소프트웨어 개발에서 나쁜 코드(Bad Code)의 영향

나쁜 코드(Bad Code) 란 ?

나쁜 코드란 소프트웨어가 올바르게 작동하기 어렵게 만드는 다양한 문제점이 널려 있는 코드를 의미합니다.

이는 읽기, 이해, 유지보수, 편집 및 추가하기 어려운 코드로, 불필요하게 복잡하거나 중복되며 일관성 없는 형식으로 작성되거나, 민감한 정보를 악용에 취약하게 만들 수 있습니다.

단순한 구문 오류나 작은 버그뿐만이 아니라 소프트웨어의 가독성, 유지 보수성, 확장성에 영향을 미치는 다양한 문제를 포함합니다.

이는 복잡하거나 제대로 구조화되지 않은 코드, 문서화가 부족한 코드 등을 말합니다.

나쁜 코드의 원인은 다양하며, 빠른 마감 기한의 압박, 코딩 지식 부족, 수동 문제 해결, 일관되지 않은 코딩 스타일 및 소프트웨어 성능을 초과한 요구 사항에서 비롯됩니다.
나쁜 코드는 또한 간단한 문제에 대해서 지나치게 복잡한 해결책, 코드 중복, 과도한 종속성으로 나타날 수 있습니다.

예시1

코드에 중복된 문자열이 있는 경우, 리팩토링 프로세스가 모든 발생 지점에서 업데이트되어야 하므로 오류 발생 가능성이 높아집니다.

이는 코드를 효율적으로 업데이트할 수 있는 단일 위치에 코드를 업데이트하는 더 효율적인 솔루션에 대한 개발자의 시간을 낭비할 수 있습니다.

Noncompliant code example

With the default threshold of 3:

```
public void run() {
  prepare("action1");                                 // Noncompliant - "action1" is duplicated
  execute("action1");
  release("action1");
}

@SuppressWarning("all")                               // Compliant - annotations are excluded
private void method1() { /* ... */ }
@SuppressWarning("all")
private void method2() { /* ... */ }

public String method3(String a) {
  System.out.println("'" + a + "'");                  // Compliant - literal "'" has less than
  return "";                                          // Compliant - literal "" has less than 5
}
```

Compliant solution

```
private static final String ACTION_1 = "action1";  // Compliant

public void run() {
  prepare(ACTION_1);                                // Compliant
  execute(ACTION_1);
  release(ACTION_1);
}
```

예시2

'null'대한 역참조 또는 접근은 결코 해서는 안 됩니다. 이런 경우 갑자기 프로그램 종료가 발생할 수도 있으며 어떠한 정리 프로세스도 실행할 수 없게 됩니다.

최악의 경우에는 공격자가 보안 조치를 우회할 수 있는 디버깅 정보를 노출할 수 있습니다.

Noncompliant code example

```
char *p1 = ... ;
if (p1 == NULL && *p1 == '\t') { // Noncompliant, p1 will be dereferenced IFF it is null
  // ...
}

char *p2 = ... ;
if (p2 != NULL) {
    // ...
}
*p2 = '\t'; // Noncompliant; potential null-dereference

char *p3, *p4;
p3 = NULL;
// ...
p4 = p3;
*p4 = 'a';  // Noncompliant
```

Compliant solution

```
char *p1 = ... ;
if (p1 != NULL && *p1 == '\t') { // Compliant, *p1 cannot be evaluated when p1 is NULL
  // ...
}

char *p2 = ... ;
if (p2 != NULL) {
    // ...
  *p2 = '\t'; // Compliant
}
```

예시3

코드는 하드코딩된 비밀이나 비밀번호가 없어야 하며, 이를 통해 민감한 정보를 악용할 수 있는 악의적인 당사자로부터 이해당사자를 보호해야 합니다.

만약 하드코딩된 비밀과 비밀번호가 유출된다면, 이는 귀하의 조직에 대해 돌이킬 수 없는 영향을 초래할 수 있습니다.

Sensitive Code Example

```
String username = "steve";
String password = "blue";
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                  "user=" + username + "&password=" + password); // Sensitive
```

Compliant Solution

```
String username = getEncryptedUser();
String password = getEncryptedPassword();
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                  "user=" + username + "&password=" + password);
```

나쁜 코드의 파급 효과

나쁜 코드의 여파는 코드 라인을 넘어 전체 개발 주기에 영향을 미칩니다.

유지보수성 및 확장성 감소

나쁜 코드는 이해하기 어려운 경향이 있습니다. 결과적으로 유지, 확장 또는 수정하는 것은 어려운 작업이 됩니다.
이는 확장성이 부족해지며, 비즈니스 요구 사항의 변화에 적응하거나 새로운 기능을 통합하는 것이 어려워집니다.

버그 발생 및 기술적 부채 증가

나쁜 코드는 버그의 번식지입니다. 은밀한 문제를 숨기고 있어 예상치 못한 시스템 실패나 기능 오동작으로 이어질 수 있습니다.
미해결된 문제가 누적되어 기술적 부채를 생성하며, 시간이 흐를수록 이를 해결하기 위해 더 많은 노력과 자원이 필요해집니다.

생산성과 효율성 저하

개발자들은 나쁜 코드를 해석하고 수정하는 데 상당한 시간을 소비합니다. 이로 인해 생산성이 감소하며, 혁신에 중점을 둘 수 없거나 새로운 기능을 개발할 수 없게 됩니다.

결과적으로 전체 개발 프로세스가 느려지며, 마감 기한을 놓치고 프로젝트 진행이 지체됩니다.

비용 및 리스크 증가

나쁜 코드의 영향은 시간이 지남에 따라 누적되어 소프트웨어 유지보수, 버그 수정, 잠재적인 개선 작업 및 기술적 부채 해결에 증가된 비용으로 나타납니다.

나쁜 코드는 또한 소프트웨어의 신뢰성, 보안 및 안정성에 리스크를 초래할 수 있어 평판 손상이나 규정 준수 문제로 이어질 수 있습니다.

나쁜 코드를 빠르게 개선하기 위해 리팩터링, 코드 리뷰, 코딩 표준 준수 등을 통해 이러한 영향을 완화할 수 있으며, 보다 건강한 개발 환경을 조성하고 소프트웨어 제품의 전반적인 품질을 향상시킬 수 있습니다.

나쁜 코드를 방지하는 방법

나쁜 코드가 코드베이스에 침투하는 것을 방지하기 위해 개발자들은 개발 라이프사이클 전반에 걸쳐 여러 가지 모범 사례를 채택할 수 있습니다.

나쁜 코드를 방지하기 위한 주요 단계들은 다음과 같습니다.

- **코딩 표준 수립 및 준수** : 일관된 코딩 규칙은 가독성과 유지보수성을 높이어, 나쁜 코드의 발생 가능성을 줄입니다.
- **코드 리뷰 실시** : 개발자들이 서로의 코드를 철저히 검토하여 잠재적인 문제를 찾고 팀 멤버 간 지식 공유를 촉진하는 강력한 코드 리뷰 프로세스를 구현합니다.
- **자동화된 코드 분석 도구 도입** : 정적 코드 분석 및 린팅을 위한 자동화된 도구를 활용하여 문제를 감지하고 코딩 표준을 강제하며 잠재적인 버그를 식별합니다.

  이러한 도구들은 개발 파이프라인에 통합되어 지속적인 피드백을 제공할 수 있습니다.

- **테스팅 자동화** : 버그를 조기에 찾기 위해 자동화된 테스트를 구현하며, 이는 유닛 테스트, 통합 테스트 및 종단 간 테스트를 포함합니다.
- **교육 및 역량 강화에 투자** : 개발 팀이 모범 사례, 새로운 기술 및 산업 표준을 최신 상태로 유지할 수 있도록 교육 및 역량 향상에 투자합니다.
- **보안 감사 실행** : 보안 감사 및 취약점 평가를 통해 잠재적인 보안 문제를 식별하고 해결합니다.
- **정기적인 코드 리팩터링** : 코드 리팩터링은 코드의 신뢰성과 유지보수성을 향상시키며 기술적 부채가 누적되는 것을 방지합니다.
- **버전 관리** : 버전 관리를 통해 변경 사항을 추적하고 협업하며 이전 상태로 되돌릴 수 있습니다. 이는 잘 문서화된 코드 변경 히스토리를 유지하는 데 도움이 됩니다.

이러한 단계를 개발 프로세스에 통합함으로써 소프트웨어의 성공에 나쁜 코드의 영향을 줄일 수 있습니다.

참고 자료

SonarQube 소개 및 Edition 별 기능 비교

출처 : https://www.sonarsource.com/solutions/our-unique-approach/