CS 246: TESTING

Reid Holmes

With content from:
Meghan Allen (CS 310 @ UBC)
Gail Alverson (CS 403 @ UWashington)

LEARNING OUTCOMES

- Differentiate various testing tactics
- Understand different levels of testing
- Be able to construct effective unit tests
- Understand how to apply various testing tools & techniques

INTRODUCTION

"Test early, test often, test automatically"

[Pragmatic Programmer]

INTRODUCTION

"Testing can show the presence, but not the absences of errors"

[Dijkstra's law]

INTRODUCTION

"Testing can show the presence, but not the absences of errors"

[Dijkstra's law]

"If Debugging Is The Process Of Removing Bugs, Then Programming Must Be The Process Of Putting Them In."

[Dijkstra]

V & V

- Validation: "Did we build the right system?"
  - Demonstrates that the system meets its requirements.
- Verification: "Did we build the system right?"
  - Demonstrates that the behaviour is correct.

TESTING TACTICS

- Black box testing:
  - Tests parts of the system without knowledge of their internal structure.
  - Simulates a "customer" experience (at the API or UI level)
  - Test as much specified behaviour as possible
- White-box testing:
  - Tests the system with complete knowledge of its internals
  - Test as much implemented behaviour as possible
- Static testing:
  - Analyze the system without executing any code.
- Dynamic testing:
  - Analyze the runtime behaviour of the system

TESTING TACTICS

	Black Box (functional)	White Box (structural)
Static	- requirements validation	- lint - Findbugs, Coverity, etc.
Dynamic	- system tests - integration tests - fuzz testing	- unit tests - mutation testing

TESTING PHILOSOPHIES

- There are no shortage of testing philosophies you can apply:
  - Unit testing
  - Integration testing
  - System testing
  - Regression testing
  - Acceptance testing (not covered)
  - Test-driven development (next class)

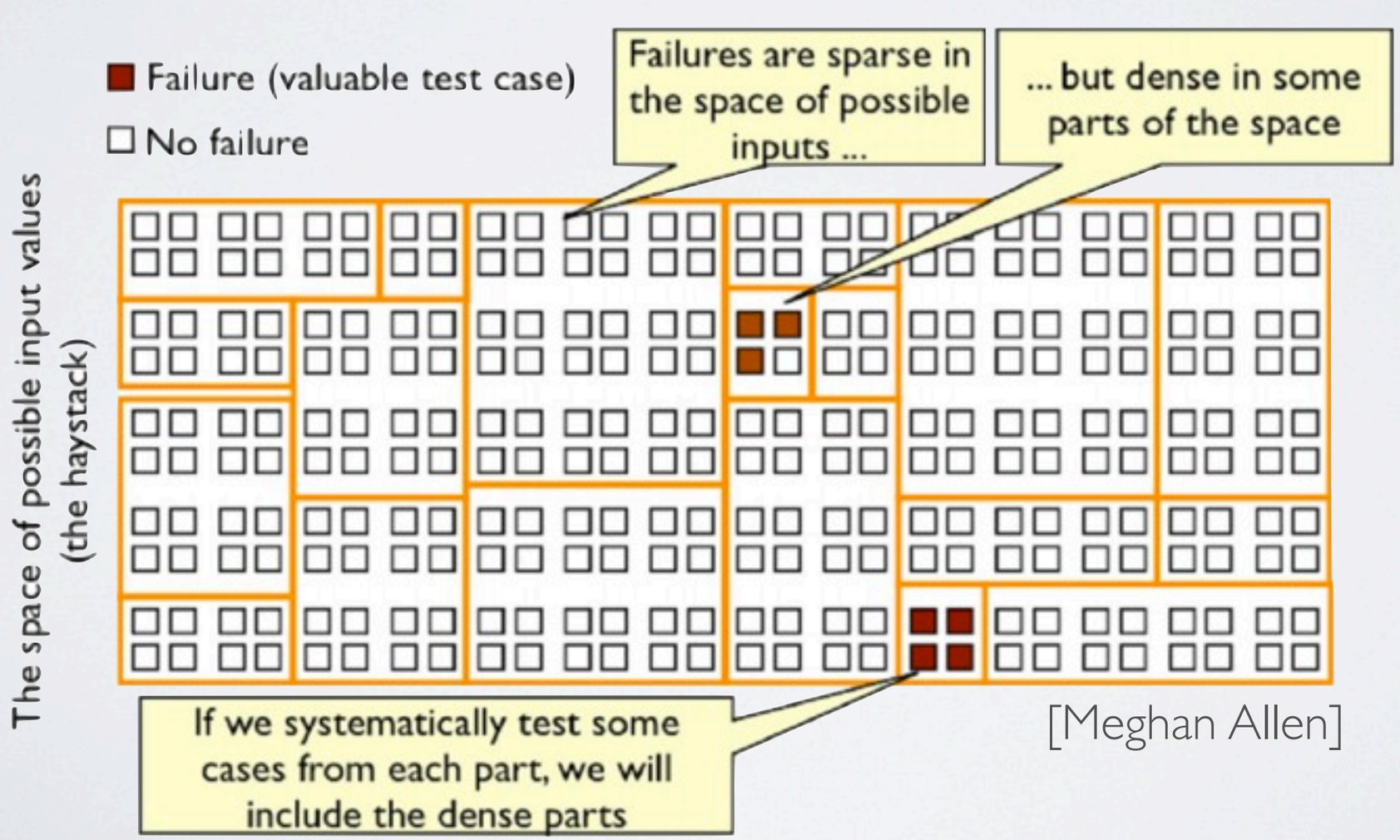
UNIT TESTING

- Basic assumption of unit tests:
  - "If the code doesn't work on its own, it won't work when the system is deployed either"
- Tests exercise a specific module (function, method, etc.)
- Often employs equivalence class partitioning and boundary testing

- Good unit tests:
  - Clearly define initial conditions and expected behaviour
  - Are specific: small granularity enables greater precision in isolating faults

EQUIVALENCE CLASS PARTITIONING

- Group inputs into categories that will be handled similarly
- Tests should exercise inputs from only one partition at a time



[Meghan Allen]

ECP EXAMPLE

- A system asks for user input between 100 and 999.
- Equivalence partitions:
  - Less than 100
  - 100 - 999
  - More than 999
- Three reasonable tests:
  - 50, 500, 1500

BOUNDARY TESTING

- Tests three kinds of values for any input:
  - Good values
  - Reasonable but invalid values
  - Unusual values
- e.g., getDaysInMonth(int month, int year):
  - reasonable: 3, 2008; 2, 2002
  - unreasonable: -1, MaxInt; MinInt, 0
  - unusual: 2, 2100 (leap year)

- Boundary / equivalence class partitioning better than random testing, but only as good as the values you test

INTEGRATION TESTING

- Ensures that multiple units or subsystems can interoperate
- Integration is a major source of errors
- Three high-level approaches:
  - big bang: no stubs, just wire it up and hope for the best
  - bottom up: integrate upwards to increasingly large tests
  - top down: test the UI and add layers to replace stubs
- Each has their tradeoffs:
  - big bang: fast, but often doesn't work
  - bottom up: more focus on units, less on UI (client focus)
  - top down: more UI focus but more infrastructure needed

SYSTEM TESTING

- Tests the deployed version of the system
- Confirms the behaviour of the complete application
- Often focuses on non-functional properties:
  - error recovery
  - security
  - stress / capacity / performance
  - usability
- System tests are sometimes used as part of the acceptance test process

REGRESSION TESTING

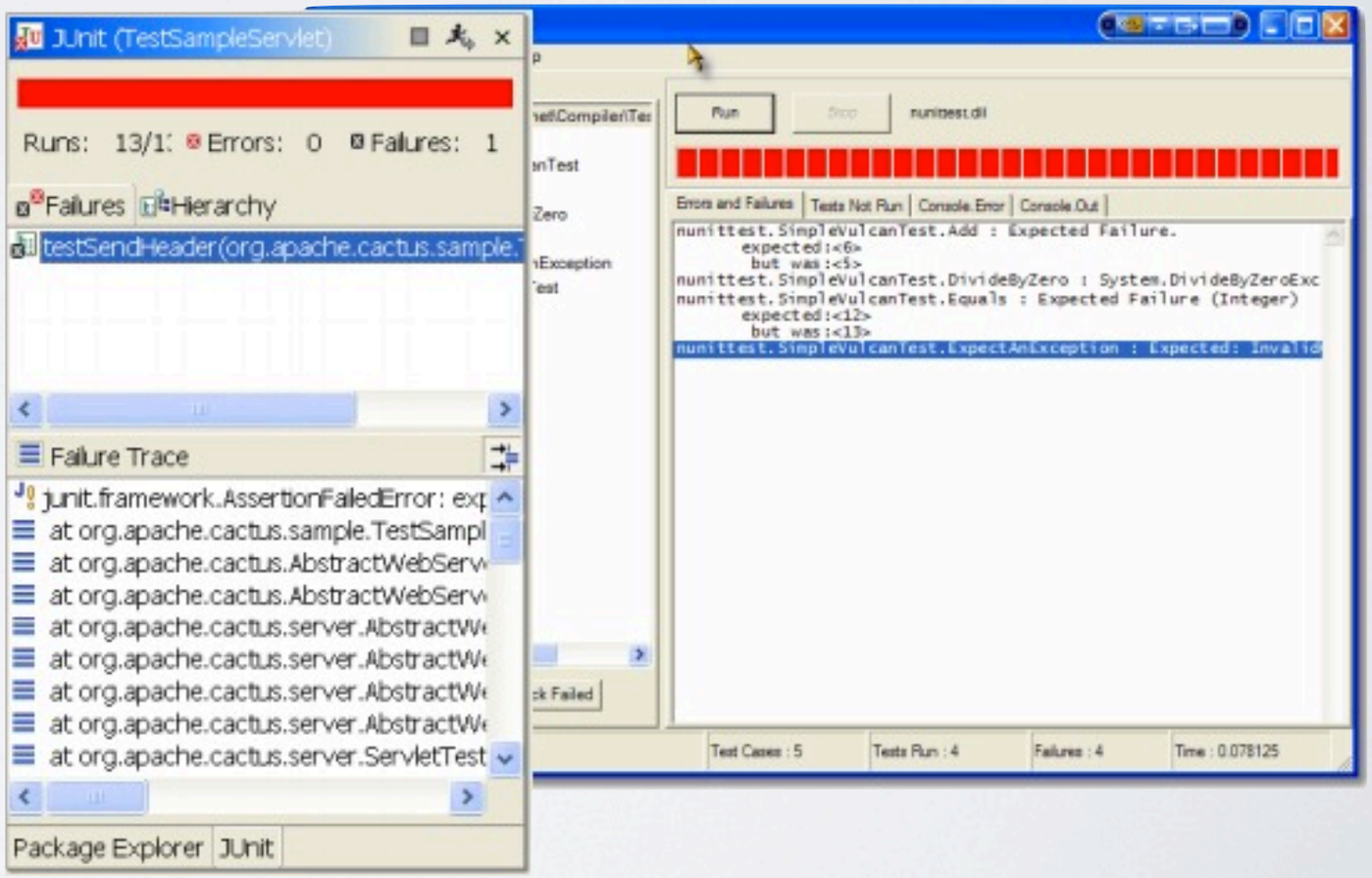
- Ensures that the system's behaviour hasn't degraded
- Run a suite of tests against every version (or at some interval)
- Commit gatekeepers can make sure code does not make it into the repository that causes new failures
- Expensive to manually perform but cheap with tooling

TESTING TOOLS

- Writing, executing, and analyzing tests is laborious
- Several testing tools have been widely adopted in industry

xUNIT

- Unit testing frameworks greatly ease test execution
  - e.g., jUnit, nUnit, cppUnit, Google Test
  - Will discuss Google Test, but they are all fairly similar
- Provide infrastructure for writing tests that can be automatically executed
- Key static components:
  - Test cases
    - Assertions
  - Test fixtures
  - Test runner



xUNIT TEST CASE

```
#include <gtest/gtest.h>

// TEST macro identifies tests (Google Test approach)
// Annotations or naming conventions often used

TEST(MyTestSuitName, MyTestCaseName) {
    int actual = 1;
    EXPECT_GT(actual, 0);
    EXPECT_EQ(1, actual) << "Should be equal to one";
}
```

xUNIT ASSERTIONS

- Main assertions:
  - ASSERT_TRUE(cond);
  - ASSERT_FALSE(cond);
  - ASSERT_EQ(expected, actual);
  - ASSERT_NE(var1, var2);
- Non-fatal checking is also available:
  - EXPECT_TRUE(cond);
  - EXPECT_... (same as ASSERTs)
- Can print custom messages with EXPECT/ASSERT:
  - EXPECT_EQ(x1, y1) << x1 << " != " << y1;

xUNIT FIXTURES P I

```
class QueueTest : public ::testing::Test {
 protected:
  virtual void SetUp() {
    q1_.Enqueue(1);
    q2_.Enqueue(2);
    q2_.Enqueue(3);
  }

  // virtual void TearDown() {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

xUNIT FIXTURES P2

```
class QueueTest : public ::testing::Test {
   ...
  // setUp() called before each TEST_F
  // tearDown() called after each TEST_F
  TEST_F(QueueTest, IsEmptyInitially) {
    EXPECT_EQ(0, q0_.size());
  }

};
```

xUNIT RUNNER

```
#include "QueueTest.h"
#include "gtest/gtest.h"

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

MOCKING

- Sometimes parts of the system under test are:
  - slow
  - non-deterministic
  - not built yet
- Mocking frameworks enable units to be tested without activating their dependencies
- Mocks adhere to an interface but simulate behaviour
- Often referred to as "stubs"

CONTINUOUS INTEGRATION

COVERAGE

> "... fundamental law of bug finding is No Check = No Bug"
>
> [Coverity]

- When executing a test suite we can instrument the program to get an idea of "how much" of the program has run.
  - Statement coverage
  - Branch coverage
  - Path coverage
- Hitting a coverage "target" is not effective, but discovering untested modules can be instructive

CODE REVIEW

- Check the code with colleagues
- Learn from more senior developers / transfer knowledge to more junior developers
- Many projects review _every_ patch e.g.,:
  - Firefox
  - Android
  - Webkit
- Encourages iteration to improve quality
- Discourages hacky solutions

CODE REVIEW