안녕하세요 오늘 코드의 가독성에 대해 발표를 하게 된 박원영입니다.

코드의 가독성에 대해 살펴보면, 원칙, 명명, 주석, 상태, 절차, 의존성 등이 있는데요, 오늘은 가독성의 도입과 원칙에 대해 살펴보겠습니다.

먼저 가독성은 네가지 특징이 있습니다. 첫번째 특징은 분명해야 합니다. 예를 들어 변수명을 설정할 때, 상대적으로 의미가 없는 flag보다는 isVisible이 변수를 다루기 수월합니다. 두번째 특징으로는 단순합니다. 우측보다는 왼쪽이 더 간단하여 이해하기가 쉽습니다. 세번째로, 가독성은 분리시켜야 합니다. 또한 코드는 구성이 잘되어 있어야 가독성이 있다고 합니다.

이러한 가독성이 필요한 이유는 크게 지속 가능한 발전과 코드를 해독하기 편하게 하기 위함입니다.

지속 가능한 발전을 위해 리팩토링, 즉 기존의 코드를 개선하는 것 만으로는 한계가 있습니다. 대신, 가독성 유지와 기술 부채의 지속적인 상환 자체를 확장 가능한 상태로 만들어야 합니다. 이를 위해서 코드를 작성하는 것보다 코드를 읽을 때 더 편리하도록 중점을 둬야 합니다. 코드 해독이 편리하기 위해서는 다른 개발자들에게 코드 검토를 요청해야 합니다. 또한 버그를 수정할 때는 수정 코드를 간단하게 작성해야 합니다.

가독성을 높여야 팀의 생산성을 향상시킬 수 있습니다. 단기적인 생산성만 중심을 맞춰서는 안됩니다. 팀의 생산성은 개인에게 초점이 맞춰질수록 줄어듭니다.

견고하고 가독성 높은 코드를 작성하기 위해서는 프로그래밍 원칙을 따르는 것이 좋습니다. 원칙에는 다섯가지가 있습니다.

첫번째 원칙은 보이스카우트 운동의 창시자의 격언을 소프트웨어 개발에 적용한 것입니다. 코드를 변경할 때는 더 깔끔하게 작성하자는 주장입니다.

예를 들어, 변수를 더 직관적인 이름으로 바꾼다거나, 코멘트를 추가하고, 사용하지 않는 코드를 삭제하는 것 등이 있습니다. 반대로, 코드를 변경할 때 거대한 요소를 추가하는 것은 지양해야 합니다. 예를 들어, 거대한 함수로 된 문장, 거대한 case문, 상속 등을 추가하는 것은 이 원칙에 반하는 것입니다.

원칙에 반하는 또 다른 경우는, 생각없이 코드를 추가하는 것입니다. 따라서 코드를 추가하기 전에는 코드가 위치한 장소가 정확한 장소인지, 더 간단하게 만들 수 있는지를 생각해봐야 합니다.

만약 코드가 이런 상태라면 ViewType의 경우, 함수에 A, B 말고도 Z를 추가할 수 있을까요? 가독성의 보이스카우트 원칙에 따라 안됩니다. 충분히 케이스가 많기 때문에 케이스가 거대해지는 것을 방지하기 위해 추가하는 것보단 전략적인 패턴을 적용하는 것이 바람직합니다. 예를 들어 기본생성자를 이용해 매개변수를 선언하고, 매개변수로 조건식을 제거하고 Z를 추가하면 됩니다.

두번째 원칙은 기능과 코드는 필요시에만 구현되어야 한다는 원칙입니다. 필요할 것 같아서 미리 구현하면 안됩니다. 그렇게 작성한 코드의 90%는 미래에 사용하지 않고 오히려 나중에 코드를 변경할 때 장애물이 될 수 있기 때문에 구조를 단순하게 하여 나중에 코드를 변경할 때 편리하도록 작성합니다. 예를 들어 사용하지 않는 함수들, 구현이 하나 이하인 추상화 레이어, 고정값 밖에 주어지지 않는 가인수 등이 있습니다. 단, 이 원칙은 코드 변경이 쉽다는 전제하에 있기 때문에 외부에 공개하는 API와 라이브러리, 마이그레이션이 어려운 데이터베이스 스키마 등은 고려해서 설계해야 합니다.

세번째 원칙은 구현 기법을 단순화하는 것입니다. 단순화하기 위한 방법에는 두가지가 있습니다. 라이브러리나 틀, 디자인의 사용을 제한하는 것과 초기구현을 사용하는 것입니다.

예를 들어 이 코드처럼 한 줄로 간단하게 끝낼 수 있는 것을

다음 코드처럼 일관성을 위해 불필요하게 코드를 사용함으로써 복잡하게 만듭니다. 이렇게 통일성의 관점에서 아름다워 보이는 코드는 항상 가독성이 좋지는 않습니다.

네번째 원칙은 solid라고 불리는 오브젝트 지향을 위한 원칙으로, 클래스가 변경될 때는 오직 한가지 이유만으로 변경되어야 합니다. 예를 들어, 사용자 정보의 구조인 이름과 메일 주소와 그 집합인 등록 완료 사용자 리스트를 하나의 클래스로 구현하려고 한다. 그러면 클래스를 변경하는 이유가 사용자 정보를 변경하고자 하는 경우와 사용자 정보 항목을 바꾸고 싶은 경우로, 2가지가 됩니다. 이 경우 어떤 변경이 다른 기능에 영향을 끼치지 않는다고 보장할 수 없습니다. 따라서 클래스의 책임과 관심의 범위는 하나로 좁혀야 합니다.

다음 코드처럼 두가지 함수를 하나의 클래스로 구현하려 하면 좋지 않습니다.

이를 해결하기 위해서는 모델 클래스를 각 엔티티 별로 분할시킵니다. 다음 코드와 같이 클래스 두 개로 선언합니다.

따라서 책임을 작게 갖기 위해서는 분할할 때의 기준을 각 엔티티에 대한 모델, 각 계층과 구성요소에 대한 로직, 각 대상 유형에 대한 유틸리티로 나눕니다.

마지막 원칙은 최적화의 효과가 작은 경우 하지 말아야합니다. 최적화를 하면 코드가 복잡해지기 때문에 효과가 클 때만 사용하기를 권장합니다. 다음 코드를 보면 arrayList의 요소를 반복문으로 검색하는 코드를 hashMap으로 치환하면, 계산량이 줄고 코드도 간결해집니다.

최적화를 할 때는 프로파일링과 견적이 중요합니다. 최적화를 잘못 사용하면 단순화를 방해할 수 있습니다. 또한 늦은 초기화와 캐시로 인한 오버헤드 비용을 요구할 수 있습니다.