6. 객체와 자료 구조 (디미터~참고문헌)

디미터 법칙

`디미터 법칙` : 모듈은 자신이 조작하는 객체의 속사정을 몰라야 한다는 법칙으로, 객체는 자료를 숨기고 함수를 공개한다.
다시 말해 **객체는 조회 함수(getter 등)로 내부 구조를 공개하면 안된다는 의미이다.**
즉, 디미터 법칙은 `클래스 C의 메서드 F는 다음과 같은 객체의 메서드만 호출해야 한다.`고 주장한다.

- 클래스 C
- F 가 생성한 객체
- F 인수로 넘어온 객체
- C 인스턴스 변수에 저장된 객체
  하지만 위 객체에서 허용된 메서드가 반환하는 객체의 메서드는 호출하면 안된다.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- 객체가 반환한 객체의 메서드를 사용하고 있다.
- 이는 디미터 법칙을 위반한 것이다.
  참고로 `자기 자신의 객체를 리턴`해서 사용하는 **체이닝 기법**은 가능하다.
  `체이닝 기법`
- 체이닝 기법은 위 상황과 달리 자기 자신의 객체를 리턴해서 사용하는 기법
- 즉, this를 반환하여 보다 깔끔하고 명확한 코드를 만들 수 있게 해준다.

기차 충돌

다음과 같은 코드를 기차 충돌train wreck(여러 객차가 한 줄로 이어진 기차처럼 보이는 현상)이라 부른다.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

일반적으로 조잡하다 여겨지며 피하자.

```
// 개선된 코드
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

위 개선된 코드는 디미터 법칙을 충족하는가?

- ctxt, Options, ScratchDir 이 객체라면 디미터 법칙을 위반 (내부 구조를 숨겨야 하므로)
- ctxt, Options, ScratchDir 이 자료 구조라면 디미터 법칙이 적용되지 않는다. (당연히 내부 구조를 공개하므로)

자료 구조 형태 - 절차 지향법

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

- 무조건 함수 없이 `public` 참조 변수들(공개 변수)을 직접 호출하기에 자료 구조 형태임을 알 수 있다.
- 코드를 위와 같이 구현한다면 디미터 법칙을 거론할 필요가 없어진다.

잡종 구조(Hybrids)

절반은 객체, 절반은 자료 구조인 구조를 잡종 구조라함.
이런 잡종 구조는 새로운 함수, 새로운 자료 구조도 추가하기 어렵다.
양쪽의 단점만 모아놓은 구조이므로 잡종 구조는 되도록 피하는 편이 좋다.

구조체 감추기

만약 ctxt, Options, ScratchDir 이 진짜 객체라면?

```
ctxt.getAbsolutePathOfScratchDirectoryOption(); // 객체에 공개해야 하는 메서드가 너무 많아짐
ctx.getScratchDirectoryOption().getAbsolutePath() // 객체가 아니라 자료구조를 반환한다고 가정
```

- 위의 개선된 코드도 잘못된 형태이다.
  - 줄줄이 사탕으로 엮여서는 안 된다.
  - 객체라면 내부 구조를 감춰야 한다.
- 위의 코드들이 왜 사용되는지 확인하고, **내부의 객체를 가져다 쓰는 것이 아닌 `메세지`를 보내 필요한 정보를 반환하도록 해야 한다.**
- 즉, ctxt 객체가 **뭔가를 하라고** 해야지, 속을 드러내면 안된다.

`outputDir` 임시 디렉토리의 절대 경로가 왜 필요할지 찾아보았고, 같은 모듈에서 아래와 같은 코드를 가져왔다.

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

결론적으로 경로를 얻으려는 이유가 임시 파일을 생성하기 위함을 알 수 있다.

```
// 위의 코드의 outputDir의 목적이 무엇인지 확인하고 수정한다.
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

ctxt는 내부 구조를 들어내지 않으며, 모듈에서 해당 함수는 자신이 몰라야 하는 여러 객체를 탐색할 필요가 없다. 따라서 디미터 법칙을 위배하지 않는다.
코드의 목적이 무엇인지 확인하고 그에 알맞게 구현하는 것이 중요하다.

자료 전달 객체

자료 구조체의 전형적인 형태 : 공개 변수(public)만 있고 함수가 없는 클래스다. -> `자료 전달 객체(Data Transfer Object : DTO` 데이터베이스와의 통신과 소켓에서 받은 메시지의 구문을 분석할 때에 유용한 구조체다.

```
public class Address {
  public String street;
  public String streetExtra;
  public String city;
  public String state;
  public String zip;
}
```

좀 더 일반적인 형태 `빈(Bean) 구조` : 비공개 변수(private), 조회/설정(Getter, Setter) 함수로 조작

활성 레코드(Active Record)

`활성 레코드` : DTO의 특수한 형태 - (Bean 구조 + save, find 메서드 포함)
1. 공개 변수가 있거나 비공개 변수에 조회 설정 함수가 있는 자료 구조
2. 대게 **save나 find와 같은 탐색 함수도 제공**한다.
3. 활성 레코드는 데이터베이스 테이블이나 다른 소스에서 자료를 직접 반환한 결과이다.
활성 레코드에 비지니스 규칙 메서드를 추가해 이런 자료 구조를 객체로 취급하는 개발자가 흔하다. 이렇게 되면 잡종 구조가 나오게 된다.

해결책은 간단하다. **활성 레코드는 자료 구조로 취급한다.** 비지니스 규칙을 담으면서, 내부 자료를 숨기는 객체는 따로 생성한다. (여기서 내부 자료는 활성 레코드의 인스턴스일 확률이 높다.)

결론

- `객체`는 동작을 공개하고 자료를 숨긴다. 그래서 기존 동작을 변경하지 않으면서 새 객체 타입을 추가하기 쉽지만 기존 객체에 새 동작을 추가하기는 어렵다.

- `자료 구조`는 별다른 동작 없이 자료를 노출한다. 그래서 기존 자료 구조에 새 동작을 추가하기는 쉬우나 기존 함수에 새 자료 구조를 추가하기는 어렵다.
- 어떤 시스템을 구현할 때 새로운 자료 타입을 추가하는 유연성이 필요하면 객체가 더 적합하다.
- 다른 경우로 새로운 동작을 추가하는 유연성이 필요하면 자료 구조와 절차적인 코드가 더 적합하다.

참고 문헌

- Refactoring:Improving the Design of Existing Code Refactoring: Improving the Design of Existing Code: Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Erich Gamma: 9780201485677: Amazon.com: Books