Session 020 소프트웨어 아키텍처

1 아키텍쳐의 설계

1.1 소프트웨어의 기본 구소, 설계도

- 소프트웨어 개발 시 원칙과 지침, 의사소통 도구
- 기능적 요구사항 구현 방법 찾고, 비기능적 요구사항으로 나타난 제약 반영
- 기본 원리
  - ■ 모듈화
  - ■ 추상화
  - ■ 단계적 분해
  - ■ 정보 은닉

2 모듈화(Modularity)

2.1 성능 향상, 수정 및 재사용, 유지 관리가 용이하도록 모듈 단위로 나누는 것

- 자주 사용되는 기능들을 공통 모듈로 구성
- 모듈 크기 너무 작으면 개수 많아짐 통합 비용 증가
- 모듈 크기 너무 크면 하나의 개발 비용 증가

3 추상화(Abstraction)

3.1 포괄적인 개념 설계 후 차례로 세분화하여 구체화

- 유사한 모델을 제작 여러가지 요인들을 테스트 가능
- 최소의 비용으로 실제 상황에 대처, 시스템 구조, 구성을 대략적으로 파악

3.2 유형

- 과정 추상화 : 전반적인 흐름만 파악
- 데이터 추상화 : 데이터 구조를 대표할 수 있는 표현으로 대체
- 제어 추상화 : 이벤트 발생에 대해 대표할 수 있는 표현으로 대체

4 단계적 분해(Stepwise Refinement)

4.1 하향식 설계 전략, 상위 -> 하위 개념으로 구체화

- 추상화의 반복에 의해 세분화
- 기능 -> 알고리즘, 자료구조 등

5 정보 은닉

5.1 다른 모듈이 접근, 변경 못하도록

- 필요한 정보가 있을 때는 인터페이스를 통해 접근
- 독립적 수행 가능, 다른 모듈에 영향X, 수정, 시험, 유지보수 용이

6 아키텍처의 품질 속성

6.1 이해 관계자들의 요구사항에 맞게 설계 되었는지를 확인 위해 평가 요소들을 구체화

6.2 시스템 측면

- 성능 : 이벤트 발생시, 적절하고 빠르게
- 보안 : 허용되지 않은 접근 막고, 허용된 것에는 적절한 서비스 제공

- 가용성 : 장애 없이 정상적으로 서비스 제공
- 기능성 : 요구 기능을 만족스럽게 구현
- 사용성 : 명확하고 편리하게
- 변경 용이성 : 다른 플랫폼에서도 동작할 수 있도록
- 확장성 : 확장시켰을 때 효과적으로 활용할 수 있도록
- 테스트 용이성
- 배치성
- 안정성

6.3 비즈니스 측면

- 시장 적시성 : 정해진 시간에 릴리즈
- 비용과 혜택
- 예상 시스템 수명 : 얼마나 오랫동안, 변경 용이성, 확장성과 연관
- 목표 시장
- 공개 일정
- 기존 시스템과의 통합

6.4 아키텍처 측면

- 개념적 무결성 : 시스템과 구성요소들 간 일관성 유지
- 정확성, 완결성 : 요구사항의 제약사항들을 모두 충족시키나
- 구축 가능성 : 모듈을 적절하게 분배 유연하게 일정을 변경할 수 있도록
- 변경성
- 시험성
- 적응성
- 일치성
- 대체성

7 설계 과정

- 설계 목표 설정 : 요구사항을 분석, 시스템 설계 목표를 성
- 시스템 타입 결정
  - ■ 대화형 시스템 : 사용자의 요청이 발생하면 이를 처리 하는 시스템
    - ◆ 온라인 쇼핑몰 등 대부분의 웹 애플리케이션
  - ■ 이벤트 중심 시스템 : 외부 상태 변화에 따라 동작
    - ◆ 전화, 비상벨 등 내장 소프트웨어
  - ■ 변환형 시스템 : 데이터 입력되면 정해진 작업들을 수행, 결과 출력
    - ◆ 컴파일러, 네트워크 프로토콜 등
  - ■ 객체 영속형 시스템 : 디비를 사용, 파일을 효과적으로 저장, 검색, 갱신 할 수 있는 시스템
    - ◆ 서버 관리 소프트웨어
- 아키텍처 패턴 적용
- 서브시스템 구체화 : 서브시스템 기능 및 서브시스템 간의 상호작용을 위한 동작과 인터페이스 정의
- 검토

Session 021 아키텍처 패턴

1 아키텍쳐 패턴의 개요

1.1 설계할 때 참조할 수 있는 전형적인 해결 방식

- 시스템 구조의 기본적인 윤곽을 제시
- 패턴에는 서브시스템, 그 역할 정의, 서브시스템 간의 관계, 규칙, 지침 등 포함(인터페이스)
- 장점
  - ■ 시행착오 줄여 개발 시간 단축, 고품질 소프트웨어
  - ■ 검증된 구조 이용, 안정적인 개발 가능
  - ■ 공통된 아키텍처 공유, 의사소통이 간편
  - ■ 구조 이해 쉬워, 개발 미 참여자도 유지보수 손쉽게 수행
  - ■ 개발 전에 예측 가능
- 패턴 종류 : 레이어, 클라이언트-서버, 파이프-필터. 모델-뷰-컨트롤러

2 레이어 패턴(Layer Pattern)

2.1 시스템을 계층(Layer)로 구분

- 서브시스템들이 계층 구조, 상위 계층은 서비스 제공자, 하위 계층은 클라이언트
- 서로 마주보는 두 개의 계층 사이만 상호작용, 변경 작업 용이
- 특정 계층만을 교체해 시스템 개선

3 클라이언트-서버 패턴(Client-Server Pattern)

3.1 하나의 서버 컴포넌트 & 다수의 클라이언트 컴포넌트 구성

- 사용자는 클라이언트와만 의사소통. 사용자는 클라이언트 통해 서버에 요청, 클라이언트가 서버로부터 응답을 받아 사용자에게 제공
- 서버는 항상 대기 상태
- 요청과 응답을 받기 위해 동기화되는 경우 제외하고는 서로 독립적

4 파이프-필터 패턴(Pipe-Filter Pattern)

4.1 데이터 스트림의 각 단계를 필터 컴포넌트 캡슐화 하여 파이프를 통해 데이터 전송

- 필터는 재사용성, 확장이 용이
- 필터를 재배치하여 다양한 파이프라인 구축 가능
- 파이프-필터 패턴은 데이터 변환, 버퍼링, 동기화 등에 사용
- 하나의 컴포넌트에서 처리가 끝나면 다음이 결과물을 받아 처리

5 모델-뷰-컨트롤러 패턴(Model-View-Controller Pattern)

- Model : 서브시스템의 핵심 기능과 데이터를 보관
- View : 사용자에게 정보 표시
- Controller : 사용자로부터 받은 입력 처리
- 각 부분은 서로 분리, 영향을 받지 않고 개발 작업을 수행
- 대화형 애플리케이션에 적합

6 기타 패턴
- 마스터-슬레이브 패턴
- 브로커 패턴
- 피어-투-피어 패턴
- 이벤트-버스 패턴
- 블랙보드 패턴
- 인터프리터 패턴

Session 022 객체지향(Obejct-Oriented)

1 객체지향의 개요
1.1 객체들을 조립해서 작성할 수 있는 기법
- 재사용 및 확장, 유지보수 용이
- 구조를 단계적 계층적으로 표현, 멀티미디어 데이터 및 병렬 처리

2 객체(Object)
2.1 데이터와 함수를 묶어 놓은 하나의 모듈
2.2 객체의 특성
- 식별 가능한 이름 가지고 있음
- 상태는 시간에 따라 변함
- 개체와 객체는 상호 연관성에 의해 관계 형성
- 메시지의 집합을 행위, 객체는 행위 특징을 나타냄
- 기억장소를 가지고 있음

3 클래스(Class)
3.1 공통된 속성과 연산을 갖는 객체의 집합. 객체의 일반적인 타입
- 각 객체들이 갖는 속성과 연산을 정의하는 틀
- Instance, Instantialtion
- 동일 클래스의 각 객체는 공통된 속성과 행위를 가지고 있지만, 각 각의 정보가 서로 달라서 여러 객체를 나타낸다.

4 캡슐화(Encapsulation)
4.1 데이터와 함수를 하나로 묶는 것
- 정보 은닉, 외부 모듈의 변경으로 인한 파급 효과 적음
- 재사용이 용이
- 인터페이스 단순, 객체 간 결합도 낮아짐

5 상속(Inheritance)
5.1 상위 클래스의 모든 속성과 연산을 하위 클래스가 물려받는것
- 하위 클래스는 다시 정의하지 않고도 자신의 속성으로 사용가능
- 새로운 속성과 연산 첨가 가능
- 재사용성을 높이는 중요한 개념
- 다중상속

6 다형성(Polymorphism)

6.1 하나의 메시지에 대해 여러가지 형태의 응답이 있다.

Session 023 모듈

1 모듈의 개요

1.1 분리된 시스템의 각 기능, 서브루틴, 서브시스템, 소프트웨어 내의 프로그램, 작업 단위 등과 같은 의미로 사용

- 단독으로 컴파일 가능, 재사용 가능
- 기능적 독립성은 각 모듈의 기능이 서로 독립됨
- 독립성이 높을 수록 다른 모듈에게는 영향 x, 오류 발생해도 쉽게 발견 및 해결
- 독립성은 결합도, 응집도로 측정. 독립성을 높이려면 결합도는 약하게, 응집도는 강하게 모듈의 크기는 작게

2 결합도

2.1 모듈 간에 상호 의존 정도

- 결합도가 약할 수록 고품질
- 강하면 구현 및 유지보수 어려움
- 자료 < 스탬프 < 제어 < 외부 < 공통 < 내용
- 약함----------강함

3 응집도

3.1 정보은닉 개념 확장, 독립적인 기능으로 정의되어 있는 정도

- 응집도가 강할수록 고품질
- 기능적 > 순차적 > 교환적 > 절차적 > 시간적 > 논리적 > 우연적
- 강함----------약함

4 팬인/팬아웃

- 팬인 어떤 모듈을 제어(호출)하는 모듈의 수
- 팬아웃 어떤 모듈에 의해 제어(호출)되는 모듈의 수
- 모듈에 들어오면 : 팬인
- 모듈에서 나가면 : 팬아웃
- 팬인이 높다는 것은 재사용측면에서 설계가 잘 되어있으나, 단일 장애점이 발생할 수 있다.(단일 장애점 : 시스템의 구성요소 중 동작하지 않으면, 전체 시스템 중단되어 버리는 요소)
- 팬아웃이 높은 경우 불필요하게 다른 모듈을 호출하고 있는지 검토필요
- 시스템 복잡도 최적화 -> 팬인 높게, 팬아웃 낮게

Session25 코드

1 코드(Code)의 개요

1.1 분류, 조합 및 집계를 용이, 특정 자료 추출 쉽게

- 정보를 신속, 정확, 명료하게 전달
- 일정한 규칙, 정보 처리의 효율과 처리된 정보의 가치에 많은 영향
- 식별 기능, 분류 기능, 배열 기능
  - ■ 식별 : 데이터간의 성격에 따라 구분
  - ■ 분류 : 특정 기준이나 동일한 유형에 해당하는 데이터 그룹화
  - ■ 배열 : 의미 부여하여 나열

2 코드의 종류

- 순차 코드(Sequence Code) : 일정 기준에 따라서 최초의 자료부터 차례로
- 블록 코드(Block Code) : 공통성이 있는 것끼리 구분, 각 블록 내에서 일련번호
- 10진 코드(Decimal Code) : 대상 항목 10진 분할, 다시 각각에 대해 10진 분할
- 그룹 분류 코드(Group Classification Code) : 대분류, 중분류, 소분류
- 연상 코드(Mnemonic Code) : 명칭, 약호와 관계 있는 숫자, 문자, 기호를 이용
- 표의 숫자 코드(Significant Digit Code) : 길이, 넓이, 부피, 지름, 높이 등 물리적 수치 그대로 코드에
- 합성 코드(Combined Code) : 2개 이상의 코드 조합

3 코드 부여 체계

3.1 이름만으로 개체의 용도와 적용 범위를 알 수 있도록 코드를 부여

- 유일한 코드를 부여, 식별 및 추출 용이하게
- 각 단위 시스템의 고유한 코드와 개체를 나타내는 코드 등이 정의 되어야
- 자릿수, 구조 등을 상세하게 명시

Session26 디자인 패턴

1 디자인 패턴(Design Pattern)의 개요

1.1 세부적인 구현 방안을 설계할 때 참조할 수 있는 전형적인 해결 방식 또는 예제

- 문제에 해당하는 디자인 패턴 참고하여 적용
- 한 패턴에 변형을 가하거나 특정 요구사항을 반영하면 다른 패턴으로 변화
- GoF 디자인 패턴 가장 일반적
- 생성 패턴 5개, 구조 패턴 7개, 행위 패턴 11개

2 생성 패턴

2.1 객체의 생성과 참조 과정을 캡슐화, 객체가 생성 또는 변경 되어도 구조에 영향을 크

게 받지 않도록 유연성 향상

- 추상 팩토리(Abstract Factory)
  - ■ 서로 연관 의존하는 객체들을 인터페이스를 통해 그룹화하여 추상적으로 표현
  - ■ 한 번에 교체하는 것 가능
- 빌더(Builder)
  - ■ 객체의 생성과 표현 방법을 분리, 동일한 객체 생성에서도 다른 결과를 만들어 낼 수 있다
- 팩토리 메소드(Factory Method)
  - ■ 객체 생성을 서브 클래스에서 처리, 캡슐화한 패턴
  - ■ 상위 클래스는 인터페이스만, 생성은 서브 클래스
- 프로토타입(Prototype)
  - ■ 원본 객체를 복제
- 싱글톤(Singleton)
  - ■ 생성된 객체를 어디서든 참조 하지만 여러 프로세스가 동시에 참조 불가
  - ■ 불필요한 메모리 낭비 최소화

3 구조 패턴

3.1 클래스나 객체들을 조합, 더 큰 구조로 만들 수 있게

- 어댑터(Adapter)
  - ■ 호환성이 없는 클래스들의 인터페이스들 다른 클래스가 이용할 수 있도록 변환
  - ■ 인터페이스가 일치하지 않을 때
- 브리지(Bridge)
  - ■ 구현, 추상으로 분리, 독립적으로 확장할 수 있도록
  - ■ 기능과 구현 두 개의 별도 클래스로 구현
- 컴포지트(Composite)
  - ■ 여러 객체를 가진 복합 객체와 단일 객체를 구분 없이 다루려 할 때
  - ■ 트리 구조로 구성
- 데코레이터(Decorator)
  - ■ 객체 간의 결합을 통해 능동적으로 기능 확장
  - ■ 다른 객체들을 덧붙이는 방식
- 퍼싸드(Facade)
  - ■ 복잡한 서브 클래스들을 피해 더 상위에 인터페이스 구성, 서브 클래스의 기능
  - ■ 서브 클래스들 간의 통합 인터페이스를 제공하는 Wrapper 객체 필요
- 플라이웨이트(Flyweight)
  - ■ 인스턴스가 필요할 때 마다 매번 생성이 아닌, 가능한 공유해서 메모리를 절약
  - ■ 다수의 유사 객체를 생성 및 조작할 때 유용

- 프록시(Proxy)
  - ■ 접근이 어려운 객체와 연결하려는 객체 사이에서 인터페이스 역할
  - ■ 네트워크 연결, 메모리의 대용량 객체로의 접근

4 행위 패턴

4.1 클래스나 객체들이 상호작용하는 방법이나 책임 분배 방법

4.2 하나의 객체로 수행할 수 없는 작업을 여러 객체로 분배해 결합도를 최소화

- 책임 연쇄(Chain of Responsibility)
  - ■ 한 객체가 요청을 처리하지 못하면 다음 객체로 넘어가는 형태
  - ■ 고리를 따라 책임이 넘어간다
- 커맨드(Command)
  - ■