29th Annual International Conference on Computer Science and Software Engineering

CASCON

On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study

Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, Fabio Palomba

RIT Rochester Institute of Technology

ÉTS ÉCOLE DE TECHNOLOGIE SUPÉRIEURE Le génie pour l'industrie Université du Québec

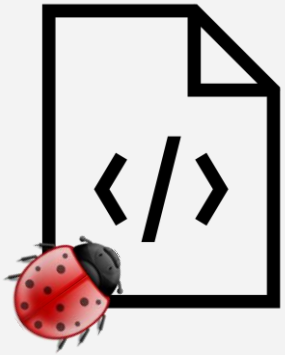
University of Zurich[UZH]

1. Introduction

Software maintenance is not cheap!

- A high-quality system need not be necessarily maintenance-friendly
- Systems built using poor design/coding practices can meet functional requirements
- In the long run, such events impact software maintenance - and maintenance is not cheap!
  - Maintenance consumes 50% to 80% of resources

Towards maintenance-friendly code



- Researchers and industry have defined and created approaches and tools to detect code in need of refactoring
  - Design/code smells - Cohesion, Coupling, God Class, etc.
  - Tools - FindBugs, PMD, Checkstyle, etc.
- Smells make code harder to understand and make it more prone to bugs and changes
- Research and tools have been primarily on production code

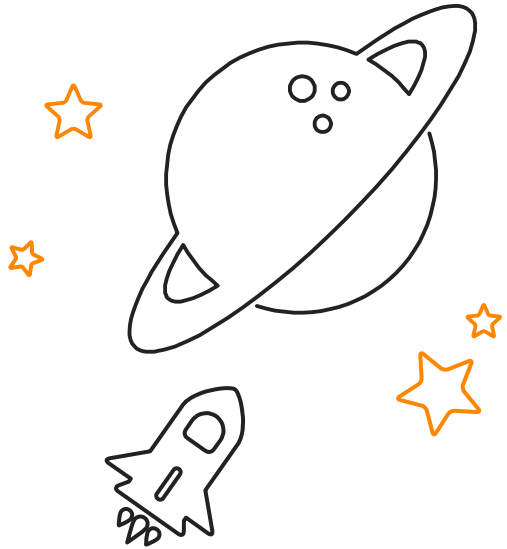
Test Smells

- Test code, like production code, is subject to smells
- Formally introduced in 2001 with 11 smell types
- Inclusion of additional smell types through the years, analysis of their evolution and longevity, and elimination patterns
- Tools to detect specific smell types
- Studies on traditional Java applications

2.6 million apps available on Google Play as of Q4 2018

Objective

Insight into the *unit testing practices of Android app developers* with the aim of providing developers a mechanism to *improve unit testing code*



Contribution

Expansion of Test Smell Types

Open-Source Test Smell Detection Tool

Understanding of Test Smells in Android apps

Replication Package Availability

Research Questions

RQ 01

How likely are Android apps to contain unit test smells?

- Are apps, that contain a test suite, prone to test smells?
- What is the frequency and distribution of test smells in apps?
- How does the distribution of smell types in Android apps compare against traditional Java applications?

RQ 02

What is the general trend of test smells in Android apps over time?

- When are test smells first introduced into the project?
- How do test smells exhibited by the apps evolve over time?

2.
Test Smells

Proposed Test Smells

- Conditional Test Logic
- Constructor Initialization
- Default Test
- Duplicate Assert
- Empty Test
- Exception Handling
- Ignored Test
- Magic Number Test
- Redundant Print
- Redundant Assertion
- Sleepy Test
- Unknown Test

Practicability

Are our proposed smells indicative of problems?

120
smelly unit test files

100
software systems

120
software developers

41.7%
response rate

Conditional Test Logic

- Conditions within the test method will alter the behavior of the test and its expected output
- Developers agree on the negative impact on code comprehension
- However, outright removal may not always be applicable – decide on a "case by case basis"

```
/*
    ** Test method contains multiple control statements **
*/
@Test
public void testSpinner() {
    /** Control statement #1 ** */
    for (Map.Entry<String, String> entry : sourcesMap.entrySet()) {
        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        /** Control statement #2 ** */
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            /** Control statement #3 ** */
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                AnswerObject answer = new AnswerObject(entry.getValue(), "",
                        new CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.
                        application, answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data
                        .size());
                /** Control statement #4 ** */
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.
                            data.get(i));
                }
            }
        }
    }
}
```

> "I actually have no idea why that for loop is there. It doesn't do anything but run the test 1000 times, and there's no point in that. I'll remove it."

Constructor Initialization

- Initialization of fields should be in the setUp() method (i.e., test fixtures)
- Most developers are aware of test fixtures
- Developers unanimously agree on using test fixtures
- Reasons for not using test fixtures include “laziness” and being “sloppy”

```
public class TagEncodingTest extends BrambleTestCase {
    private final CryptoComponent crypto;
    private final SecretKey tagKey;
    /* ** Constructor initializing field variable ** */
    public TagEncodingTest() {
        crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
        tagKey = TestUtils.getSecretKey();
    }
    @Test
    public void testKeyAffectsTag() throws Exception {
        for (int i = 0; i < 100; i++) {
            .....
            /* ** Field variable utilized in test method ** */
            crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
            assertTrue(set.add(new Bytes(tag)));
    .....
}
```

> “I have already made this change since you pointed it out so the code is clearer now”

Default Test

- Default test class meant to serve as an example
- Should either be removed
- A test-first approach will force developers to remove the file
- Unanimous agreement among developers that the file “serves no concrete purpose” and that it may lead to confusion

```
/*
    ** Default test class created by Android Studio **
*/
public class ExampleUnitTest {
    /*
        ** Default test method created by Android Studio **
    */
    @Test
    public void addition_isCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }

    /*
        ** Actual test method **
    */
    @Test
    public void shareProblem() throws InterruptedException {
        .....
        Observable.just(200)
        .subscribeOn(Schedulers.newThread())
        .subscribe(begin.asAction());
        begin.set(200);
        Thread.sleep(1000);
        assertEquals(beginTime.get(), "200");
        .....
    }
    .....
}
```

“Removed useless example unit test”

Duplicate Assert

- The same condition is tested multiple times within the same test method
- The name of the test method should be an indication of the test
- Mixed responses - some developers preferred to split the assertion statement into separate methods

```
@Test
public void testXmlSanitizer() {
    .....
    valid = XmlSanitizer.isValid("Fritz-box");
    /* ** Assert statements are the same ** */
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    /* ** Assert statements are the same ** */
    assertEquals("Minus is valid", true, valid);
       System.out.println("Minus test - passed");
    .....
}
```

> “I might enforce it on some bigger projects”

Empty Test

- When a test method has no executable statements
- JUnit will indicate that the test passes even if there are no executable statements present in the method body
- Unanimous agreement among developers that such test methods should be removed from the test suite

```
/* ** Test method without executable statements ** */
public void testCredGetFullSampleV1() throws Throwable{
//  ScrapedCredentials credentials =  innerCredTest(FULL_SAMPLE_v1);
//  assertEquals("p4ssw0rd", credentials.pass);
//  assertEquals("user@example.com",credentials.user);
}
```

“Yes definitely should be removed”

Exception Handling

- Passing or failing of a test method is explicitly dependent on the production method throwing an exception
- Developers should utilize JUnit's exception handling features to automatically pass/fail

```
@Test
public void realCase() {
    .....
    /* ** Fails the test when an exception occurs ** */
    try {
        a.compute();
    } catch (CalculationException e) {
    Assert.fail(e.getMessage());
    }
    Assert.assertEquals("233.2405", this.df4.format(a.getResults().get(0).
          getUnknownOrientation()));
    .....
```

Ignored Test

- Ignored test methods result in overhead with regards to compilation time and an increase in code complexity and comprehension
- Mixed responses - investigate problems or serve as a means for new developers “to understand behavior”

```
@Test
/* ** This test will not be executed due to the @Ignore annotation ** */
@Ignore("disabled for now as this test is too flaky")
public void peerPriority() throws Exception {
    final List<InetSocketAddress> addresses = Lists.newArrayList(
        new InetSocketAddress("localhost", 2000),
        new InetSocketAddress("localhost", 2001),
        new InetSocketAddress("localhost", 2002)
    );
    peerGroup.addConnectedEventListener(connectedListener);
    .....
}
```

“would not tolerate to have ignored tests in the code”

Magic Number Test

- Test method contains unexplained and undocumented numeric literals
- Developers agree that the use of constants over magic numbers improve code readability/understandability
- Not a blanket rule - a constant should only be used so that its "name adds useful information"

```
@Test
public void testGetLocalTimeAsCalendar() {
    Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D),
            Calendar.getInstance());
    /* ** Numeric literals are used within the assertion statement ** */
    assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
    assertEquals(30, localTime.get(Calendar.MINUTE));
}
```

> "If the numerical value has a deeper meaning (e.g. flag, physical constant, enum value) then a constant should be used."

Redundant Assertion

- Assertion statements that are either always true or false
- Common reason for the existence of this smell is due to developer mistakes
- Developers confirmed that such code “is not needed”, “bad style” and “should probably be removed”
- Might exist to support edge cases

```
@Test
public void testTrue() {
    /* ** Assert statement will always return true ** */
    assertEquals(true, true);
}
```

Redundant Print

- Unit tests are executed as part of an automated script
- They can consume computing resources or increase execution time
- Unanimous agreement that print statements do not belong in test suites
- A common reason for the existence of this smell is due to developer debugging

```
@Test
public void testTransform10mNEUAndBack() {
    Leg northEastAndUp10M = new Leg(10, 45, 45);
    Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
    /* ** Print statement does not serve any purpose ** */
    System.out.println("result = " + result);
    Leg reverse = new Leg(10, 225, -45);
    result = transformer.transform(result, reverse);
    assertEquals(Coord3D.ORIGIN, result);
}
```

> “a waste of resources (cpu+disk space)”

Sleepy Test

- Explicitly causing a thread to sleep can lead to unexpected results as the processing time for a task differs when executed in various environments and configurations
- Developers confirmed that there are risks (i.e., inconsistent results) involved with causing a thread to sleep

```
public void testEdictExternSearch() throws Exception {
    .....
    assertEquals("Searching", entry.english);
    /* ** Forcing the thread to sleep ** */
    Thread.sleep(500);
    final Intent i2 = getStartedActivityIntent();
    .....
}
```

> “ the alternative requires more code ”

Unknown Test

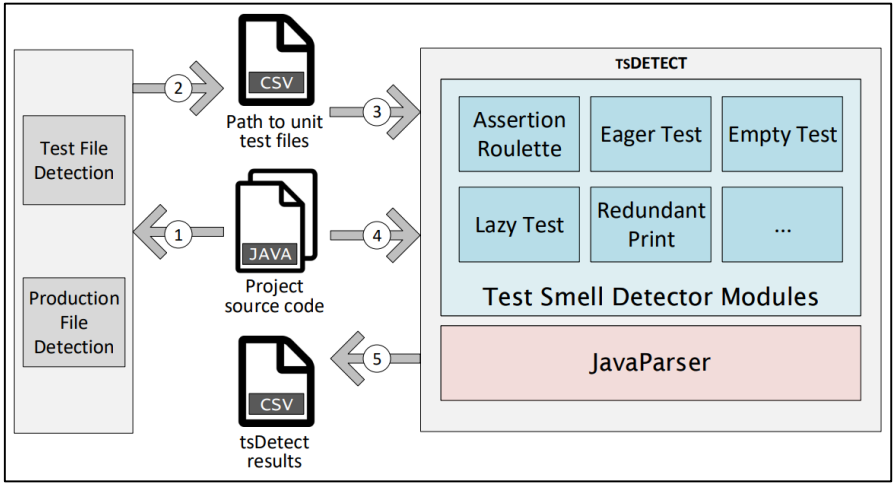
- The assertion statement helps to indicate the purpose of the test
- JUnit will show the test method as passing
- Majority of the developers are in favor of having assertion statements in a test method
- Missing assertions were due to mistakes

```
/* ** Test method without an assertion statement and non-descriptive name  ** */
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

> “It looks like just sloppy coding there. I'll look to fix that test”

TSDetect

- Open-source, Java-based, static analysis
- Available as a standalone jar and requires a list of file paths as input
- Utilizes an abstract syntax tree to parse and detect test smells
- Detects 19 test smells (12 proposed + 7 existing)
- Average F-Score of 96.5%



High-level architecture of TSDetect

3.
Experiment Methodology

Data Collection Phase

2,011
cloned apps

1,037,236
commits

6,379,006
java files affected by commits

+3.5 GB
java files collected

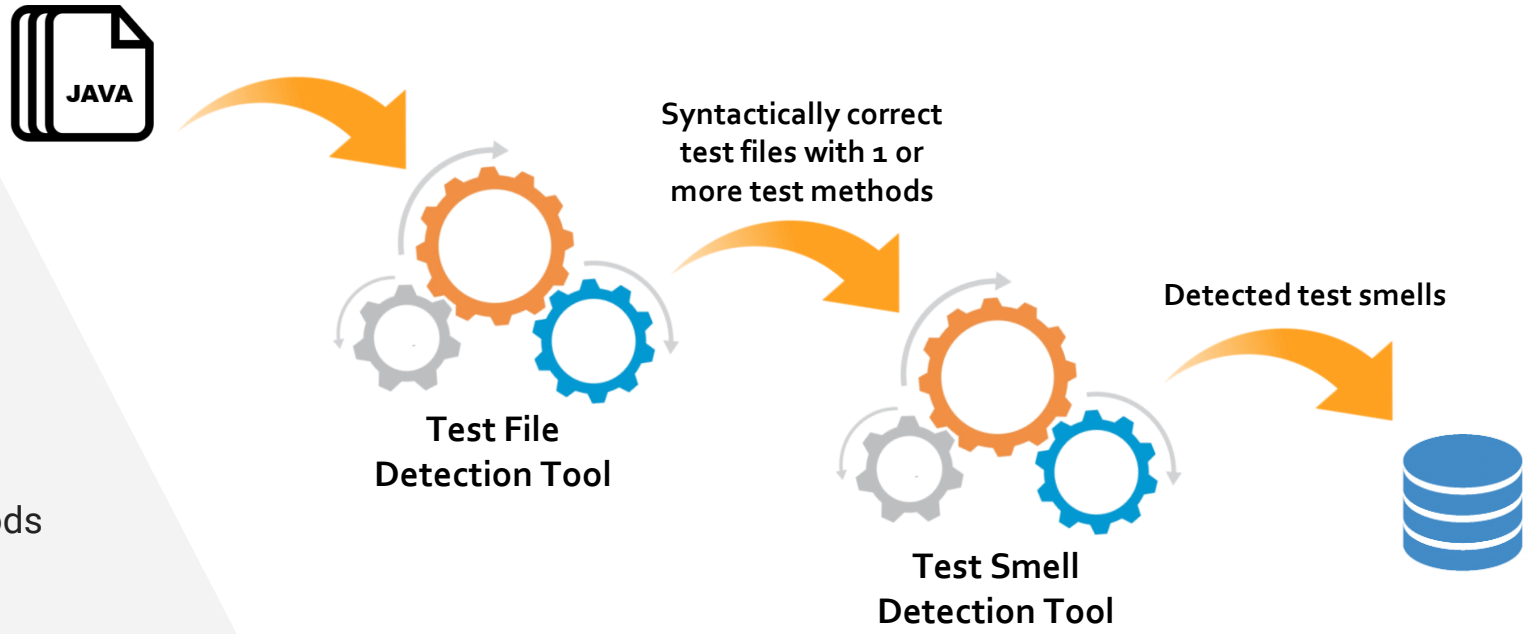
Detection Phase

656
analyzed apps

206,598
detected test files

1,187,055
analyzed test methods

175,866
test files with 1 or more smells



4.
Analysis & Discussion

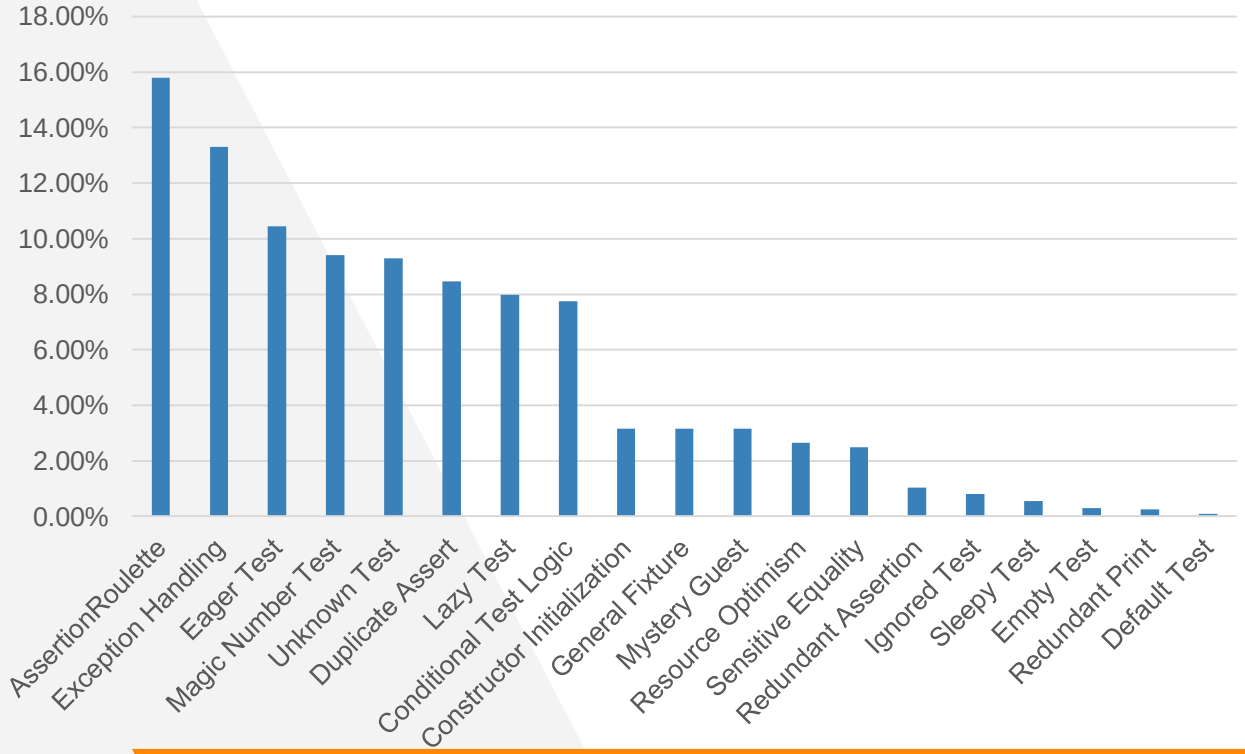
RQ1 – Test Smell Occurrence

Test Smell Occurrence & Distribution

- 97% of the analyzed apps contained test smells
- Assertion Roulette occurred the most (in over 50% of the analyzed apps and test files)
- All smell types had a high co-occurrence with Assertion Roulette
- Similar distribution of test smells between Android and non-Android applications

RQ1 – Test Smell Occurrence

Smell Type Distribution



Smell Type Occurrence

Smell Type	Smell Exhibition In Apps	Files
Assertion Roulette	52.28%	58.46%
Conditional Test Logic	37.32%	28.67%
Constructor Initialization	20.47%	11.70%
Default Test	42.20%	0.32%
Duplicate Assert	31.81%	31.33%
Eager Test	42.99%	38.68%
Empty Test	16.38%	1.08%
Exception Handling	84.57%	49.18%
General Fixture	25.51%	11.67%
Ignored Test	15.28%	3.00%
Lazy Test	39.06%	29.50%
Magic Number Test	77.01%	34.84%
Mystery Guest	36.38%	11.65%
Redundant Assertion	12.91%	3.87%
Redundant Print	14.02%	0.92%
Resource Optimism	15.75%	9.79%
Sensitive Equality	21.10%	9.19%
Sleepy Test	12.60%	2.04%
Unknown Test	47.09%	34.38%

RQ1 – Test Smell Occurrence

Smell Type Co-Occurrence

Smell Type	ASR	CTL	CNI	DFT	EMT	EXP	GFX	MGT	RPR	RAS	SEQ	SLT	EGT	DAS	LZT	UKT	IGT	ROP	MNT
ASR		31%	9%	0%	1%	49%	13%	13%	1%	3%	11%	2%	54%	46%	37%	23%	3%	13%	52%
CTL	62%		18%	0%	2%	58%	14%	25%	2%	7%	9%	5%	44%	39%	33%	46%	6%	20%	40%
CNI	43%	44%		0%	1%	84%	12%	22%	1%	3%	3%	6%	32%	24%	24%	57%	2%	12%	18%
DFT	0%	0%	0%		1%	99%	0%	23%	1%	0%	0%	0%	0%	0%	0%	2%	0%	0%	76%
EMT	69%	45%	10%	0%		42%	28%	8%	0%	0%	4%	1%	35%	32%	18%	100%	2%	2%	47%
EXP	58%	34%	20%	1%	1%		15%	19%	1%	5%	6%	4%	35%	32%	32%	40%	3%	18%	39%
GFX	66%	35%	12%	0%	3%	63%		10%	1%	1%	10%	3%	49%	42%	47%	43%	3%	8%	38%
MGT	67%	61%	22%	1%	1%	79%	10%		1%	3%	5%	4%	42%	40%	29%	46%	2%	63%	42%
RPR	46%	74%	7%	0%	1%	46%	19%	6%		1%	9%	1%	25%	22%	21%	61%	2%	5%	32%
RAS	45%	50%	8%	0%	0%	70%	4%	10%	0%		2%	3%	46%	14%	40%	4%	8%	7%	40%
SEQ	71%	28%	4%	0%	0%	34%	13%	6%	1%	1%		2%	48%	44%	35%	20%	3%	3%	52%
SLT	60%	67%	36%	0%	0%	100%	18%	20%	0%	5%	9%		48%	38%	31%	53%	5%	14%	26%
EGT	82%	33%	10%	0%	1%	45%	15%	13%	1%	5%	11%	3%		46%	61%	19%	1%	11%	49%
DAS	86%	36%	9%	0%	1%	51%	16%	15%	1%	2%	13%	2%	57%		44%	26%	3%	13%	60%
LZT	72%	32%	10%	0%	1%	53%	19%	11%	1%	5%	11%	2%	79%	47%		26%	1%	9%	47%
UKT	39%	38%	19%	0%	3%	57%	15%	16%	2%	0%	5%	3%	21%	24%	22%		7%	14%	25%
IGT	50%	53%	7%	0%	1%	49%	10%	6%	1%	10%	8%	3%	19%	32%	13%	75%		6%	35%
ROP	77%	60%	15%	0%	0%	92%	10%	75%	0%	3%	3%	3%	44%	41%	26%	48%	2%		45%
MNT	88%	33%	6%	1%	1%	55%	13%	14%	1%	4%	14%	2%	55%	54%	40%	25%	3%	13%	

Abbreviations:

ASR = Assertion Roulette | CTL = Conditional Test Logic | CNI = Constructor Initialization | DFT = Default Test | EMT = Empty Test | EXP = Exception Handling | GFX = General Fixture | MGT = Mystery Guest | RPR = Redundant Print | RAS = Redundant Assertion | SEQ = Sensitive Equality | SLT = Sleepy Test | EGT = Eager Test | DAS = Duplicate Assert | LZT = Lazy Test | UKT = Unknown Test | IGT = Ignored Test | ROP = Resource Optimism | MNT = Magic Number Test |

RQ2 – Test Smell Trend

Test Smell Introduction

- The first inclusion of a smelly file occurs approximately 23% of the way through the total app commits
- A test file is added with 3 smell types
- Assertion Roulette is the frequently the first smell type introduced
- Smells exhibited by a file remains constant throughout all updates to the file

5.
Conclusion

Summary

- Extended the catalog of known unit test smells
- Open source test smell detection tool
- A study of 656 Android apps showed a high prevalence of test smells in test suites
- Smells are introduced early on into the codebase and exist during the lifetime of the app
- Comprehensive project website: https://testsmells.github.io

Thanks!

https://testsmells.github.io