

12

안전한 프로그래밍

차례

- 소프트웨어 안전성 개요
- 동적 분석
 - 수동 테스트
 - 자동 테스트 케이스 생성
- 정적 분석
- 미래 기술
 - 프로그램 자동 합성

소프트웨어 오류

'/' 응용 프로그램에 서버 오류가 있습니다.

인덱스가 배열 범위를 벗어났습니다.

설명: 현재 웹 요청을 실행하는 동안 처리되지 않은 예외가 발생했습니다. 스택 추적을 생성한 위치에 대한 자세한 정보를 확인하십시오.

예외 정보: System.IndexOutOfRangeException: 인덱스가 배열 범위를 벗어났습니다.

소스 오류:

```

줄 192: {
줄 193:     new_link_aid = Regex.Split(rel_article_list[
줄 194:     ],
줄 195:     );
줄 196: }

```

소스 파일: d:\WEB\mnews.jib





다양한 소프트웨어 오류 종류

- 안전성 오류 (safety error)
 - 0으로 나누기 (divide-by-zero)
 - 잘못된 배열 접근 (array-out-of-bounds error)
 - 정수 흘러넘침 (integer overflow)
 - 널 접근 (null dereference)
 - 자원 누수 (resource leak) 등...
- 기능성 오류 (functionality error)
 - 바람직한 불변식 위반 (invariant violation)
 - 성능 저하 오류 (performance bug)
 - 원치 않는 무한 루프 (infinite loop) 등 ...

소프트웨어 오류 피해 사례

- 아리안 로켓 5 폭발
 - \$1억 손해, 아리안 프로그램 몇년 후퇴
 - https://youtu.be/PK_yguLapgA?t=50s
- 정수 넘침 에러 (integer overflow)
 - 64비트 실수 (double) 타입 변수를 16비트 정수로 안전하지 못한 방식으로 타입 변환 -> 오버플로우!
 - 로켓의 onboard 컴퓨터에 진행방향을 바꾸라는 신호로 잘못 인식
 - <https://around.com/ariane.html>

보안 취약점 (Security Vulnerabilities)

- 프로그램에 존재하는 에러를 악용
- 다양한 나쁜 웨어 들 (malware)
 - Moonlight Maze (1998)
 - Code Red (2001)
 - Titan Rain (2003)
 - Stuxnet Worm (2010)
- 악성 스마트폰 앱
- 점점 더 위험하고 많아짐

소프트웨어 개발 보안(Secure Coding)을 위한 국내 노력

- SW 개발 과정에서 지켜야 할 일련의 보안활동
 - 소스코드에 존재할 수 있는 잠재적 보안 취약점을 제거
 - 보안을 고려하여 기능을 설계 및 구현
- SW 개발 시 보안 취약점을 악용한 해킹 등 내외부 공격으로부터 시스템을 안전하게 방어할 수 있도록 코딩
- 행정안전부 2012년 5월 시큐어 코딩 의무화 법안: 개발비 40억원 이상 정보화 사업에 시큐어 코딩을 위한 가이드라인을 따르는 것을 의무화

프로그램 분석 (Program Analysis)

- 프로그램들에 대한 유의미한 사실들을 발견하기 위한 기술
- 세 가지 종류:
 - 동적 분석 (static analysis) (실행 해보면서 성질 찾기)
 - 정적 분석 (dynamic analysis) (실행 전에 실행 해보지 않고 소스코드로부터 성질 유추)
 - 하이브리드 (hybrid analysis) (동적 + 정적)
- 자바 동적 분석
 - Randoop, Korat, ...
- 자바 정적 분석
 - Facebook Infer, SonarQube, ESC/Java, ...

동적 vs. 정적 분석

- 안전 (soundness): 존재하는 모든 오류를 찾음
- 완전 (completeness): 정확히 오류만 찾음 (허위 경보 없이)

	동적	정적
비용	프로그램 실행 시간에 비례	프로그램 코드 크기에 비례
효과	불안전 (에러를 놓칠 수 있음)	불완전 (실제 에러가 아닌 것 보고)

프로그램 성질 파악의 어려움

- 임의의 프로그램 대상, 프로그램 분석이 안전하고 완전할 수 있을까? → “일반적으로” 불가능!
- 동적 분석: 가능한 테스트 케이스가 무한히 많을 수 있음
- 정적 분석: 그러한 분석이 있다면, 그것을 이용해서 “풀 수 없는 문제”를 풀 수 있음 (모순!)
- 결국 안전/완전성 둘 중 하나를 포기하거나 둘 다 포기
- 프로그램 분석은 분석 비용, 안전성, 완전성 사이에서 교묘하게 줄타기하는 것

번외: 풀 수 없는 문제

- 멈춰요 문제 (Halting problem): 임의의 프로그램 p 를 받아서 p 가 유한 시간 안에 끝나는 프로그램이면 참, 아니면 거짓을 반환하는 프로그램 만들기
- 다음의 이유로 그러한 프로그램은 존재할 수 없음
 - 그러한 프로그램을 H 라고 하자. 프로그램 p 를 받아서 p 가 유한 시간안에 끝나면 $H(p)$ 는 참, 아니면 거짓.
 - 그 H 를 가지고 다음과 같은 모순된 함수 정의 가능:
 - $f(\cdot) \{ \text{if } H(f) \text{ while(true)} \}$

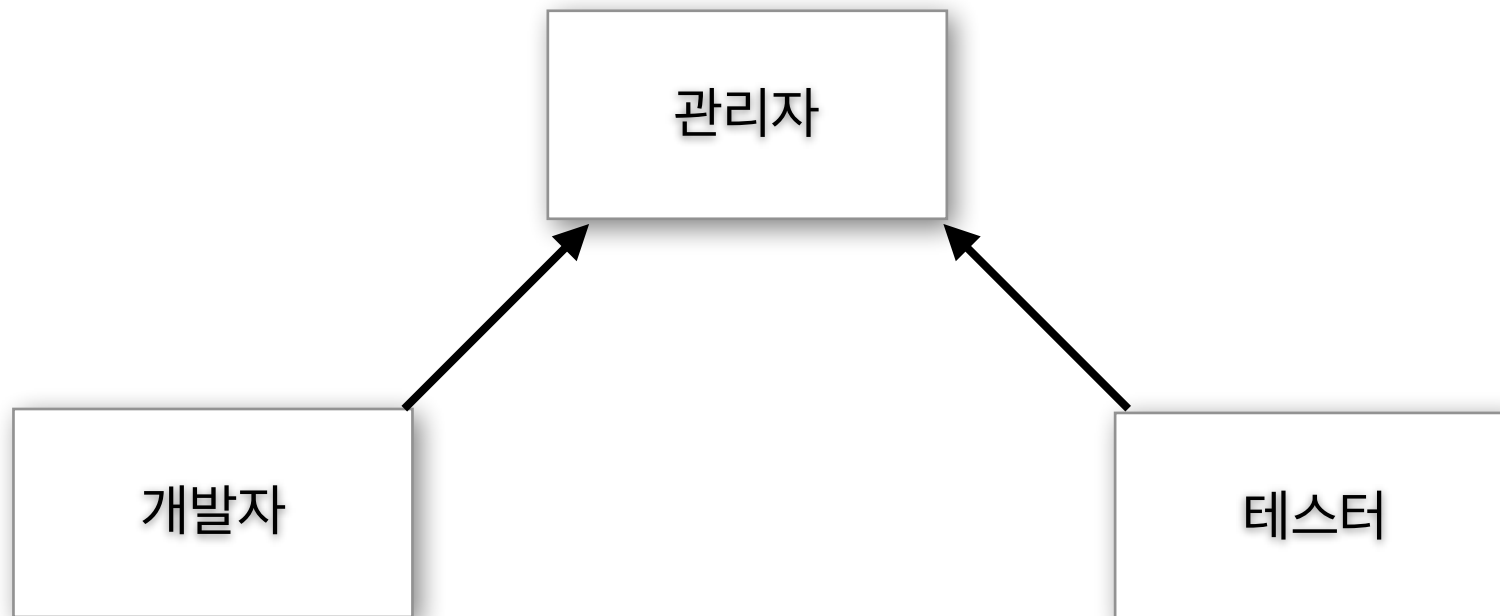
번외: 풀 수 없는 문제

- f 는 끝나는가?
 - 끝난다면 안끝나야 한다
- f 는 안끝나는가?
 - 안끝난다면 끝나야 한다. 모순!
- 어떠한 특정 프로그램 성질에 대해 안전하고 완전한 분석을 고안하는 일 = 멈춤 문제를 푸는 일
- 그러나 포기할 필요는 없음
 - 안전 & 완전 하지 않아도 쓸모있는 분석은 충분히 많음

차례

- 소프트웨어 안전성 개요
- **동적 분석 (테스팅)**
 - 수동 테스트
 - 자동 테스트 케이스 생성
- 정적 분석
- 미래 기술
 - 프로그램 자동 합성

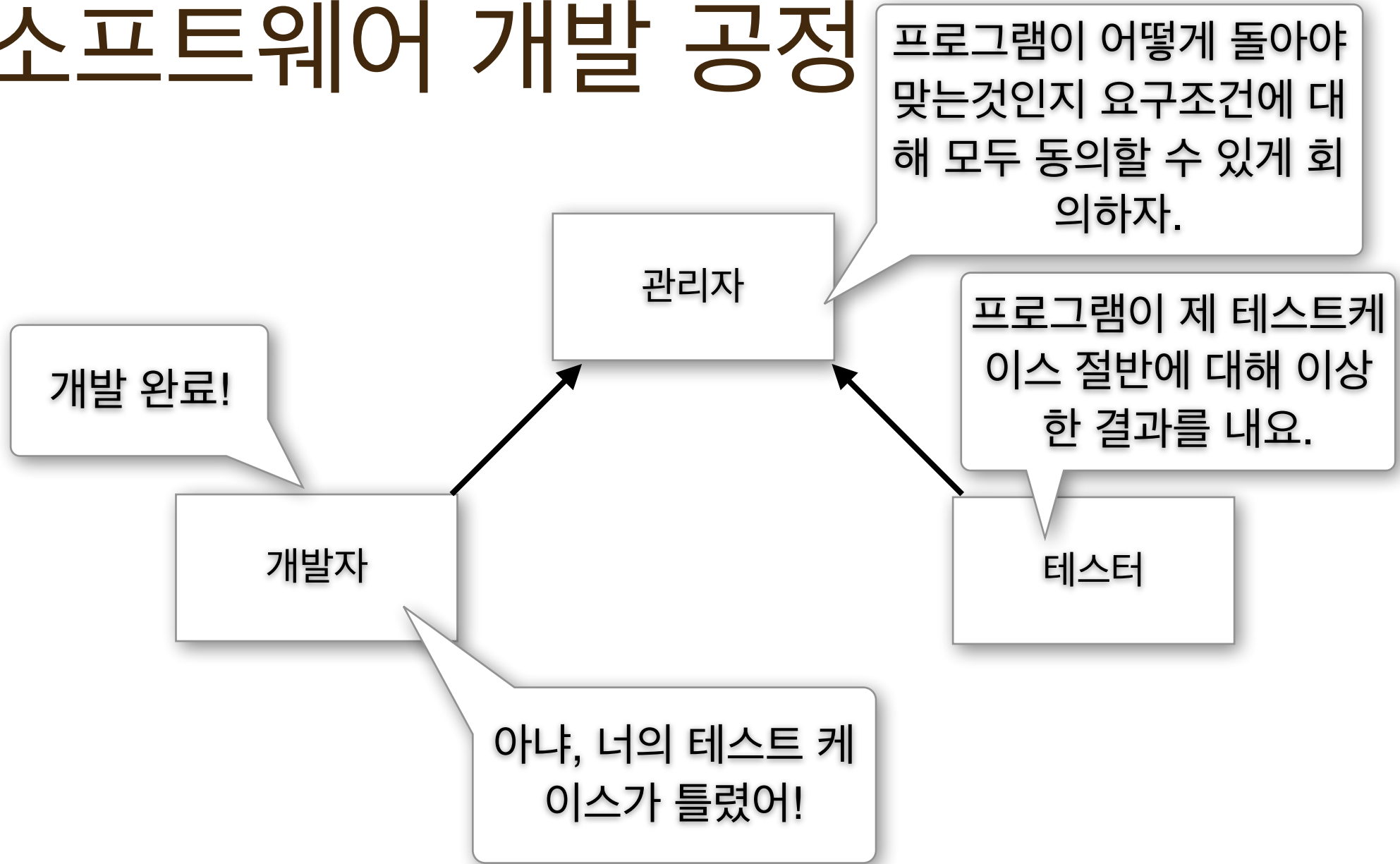
소프트웨어 개발 공정



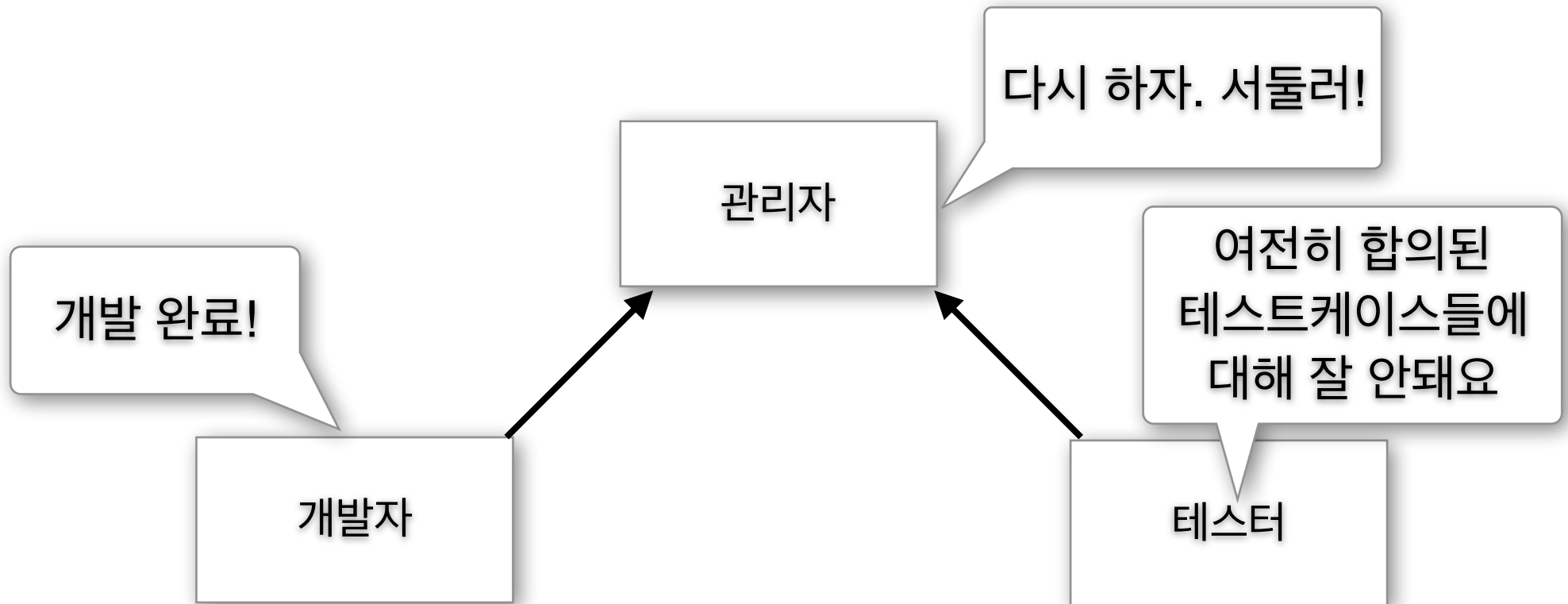
소프트웨어 개발 공정



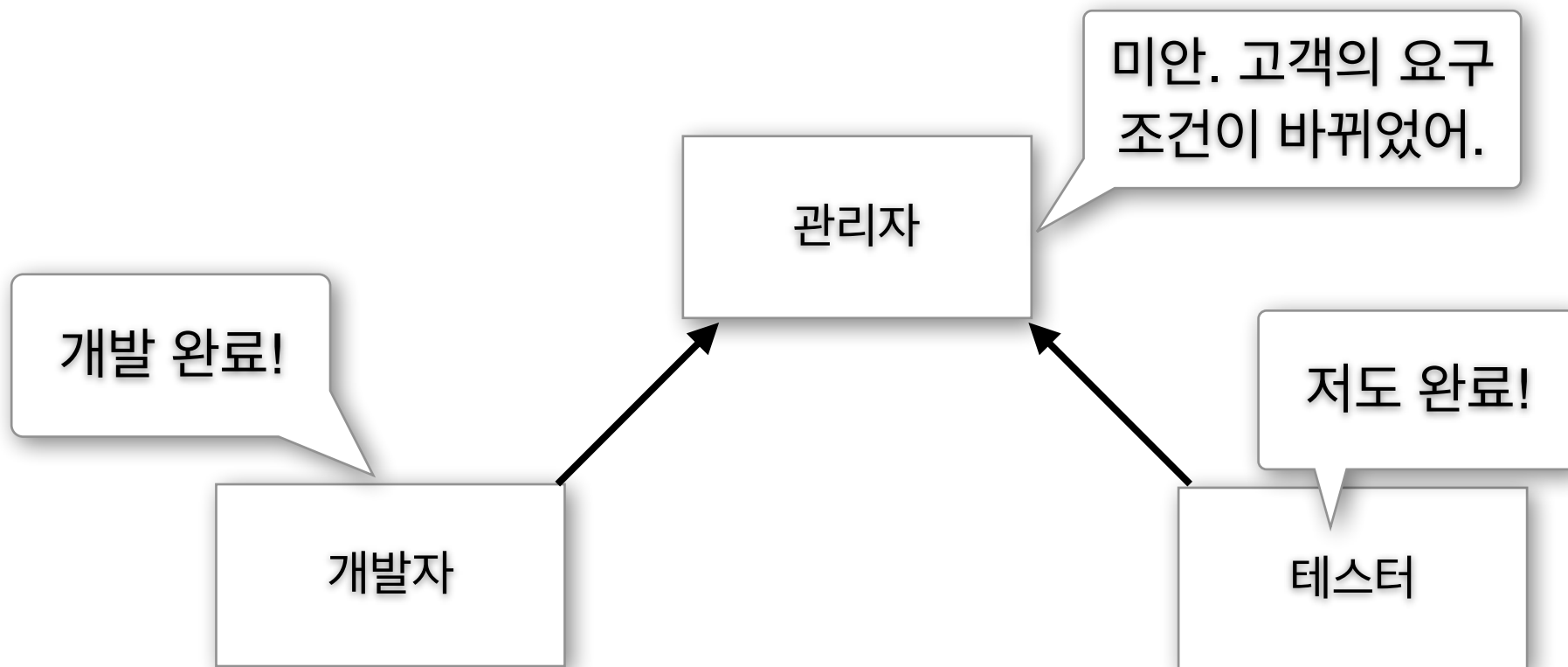
소프트웨어 개발 공정



소프트웨어 개발 공정



소프트웨어 개발 공정



중요한 관찰들

- 요구조건 (spec)은 분명해야 함.
- 개발과 테스트는 따로 진행
- 사용할 수 있는 자원(시간, 인력 등)은 제한적
- 요구조건은 계속 변화
 - 테스트도 그에 맞게 업데이트 되어야 함.

요구조건 (Specification)

- 프로그램 구현이 요구조건에 맞는지 확인하기 위해 테스트 수행
- 요구조건이 없이는 테스트할 것이 없음.
- 요구조건은 오해의 여지 없이 올바르게 쓰여야.
 - 예: 입/출력 예제, 실행 전/후 만족시켜야할 조건식

수동 vs. 자동

○ 수동 테스트

- 프로그램에 대한 이해와 함께 작성될 경우 적은 수의 테스트 케이스로 효율적으로 테스트 수행 가능
- 프로그램이 바뀌면 테스트 케이스도 함께 바뀌어야.

○ 자동 테스트

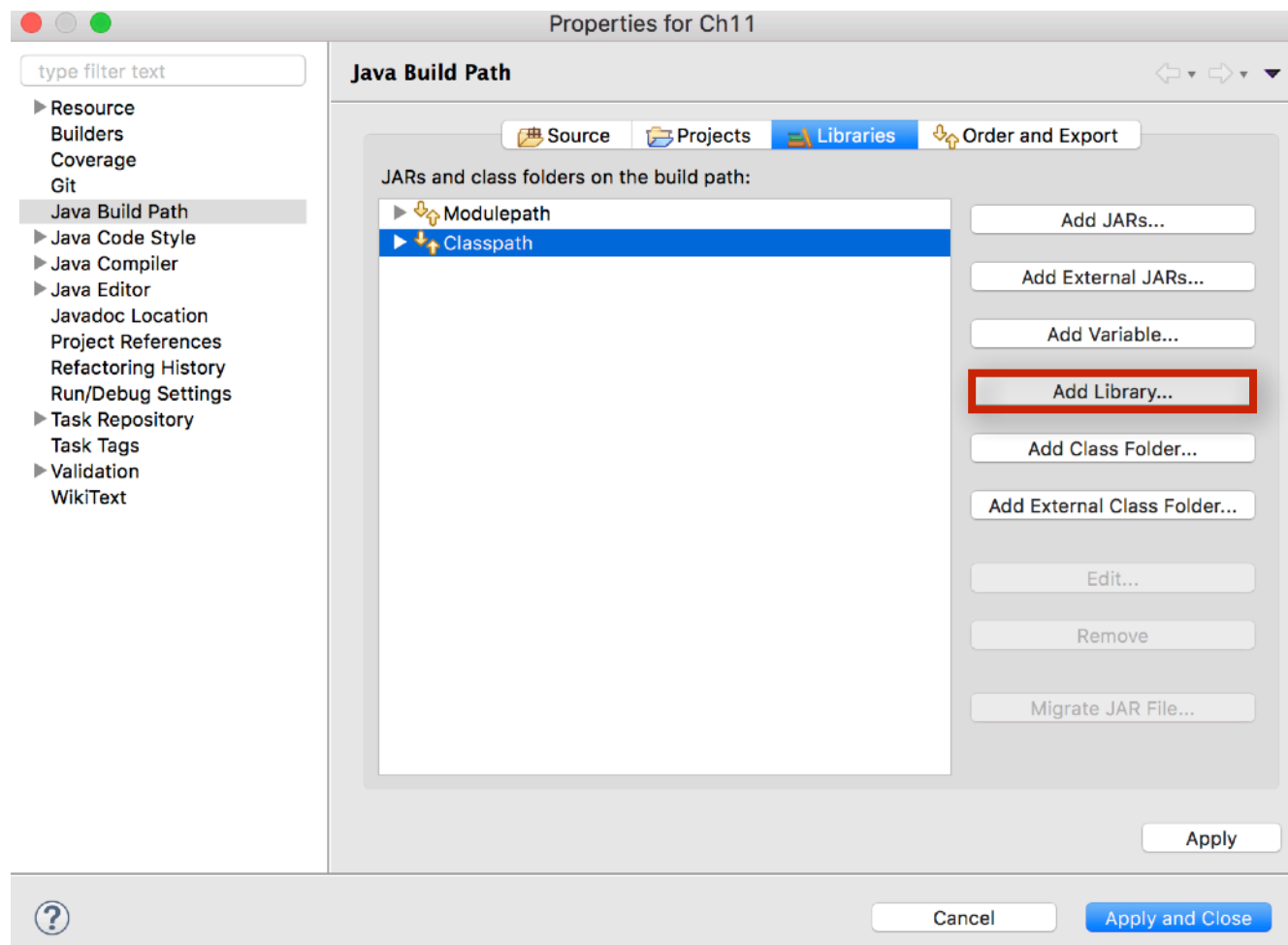
- 오류를 더 빨리 찾을 수 있음
- 테스트 케이스 수동으로 작성 불필요.
- 프로그램이 바뀌어도 수동으로 바꿀 필요 없음

수동 단위 테스트 (Unit testing)

- 코드 기본 단위(예: 메소드)에 대해 테스트를 수행
- 프로그램 전체를 테스트하는 대신, 단위 별로 테스트를 수행하는 것의 이점:
 - 테스트 케이스를 작성하기 수월
 - 문제를 파악하기 더 수월
- 자바: JUnit 단위 테스트 Framework

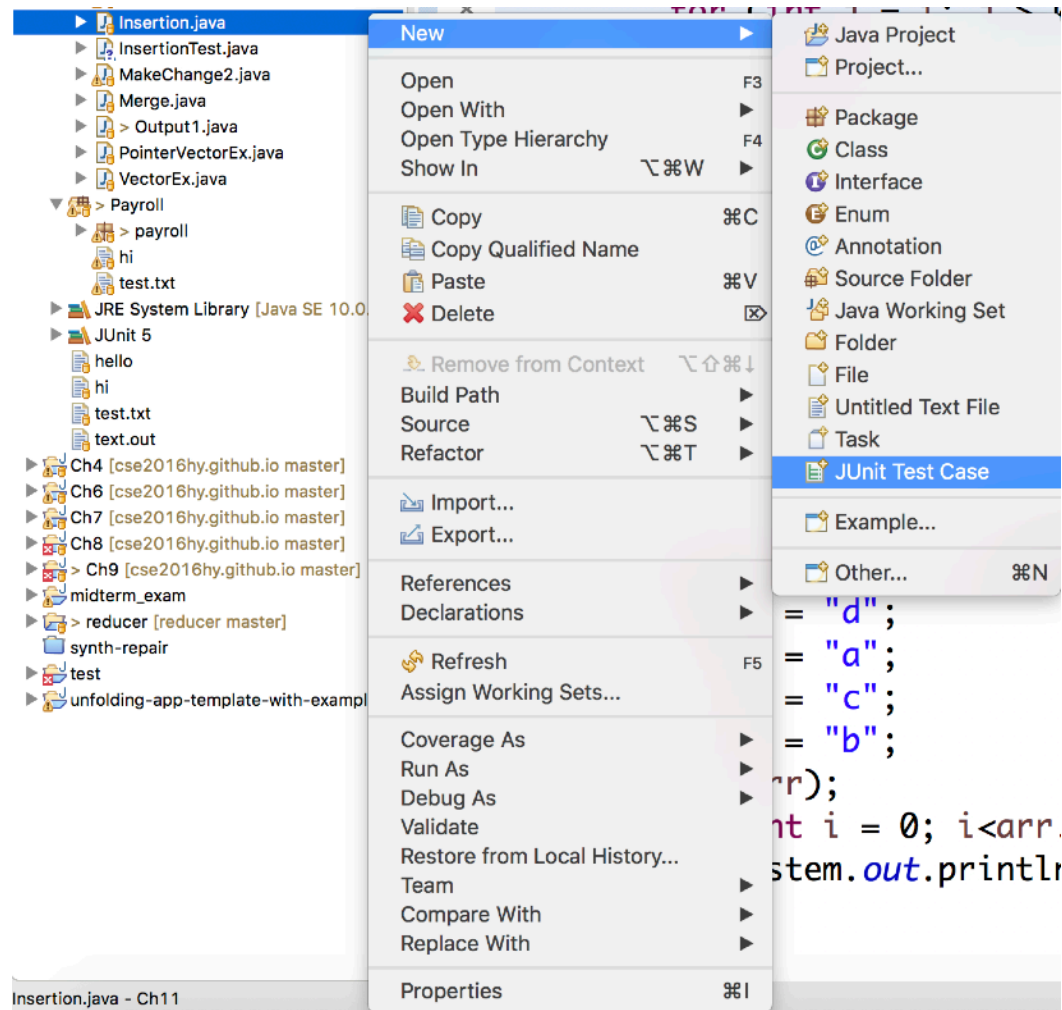
Eclipse 에 JUnit 설정하기

- Java Project 생성
- Project 이름에 오른쪽 클릭 -> 성질 (Properties) 선택
- Java Build Path 선택
- Libraries 탭을 선택
- Modulepath, Classpath 중 아무거나 선택
- Add Library... 선택
- JUnit 선택 -> Next 버튼 -> 버전 선택 (JUnit 5), 완료 버튼



Eclipse 에서 JUnit 사용하기

- 테스트 할 클래스 파일 혹은 패키지 오른쪽 클릭 -> New -> JUnit Test Case 선택 -> Next 버튼
- 테스트할 메소드 선택 후 완료 버튼



Eclipse 에서 JUnit 사용하기

- 다음과 같은 테스트 코드 생성
- `@Test` 의 의미: `testSort` 메소드가 단위 테스트를 위한 메소드임을 지정한다는 뜻

```
import static
org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class InsertionTest {
    @Test
    void testSort() {
        fail("Not yet implemented");
    }
}
```

Eclipse 에서 JUnit 사용하기

- 테스트 코드 작성
 - `assertTrue(조건):`
조건이 참이면 테스트 성공, 아니면 실패

```
import static
org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class InsertionTest {
    @Test
    void testSort() {
        String[] arr = new String[4];
        arr[0] = "d";
        arr[1] = "a";
        arr[2] = "c";
        arr[3] = "b";
        Insertion.sort(arr);
        assertTrue(arr[0].equals("a"));
    }
}
```

Eclipse 에서 JUnit 사용하기

- 테스트 실행: 테스트 클래스 오른쪽 클릭
→ Run As → JUnit Test 선택

```

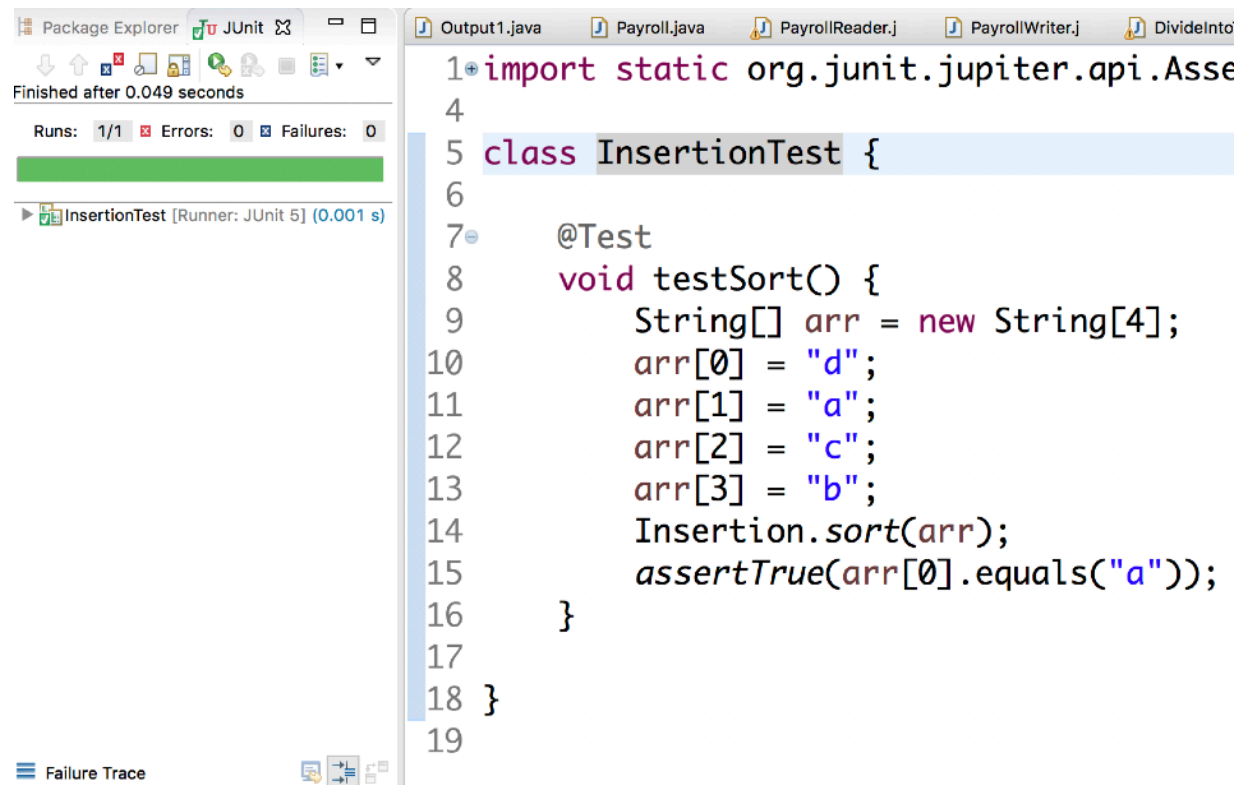
arr[0] = "a";
arr[1] = "a";
arr[2] = "c";
arr[3] = "b";
Insertion.sort(
assertTrue(arr[

```

The screenshot shows the Eclipse IDE's context menu for a Java class. The 'Run As' option is highlighted, and a submenu is open showing 'JUnit Test' as the selected option. The submenu also includes 'Run Configurations...'. The background shows a project structure with folders like 'Ch4', 'Ch6', 'Ch7', 'Ch8', 'Ch9', 'midterm_exam', 'reducer', 'synth-repair', and 'test'.

Eclipse 에서 JUnit 사용하기

- 테스트가 성공시 다음과 같음 (실패 시 에러 출력)



```
1 import static org.junit.jupiter.api.Asse
4
5 class InsertionTest {
6
7     @Test
8     void testSort() {
9         String[] arr = new String[4];
10        arr[0] = "d";
11        arr[1] = "a";
12        arr[2] = "c";
13        arr[3] = "b";
14        Insertion.sort(arr);
15        assertTrue(arr[0].equals("a"));
16    }
17
18 }
19
```

대표적인 단정문

- `assertArrayEquals(a,b)` : 배열 a와b가 일치함을 확인
- `assertEquals(a,b)` : 객체 a와b의 값이 같은지 확인
- `assertSame(a,b)` : 객체 a와b가 같은 객체임을 확인
- `assertTrue(a)` : a가 참인지 확인
- `assertNotNull(a)` : a객체가 null이 아님을 확인
- 참조: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

추가 정보 기입 활용

```
import static org.junit.jupiter.api.Assertions.*;
import static java.time.Duration.*;
import org.junit.jupiter.api.Test;
@Test
    public void test() { ... }
```

- 테스트 메소드 수행시간 제한 (시간단위: 밀리 초)

```
@Test
public void test() {assertTimeout(ofMillis( 시간 ), ()->{ 할일 })}
```

- 특정 예외가 발생해야 성공

```
@Test
public void test() {assertThrows(예외타입(예: RuntimeException).class,
    ()->{ 할일 })}
```

유닛 테스트 예

- 중간고사 채점 테스트 코드 (JUnit 4 기반)
 - <https://github.com/cse2016hy/cse2016hy.github.io/tree/master/code/midterm/class1/tests>
 - <https://github.com/cse2016hy/cse2016hy.github.io/tree/master/code/midterm/class2/tests>

좋은 테스트?

- 작성한 테스트 케이스가 좋은것인지 어떻게 판단?
 - 너무 적은 테스트 케이스: 오류를 놓칠 수 있음
 - 너무 많은 테스트 케이스: 테스트 비용 증가, 중복되거나 필요치 않은 테스트 케이스 존재, 프로그램 변화에 따라 테스트 케이스 업데이트하기 어려워짐
- 흔히 실행되는 코드 양 (code coverage) 으로 판단

실행되는 코드 양 (Code Coverage)

- 테스트 케이스들에 의해 프로그램 코드의 얼마나 많은 부분이 실행되는지 측정하는 척도 (%)
- 100% 는 달성하기 어려움
 - 모든 부분을 커버하는 테스트 케이스 작성 어려움
 - 일부 부분은 어느 입력이 주어지든 아예 실행되지 않을수도 (dead code)
 - 하지만 안전이 중요한 (safety-critical) 소프트웨어에는 간혹 달성이 요구됨

척도의 종류

- 함수 coverage: 테스트 케이스들에 의해 얼마나 많은 함수가 호출되었는가?
- 라인 coverage: 얼마나 많은 코드 줄이 실행되었는가?
- 분기 coverage: 얼마나 많은 조건문 분기가 실행되었는가?
- 이클립스에서 도출 방법
 - 메뉴 “Run” → “Coverage”
 - 하이라이트 효과 끌 때: 메뉴 “Windows” → “Show View” → “Other...” → 텍스트 창 “Coverage” 입력 후 클릭 → 새로 생긴 Coverage View 창에 Remove all sessions 버튼 클릭

척도의 종류

- 테스트 입력: `foo(1, 0)`
- 라인 coverage: 80%
- 분기 coverage: 50%
- 두 coverage 를 100%로 만들기 위해 필요한 추가 테스트 입력은?
→ `foo(1, 1)`

```

int foo (int x, int y) {
  int z;
  if (x <= y) {
    z = x;
  }
  else {
    z = y;
  }
  return z;
}

```

자동 테스트 코드 생성

- 소프트웨어 도구 Randoop 에 의해 자동으로 생성된 코드 예

```
public void test1() throws Throwable {
    BankAccount bankAccount1 = new BankAccount(0);
    BankAccount bankAccount3 = new BankAccount(0);
    bankAccount3.getBalance();
    BankAccount bankAccount6 = new BankAccount(0);
    bankAccount6.deposit(1);
    BankAccount bankAccount10 = new BankAccount(10);
    BankAccount[] bankAccountArray11 = new BankAccount[]
{ bankAccount1, bankAccount3, bankAccount6, bankAccount10 };
    double[][] doubleArray30 = ...;
    AccountController accountController31 = new
AccountController(bankAccountArray11, doubleArray30);

    // 여기서 잘못된 배열 접근 오류 발생! (ArrayIndexOutOfBoundsException)
    accountController31.computeBalancesAfterMonths(1);
}
```

Randoop

- 자바 프로그램 단위 테스트 코드 자동 생성 프로그램
- 기본 아이디어: 랜덤하게 새 테스트 케이스를 생성

테스트 코드 = 연속된 메소드 호출

- 문제: 올바르지 못한 테스트 케이스 다수 생성 가능. 이를 그 전에 생성된 테스트 케이스들의 실행 결과를 이용함으로써 해결
- 방법:
 - 새 테스트 코드를 점진적으로 증가시킴(기존의 테스트 코드를 늘리며)
 - 코드가 생성되면 실행해봄
 - 실행 결과를 다음 테스트 코드 생성하는데 이용

자동 테스트 케이스 생성

입력

- 테스트 할 클래스
- 시간 제한
- 검사할 성질 (사용자 설정 가능)
 - 예: toString 호출 뒤 실행 중 예외가 발생해선 안됨, 임의의 객체 a에 대해 a.equals(a) 는 true 를 반환해야함

출력

- 검사할 성질을 위반하는 테스트 케이스 (코드)

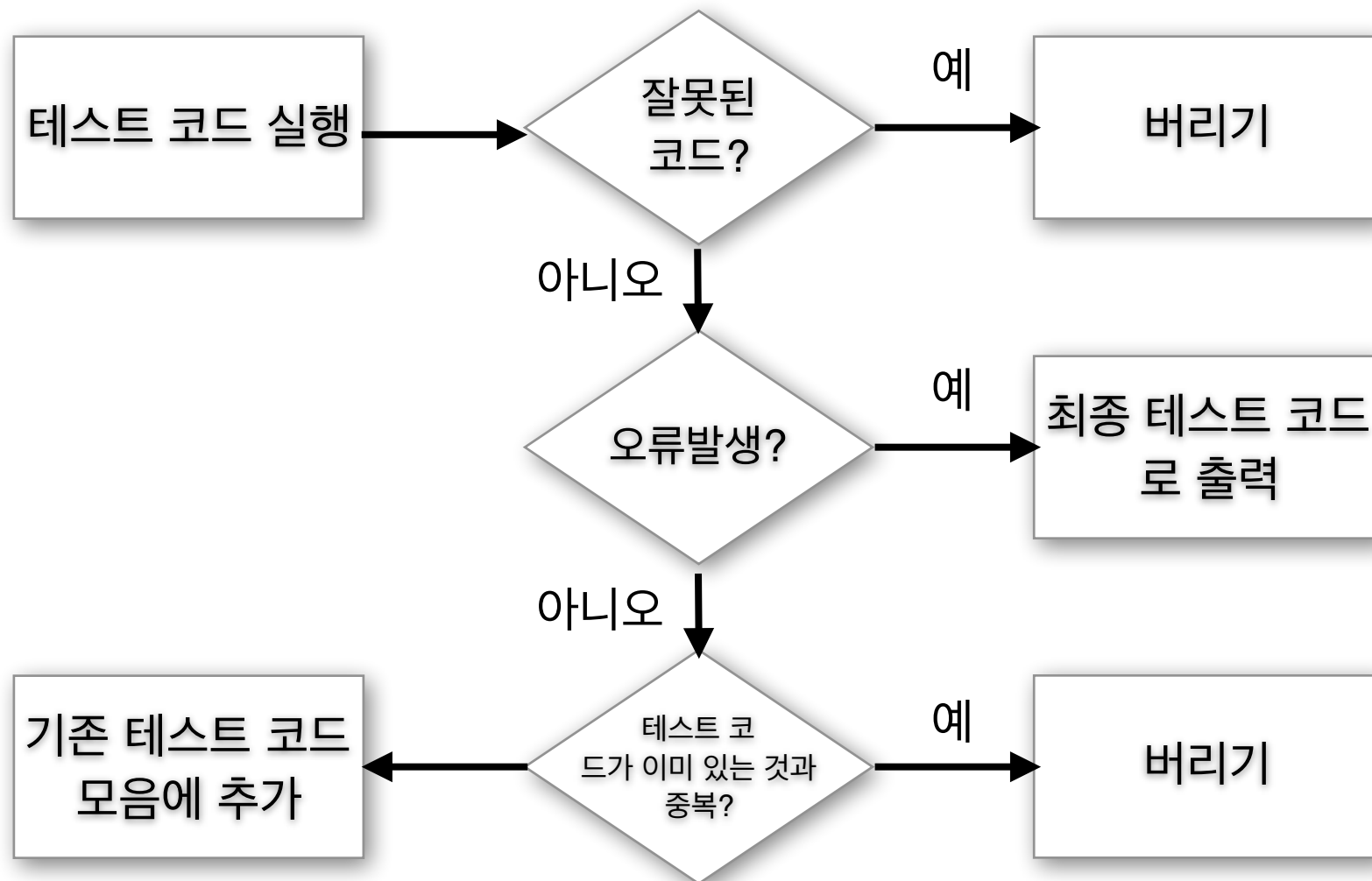
번외: Randoop 알고리즘

- 부품들: 지금까지 생성된 테스트 코드 집합
(처음엔 한줄짜리들만: { int i = 0 ; , boolean b = false; ... })

다음은 시간 제한 초과할때까지 반복:

- 새로운 테스트 코드 (연속된 메소드 호출) 생성
 - 랜덤하게 한 메소드 호출을 선택: $T_{ret} \ m(T_1, \dots, T_n)$
(메소드 이름이 m 이고 인자들 타입은 $T_{1..n}$, 반환 타입은 T_{ret})
 - 각 인자 타입 T_i 에 대해서, T_i 타입 객체 v_i 를 생성하는 테스트 코드 S_i 를 기존에 생성된 테스트 코드들로부터 (처음에는 기본 부품들) 랜덤하게 고름
 - 새 테스트 코드 생성: $S_{new} = S_1; \dots; S_n; T_{ret} \ v_{new} = m(v_1, \dots, v_n)$
- 새로운 테스트 코드 S_{new} 를 (1) 버리거나, (2) 부품들 모음에 추가하거나, (3) 완성된 테스트 코드로 출력하거나를 실행 결과에 따라 선택

번외: Randoop 테스트 코드 분류



잘못된/중복된 테스트 코드?

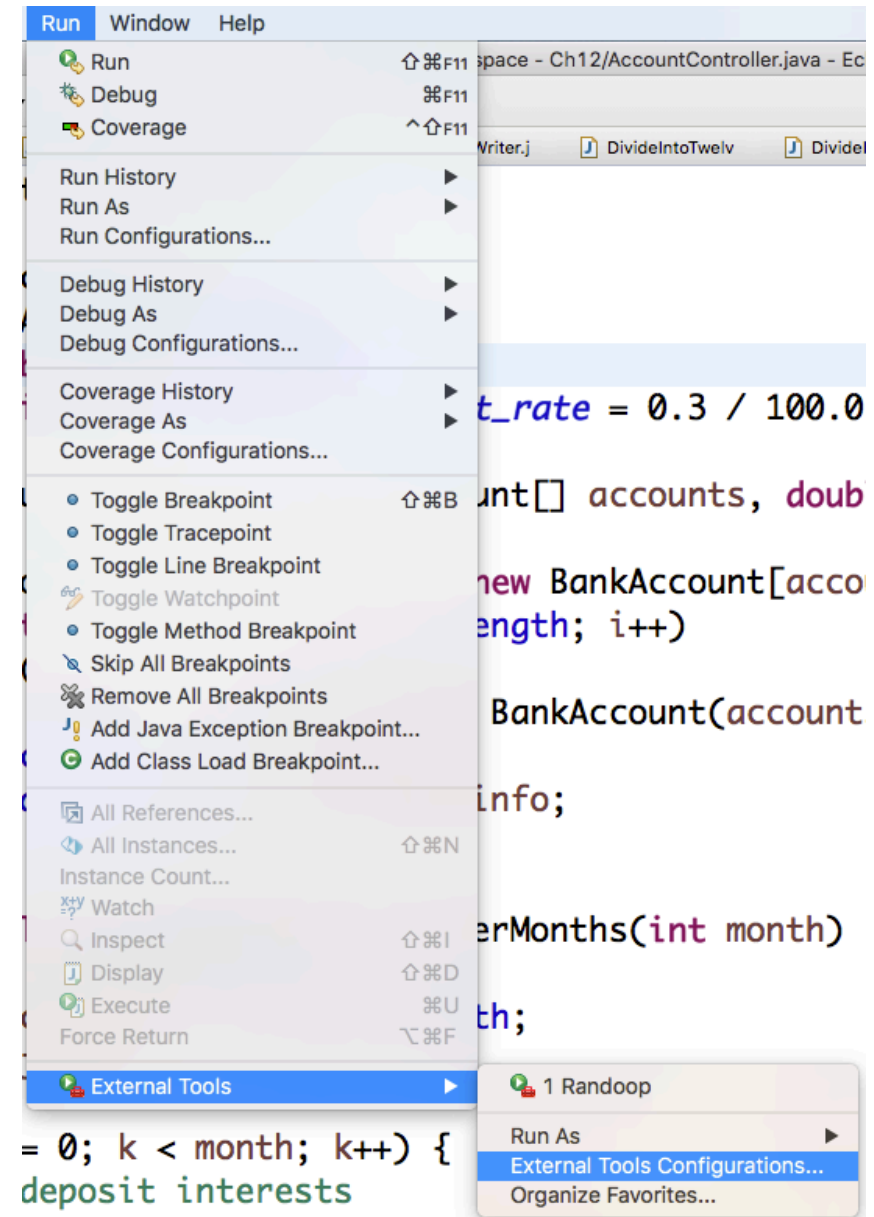
- 잘못된 테스트 코드: 테스트 대상 클래스를 테스트하기도 전에 죽는 코드 (예: 예외 발생 등...)
- 중복된 테스트 코드
 - 지금까지 생성한 테스트 코드들이 마지막으로 만들어내는 객체 보관
 - 위에 보관하고 있는 것과 동일한 (equals 메소드로 체크) 객체를 만들어내는 테스트 코드는 “중복”으로 판단되어 버려짐

Randoop으로 찾은 오류들

- 다음 자주쓰이는 라이브러리에 치명적인 오류들 검출
 - 자바 표준 라이브러리 (JDK)
 - Apache 라이브러리
 - 마이크로소프트 .Net 프레임워크
- 다운로드: <https://github.com/randoop/randoop/releases/download/v4.1.0/randoop-4.1.0.jar>
- 메뉴얼: <https://randoop.github.io/randoop/>

실행 방법

- 메뉴 “Run” → “External Tools” → “External Tools Configuration”
- 왼쪽 탭에 Program 선택, 상단 New 버튼 선택
- 다음 슬라이드 화면과 같이 설정



External Tools Configurations

Create, manage, and run configurations

Run a program

Name: Randoop

Location: /System/Library/Frameworks/JavaVM.framework/Versions/A/Commands/java

JDK 설치 위치

Browse Workspace... Browse File System... Variables...

Working Directory: \${container_loc}

Browse Workspace... Browse File System... Variables...

Arguments: **Randoop 다운로드 위치**

```
-cp ${container_loc}:/Users/woosuk/course/randoop/randoop-all-4.1.0.jar randoop.main.Main gentests --classlist=${container_loc}/target_classes.txt --checked-exception=ERROR --unchecked-exception=ERROR --output-limit=100
```

Variables...

Note: Enclose an argument containing spaces using double-quotes (").

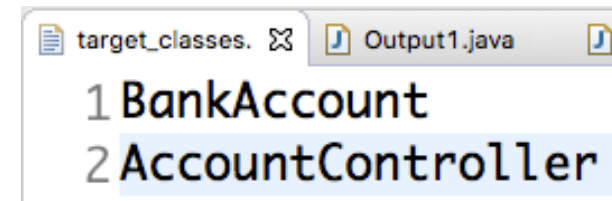
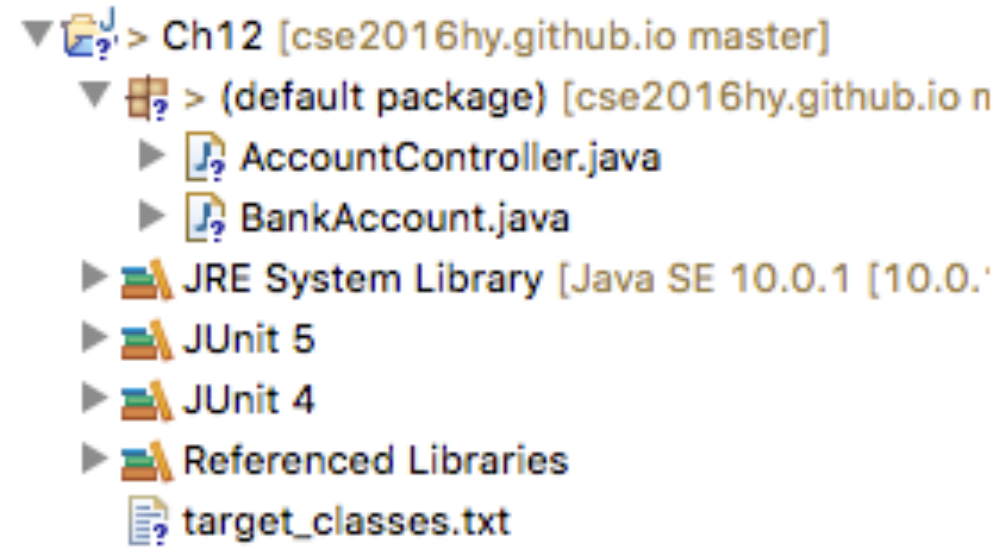
Filter matched 3 of 3 items

Revert Apply

Close Run

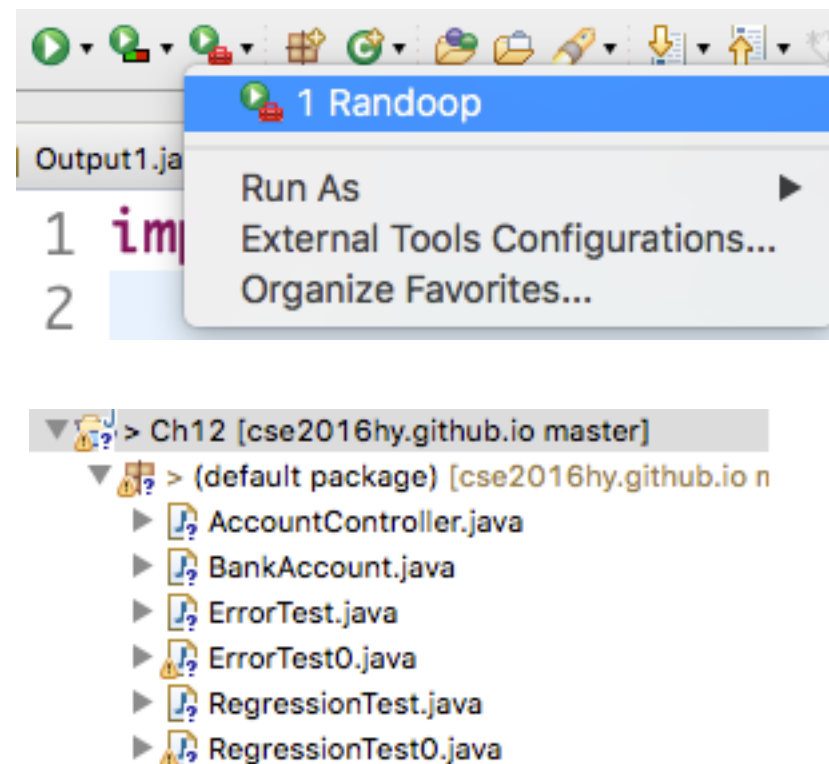
실행 방법

- 작업 중인 프로젝트 오른쪽 클릭
→ “New” → “File” → 파일이
름에 “target_classes.txt” 입
력
- 우측 프로젝트 explorer 창에
새 파일 target_classes.txt
생긴 것 확인 후, 파일 안에 테
스트할 클래스 이름들 입력. 예:
- BankAccount
AccountController



실행 방법

- 그림과 같이 Randoop 실행
- 프로젝트 우클릭 →
“Refresh” 클릭 시 새로운
*Test.java 파일들이 생성
되었음 확인
- ErrorTest*.java 우 클릭
후 오른쪽 클릭 → Run As
→ JUnit Test 선택



차례

- 소프트웨어 안전성 개요
- 동적 분석 (테스팅)
 - 수동 테스트
 - 자동 테스트 케이스 생성
- **정적 분석**
- 미래 기술
 - 프로그램 자동 합성

정적 분석 도구들

- Facebook Infer: 페이스북에서 개발된 널리 쓰이는 정적 분석 도구
 - 페이스북, UBER, Instagram, Spotify, Mozilla, ... 등 주요 IT 기업들에서 널리 사용되는 중
 - <https://fbinfer.com>
 - 방문 후 “TRY INFER IN YOUR BROWSER” 클릭하여 웹 브라우저 상에서 사용할 수 있음
 - 상단 메뉴 바 “Project” → “Add file” 로 소스파일 업로드 후 메뉴 바 “Actions” → “Analyze”
- SonarLint: <https://www.sonarlint.org>
- FindBugs: <http://findbugs.sourceforge.net>

정적 분석은 불안전 / 불완전

- 불안전 혹은 불완전한 이유: 안전하고 완전한 분석을 만드는 문제는 풀 수 없는 “멈춤 문제”로 귀결됨
- 멈춤 문제 (halting problem): 임의의 프로그램이 유한한 시간 안에 종료될지를 판단하는 문제
- 예: 잘못된 배열 접근 오류 (array out-of-bounds error) 검출 분석
- 멈춤 문제를 풀 수 있으면 안전&완전한 배열 접근 오류 검출기 도출 가능
 - 먼저 모든 프로그램 종료 지점을 무한루프 (while(true);) 로 바꿈
 - 그럼 프로그램은 무한히 실행되는 것이 보장됨
 - 모든 `a[i]` (배열 접근) 부분을 `(i >= 0 && i < a.length) ? a[i] : exit();` 로 바꿈
 - 만약 우리가 멈춤 문제를 풀 수 있다면 안전하고 완전한 배열 접근 오류 검출기를 만들 수 있음.

정적 분석은 불안전 / 불완전

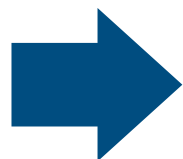
- 안전&완전한 배열 접근 오류 검출기를 만들 수 있으면 멈춤 문제를 풀 수 있음
 - 모든 `a[i]` (배열 접근) 부분을 `(i >= 0 && i < a.length) ? a[i] : exit();` 로 바꿈
 - 이제 잘못된 배열 접근이 없음이 보장됨
 - 모든 프로그램 종료 지점 앞에 잘못된 배열 접근 추가 (예: `a[a.length + 10]`)
 - 만약 분석기가 잘못된 배열 접근을 찾아낼 수 있다면 원본 프로그램이 종료한다는 것도 알 수 있음
 - 즉, 그러한 분석기를 이용하여 멈춤 문제를 풀 수 있음.
- 결론: 안전&완전한 배열 접근 오류 검출기를 만들기 = 멈춤 문제 풀기

차례

- 소프트웨어 안전성 개요
- 동적 분석 (테스팅)
 - 수동 테스트
 - 자동 테스트 케이스 생성
- 정적 분석
- **미래 기술**
 - 프로그램 자동 합성

프로그램 자동 생성 (Program Synthesis)

Specification



Synthesizer



Program

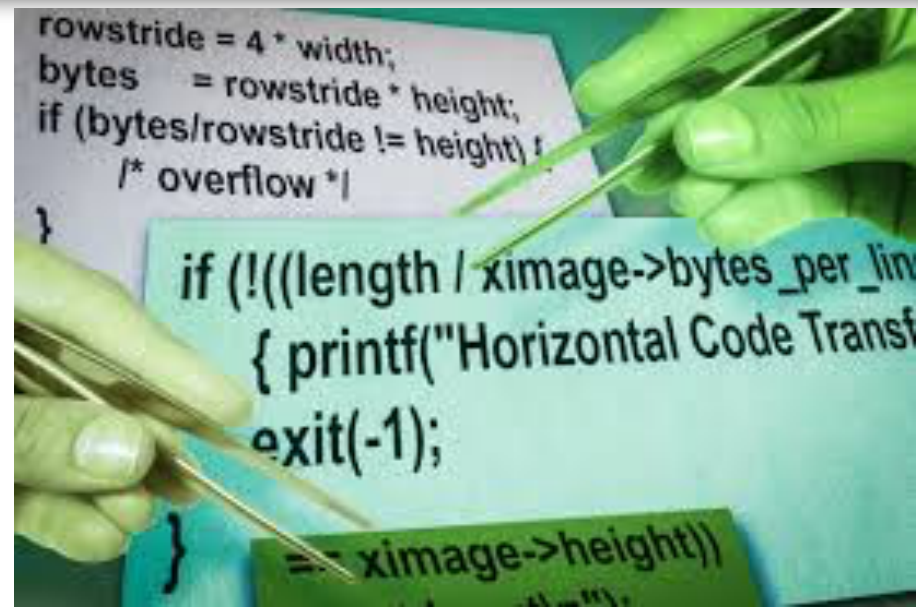


- 컴퓨터가 사용자가 원하는 프로그램의 요구조건을 받아, 그것을 만족시키는 프로그램을 자동으로 만들어내는 일
- 목표: 깊은 프로그래밍 지식을 갖추지 않은 사람들도 손쉽게 프로그램을 작성할 수 있게 하여 컴퓨팅 자원 이용 효율 극대화
- 공개된 대규모 오픈소스 프로그램들, 전산 논리학의 발달, 인공지능 기술 발달 등에 힘입어 최근 비약적 발달

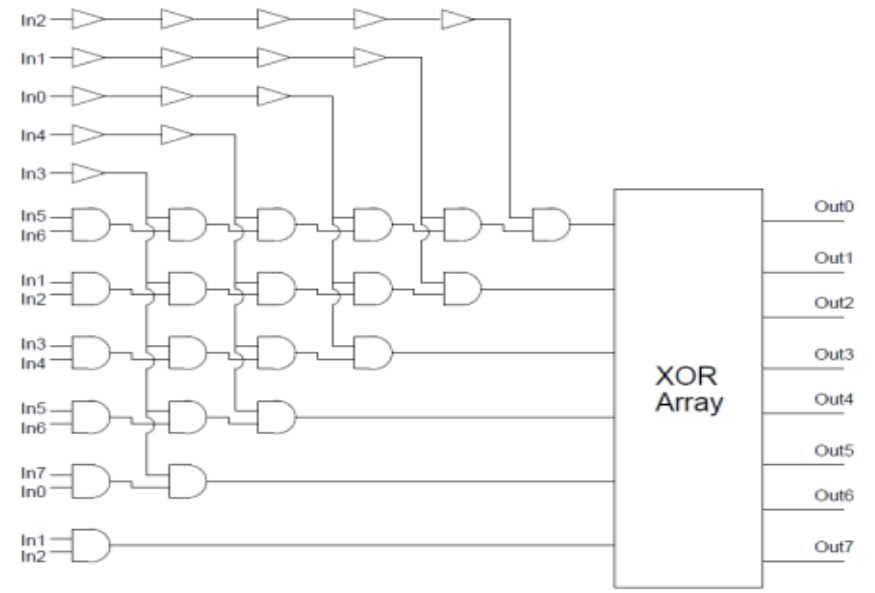
프로그램 합성 예

	A	B	C	D
1	Number	Phone		
2	02082012225	020-8201-2225		
3	02072221236	020-7222-1236		
4	0208123654	020-8123-654		
5	0207236523	020-7236-523		
6	02082012222	020-8201-2222		
7				
8				

End-user Programming (e.g., Excel Flash Fill)



Program Repair



Circuit Transformation

프로그램 합성 예

- Bayou 시스템: <http://askbayou.com>
- 작성하고자 하는 함수의 빈 몸통에 사용할 자바 API 함수 및 API 클래스 타입 등을 입력하면 자동으로 자바 프로그램 생성
- <https://info.askbayou.com/how-to-use-bayou/>

The screenshot displays the Bayou web application interface. At the top left is the Bayou logo and the tagline "Deep Generation of API Usage Idioms". On the right are "Info" and "About" links. The main interface is split into two panels:

Enter Your Code Here: This panel contains a text area with the following Java code snippet:

```

1 import java.io.*;
2 import java.util.*;
3 public class TestIO {
4     void read(File file) {
5         {
6             /// call:readLine type:FileReddler type:BufferedReader
7         }
8     }
9 }
10

```

Below the text area is a dropdown menu set to "File Read" and a search icon.

Results: This panel shows the code generated by the system:

```

1 import java.io.*;
2 import java.util.*;
3 import java.io.IOException;
4 import java.io.File;
5 import java.io.BufferedReader;
6 import java.io.FileReader;
7 import java.io.FileNotFoundException;
8
9 public class TestIO {
10     void read(File file) {
11         {
12             FileReader fr1;
13             BufferedReader br1;
14             String s1;
15             try {
16                 fr1 = new FileReader(file);
17                 br1 = new BufferedReader(fr1);
18                 while ((s1 = br1.readLine()) != null) {}
19                 br1.close();
20             } catch (FileNotFoundException _e) {}
21             } catch (IOException _e) {}
22         }
23         return;
24     }
25 }
26 }
27

```