소프트웨어 디자인 패턴 중간고사

20190839 경제학과 이승민

1. [10점] Agile Manifesto의 기본 정신은 무엇이고, 이것이 소프트웨어 개발에는 어떤 영향을 끼쳤는지 설명해 보라. 또, S/W quality 와는 어떤 관계가 있는가? 설명하라.

Agile Manifesto의 기본 정신은 소프트웨어 개발에서 점점 복잡해지고 경직된 프로세스를 해결하고자 하는 목적에서 시작되었다. Agile Manifesto는 소프트웨어 팀이 더 빠르고 유연하게 작업할 수 있도록 하는 가치관과 원칙을 제시하며, 그 핵심 정신은 다음 네 가지로 요약된다.

1) Individuals and interactions over processes and tools : 성공적인 소프트웨어 개발의 핵심은 팀원 간의 협력과 소통이다. 좋은 프로세스와 도구가 중요한 역할을 하더라도, 팀 구성원의 능력과 상호작용이 더욱 중요하다. 소통이 잘 이루어지는 팀이 그렇지 않은 팀보다 성공 가능성이 높다.

2) Working software over comprehensive documentation : 문서화는 시스템의 구조와 설계 의도를 설명하는 데 필요하지만, 과도한 문서는 관리에 어려움을 준다. Agile은 고객에게 가치 있는 동작하는 소프트웨어를 우선시하며, 필요한 문서는 최소한으로 유지한다.

3) Customer collaboration over contract negotiation : 성공적인 프로젝트를 위해 고객과의 지속적이고 빈번한 협력이 필요하다. 계약서에 의존하기보다는, 고객의 피드백을 기반으로 소프트웨어를 발전시키는 것이 중요하다. 이를 통해 고객의 요구사항 변경에도 신속히 대응할 수 있다.

4) Responding to change over following a plan : 소프트웨어 개발 환경은 빠르게 변화하며, 고객의 요구도 변할 수 있다. Agile은 이러한 변화를 수용하고, 유연하게 대응하는 것을 목표로 한다. 고정된 계획에 얽매이기보다는 짧은 주기의 계획을 세우고, 이를 지속적으로 수정해 나가는 것이 더 효과적이다.

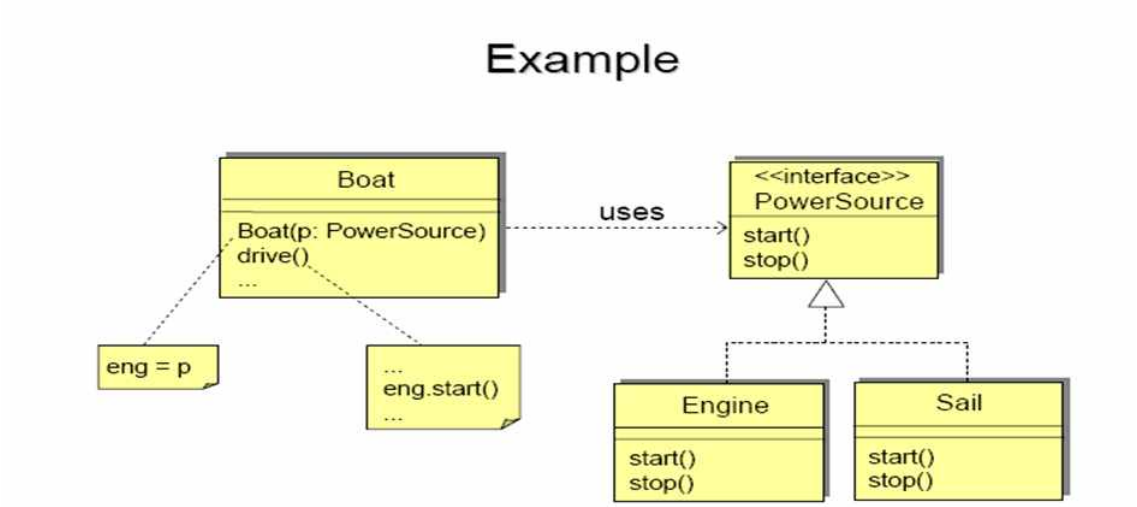
Agile이 소프트웨어 개발에 미친 영향은 크게 두 가지로 요약할 수 있다. 첫째, 고객 중심의 개발. Agile은 고객의 요구와 피드백을 정기적으로 반영하여 개발 과정에서의 오류와 비효율을 최소화한다. 둘째, 지속적인 개선과 빠른 배포를 통해 소프트웨어 개발 주기를 단축했다. 짧은 반복 주기와 자주 공개되는 배포는 개발 팀이 고객의 요구에 신속하게 대응할 수 있게 하여, 더 높은 고객 만족도를 이끌어낸다.

S/W Quality 측면에서 Agile은 중요한 의미를 가진다. 품질에 대한 정의는 크게 Conformance to specification와 Meeting customer needs으로 나뉘는데, Agile은 이 두 가지를 모두 충족한다. 지속적인 피드백과 짧은 반복 주기를 통해 고객의 요구를 충족하면서, 동시에 소프트웨어의 Maintainability, Usability, Reliability와 같은 품질 속성을 높이는 데 중점을 둔다. TDD와 CI를 통해 사전에 결함을 방지하고, 지속적인 피드백 루프를 통해 개선을 반복적으로 수행하여 품질을 관리한다. TDD는 기능을 구현하기 전에 테스트를 작성함으로써 오류를 사전에 예방하고, CI는 코드 변경 사항을 지속적으로 통합하여 문제를 조기에 발견하고 해결할 수 있게 한다. 이를 통해 품질 관리가 지속적으로 이루어지며, 고객의 요구에 신속히 대응할 수 있다.

즉, Agile Manifesto는 개발의 복잡성을 줄이고 개발팀의 효율성을 제고할 수 있는 가치관을 제시한다.

3. [10점] Dependence Management는 어떤 문제를 어떻게 풀겠다는 것인가? 또한 이들은 S/W quality와 어떤 관계가 있는가? 예를 들어 설명하라.

Dependence Management는 객체지향 프로그래밍(OOP)에서 발생하는 의존성 문제를 해결하려는 시도이다. OOP는 객체와 데이터 간의 관계를 중심으로 문제를 해결하는 패러다임이지만, 객체 간의 의존성이 복잡하게 얽히게 되면 Maintainability이 떨어지고 재사용이 어려워지는 문제를 발생시킨다. 이러한 Dependency는 클래스나 객체가 다른 클래스나 객체에 기능적으로 의존하는 것을 의미하며, 이러한 의존성이 많아질수록 코드 변경의 영향이 연쇄적으로 발생하고 시스템이 점점 관리하기 어려워진다.
Dependence Management (DDM)의 핵심 목표는 각 클래스가 자신의 역할을 수행하기 위해 필요한 최소한의 정보만을 알고 있도록 하는 것이다. 즉, 클래스 간의 의존성을 최소화하고 Coupling를 낮추며 Cohesion를 적절히 관리하여 클래스가 독립적으로 작동할 수 있게 한다.
Defining the form of services with interfaces:



PowerSource라는 인터페이스를 도입하여 Boat 클래스는 실제로 어떤 종류의 동력원이든지 상관없이 동작할 수 있게 설계되었다. Boat는 Engine이나 Sail 클래스에 대해 구체적으로 알 필요가 없으며, 대신 PowerSource 인터페이스를 통해 start()와 stop() 메서드를 호출한다. 이렇게 함으로써, 클래스 간의 의존성을 최소화하고 Coupling을 낮출 수 있다.
인터페이스는 서비스 제공자를 추상화하는 역할을 한다. Boat는 실제로 어떤 동력원인지에 대해 전혀 신경 쓰지 않고, 단지 동력원이 PowerSource 인터페이스를 구현하기만 하면 된다. 이로 인해 Boat는 Engine이든 Sail이든 어떤 동력원이든 사용할 수 있어 유연한 설계가 가능해진다. 상속 대신 인터페이스를 통해 의존성을 관리함으로써 Flexibility을 높일 수 있다.
Boat는 PowerSource 인터페이스만을 알면 되기 때문에 Coupling이 낮다. 즉, Boat는 구체적인 Engine이나 Sail의 동작 방식에 의존하지 않으며, 각 클래스는 자신의 역할에 충실하다. Engine은 엔진을, Sail은 돛을 담당하고, Boat는 동력원의 구체적인 구현에 구애받지 않고 이를 사용할 수 있다. 이로써 각 클래스는 자신의 역할에만 집중하게 된다.
결론적으로, Dependence Management는 객체 간 의존성을 체계적으로 관리함으로써 소프트웨어 품질의 핵심 요소인 Maintainability, Reusability, Flexibility을 개선할 수 있다.

4. [10점] OCP는 목표이고 DIP는 Mechanism이고 LSP는 보험이라고 했다. 무슨 의미인가? 예를 들어 설명해보라.

(Chapter 11의 Button과 Lamp 예제를 활용)
어떤 스마트 홈 시스템에서 Button을 통해 다양한 장치를 제어한다고 가정해보면. 처음에는 Lamp를 켜고 끄는 기능만 있었다. OCP에 따라, 우리는 시스템을 확장할 때 기존 코드를 수정하지 않고도 새로운 기능을 추가할 수 있어야 한다. 예를 들어, Lamp 외에도 팬이나 에어컨 같은 새로운 장치를 추가할 수 있어야 하는데, 이를 위해 Button의 코드를 수정하지 않고도 이 장치들을 제어할 수 있도록 해야 한다.
이를 실현하기 위해 DIP를 적용하여, Button은 특정 장치(예: Lamp)에 의존하지 않고 추상화된 인터페이스에 의존하도록 설계한다. 이렇게 하면 Button은 특정 장치가 아니라 "Switchable Device"라는 추상 인터페이스를 통해 다양한 장치를 제어할 수 있게 된다. 즉, Button은 팬이나 에어컨 같은 새로운 장치가 추가되더라도 이 추상 인터페이스(Polymorphism)를 통해 제어할 수 있으므로, 시스템의 유연성이 높아진다.
마지막으로, LSP는 서브타입이 상위 타입을 대체할 수 있음을 보장하여 설계의 안정성을 유지하는 역할을 한다. Button이 사용하는 "Switchable Device" 인터페이스를 구현한 모든 장치는 Button과의 호환성이 유지되어야 한다. 예를 들어, 팬이 "Switchable Device"로서 Lamp 대신 Button에 연결되더라도 동일하게 동작해야 한다. LSP는 이러한 치환 가능성을 보장함으로써 시스템의 안정성을 유지한다.
결론적으로, OCP는 시스템이 확장 가능하도록 하는 목표이며, DIP는 이를 위해 추상화에 의존하는 설계를 가능하게 하는 메커니즘이고, LSP는 서브타입이 상위 타입을 대체할 수 있음을 보장하여 설계의 안정성을 유지하는 역할을 한다. 이 세 가지 원칙을 통해 고객의 요구사항 변화에 유연하게 대응할 수 있는 시스템을 구축할 수 있다

5. [30점] ASD(Agile Software Development) 111쪽 맨 아래부분에 Liskov 의 principle에 대한 정의가 나온다.

a. [10점] 무슨 의미인지 해석해 하고, 이것이 왜 중요한지 예를 들어 설명해 보라.

Liskov Substitution Principle(LSP)은 "subtype은 그것의 base type으로 치환 가능해야 한다"는 의미이다. 즉, 프로그램 내에서 어떤 객체의 부모 타입을 해당 객체의 서브타입으로 대체하더라도 프로그램의 올바른 동작을 보장해야 한다는 뜻이다. 이 원칙은 객체지향 프로그래밍(OOP)에서 polymorphism을 지키는 중요한 원칙이다.
LSP의 중요성은 소프트웨어 시스템이 maintainability과 extensibility을 갖출 수 있게 해주는 데 있다. 예를 들어, 어떤 클래스 D가 클래스 B를 상속받는 경우, 함수 f가 B의 객체를 인자로 받을 때, 이를 D의 객체로 대체해도 함수 f가 올바르게 동작해야 한다. 만약 D가 B의 행위를 변경하여 f가 예상하지 못한 결과를 내놓는다면, D는 LSP를 위반한다. 이 경우 f가 B의 모든 파생

클래스의 변경에 대해 closed 되어 있지 않기 때문에, OCP 또한 위반한다. 즉, 소프트웨어의 Rigidity, Fragility, Immobility 등이 증가한다.

b. [10점] 112쪽에 “A simple example of a Violation of the LSP”에서 하고자 하는 말이 무엇인가? 그래서 어떻게 하라는 이야기 인가? 설명해 보라,

112쪽의 "A simple example of a Violation of the LSP"에서는 Run-Time Type Information (RTTI)를 사용하는 예시로 LSP가 위반되는 상황을 보여준다. 코드에서 if문을 통해 객체의 타입을 검사하고 타입에 따라 행동을 달리하는 것은 객체지향 설계에서 좋지 않은 설계이다. DrawShape 함수에서 Shape 타입의 객체를 검사하여 Circle이나 Square에 따라 다른 함수를 호출하는 방식은, 새로운 도형이 추가될 때마다 DrawShape 함수를 수정해야 하므로 OCP도 위반한다.

이 문제를 해결하려면 polymorphism을 사용해야 한다. 부모 클래스에 virtual function을 선언하고, 각 자식 클래스에서 이 함수를 재정의하여 타입에 따라 다르게 동작하도록 구현함으로써, DrawShape 함수의 코드 수정 없이도 새로운 파생 클래스를 추가할 수 있게 만든다. 이를 통해 객체가 상위 타입으로 대체되더라도 올바르게 동작하며, LSP를 준수할 수 있다.

c. [10점] 113쪽에 “square and Rectangle, a more subtle violation”에서 하고자 하는 말은 무엇인가? 그래서 어떻게 하라는 이야기 인가? 설명해 보라,

113쪽의 "square and Rectangle, a more subtle violation"에서는 Square와 Rectangle 클래스 간의 상속 관계가 LSP를 subtle하게 위반하는 경우를 설명한다. 문제는 함수 g의 작성자는 Rectangle의 가로 길이를 바꾸는 것이 세로 길이를 바꾸지는 않을 것이라고 생각한다'는 데 있다. 하지만, 상속의 관계와 설계의 유효성은 고객의 관점에서 생각되어야 한다. 일반적으로 "사각형(Rectangle)"은 "정사각형(Square)"의 상위 개념으로 간주될 수 있지만, 함수 g의 행위 측면에서 볼 때 Square는 Rectangle이 아니다. 즉, Square는 Rectangle을 완전히 대체할 수 없으며, 이는 LSP를 위반하게 된다.

이 문제를 해결하기 위해, Square는 Rectangle의 특성을 모두 만족시키지 못하기 때문에 상속 관계로 두기보다는, 공통된 인터페이스를 구현하는 별도의 클래스로 두는 것이 더 적절하다. 이렇게 하면 각 클래스가 자신만의 고유한 특성을 유지하면서도 필요에 따라 공통된 인터페이스를 통해 사용될 수 있다.

6. [10점] Agile Software Development 134쪽 Conclusion 부분에 나오는 DIP의 중요성에 대한 설명 중 “The principle of dependency inversion is fundamental low-level mechanism behind many of the benefits claimed ...” 라는 구문이 나온다. 어디에 어떻게 작용 했는지 예를 들어 설명해 보라. 그래서 소프트웨어 퀄러티가 어떻게 좋아지는가? 설명하라.

Button과 Lamp 예제를 살펴보면. 초기에는 Button 객체가 Lamp 객체에 직접 의존하고 있었기 때문에 Lamp가 변경될 때마다 Button도 수정해야 했다. 이러한 밀접한 의존성은 Button이 Lamp를 제어하는 것 이외의 목적으로 재사용될 수 없게 하여 설계의 확장성과 유연성을 제한했

다.

이 문제를 해결하기 위해 DIP가 적용되며, 추상화인 ButtonServer 인터페이스가 도입된다. Lamp와 직접 상호작용하는 대신, Button은 필요한 기능(이 경우, 켜기 또는 끄기 메서드)을 정의하는 추상 인터페이스인 ButtonServer에 의존하게 된다. Lamp 클래스는 이 인터페이스를 구현하며, 이제 ButtonServer라는 추상화에 의존하게 된다. 이러한 종속성 반전을 통해 Button은 Lamp와 완전히 분리된다. 결과적으로 Button은 내부 논리를 변경하지 않고도 ButtonServer 인터페이스를 구현하는 모든 객체를 제어할 수 있게 된다.

더 나아가 ButtonServer의 이름을 SwitchableDevice와 같은 더 일반적인 이름으로 변경함으로써 인터페이스의 reusability와 mobility을 향상시킬 수 있다. 이러한 변경은 on/off 동작을 구현할 수 있는 다양한 장치에 추상화를 적용할 수 있도록 한다. 이 인터페이스는 더 이상 Button에만 속하지 않으며, switchable 장치와 상호작용해야 하는 모든 클라이언트가 사용할 수 있게 된다.

위 예제에서 볼 수 있듯이, DIP는 flexibility, durability, maintainability, scalability가 향상된 설계를 가능하게 한다.

7. [10점] Agile Software Development 144쪽 The Polyad v. the Monad에서 하고 싶은 이야기는 무엇인가? 즉, 무엇이 문제이고 그래서 어떻게 하라는 이야기인가? 145쪽 Conclusion에서 하는 이야기 와는 어떻게 연관이 되는가? 설명해보라.

Polyad와 Monad의 비교에서 강조하는 핵심은 불필요한 coupling을 줄이는 것이다. 여기서 문제는 모든 UI 인터페이스를 단일 객체로 사용하는 Monad 형태가, 코드 변경 시 불필요하게 많은 영향력을 미친다는 점이다. 예를 들어, WithdrawUI에 변화가 생길 경우 g 함수뿐만 아니라 g의 클라이언트들 모두 영향을 받게 된다. 이는 시스템의 rigidity를 높이고, 유지보수를 어렵게 만든다. 반면 Polyad 형태에서는 필요에 따라 각 UI 인터페이스를 개별적으로 전달하므로, 특정 인터페이스의 변경이 다른 부분에 미치는 영향을 최소화할 수 있다. 즉, 클라이언트간의 의존성을 줄일 수 있다.

145쪽 Conclusion에서 언급된 내용과의 연결점은 이러한 fat 클래스 문제를 해결하는 데 있다. 결론에서는 클라이언트가 실제로 사용하는 메서드에만 의존하도록 설계하는 중요성을 강조하며, 이를 위해 fat 클래스를 여러 클라이언트 전용 인터페이스로 분해할 필요가 있다고 한다. 이렇게 하면 클라이언트는 자신이 호출하지 않는 메서드에 대한 의존성을 제거하여 상호 결합을 줄이고, 시스템의 Maintainability, Reusability, Flexibility를 높일 수 있다. 즉, 특정 기능에 필요한 인터페이스와 결합하도록 설계하면 유지보수가 쉬워지고, 각 인터페이스의 독립적 재사용 가능성이 커지며, 변경이 다른 기능에 미치는 영향을 최소화해 시스템의 유연성도 확보할 수 있다.

감사합니다