CMPT 276 Class 03: Testing

Dr. Jack Thomas

Simon Fraser University

Fall 2020

Why Test?

- You don’t actually need me to tell you why testing is important, come on.

Let’s pretend though

- You don’t want to be the engineer responsible for this:

Image Credit: Honda’s Asimo, .gif found at http://iruntheinternet.com/05148

Today’s Topics

1. What are common **types of testing**?
2. **Testing like a User**: through the UI.
3. **Testing like a Dev**: through the code.
4. **JUnit 5**: writing code to test code.
5. How to do effective **unit testing**?
6. What makes a good **bug report**?

Types of Testing

- Testing is meant to **find bugs** and **prove the product works**
- For software, testing can be broadly split into:
  1. **Acceptance Testing**
     - Test overall application's features
     - "Is the program acceptable to the customer?"
  2. **Unit Testing**
     - Test each class in isolation
     - "Does this class do anything wrong?"
- Testing can be done by a human (**manually**) or by code (**automatically**).

White Box vs. Black Box

- When creating tests, do you have access to the system’s code/design?
  - Knowing the code can help you make better, more complete tests
  - Not knowing the code can help you see the big picture and fix incorrect assumptions.
- **White Box Testing**
  - Can see source code when writing tests.
  - Also called clear box or glass box.
- **Black Box Testing**
  - Have no access to system internals.
  - Often for user interface testing.

Acceptance Testing

- Testing products from the customer’s perspective.
  - Are needed features included?
  - Do the features work as expected?
- Can generate acceptance tests from user’s requirements.

Ex: Requirements for a Scroll Bar

Requirements

- Scroll bar's slider shows the proportion of how much of the content is shown in the window.
- Scroll bar only visible when all content can not be shown in window at once.

Acceptance Tests

- With enough content to need scroll bar, double amount of *content* and slider should be half as tall.
- With enough content to need scroll bar, double *window height* and slider height should double.

Acceptance Testing in Practice

- Acceptance tests are often done manually.

Quality Assurance Tester Job:
- Writing Test Cases and Scripts based on business and functional requirements
- Executing high complexity testing tasks
- Recording and reporting testing task results
- Proactively working with project team members to improve the quality of project deliverables

- Acceptance testing can even roll into product deployment:
  – In **Alpha**, early builds of software are made available to some users under controlled circumstances.
  – In **Beta**, software gets deployed to customers pre-release.

http://www.bctechnology.com/jobs/Avocette-Technologies/127103/Quality-Assurance-Tester-(6-Month-Contract-and-Permanent).cfm

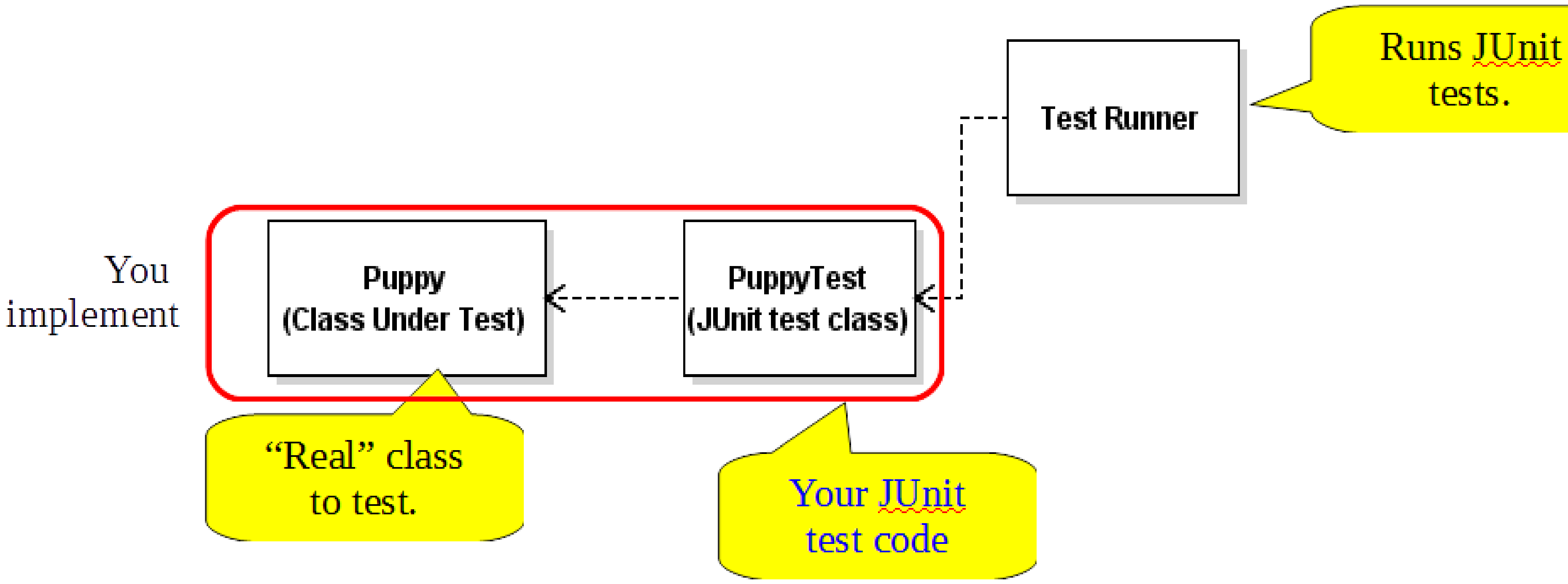
Unit Testing: Intro to JUnit

- **Unit Tests** mean testing a class in isolation.

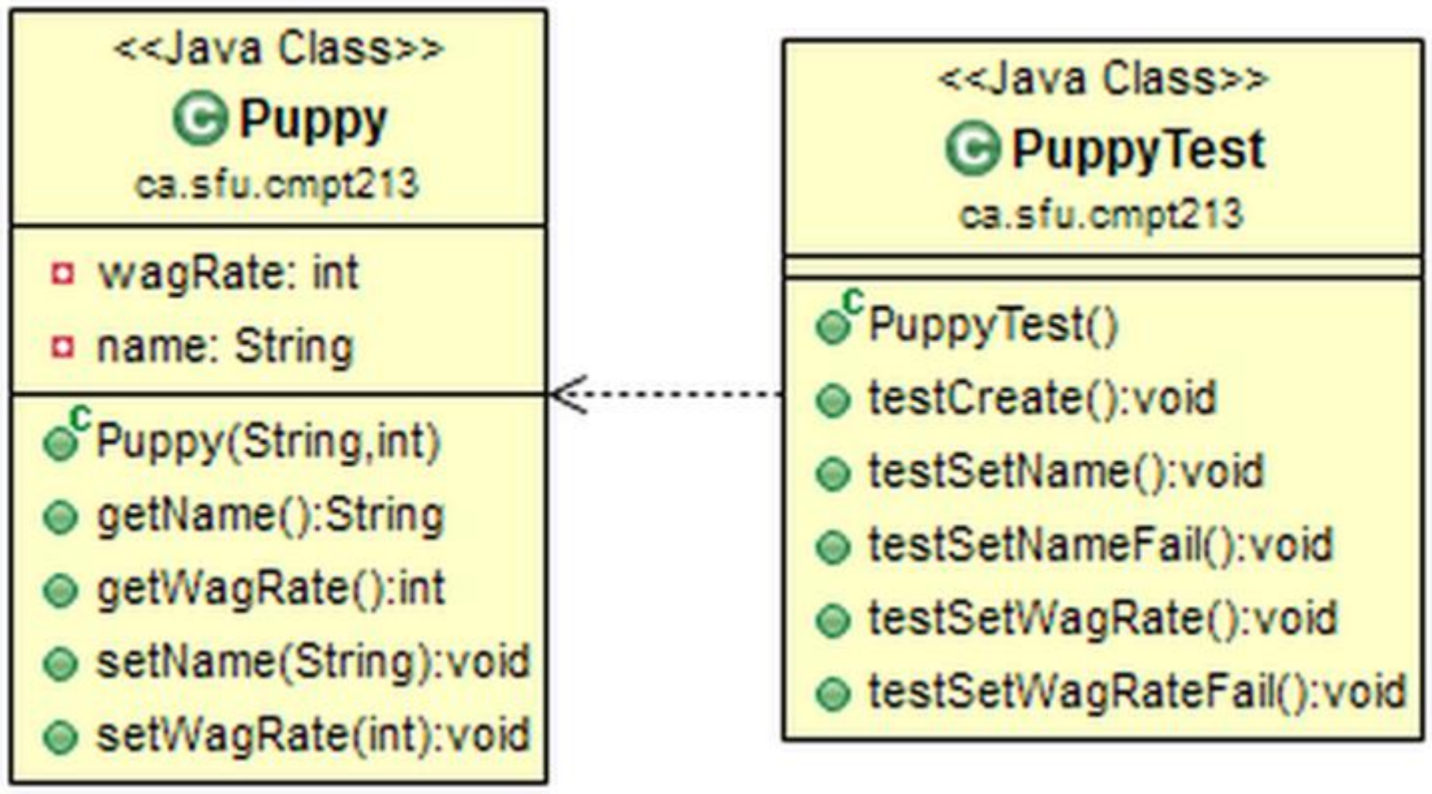
- Purpose:
  - Gives you reason to believe your code works.
  - Should test ~100% of a class.
  - Helps improve quality of code.
  - Supports aggressive refactoring because you can quickly check your code is correct.

JUnit Concept

- You create a **test class** which is paired with the class you want to test.
- The JUnit **test runner** executes your test class.



Basic JUnit Architecture



- JUnit: “Test Runner” executes methods with a @Test annotation.

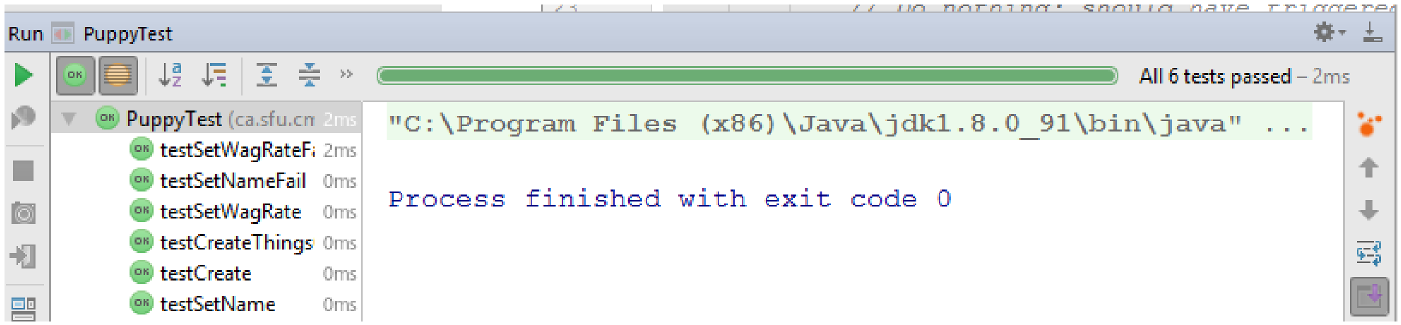
Junit 5 Example

```
package ca.cmpt276.junit5;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class PuppyTest {
        @Test
        void testCreate() {
                Puppy rover = new Puppy("Rover", 100);
                assertEquals("Rover", rover.getName());
                assertEquals(100, rover.getWagRate());
        }

        @Test
        void testSetName() {
                Puppy rover = new Puppy("Rover", 100);
                rover.setName("Fluffy");
                assertEquals("Fluffy", rover.getName());
        }

        //... more tests omitted.
}
```

- Test runner executes all methods with @Test annotaiton
- Tests are done using Junit's Asserts
- New instance of PuppyTest created for each JUnit test method: Behaviour of one does not affect the others

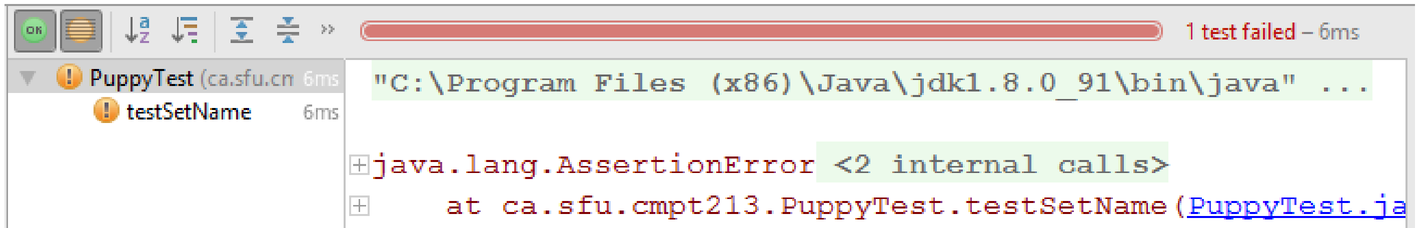
Test Runner

- Test runner executes @Test methods in test class.
- Displays results and a coloured bar:
  - Green-bar means all tests successful



```
"C:\Program Files (x86)\Java\jdk1.8.0_91\bin\java" ...

Process finished with exit code 0
```

  - Red bar means some tests failed



```
"C:\Program Files (x86)\Java\jdk1.8.0_91\bin\java" ...

java.lang.AssertionError <2 internal calls>
    at ca.sfu.cmpt213.PuppyTest.testSetName(PuppyTest.ja
```

JUnit 5 Asserts: Basics

```
public class JUnitAssertTest {
		@Test
		public void demoAssertEquals() {
				String name = "Dr. Evil";
				assertEquals("Dr. Evil", name);
		}
		@Test
		public void demoOtherAsserts() {
				int i = 10;
				assertEquals(10, i);
				assertTrue(i == 10);
				assertFalse(i == -5);
		}
		@Test
		public void demoAssertEqualsOnDouble() {
				double weight = (1 / 10.0);
				assertEquals(0.1, weight, 0.000001);
		}
		// Array support: assertArrayEquals()
}
```

Doubles have limited precision.
3rd arg is the “delta” to tolerate

JUnit 5 Asserts: Exceptions

```
public class JUnitAssertTest {
	private void throwOnNegative(int i) {
			if (i < 0) {
					throw new IllegalArgumentException();
			}
	}
	@Test
	void testThrows() {
			assertThrows(IllegalArgumentException.class, () -> {
					throwOnNegative(-1);
			});

	}
	@Test
	void testNoThrows() {
			throwOnNegative(1);
	}
}
```

Code likely in class being tested (shown here for simplicity)

Use to test exception throwing. Test fails unless it throws IllegalArgumentExecption

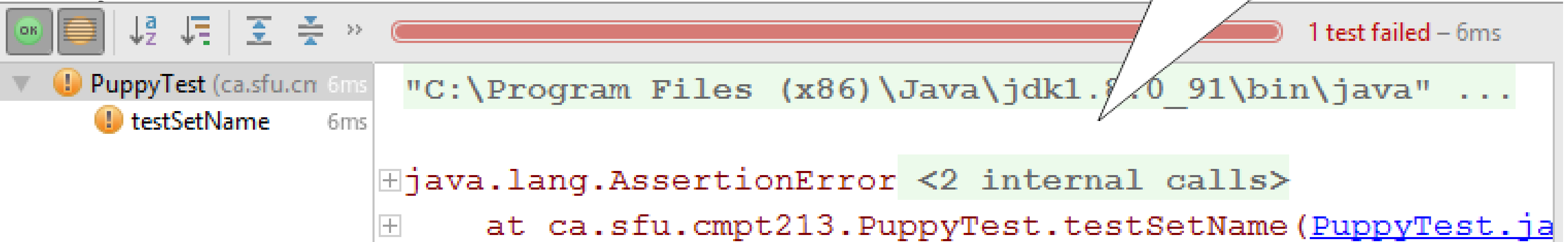
Tests that exception isn’t thrown when no error is present.

JUnit 5 Asserts: Disable

```
public class JUnitAssertTest {

        @Disable("DB does not yet support reconnecting.")
        @Test
        void testDBReconnect() {
// ... put your JUnit tests of the not-yet implemented code....
                fail();             // Automatic fail...
        }
```

Ignore the test so "to-be-done" style tests do not break testing.

Gives warning message to highlight that some tests not yet enabled.



```
"C:\Program Files (x86)\Java\jdk1.8.0_91\bin\java" ...

java.lang.AssertionError <2 internal calls>
    at ca.sfu.cmpt213.PuppyTest.testSetName(PuppyTest.ja
```

IntelliJ Demo

1. Create JUnit Test Class:
   1. Open class under test,
   2. Click class name, alt-enter --> Create Test
   3. Select JUnit 5, click OK
   4. Select ...\src\test..... folder
2. Execute Tests:
   1. Run --> Run... (alt-shift-F10)
   2. Select your JUnit test class.
3. Run test: Run --> Run...; select whole test file or individual tests

IntelliJ JUnit Video Tutorials: Basics:
https://www.youtube.com/watch?v=Bld3644bIAo&t
More: https://www.youtube.com/watch?v=xHk9yGZ1z3k&t

Effective Unit Tests

- Unit testing should be **automated**
- Design tests meant to prove a class will both:
  - Work with **expected normal inputs**.
  - Work with **extreme or invalid inputs**.
- Testing strategies:
  1. **Partition Testing**
     - Group together input values which are "similar"
     - Test based on these groupings.
  2. **Guideline-based Testing**
     - Follow guidelines to choose test cases.
     - Guidelines cover common programming errors.

1. Partition Testing

- Identify “groups”, as in regions of values in the input data and output results which should behave similarly.
  - Ex: Multiplying two integers.
    - Input: Positive vs. negative input values
    - Output: Positive vs. negative result.
- Each of these groups is an **equivalence class**:
  - The program behaves in an equivalent way for each group member.
- Test cases should be chosen from each partition.
  - test the extremes of the partitions (min and max)
  - test a middle value of the partition

Equivalence Classes

- Identify the equivalence classes (partitions):

```
/** Return a grade based on the percent:
 *   50 to 100 = 'P'
 *   0 to <50 = 'F'
 *   otherwise throw an exception.
 */
char assignGrade(int percent);
```

2. General Testing Guidelines

- Think like a programmer.
- Choose test inputs that will:
  - Generate all error messages
  - Cause buffers to overflow;
  - Force calculation result to be too large or small (overflow & underflow).
- For example, testing with Arrays:
  - Different # elements. Ex: 0, 1, 2, 100, 32000…
  - Put desired element in first, last, in the middle…

Code Coverage

- **Code Coverage** is the % of each class’s lines of code run during your test.
- The goal is ~100% Code Coverage
  - All lines of code executed at least once.
  - Quite hard to achieve (complex error cases, asserts, ..)
  - This should almost be the bare minimum: tests run each line perhaps only once!
- Let’s run a quick demo in IntelliJ.

Test Code Quality

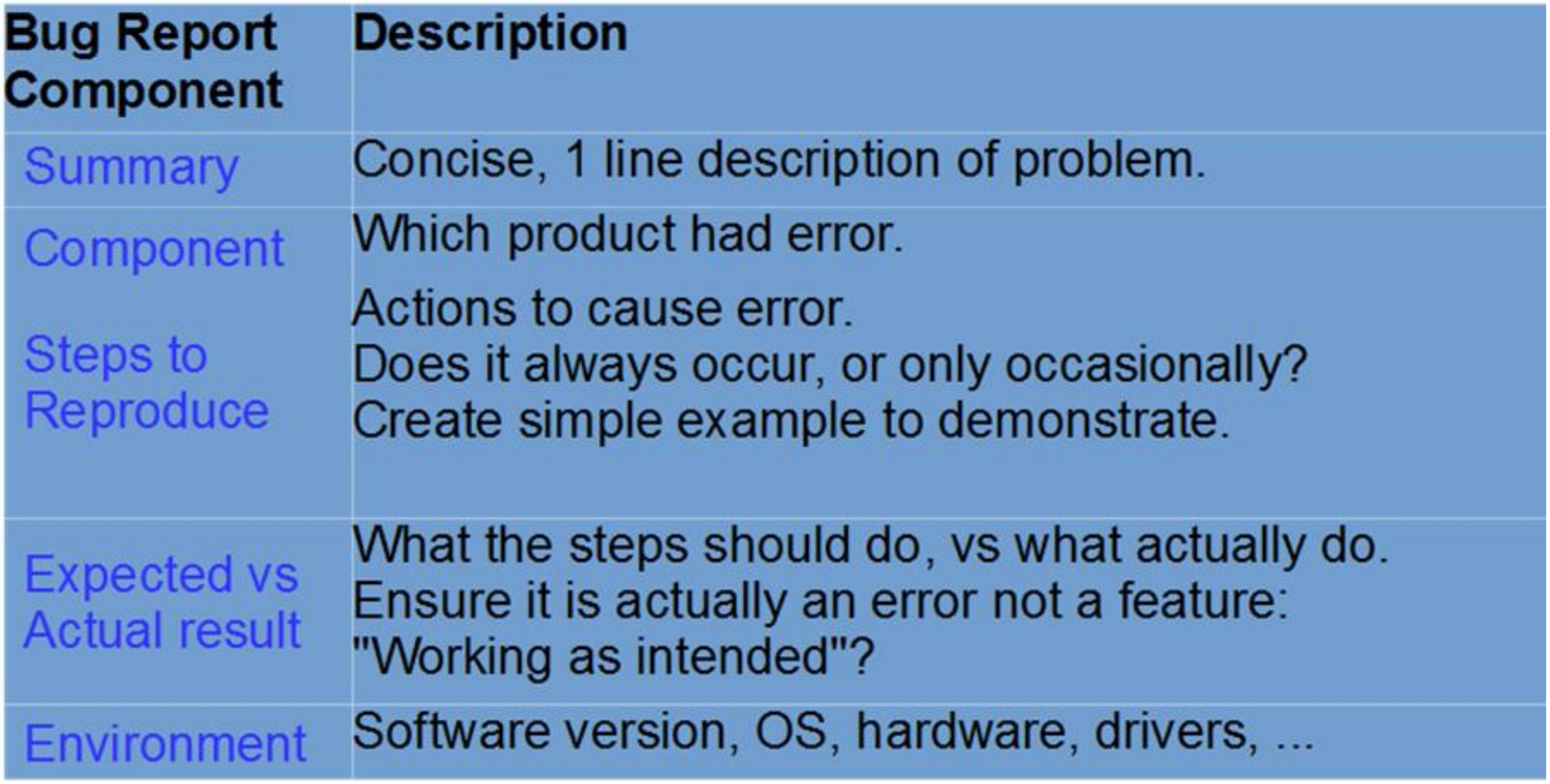
- Unit tests are an integral part of software development. Write tests to same code quality standards as the rest of the project.
  - Only possible if you don’t think of tests as throw-away or beneath your coding skill.
- **Good code** quality makes maintenance easier, and keeps tests current and relevant
- **Poor code** makes tests obsolete fast. Unreliable tests cause developers to lose trust.

Finding Many Bugs

- If you find a function which is quite buggy, don’t debug it: Rewrite the function!
  - Good unit testing only finds 30% of defects
  - A hacked together routine indicates poor understanding of its requirements:
    - If many bugs are discovered now, then many bugs will be encountered later!
- More tests cannot solve this problem:
- *Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often*. (McConnell 2004)

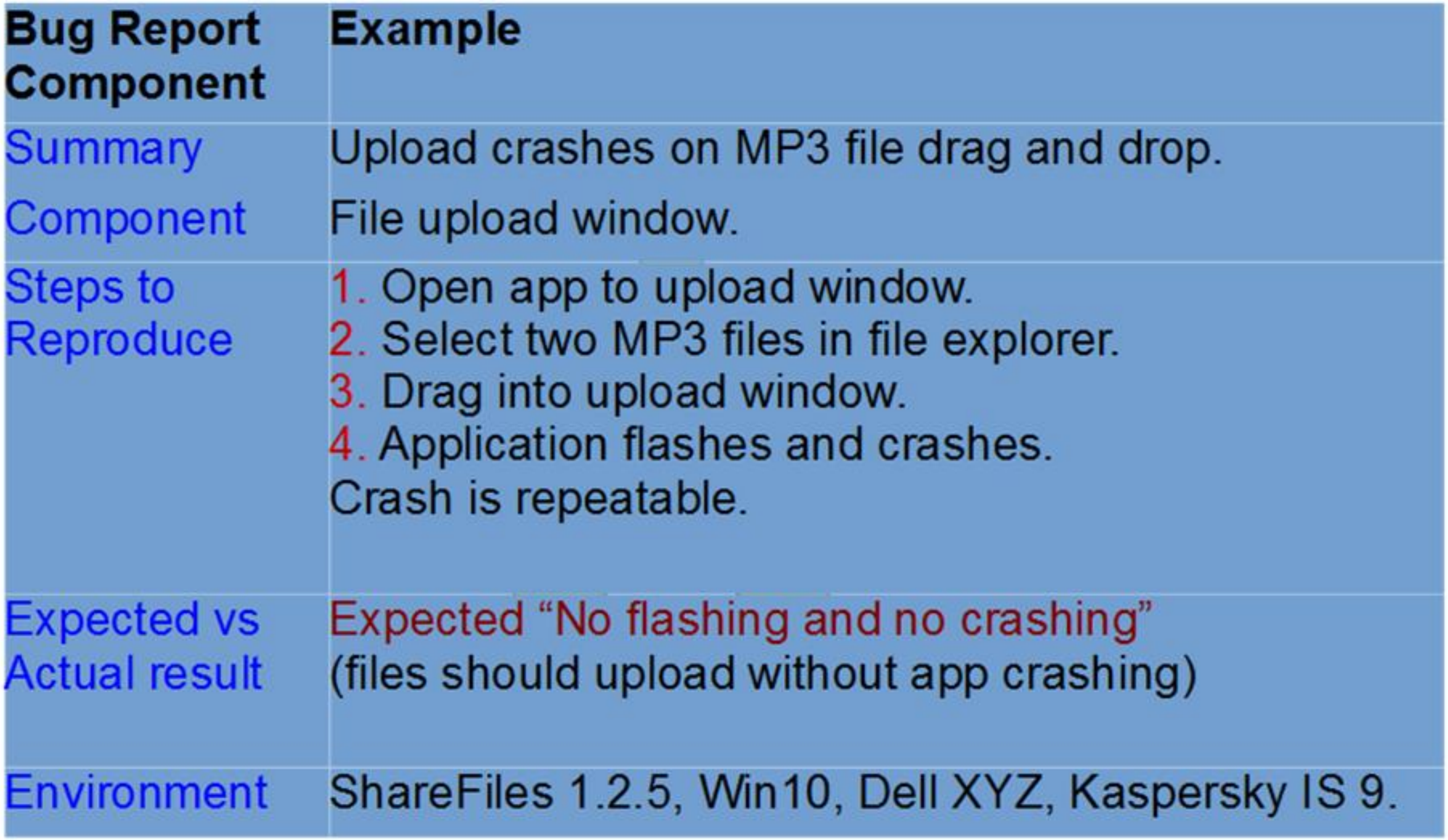
Bug Reports

- Submit a bug report when a defect is found.



Bug Report Component	Description
Summary	Concise, 1 line description of problem.
Component	Which product had error.
Steps to Reproduce	Actions to cause error. Does it always occur, or only occasionally? Create simple example to demonstrate.
Expected vs Actual result	What the steps should do, vs what actually do. Ensure it is actually an error not a feature: "Working as intended"?
Environment	Software version, OS, hardware, drivers, ...

Example Bug Report



Bug Report Component	Example
Summary	Upload crashes on MP3 file drag and drop.
Component	File upload window.
Steps to Reproduce	1. Open app to upload window. 2. Select two MP3 files in file explorer. 3. Drag into upload window. 4. Application flashes and crashes. Crash is repeatable.
Expected vs Actual result	Expected "No flashing and no crashing" (files should upload without app crashing)
Environment	ShareFiles 1.2.5, Win10, Dell XYZ, Kaspersky IS 9.

Bug Report Suggestions

- The better the bug report, the more likely the developer is to identify the problem and fix it.
- Example files:
  - For an office application, or a compiler, provide an example file which causes the problem.
- Screenshots:
  - Show, don't tell. It's usually easier to grasp.
  - elopers are often sceptical of problems they t reproduce. Proof helps your case!

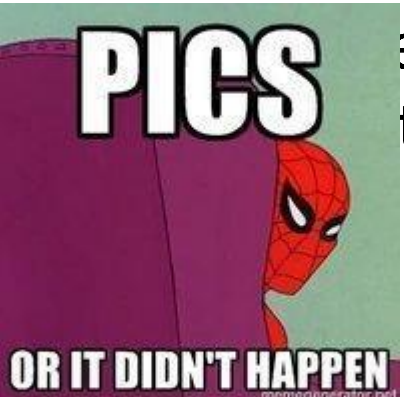
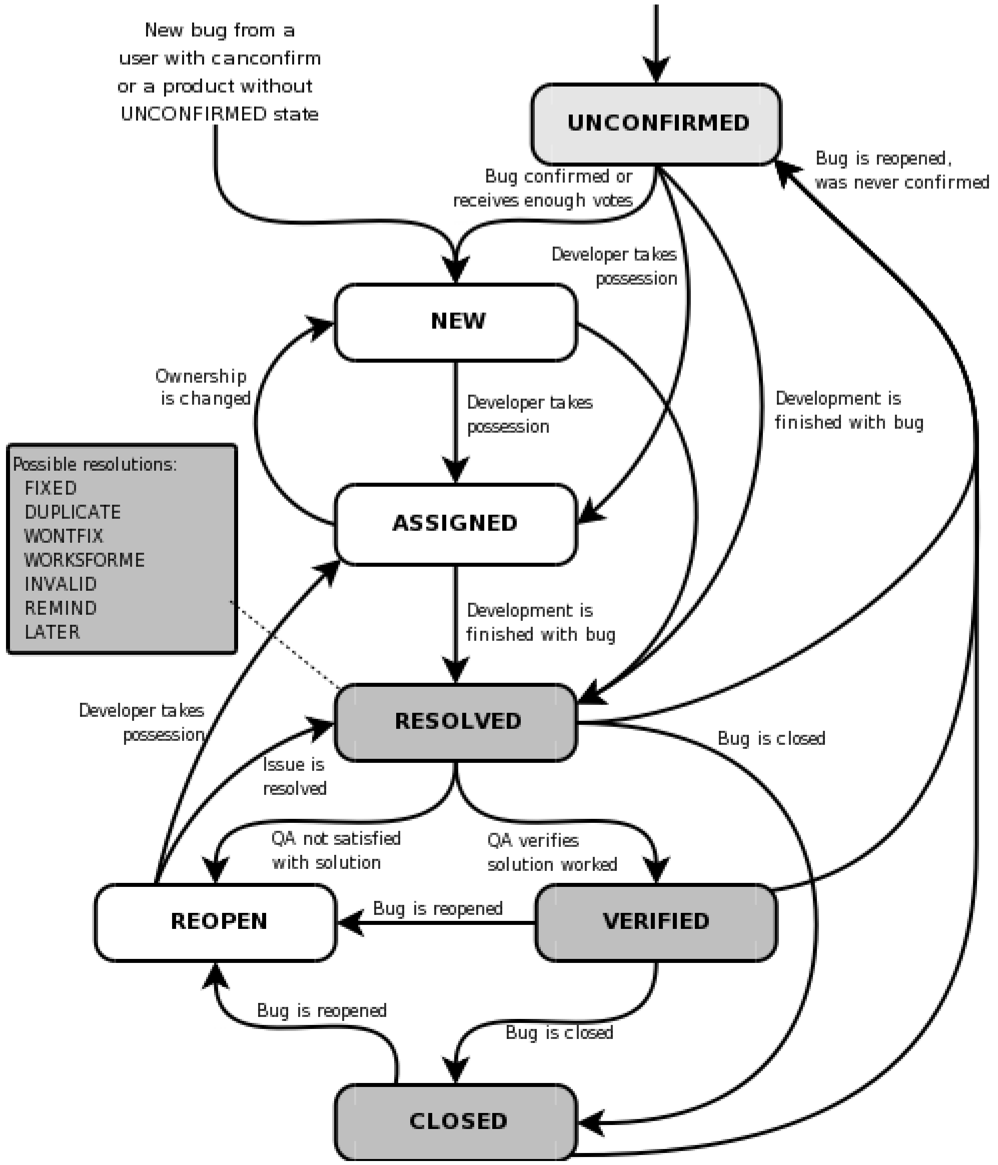


Image Credit: https://knowyourmeme.com/memes/pics-or-it-didnt-happer

Life Cycle Of A Bug

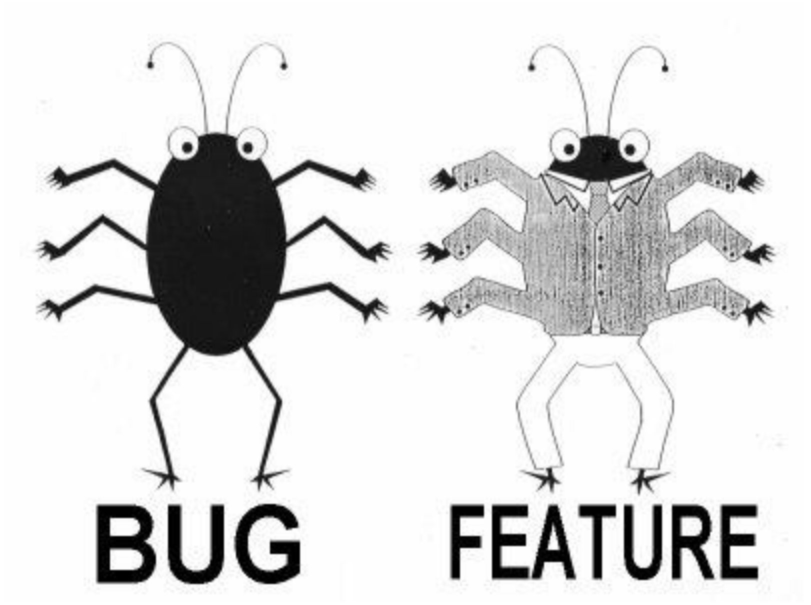


New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug is reopened, was never confirmed
Bug confirmed or receives enough votes
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
ASSIGNED
Possible resolutions: FIXED DUPLICATE WONTFIX WORKSFORME INVALID REMIND LATER
Development is finished with bug
RESOLVED
Developer takes possession
Issue is resolved
Bug is closed
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

Image Source: Bugzilla – lifecycle.

Bug Report Resolutions

- Some **resolutions**:
  1. Fixed
  2. Duplicate
  3. Won't Fix
  4. Cannot Reproduce
  5. Working as Intended
     - "ID-10-T"
     - "PLBKAC"
  6. Enhancement / feature request



Recap – Bugged About Testing

- **White-box**: Knowing the inside of the code.
- **Black-box**: Working only from input and output.
- **Acceptance Testing** for determining if your software's features satisfy the user.
- **Unit Testing** via **JUnit**, including how to use: assert...(), @Test, @Disable, and assertThrows().
- Good JUnit tests:
  - **Partition testing** using **equivalence classes**, or following standardized **test guidelines**.
  - Write **high-quality, maintainable code**.
- **Bug reports** should include a description, component, steps to reproduce, expectations, environment info.