CS 기술 면접 대비

Library와 Framework를 설명

* Library와 Framework 모두 특정한 기능들을 모듈화한 미리 작성된 코드이다. flow의 제어권이 프로그래머에게 있는지 아니면 framework에 있는지로 Library와 Framework가 구별된다.

Singleton Pattern

* 클래스 하나에 오직 하나의 인스턴스를 가지도록 하는 패턴.
* 인스턴스를 마구 생성하는 것에 비해 여러 비용(메모리와 속도)측면에서 이점을 가질 수 있음
* 하나의 인스턴스이기에 객체 상태의 공유가 유의미한 환경에서 잘 사용된다.
* 다만 TDD의 걸림돌이 될 수 있다. 테스트는 서로 독립적이어야 하고 어떤 순서로든 실행될 수 있어야 하는 반면 싱글톤 패턴은 미리 생성된 하나의 인스턴스를 기반으로 동작하기에 독립적인 인스턴스를 생성하기 어려움.
* 하지만 싱글톤 내에 상태를 초기화하는 메서드를 하나 구현하면 독립적으로 사용할 수 있을 것.
* 또 다른 싱글톤 패턴의 단점은 모듈간의 결합을 강하게 만든다는 것. 이것은 의존성 주입을 통해 모듈간의 결합을 조금 더 느슨하게 만들어 해결할 수 있음

Simple Factory Pattern

* 객체의 생성 부분을 떼어내어 추상화한 패턴. (객체의 생성을 담당하는 Factory 클래스가 있다)
* 다음은 Simple Factory Pattern의 모습

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* SimplePhoneFactory에서 type을 따져 휴대폰의 객체를 직접 생성하는 것을 볼 수 있음
* 객체 생성의 세부 사항을 Phone을 구현하는 각 클래스에 두고 객체를 만드는 작업을 하나의 팩토리 클래스에 모아둔 모습을 볼 수 있음 (생성하는 부분만을 떼어내어 Factory 클래스에 두었다)

Factory Method Pattern

* 객체를 생성하기 위한 팩토리 인터페이스를 정의, 어떤 클래스의 인스턴스를 만들지는 서브클래스에서 결정하도록 하는 패턴
* 다음은 Factory Method Pattern의 모습

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* PhoneFactory를 상속, 혹은 구현한 서브 클래스에서 객체를 만들게끔 하는 것이 팩토리 메서드 패턴
* 구체적인 클래스 대신 인터페이스 메서드를 통해 phone을 생성 가능 유연하고 확장성 높다

Abstract Factory Pattern

* 구체적인 클래스에 의존하지 않고 서로 연관되거나 의존적인 객체들의 조합을 만드는 인터페이스를 제공
* 예로 IphoneFactory에서는 폰을 만들 때 createOS를 통해 IOS를 넣고 AndroidPhoneFactory에서는 createOS를 통해 AndroidOS를 넣으면 객체의 조합을 반영하여 생성 로직을 분리한 케이스가 된다.

Strategy Pattern

* 캡슐화된 알고리즘을 갈아낄 수 있도록 함으로써 코드의 동작을 다양화하는 패턴. 다형성을 이용해서 구현할 수 있다. OCP를 지킬 수 있음

의존성 주입

* 의존성 : A가 B에 의존성이 있다는 말은 B의 변경에 A가 반드시 변해야 함을 의미
* 의존성 만을 주입하는 중간자(Ioc Container)에 필요한 모듈들을 등록하고 의존성을 주입시키는 방법. (개발자가 직접 주입하는 것이 아니라 중간자가 개입함으로 제어의 역전이 일어남) 모듈간의 결합을 느슨히 하여 모듈간 전환을 부드럽게 하며 SRP OCP를 위배하지 않도록 도움을 준다.

Observer Pattern

* 관찰자가 객체의 상태변화를 관찰하고 상태가 변화할 때 마다 옵저버들에게 변화 상황을 알리는 패턴

자바 상속 vs 구현

* 상속: 자식 클래스가 부모 클래스의 속성과 메서드를 상속받아 추가 및 확장을 가능하도록 하는 기능. 재사용성, 중복성의 최소화를 통해 객체지향의 특징을 강화해준다.
* 구현: 부모 인터페이스를 자식 클래스에서 재정의하여 구현하는 것을 말함 상속과 달리 반드시 부모 클래스의 메서드를 재정의해야 함
* 자바에서는 diamond problem으로 다중 상속을 허용하지 않는다. 하지만 인터페이스는 다중 구현이 가능함 이것 또한 상속과 구현의 차이
* Diamond problem
  + 텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

    자동 생성된 설명
  + Child는 Father를 extends하고 Mather도 extends 다만 Person의 특정 메소드가 Father와 Mother에서 다르게 구현되어 있다면 child는 Person의 특정메소드를 실행할 때 Father 것으로? Mother의 것으로? -> diamond problem

프록시 패턴, 프록시 서버

* 프록시 패턴이란 대상 객체의 앞단에 흐름을 가로챌 수 있는 가상 객체를 두어 인터페이스 역할을 할 수 있도록 설계한 패턴을 말함. 이를 통해 프록시 서버를 만들 경우 보안적인 측면에서 공격의도가 있는 요청을 걸러낼 수도 있고 CDN등을 구축하여 서버 응답성을 높일 수도 있음 또한 모든 요청을 곧이곧대로 서버에 보내어 SPOF문제를 직면하기 보다 SPOF 문제를 해결할 수 있는 것임

DDOS 공격

* Distributed Denial Of Service: 공격대상이 되는 서버의 서비스 중단을 목적으로 표적 서버에 대량의 트래픽을 유발하는 공격 -> 프록시 패턴을 이용한 CloudFlare를 이용하면 사용자가 아닌 시스템으로부터 오는 트래픽을 자동으로 차단하여 공격을 쉽게 막아낼 수 있다 (프록시 패턴 이용)

CDN (Contend Delivery Network)

* 분산된 형태의 서버 네트워크를 말함 콘텐츠를 사용자에게 가까운 여러 프록시 서버에 데이터를 캐싱, 배포해놓음으로써 요청에 대한 응답성을 높일 수 있음(SPOF 문제 해결) 이러한 특징 때문에 주로 CDN은 한번 배포하면 수정이 적은 특징의 데이터를 위한 서버 네트워크를 구축할 때 사용이 됨. 영상 컨텐츠 등이 그러하기에 넷플릭스 왓챠 등 여러 OTT에서 CDN이 사용되는 것을 확인할 수 있음. CDN의 정책에는 여러가지가 있는데 사용자가 여러 서버노드들 중 핑을 날려보고 응답성이 가장 좋은 프록시 서버에게 요청하도록 하는 정책을 NETFLIX에서 사용한다고 분산시스템 과목에서 배운 기억이 있음

Iterator Pattern

* Iterator 라는 인터페이스로 컬렉션을 순회할 수 있도록 설계한 패턴. 자료형에 구애 받지 않고 하나의 인터페이스로 순회가능한 것이 특징

Reveal Module Pattern

* 자바스크립트와 같은 언어는 접근 제어자가 없음 이러한 언어에서 접근 제어자 역할을 할 수 있는 것을 만드는 패턴

MVC Pattern

* 애플리케이션 구성요소를 Model, View, Controller로 나눈 패턴. 이렇게 나눔으로써 애플리케이션 개발 프로세스에서 코드 재사용성과 확장성을 높일 수 있다.
* Model : 어플리케이션의 데이터에 해당하는 부분(데이터 베이스, 상수, 변수 등등)
* View : 유저가 마주하게 되는 인터페이스 요소들
* Controller : 한 개 이상의 View와 한 개 이상의 Model을 연결하는 중간 다리 역할을 하는 요소

MVP Pattern

* MVC Pattern의 Controller 요소가 Presenter로 바뀐 패턴. Presenter는 Controller와 다르게 뷰와 Presenter가 1:1로 강한 결합을 가진다는 특징이 있음

MVVM Pattern

* MVC Pattern의 Controller 요소가 View Model로 바뀐 형태의 Pattern View Model은 View의 추상화된 버전으로 커맨드와 데이터 바인딩을 가지는 것이 MVC Pattern과 구별되는 차이점. 뷰와 뷰모델 사이의 양방향 데이터 바인딩을 지원한다.

Programming Paradigm

* Declarative 선언적 언어 (함수형 언어)
  + 컴퓨터에게 목적을 알린다. ”프로그램은 함수로 이루어진 것이다”라는 명제를 깔고 감
  + 함수형 언어에선 함수가 일급 객체다.
    - 일급 객체
      * 함수에 인자로 넘기기 가능
      * 변수나 메서드에 할당 가능
      * 함수가 반환 가능
      * 위 3가지 조건을 만족하는 객체는 일급 객체
  + 즉 함수형 언어에서 함수는 변수나 메서드에 할당 가능하고 함수가 함수를 반환가능하며 함수에 함수를 인자로 넘길 수 있다.
* Imperative 명령형 언어
  + 컴퓨터에게 알고리즘, 즉 어떻게 해야 할 지를 알린다. 변수의 값을 계속 수정하는 데에서 함수의 결과가 계속 바뀔 수 있는 Side Effect가 발생하는 것이 특징. 명령형 언어는 이러한 side effects를 십분 활용하는 패러다임이라고 볼 수도 있음
  + 하지만 요새는 람다와 같은 것들이 도입되면서 함수형 언어와 명령형 언어를 나누는 clear cut이 없음
* 절차형 언어
  + 수행되어야 할 연속적인 계산 과정으로 이루어짐
  + 일이 진행되는 방식으로 그저 코드를 구현하기만 하면 되기 때문에 코드의 가독성이 좋으며 실행 속도가 빠름
* 객체지향형 언어
  + 객체의 집합으로 프로그램의 상호작용을 표현하는 프로그래밍 패러다임
  + 데이터를 객체로 취급하여 객체 내부에 선언된 메서드를 활용하는 방식
  + 설계에 많은 시간이 소요되며 처리속도가 다른 프로그래밍 패러다임에 비해 느리지만 코드의 재사용성을 올려주어 개발 생산성을 많이 높임 (SW crisis를 객체지향으로 극복했다)
  + 객체지향 프로그래밍의 특징
    - 추상화 : 핵심적인 개념 또는 기능들을 간추리는 것을 의미
    - 캡슐화 : 객체의 속성과 메서드를 하나로 묶고 일부를 외부에 감추어 은닉하는 것
    - 상속성 : 상위 클래스의 특성을 하위 클래스가 이어 받아서 재사용하거나 추가 확장하는 것을 의미
    - 다형성 : 하나의 메서드나 클래스가 다양한 모습으로 동작하는 것을 의미 대표적으로 오버로딩과 오버라이딩이 다형성의 예이다.
      * 오버로딩: 같은 이름을 가진 메서드가 여럿 있을 수 있는 것을 의미. 메서드의 타입, 매개 변수 유형, 개수 등의 차이를 두며 여러 개를 둘 수 있으며 컴파일 중에 발생하는 정적 다형성
      * 오버라이딩: 상위 클래스의 메서드를 하위 클래스가 재정의하는 것을 의미 runtime 중에 발생하는 동적 다형성
  + 객체지향 프로그래밍의 설계 원칙
    - SRP (Single Responsibility Principle)
      * 모든 클래스는 각각 하나의 책임만을 가져야 하는 원칙
    - OCP (Open Closed Principle)
      * 확장에는 열려있고 변화에는 닫혀있어야 하는 원칙 즉, 기존의 코드는 잘 수정하지 않으면서 확장은 쉬워야 한다는 것
    - LSP (Liskov Substitution Principle)
      * 프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 하는 것을 의미. 부모 객체에 자식 객체를 넣어도 시스템이 문제없이 돌아가게 만드는 것을 뜻한다.
    - ISP (Interface Segregation Principle)
      * 하나의 일반적인 인터페이스보다 구체적인 여러 개의 인터페이스를 만들어야 하는 원칙
    - DIP (Dependency Inversion Principle)
      * 자신보다 변하기 쉬운 것에 의존하던 것을 추상화된 인터페이스나 상위 클래스를 두어 변하기 쉬운 것의 변화에 영향 받지 않게 하는 원칙, 즉 상위 계층은 하위 계층의 변화에 대한 수현으로부터 독립해야 한다. 추상화에 의존해야지 구체화에 의존하면 안된다!