문제 1 : 아직 따라가지 못하고 힘들지만 강사님이 상담해주신 대로 복습해서 따라가보도록 하겠습니다,

문제 2 : 객체 지향에 기반한 재사용성,확장성, 그리고 유지 보수의 편리함

문제 3 :

3-1 : 코드품질향상 - TDD는 코드를 철저히 테스트하고 오류 및 버그를 확인한다. 반복적인 테스트가 더 쉬워지므로 간단하고 유지 관리 가능한 코드를 작성할 수 있게 된다. 결과적으로 유지 관리와 확장이 용이한 고품질의 코드를 생산할 수 있게된다.

3-2 : 빠른 개발 - 장기적으로 보았을 때 개발 속도를 높이게 되는데 테스를 먼저 작성함으로써 개발자는 코드를 고도화 함과 동시에 테스트를 실행함으로써 오류를 조기에 잡아낼 수 있다. 따라서 코드에 오류가 있다면 초기에 수정 작업을 할 수 있다.

3-3 : 협업에 용이 - TDD로 개발을 할 경우 개발 단계에서도 요구사항에 대한 테스트 코드가 존재하므로 개발에 참여한 모든 구성원이 해당 코드에 대해 더 잘 이해할 수 있다.

문제 4 : DDD의 주요 목표는 도메인 전문가와 소프트웨어 개발자 사이의 커뮤니케이션을 향상시킴으로써 복잡한 소프트웨어를 개발하는데 초점을 맞춥니다. 하지만 TDD는 소프트웨어 품질을 향상시키는데 초점을 맞추고 있으며 서로 다른 초점과 목표를 가지고 있지만 같이 사용함으로써 서로를 보완하며 소프트웨어 개발의 효율성과 품질을 동시에 향상 시킬수 있습니다.

문제 5 :

5-1 : 팀원 간의 커뮤니케이션 - 팀원 간의 활발하고 명확한 커뮤니케이션은 필수적이라고 생각합니다. 기술적인 문제뿐만 아니라, 프로젝트의 목표, 일정, 역할 분배 등 모든 측면에서 적용되는 것이고 커뮤니케이션은 서로의 이해를 향상시키고 오해를 방지하며 팀의 일관성을 유지하는 데 도움이 됩니다.

5-2 : 코드 관리 - 소스 코드의 버전 관리는 팀원 간의 작업을 조율하고, 코드 변경 사항을 추적하고, 문제가 발생했을 때 이를 빠르게 해결하는 데 필수적입니다. Git과 같은 버전관리 시스템을 사용하여 코드의 일관성을 유지하고, 팀원 간의 작업 충돌을 최소화하는 것이 중요합니다.

5-3 : 코드 리뷰 - 코드 리뷰는 코드의 품질을 높이는 데 중요한 역할을 합니다. 팀원들이 서로의 코드를 검토하면서 버그를 발견하고, 코드의 가독성과 유지 보수성을 향상시킬수 있습니다.

문제 6 : IoC 관점에서 함수 포인터 테이블을 사용하는 이유는 프로그램의 유연성과 확장성을 향상시키기 위함입니다.

문제 7 : 다형성 구현 - 상위 클래스의 포인터나 참조를 통 하위 클래스의 메소드를 호출할 수 있습니다. 이를 통해 런타임에 객체의 실제 타입에 따라 다른 메소드를 실행할 수 있습니다.

문제 8 : 문제에 대한 원인과 영향에 대해서 파악한 후 문제의 심각성을 파악합니다. 그 문제를 나 혼자 해결할 수 있는가를 판단합니다. 그리고 난 후 혼자 해결할 수 없다는 판단이 된다면 다른 팀원이나 관련 전문가에게 도움을 청합니다. 만약 혼자 해결할 수 있다는 판단이 된다면 문제에 대한 가능한 모든 정보를 수집합니다. 그리고 가능한 모든 해결책을 탐색한 후 가장 효과적이고 현실적인 해결책을 선택하고 이를 구현합니다. 하지만 이 과정에서 도움을 청할 수 있고 여러번의 실패를 겪을 수 있습니다. 해결책을 구현하게 된다면 문제와 해결 과정을 문서화합니다. 이를 통해 비슷한 문제가 다시 발생했을 때 빠르게 대응할 수 있도록 합니다.

문제 9 :

9-1 : 일정 관리 - Backlog는 개발해야 할 기능들을 정리하고 우선순위를 정하는 데 도움이 됩니다. 이를 통해 프로젝트 일정을 관리하고 효율적으로 작업을 할당할 수 있습니다.

9-2 : 목표 설정 - Backlog에는 프로젝트의 목표와 이를 달성하기 위한 세부적인 작업들이 포함됩니다. 이를 통해 프로젝트의 전반적인 목표를 이해하고 어떤 작업을 해야 하는지 명확하게 알 수 있습니다.

9-3 : 커뮤니케이션 향상 - Backlog는 프로젝트의 상태를 공유하고 팀원들 간의 커뮤니케이션을 촉진하는 데 도움이 됩니다. 이를 통해 작업의 중복을 방지하고 프로젝트의 진행 상황을 파악할 수 있습니다.

문제 10 :

10-1 : 명확한 통신 부재 - 팀원들 간의 커뮤니케이션과 이해가 부족할 경우 각자의 작업물이 다른 사람의 작업물과 잘 맞지 않을 수 있습니다.

10-2 : 지속적인 통합 부재 - 각자의 작업물을 주기적으로 통합하고 이를 테스트하여 문제를 조기에 발견하고 해결하는 것이 중요합니다. 이 과정이 부재할 경우 문제가 누적되어 결국 큰 문제로 이어질 수 있습니다.

문제 11 :

11-1 : 우선순위 결정 어려움 - 기능 단위로만 Backlog를 작성하면 어떤 기능이 더 중요한지 혹은 먼저 개발되어야 하는지 결정하기 어려울 수 있습니다.

11-2 : 유지보수 어려움 - 각 기능이 독립적으로 개발되면 이들 간의 상호작용이나 의존성을 파악하기 어려워 유지보수가 어려울 수 있습니다.

11-3 : 전체 시스템의 흐름 무시 - 개별 기능에만 초점을 맞추다 보면 전체 시스템의 흐름이나 아키텍처를 놓칠 수 있습니다. 이로 인해 각 기능 간의 연결성이 떨어지거나 시스템 전체의 일관성이 떨어질 수 있습니다.

문제 12 ~ 20 :
https://github.com/ajun984/SDC-Comprehensive-Evaluation/tree/main/1%EA%B8%B0/first/junyeonkim/comp12

문제 22 :

22-1 : 동적 메모리 할당 - 포인터를 사용하면 프로그램 실행 중에 필요한 만큼의 메모리를 동적으로 할당하거나 해제할 수 있습니다. 이는 프로그램의 유연성을 높이고 메모리 사용을 최적화하는데 도움이 됩니다.

22-2 : 데이터 구조구현 - 포인터는 복잡한 데이터구조를 구현하는 데 필요합니다. 이런 데이터 구조는 데이터 간의 관계를 표현하고, 효율적인 데이터 관리를 가능하게 합니다.

22-3 : 다형성 구현 - C++과 같은 언어에서는 포인터를 이용해 상속과 가상 함수를 통한 다형성을 구현할 수 있습니다. 이를 통해 코드의 재사용성을 높이고 유연한 프로그래밍을 가능하게 합니다.

문제 23 :

23-1 : 독립성 유지 - 각 함수는 자신만의 변수와 연산을 관리하는 독립적인 공간이 필요합니다. 만약 함수들이 스택을 공유한다면, 한 함수에서의 작업이 다른 함수의 작업에 영향을 미칠 수 있어 예측하지 못한 결과나 버그를 초래할 수 있습니다.

23-2 : 실행 컨텍스트 유지 - 함수가 호출될 때마다 해당 함수의 실행 컨텍스트(로컬 변수, 매개변수, 복귀 주소 등)는 스택에 저장됩니다. 함수의 실행이 완료되면 그 정보는 스택에서 제거됩니다. 이렇게 각 함수가 독립적인 스택을 가지게 되면, 함수의 호출과 종료가 명확하게 관리됩니다.

따라서 함수들이 각자의 스택 공간을 사용하는 것은 프로그램의 안정성, 예측 가능성, 그리고 메모리 관리 효율성을 위해 필요합니다.

문제 33 :

33-1 : 도메인 분리 - 복잡도가 높아진 멤버 변수와 관련된 로직을 새로운 클래스나 도메인 모델로 분리합니다. 이렇게 하면 관련 로직을 더 효과적으로 관리하고 테스트할 수 있습니다.

33-1 : 캡슐화 - 새로운 도메인 모델 내에 로직을 캡슐화하여 해당 모델과 관련된 연산을 모델 자체가 수행하도록 합니다. 이렇게 하면 코드의 응집력이 높아지고 다른 부분에서 이 로직을 재사용하기가 더 쉬워집니다.

33-2 : 리팩토링 - 기존 코드를 수정하여 새로운 도메인 모델과 잘 연동되도록 만듭니다. 이 과정에서 기존 코드의 버그를 발견하고 개선할 수도 있습니다.

33-3 : 테스트 - 새로운 도메인 모델과 관련된 테스트 케이스를 작성하여 모델이 올바르게 동작하는지 확인합니다. 이는 새로운 버그를 방지하고 코드의 안정성을 높이는 데 도움이 됩니다.

이러한 작업은 초기에는 시간과 노력이 필요하지만 시스템의 복잡도가 증가함에 따라 이를 관리하는 데 큰 도움이 됩니다. 이는 코드의 가독성, 유지 관리성, 그리고 확장성을 향상시키는 데 기여합니다.

문제 34 ~ 36 :
https://github.com/ajun984/SDC-Comprehensive-Evaluation/tree/main/1%EA%B8%B0/first/junyeonkim/comp34