스들은 함께 재사용되어야 한다. 어떤 패키지의 클래스 하나를 재사용한다면 나머지 모두 재사용한다.

이 원칙은 어떤 클래스들이 패키지에 포함되어야 하는지 결정할 때 도움이 된다. 이 원칙에 따르면 자주 함께 재사용되는 클래스들은 동일한 패키지에 속해 있다.

단독으로 재사용되는 클래스는 거의 없다. 대부분의 경우, 재사용 가능한 클래스들은 재사용 가능에 대해 같은 추상적 범주에 속해 있는 다른 클래스와 협력한다. (모두 동일한 패키지에 속해 있다. Ex) iterator)

반면 CRP는 어떤 클래스를 같은 패키지에 넣지 않아야 할지도 말해준다. 패키지가 다른 패키지를 사용하면 둘 사이에 의존 관계가 생긴다.

🡺두 번째 패키지가 변경되면 첫번째 패키지도 재검증하고 다시 릴리즈 해야 한다. 그것이 첫번째 패키지와 전혀 상관없는 클래스 하나에 생긴 변화 때문일지라도…

따라서 어떤 패키지에 의존한다면 그 패키지의 모든 클래스에 의존하는지 확실히 해두어야 한다.

다시말해 패키지에 넣는 클래스들이 서로 뗄 수 없는 관계여서 일부에만 의존하고 나머지에는 의존하지 않는 일이 불가능한지 확실히 해두어야 한다.

**3)공통 폐쇄 원칙(CCP : Common-Closure Principle)**

같은 패키지 안의 클래스들은 동일한 종류의 변화에는 모두 폐쇄적이야 한다. 패키지에 어떤 변화가 영향을 미친다면, 그 변화는 그 패키지의 모든 클래스에 영향을 미쳐야 하고 다른 패키지에는 영향을 미치지 않아야 한다.

CCP는 대상이 패키지인 단일 책임 원칙(SRP) 이다.

CCP는 동일한 이유로 변할 것 같은 클래스들은 한 장소에 모아놓으라고 권장한다.

클래스 2개가 물리적으로나 개념적으로나 단단히 결합되어 있어서 변경해야 된다면?

🡺이들은 같은 패키지에 속하고 이렇게 하면 릴리즈, 재검증, 재배포하는 일과 관련된 작업량을 최소로 줄일 수 있다.

**CCP vs CRP**

CCP를 확대하면 CRP가 축소 : 전체 패키지 수가 줄어듬 : 개발 용이성

CRP를 확대하면 CCP가 축소 : 전체 패키지 수가 늘어남 : 재사용성

**안정성: 패키지 결합도의 원칙**

세가지 원칙은 패키지 상호 관계에 대해 다룬다.

**1)의존 관계 비순환 원칙(ADP : Acyclic-Dependencies Principle)**

패키지 의존성 그래프에서 순환을 허용하지 말라

**주간 빌드**

4일동안 다른 개발자를 신경 쓰지 않고 개발한다. 그리고 금요일날 통합해서 시스템을 빌드해본다.

🡺프로젝트의 규모가 커질수록 복잡해진다. 효율성도 따라서 내려간다.

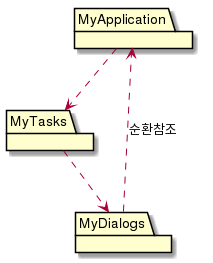
**의존관계 순한 없애기(ADP)**

개발 환경을 릴리즈로 만들 수 있는 패키지로 분할하는 것이다.

패키지가 작업의 단위가 되고, 개발자나 팀은 패키지를 체크아웃해서 작업한다.

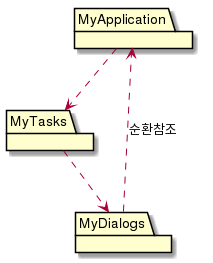
패키지가 새로 릴리즈되면, 다른 팀은 새 릴리즈를 즉시 채택할지 말지 결정할 수 있다.

이 밥ㅇ법이 제대로 동작하려면 여러분이 패키지의 의존 관계 구조를 관리해주어야 한다. 의존 관계 구조에 순환이 있으면 안 된다.



순환 참조가 있으면 안된다. 비 순환 방향 그래프이어야 한다.

**패키지 의존 관계 그래프에 순환이 있을 경우 생기는 결과**

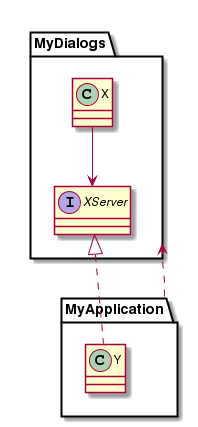


릴리즈 시 한 번에 모든 패키지의 버전을 다 올려서 배포해야 된다. (거대한 패키지가 된다.)

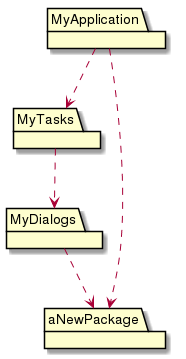
테스트 시도 전체 빌드를 해야 된다.

**순환을 끊기**

1. 의존 관계 역전 원칙(DIP)



2. 둘다 의존하는 새로운 패키지를 만든다.



**하향식 설계**

초기에 **SRP, CCP**에 주의를 기울여서 함께 변경되기 쉬운 클래스들은 함께 묶어놓기 시작한다.

애플리케이션이 계속 성장하면서 **CRP**가 패키지 구성을 지배하게 된다.

마지막으로 순환이 나타나면서 이제 **ADP**가 적용되고 패키지 의존 관계 그래프는 조금씩 흔들리면서 성장한다.

클래스를 설계하기 전에 패키지 의존 관계 구조 설계를 먼저 시도한다면 상당히 큰 실패를 맛볼 가능성이 크다

🡺하향식 설계를 할 수 없다. 시스템의 논리적 설계와 함께 성장하고 진화해야 한다.

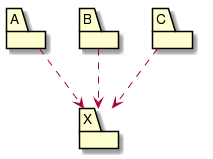
**2)안정된 의존 관계 원칙 (SDP : Stable-Dependencies Principle)**

의존은 안정적인 쪽으로 향해야 한다.

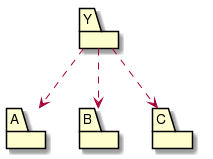
쉽게 바뀔 것이라고 예상되는 패키지들이 바뀌기 어려운 패키지들의 의존 대상이 되어서는 안된다. 이렇게 되면 쉽게 바뀔 패키지들도 바꾸기 어렵게 되어버린다.

🡺SDP를 지킴으로써, 쉽게 변경할 수 있도록 의도한 모듈이 변경하기 어려운 모듈의 의존 대상이 되지 않도록 보장할 수 있다.

**안정성**



X : 세 패키지에 책임이 있다. X는 독립적이다 (안정적)



Y : 책임이 없다. Y는 패키지 3개에 의존하기 때문에, 변경을 일으킬 가능성이 많다. (의존적)

**모든 패키지가 안정적일 필요는 없다**

만약 시스템에서 모든 패키지의 안정성이 가장 중요하다면 시스템은 변경할 수 없게 될 것이며, 이것은 그다지 바람직한 상황이 아니다.

**높은 차원의 설계는 어디에 두어야 하는가?**

시스템에서 높은 차원의 설계를 캡슐화한 소프트웨어는 안정적인 패키지 (I=0)에 넣어야 한다.

불안정한 패키지(I=1)는 변경될 가능성이 높은 소프트웨어만 포함해야 한다.

하지만 높은 차원의 설계가 안정적인 패키지에 들어간다면 이 설계를 나타내는 소스 코드를 변경하기가 어려워진다.

어떻게 해야 안정적인(I=0) 패키지를 , 변화를 견딜 수 있도록 유연하게 만들 수 있을까?

🡺OCP가 답이다. 추상클래스

**3)안정된 추상화 원칙(SAP : Stable-Abstractions Principle)**

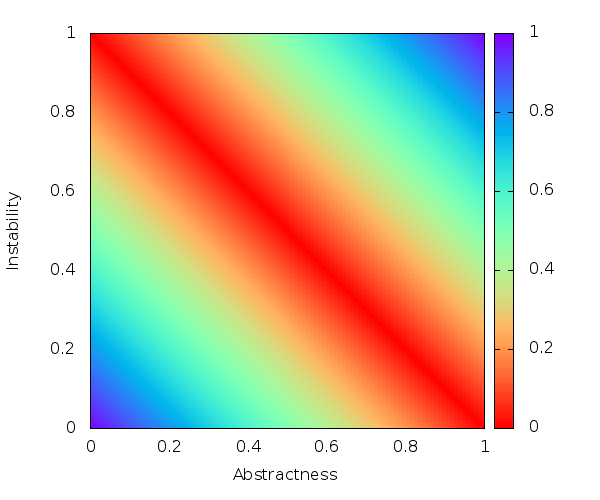
패키지는 자신이 안정적인 만큼 추상적이기도 해야 한다.

안정된 추상화 원칙은 안정성과 추상성 사이의 관계를 정한다.

어떤 패키지가 안정적이라면 확장할 수 있도록 추상 클래스들로 구성되어야 한다. 확장이 가능한 안정적인 패키지는 유연하며, 따라서 설계를 지나치게 제약하지 않는다.

**SAP + SDP = DIP 패키지판**

**주계열**



**0,0 :** 안정적이고 구체적 : 고통의 지역(Zone of Pain)

ex) DB, StringUtils

**1,1 :** 의존성이 없고(불안정적) 추상적 : 쓸모없는 지역(Zone of Uselessness)

**1,0 ~ 0, 1 붉은색 직선(주계열) :** 너무 추상적이지도 않고 너무 불안정적 하지도 않음

주계열 양 끝점이 이상적인 위치