객체지향 프로그래밍

1. 객체

- 객체지향: 객체 내에 데이터와 프로시저(기능)가 들어 있는 형태
 ㄴ 다른 객체의 프로시저를 호출하여 사용한다.
 ㄴ 절차지향: 데이터를 여러 프로시저가 공유하는 방식
- 메시지: 객체와 객체 간의 상호작용을 가리킨다.
 ㄴ ex) 메서드 호출 메시지, 리턴 메시지, 익셉션 메시지 등
- 데이터를 저장하고 호출(get/set)하는 기능만 존재하는 클래스는 객체지향 프로그래밍에서 의미하는 '객체'보다는 데이터에 가까우며, 데이터 클래스라고도 한다.
 ㄴ Java는 구조체가 없어 클래스를 대신 사용하며, 구조체가 있는 언어도 참조를 사용하기 위해 데이터 클래스를 사용하기도 한다.

2. 캡슐화

- 데이터와 관련된 기능을 데이터와 묶는 것
- 외부에 영향 없이 객체 내부 구현을 변경할 수 있다.
- 정보 은닉: 기능의 구현을 외부에 감춘다.
 ㄴ 객체가 기능을 구현할 때 어떤 타입의 데이터를 어떻게 사용하는지 감춤

---

* 예시: 사용자 계정을 5년 이상 사용 시 1달간 프리미엄 혜택을 제공한다는 수정 요청
 - 캡슐화 X : 사용자 계정의 데이터에 접근하는 부분 전체에 대한 수정 필요

```
Before: if (account.GetState() == State.Premium &&
          account.GetExpireDate().IsAfterNow())
                  ... 프리미엄 회원 혜택 ...
After:   if((account.GetState() == State.Premium &&
          account.GetExpireDate().IsAfterNow()) ||
         (account.JoinDate().AddYears(5) > ServerTime.Now &&
         account ... ) ...
```

 - 캡슐화 O : 내부 데이터가 아닌 프리미엄 회원인지 확인하는 기능만 사용

```
Before: if (account.IsPremium) ... 프리미엄 회원 혜택 ...
After:   위와 동일
```

- 캡슐화를 시도하면서 기능에 대한 이해를 높일 수 있다.
 ㄴ 위 예시에서 State == Premium을 확인하는 이유는 프리미엄 권한을 확인하려는 의도

2.1. 캡슐화를 위한 규칙

(1) Tell, Don't Ask: 데이터를 달라고 하지 말고 해달라고 할 것
- 계정의 데이터를 가져와서 비교하는 대신, 프리미엄 유저인지 확인해달라고 한다.
(2) Demeter's Law(데미테르 법칙): 메서드에서 생성한 객체, 파라미터로 받은 객체, 필드로 참조하는 객체의 메서드만 호출할 것
- account.GetExpireDate().IsAfterNow() -> account.IsExpired()
- Date date = account.GetExpireDate(); date.IsAfter(ServerTime.Now);
-> account.IsValid(ServerTime.Now);

2.2. 캡슐화 예시

1) 로그인 인증 코드
- 유효성이 인증된 이메일은 숫자 2로 표시한다고 가정한다.

```
-----------------------------------------------------------------
public AuthResult Authenticate(string id, string pw) {
        Member mem = FineOne(id);
        if (mem == null)
                return AuthResult.NoMatch;
        // if (mem.GetVerificationEmailStatus() != 2) Tell, Don't Ask
        // 데이터를 가져와 비교하지 말고, 유효한지 확인해달라고 하기
        if (!mem.IsEmailVerified())
                return AuthResult.NoEmailVerified;
        if (passwordEncoder.IsPasswordValid(mem.Password, pw, mem.Id)
                return AuthResult.Success;
        return AuthResult.NoMatch;
}
-----------------------------------------------------------------
```

2) 이메일 유효성 검사

```
-----------------------------------------------------------------
public void VerifyEmail(string token) {
        Member mem = FindByToken(token);
        if(mem == null) throw new BadTokenException();

/*      데이터를 가져와서 변경하는 식의 동작은 해당 클래스로 이관한다.
        if(mem.GetVerificationEmailStatus() == 2)
                throw new AlreadyVerifiedException();
        else
                mem.SetVerificationEmailStatus(2);
*/
        mem.VerifyEmail();

        SaveOnDatabase();
```

```
}
```

---

3) 영화 대여 시 적립할 포인트 계산

```
public class Rental {
        private Movie movie;
        private int rentalDay;
        public int GetMembershipPoints() {
/*
                if (movie.PriceCode == Movie.NewRelease || rentalDay > 2)
                        return 150;
                else
                        return 100;
*/      // 계산에 필요한 데이터를 넘겨서 계산을 요청
                return movie.CalculatePoints(rentalDay);
        }
}
```

---

4) 시간 측정하기

```
public async void CalculateTime(Func<Task> func){
        Timer t = new Timer();
        // t.startTime = System.GetCurrentTime();
        t.Start();
        await func();

        // t.stopTime = System.GetCurrentTime();
        t.Stop();
        // Print(t.stopTime - t.StartTime);
        Print(t.ElapsedTime);
}
```

---

3. 추상화

- 추상화(Abstraction): 데이터나 프로세스 등을 의미가 비슷한 개념이나 의미 있는 표현으로 정의하는 과정

 1) 특정한 성질 추출: 사용자의 아이디, 이름, 이메일 -> User 클래스로 추상화

2) 공통 성질 일반화: GTX 1660, RX 5900 -> GPU 클래스로 추상화
ㄴ 서로 다른 구현에 공통된 특징이 있으면 상위 클래스로 추상화할 수 있다.

- 다형성(Polymorphism): 한 객체가 여러 타입을 갖는 것. 상속을 통해 구현된다.
ㄴ 구현 클래스들의 공통 성질을 인터페이스나 기반 클래스로 추상화한다.

- 장점: 유연한 변경이 가능하다.
ㄴ 추상 타입(기반 클래스, 인터페이스 등)을 사용하여 프로그래밍하면, 사용하는 곳의 로직을 수정하지 않고도 구체적인 동작의 변경이 가능하다.

- 단점: 프로그램 복잡도가 증가한다.
ㄴ 추상화를 하게 되면 추상 타입이 늘어나고, 그만큼 복잡도도 증가한다.
ㄴ 아직 존재하지 않는 기능에 대해 미리 추상화해두는 것은 좋지 않음
ㄴ 실제로 기존 기능의 변경/확장이 발생할 때, 변경 원인을 생각하여 추상화 시도할 것

ex) 주문 접수 후 문자 보내는 함수 -> 이메일 알림도 보낼 수 있게 해달라는 요구가 생기면 -> 변경 원인이 전송 방식의 다양화임을 생각 -> 문자 전송을 '알림 전송'으로 추상화하고, 여기서 문자/이메일/푸시알람 등을 구현

3.1. 추상화 예시 - 클라우드 파일 관리 기능

- 클라우드 종류: MYBOX, 드롭박스
- 기본적으로 아래 클래스 및 열거형이 선언되어 있다고 가정한다.

```
---------------------------------------------------------------
public enum Cloud { MyBox, Dropbox }
public class FileInfo {
        public Cloud CloudId { get; private set; }
        public string FileId { get; private set; }
        public string Name { get; private set; }
        public long Length { get; private set; }
}
그 외 각 클라우드별 내부 기능들을 지원하는 클래스들
---------------------------------------------------------------
```

- 추상화 미적용

```
---------------------------------------------------------------
// 파일 목록 조회 기능
public List<FileInfo> GetFileInfos(CloudId cloudId) {
        List<FileInfo> ret = new List<FileInfo>();
        if (cloudId == Cloud.Dropbox) {
                List<DropBoxData> datas = DropboxHelper.GetFiles();
                foreach (data in datas)
```

```
                        ret.Add(DropboxHelper.ToFileInfo(data));

                return ret;
        } else if (cloudId == Cloud.MyBox) { ... }
        //지원하는 클라우드가 늘어날 때마다 if문이 늘어난다.
}
// 다운로드, 검색, 업로드, 복사, 삭제 등 다른 기능도 비슷한 형태가 된다.
```

- 시간이 지날수록 코드 길이와 복잡도가 기하급수적으로 증가 -> 개발 시간 증가
 ㄴ 기능이 추가될 때마다 if ~ else if 나열이 추가된다.
 ㄴ 지원하는 클라우드가 늘어날 때마다 각 함수에 else if문을 중첩시켜야 한다.
 ㄴ 특히 클라우드간 복사 기능은 경우의 수를 클라우드 종류의 제곱만큼 고려해야 한다.

- 추상화 적용
- 아래 클래스를 설계해둔다.

```
interface ICloudSystem {
        // FileInfo는 IFileInfo 인터페이스를 상속받는다고 가정
        List<IFileInfo> GetFiles();
        List<IFileInfo> Search(string keyword);
        IFileInfo GetFile(string name);
        void AddFile(string name, IFileInfo fileInfo);
}
public static class CloudSystemFactory {
        static readonly Dictionary<Cloud, ICloudSystem> systems =
                new Dictionary<Cloud, ICloudSystem> {
                        Cloud.Dropbox, new DropboxSystem(),
                        Cloud.MyBox, new MyBoxSystem()
                };
        public static ICloudSystem Get(Cloud id) => systems[id];
}
```

```
// 파일 목록 조회 기능
public List<FileInfo> GetFileInfos(Cloud cloudId) {
        return CloudSystemFactory.Get(cloudId).GetFiles();
}
// 검색 기능
public List<FileInfo> Search(Cloud cloudId, string keyword) {
        return CloudSystemFactory.Get(cloudId).Search(keyword);
```

}
// 기타 추가되는 기능들도 유사한 방식으로 추가 가능

- 이 클래스의 수정 없이도 새로운 클라우드를 추가할 수 있다.
 ㄴ OCP(Open-Closed Principle)원칙: 확장엔 열려 있고, 수정엔 닫혀 있는 구조

4. 상속보단 조립

- 상속은 상위 클래스의 기능을 재사용/확장하는 방법으로 많이 활용되지만, 몇 가지 주의할 점이 있다.
 1) 상위 클래스 변경이 어려움: 변경이 모든 하위 클래스에 영향
 - 하위 클래스가 늘어날 때마다 상위 클래스의 캡슐화가 약해진다.
 2) 클래스 수 증가: 몇 가지 하위 클래스의 기능을 조합할 때, 상속으로 처리하면 조합의 수만큼 클래스 수가 증가
 3) 상속 오용: 상속받은 클래스의 사용법을 숙지하지 않으면 잘못 사용할 가능성이 커짐
 - 클라이언트에서 하위 클래스 사용 시, 상위 클래스의 기능들까지 노출된다.
 - 상위 클래스 기능을 오버라이드하는 과정에서 기대와 다른 동작이 발생할 수 있다.

- 조립은 기능을 재사용하고 싶은 클래스가 있을 때, 해당 클래스를 필요한 시점에 인스턴스화하여 사용한다.
 ㄴ ex) A클래스가 B클래스의 기능이 필요할 때, B클래스를 멤버 변수로 선언하여 사용

- 다른 클래스의 기능이 필요할 경우 조립으로 풀 수 없는지 검토해보고, 기능을 사용하는 클래스가 정말로 대상 클래스의 하위 타입일 경우에만 상속을 사용한다.

5. 기능과 책임 분리

- 하나의 기능은 여러 개의 하위 기능으로 분해할 수 있다.
 ㄴ ex) 암호 변경 기능 = 변경 대상 확인 + 대상 암호 변경
  ㄴ 변경 대상 확인 = 변경 대상 구하기 + 대상 없을 때 오류 안내
  ㄴ 대상 암호 변경 = 암호 일치여부 확인 + 암호 데이터 변경 + 불일치 시 안내
- 기능은 곧 책임
 -> 기능을 분리하고, 분리한 기능을 알맞게 분배해야 한다.
- 소스 코드가 방대할수록, 여러 기능이 한 클래스나 함수에 섞인다.
 ㄴ 변수를 많은 클래스나 메서드가 공유하게 된다.
 ㄴ 분리되지 않은 코드는 일부 기능만 테스트하기 어려워진다.
 -> 책임에 따라 코드 분리 필요

5.1. 코드 분리 방법

1) 패턴 적용: 역할 분리 패턴들을 사용하는 방식. GoF를 예로 들면 팩토리, 빌더, 스트레티지, 템플릿 메서드, 데코레이터 등이 있다.
2) 계산 분리: 함수가 길어질 경우 계산 부분을 별도의 함수나 클래스로 분리
3) 연동 분리: 네트워크, 메시지, 파일 등 외부와 연동하는 코드는 별도의 클래스로 분리
4) 조건 분기 추상화: 연속적인 if-else/switch 등 긴 분기문이 있을 때, 각 블록에서 하는 일이 비슷하다면 공통점을 이용해서 추상화할 수 있다.

- 코드 분리 시 의도가 잘 드러나는 이름을 사용해야 한다.
ㄴ ex) HTTP로 추천 데이터를 읽어오는 기능은, 'RecommendService' 'HttpDataService' 가 좋다.

6. 의존과 DI

- 의존: 기능 구현을 위해 다른 요소를 사용하는 것
ㄴ ex) 객체 생성, 함수 호출, 데이터 사용 등
- 의존 대상이 바뀌면 자신도 변경될 가능성이 높다.
ㄴ 클래스, 패키지, 모듈 등 모든 수준에서 순환 의존은 피하는 것이 좋다.
ㄴ 의존하는 대상이 많은 클래스(모듈)는 변경될 가능성도 그만큼 높다.

6.1. 의존 대상 줄이기

1) 클래스를 기능별로 분리
- 한 클래스가 다양한 기능을 제공하고 있을 경우, 기능별로 클래스를 분리할 수 있다.
ㄴ 개별 기능 테스트도 더 쉬워짐
2) 의존하는 대상들을 묶어서 추상화
- 의존 대상이 많을 때, 대상들 중 몇 가지를 묶어서 추상화하면 의존 대상이 줄어든다.
3) 의존 대상 객체 생성 추상화
- 대상 객체를 직접 생성하지 않고, 팩토리, 빌더, 의존 주입 등을 활용할 수 있다.

6.2. 의존 주입 (DI, Dependency Injection)

- 의존하는 대상을 직접 생성하는 대신, 생성자나 메서드를 통해 전달하는 방식

```
public class ScheduleService {
        private UserRepository repo;
        private Calculator cal;
        // 외부에서 객체 생성 후 생성자나 Setter로 전달한다.
        public ScheduleService(UserRepository repo) { this.repo = repo; }
        public SetCalculator(Calculator cal) { this.cal = cal; }
}
```

- 조립기(Assembler)가 객체를 생성해서 의존 주입을 처리한다.

```
public class CustomConfig {
        private ScheduleService schedule;

        public CustomConfig() {
                schedule = new ScheduleService(GetRepo());
                schedule.SetCalculator(GetCalc());
        }
        private UserRepository GetRepo() => new CustomUsetRepo();
        private Calculator GetCalc() => new Calculator();
        public GetSchedule() => schedule;
}
```

- 의존 대상이 바뀌면 조립기의 설정만 변경하면 된다.
 ㄴ 예를 들어 Config만 변경해서 Calculator -> CustomCalculator 로 변경 가능
 ㄴ 테스트를 위해 더미 클래스를 사용하게 할 때도 마찬가지