An Introduction To Software Agitation

Alberto Savoia
Chief Technology Officer
Agitar Software Inc.
www.agitar.com

Agenda

- The Developer Testing Revolution
- Four Types of Tests
- Software Agitation

The Developer Testing Revolution

- Agile/XP methodology are cool
- Developer testing is part of Agile/XP
- → Developer testing is cool

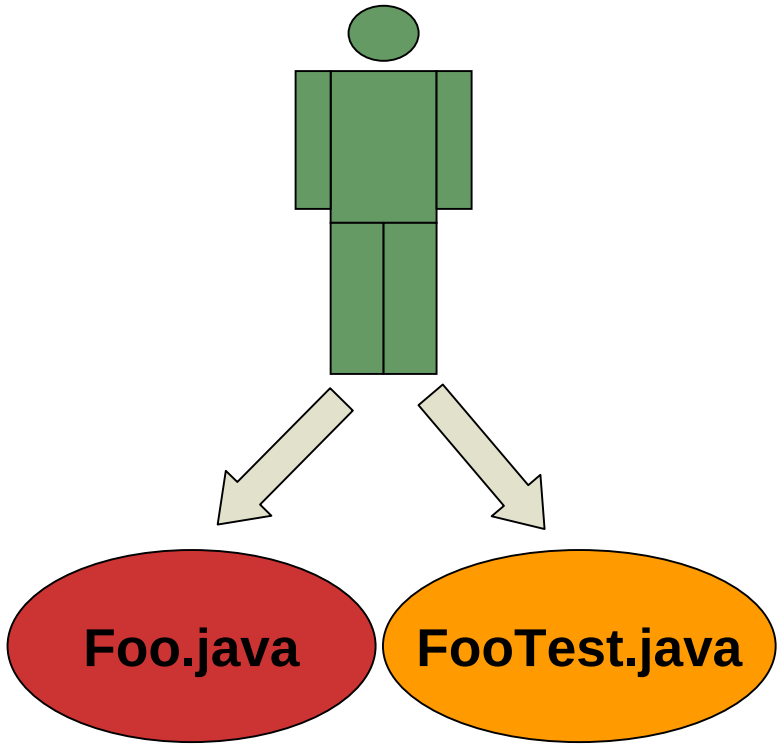
Developer Testing Trends

- 5 years ago – Ignorance and resistance
- Today – The Developer Testing Paradox
- 5 Years from now – Common practice???

Test

- It’s your code, and your responsibility

- Do it for your current colleagues
- Do it for future *generations* of colleagues
- Do it for yourself



My View of Unit Testing

What's a Unit in Unit Testing?

- **What's a Unit**
  - A single method/function/procedure
  - A collection or related methods/functions/procedures (e.g. a Java class)
- **In an ideal world …**
  - A unit is independent, self-sufficient, stand-alone
  - No need to deal with other units for testing purposes
- **In the real world …**
  - Most code has lots of dependencies
  - Testing a unit, typically involves other units

Basic Structure of Unit Tests

1. Set-up
   - Create initial state
   - Initialize method parameters
   - Store pre-execution values
2. Execute code under test
3. Compare actual results against expected results

Partial Correctness Assertions

- Notation introduced by C.A.R. Hoare in the context of formal verification

{P}S{Q}

If P is true at the time S executes, then Q must be true after S completes

```
{
   IntStack s
   s.size() == 0
}
   s.push(42);
   int val =  s.top();
{
   val == 42;
   s.size() == 1;
}
```

- **Think of tests as executable PCAs**
  - **Make P true**
  - **Execute S**
  - **Check if Q is true**

Example

```
public void testIntStackPushTop() {

   // setup : make P true
   IntStack s = new IntStack();

   // execute code under test: S
   s.push(42);
   int val = s.top();

   // compare actual vs. expected results: check Q
   assertEquals(42, val);
   assertTrue(s.size() == 1);
}
```

Basic Components of Unit Testing

- **Code under test**
- **Test data**
  - Several *interesting* instances of each of the types and objects needed to execute the code under test
- **Test assertions**
  - Boolean-valued expressions built from a dictionary of predicates and various logical/mathematical operators

Four Test Modes

- **Test Data** can be specific or general
  - Specific: **`int acctNum = 1234`**
  - General: **`forall int acctNum [acctNum > 0]`**
- **Test Assertions** can be weak or strong
  - Weak: **`getBalance(acctNum) >= MIN_BALANCE`**
  - Strong: **`getBalance(acctNum) == 344.32`**

Weak Assertions

- Weak Assertions
  - An assertion is considered weak if it can evaluate to true even if the aspect of the implementation that it's testing is incorrect:

  Example of weak assertion:

```
// after the withdraw operation completes
getBalance(acctNum) >= MIN_BALANCE
```

  - Weak assertions are still useful because they **can** detect problems when they evaluate to false

  - WA == false → bug
  - Bug !→ WA == false

Strong Assertions

- Strong Assertions
  - An assertion is considered strong if it will evaluate to true iff the aspect of implementation that it's testing is correct (for the test data used in the test case)
    - Example:

```
// after the withdraw operation completes
getBalance(acctNum) == 344.32
```

  - SA == false → bug
  - Bug → SA == false

Strong != Infallible

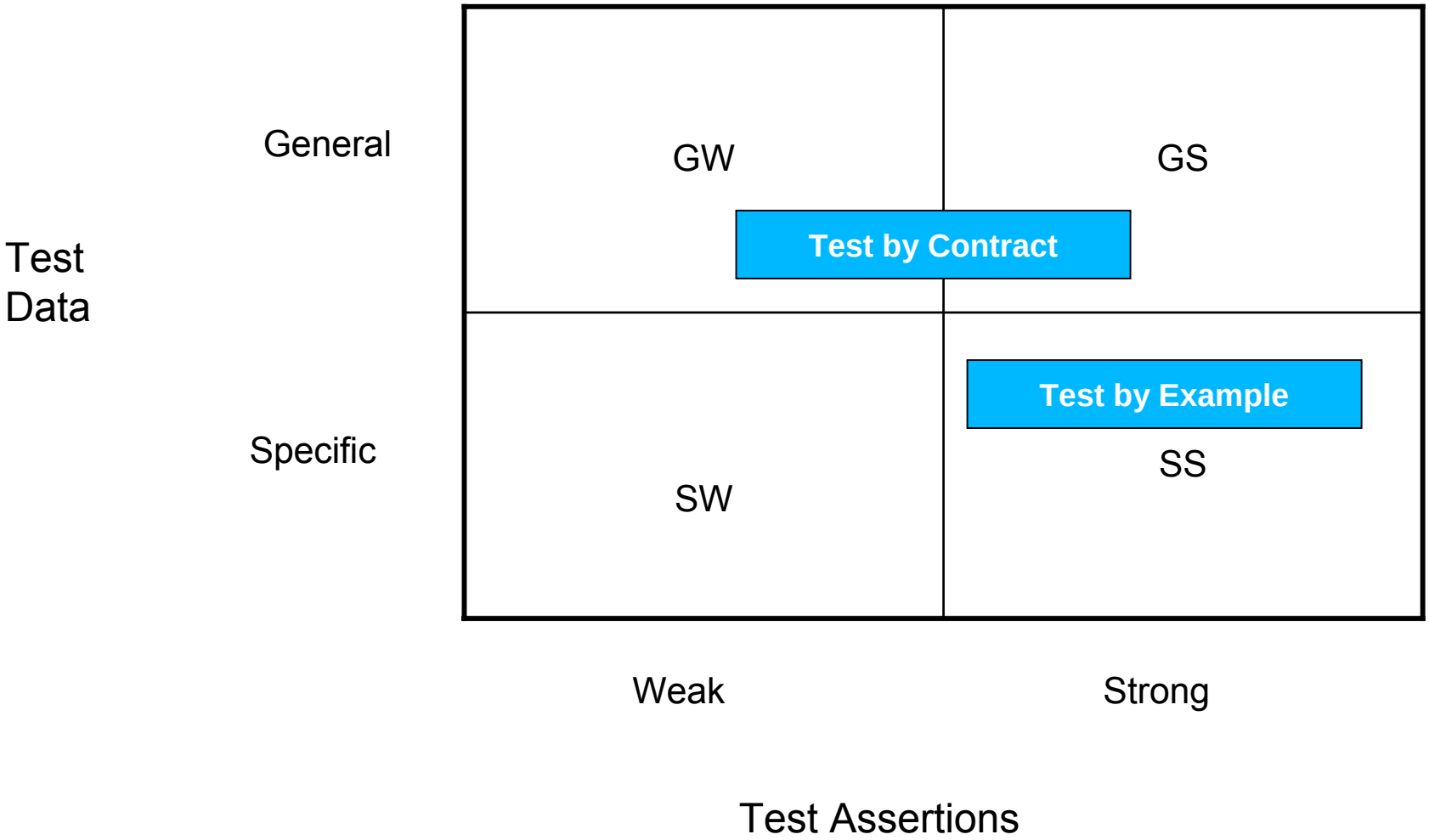
- It's possible to have a faulty implementation even if all strong assertions pass.

- Consider the following test:

```
{
   Bank bank
   bank.totalDeposits() == 10000000.0
   bank.getBalance(1234) == 1000.0
}
   bank.deposit(1234, 500.0)
{
   bank.totalDeposits() == 1000500.0
   bank.getBalance(1234) == 1500.0
}
```

- The 2 strong assertions can evaluate to true even though deposit has unwanted side effects

Four Test Modes



Two Basic Types Of Tests

Test by *Example*

```
{
   IntStack s
   s.size == 0
}
   s.push(42);
   int val = s.top();
{
   val == 42
   s.size() == 1
}
```

- Characterized by:
  - Single initial state
  - Single/specific set of test data
  - Strong assertions

Test by *Contract*

```
{
   IntStack s
   int n
   s.size < IntStack.MAX_SIZE
}
   s.push(n);
   int val = s.top();
{
   val == n
   s.size() == @PRE(s.size()) + 1
}
```

- Characterized by:
  - Multiple initial states
  - Multiple sets of test data
  - Strong and weak assertions

Four Modes Exercise

- Let's create tests for the method

```
int nextPrime(int n)
```

- Test by Example
  - Specific data

- Test by Contract
  - General data

Test by Example for nextPrime(int n)

```
{
    n == 0
}
    np = nextPrime(n)
{
    np == 2
}
```

```
{
    n == 2
}
    np = nextPrime(n)
{
    np == 3
}
```

```
{
    n == 31
}
    np = nextPrime(n)
{
    np == 37
}
```

```
{
    n == 0
}
    np = nextPrime(n)
{
    np > 0
}
```

```
{
    n == 2
}
    np = nextPrime(n)
{
    isPrime(np)
}
```

```
{
    n == 3
}
    np = nextPrime(n)
{
    np % 2 == 1
}
```

Test by Contract for nextPrime(int n)

```
{
    n >= 0
}
    np = nextPrime(n)
{
    np % 2 == 1
}
```

```
{
    n >= 0
}
    np = nextPrime(n)
{
    isPrime(np)
    np > n
}
```

```
{
    isPrime(n)
}
    np = nextPrime(n)
{
    prevPrime(np) == n
}
```

Can you spot the bug in this example?

Test by Contract

By combining multiple weak assertions you can create a strong test

```
{
   n >= 0
}
   np = nextPrime(n)
{
   isPrime(np)
   np > n
   numOfPrimesBetween(n, np) == 0
}
```

4 Modes Summary

Test Data \ Test Assertions	Weak	Strong
General	**GW** nextPrime(n) > n isPrime(nextPrime(n))	**GS** isPrime(n)→ prevPrime(nextPrime(n)) == n
Specific	**SW** nextPrime(7) > 7 isPrime(nextPrime(7))	**SS** nextPrime(7) == 11

Class Invariants

- A class invariant is a property that is true of all objects of a given class before and after each public method call
  - Examples for IntStack class
    - `size() >= 0`
    - `size() <= MAX_SIZE`
  - Examples for Employee class
    - `hourlySalary >= HRSystem.MINIMUM_WAGE`
    - `getManager() != null`
    - `getSSN.matches("[0-9]{3}-[0-9]{2}-[0-9]{4}")`
- Class invariants are a *cheap* and powerful testing *tool*, but unfortunately rarely used in manual unit testing

The Bottom Line

- Unit testing is not easy
- Solid testing effort ~=> implementation effort
  - ~3-4 lines of JUnit for every 1 lines of Java for 90-100% code coverage
- Good set of tests include
  - Specific and general test data combined with
  - Strong and weak assertions
  - (general test data + strong assertions is best but can be hard to achieve)
- Additional challenges
  - It's difficult to create test data/tests for situations you did not consider when writing the code
  - It's difficult to be aware of all the properties and behaviors of code that you depend on
- As a result:
  - Currently >90% of unit tests are of the test-by-example variety
  - Most unit tests focus on normal conditions and most common paths

The Bottom Line

- The nature and challenges of unit testing make automation and computer assistance a necessity

- Unit testing tools can help:
  - Simplify test creation
  - Test data generation
  - Assertion generation
  - Test execution
  - Test analysis
    - Code coverage
    - Results

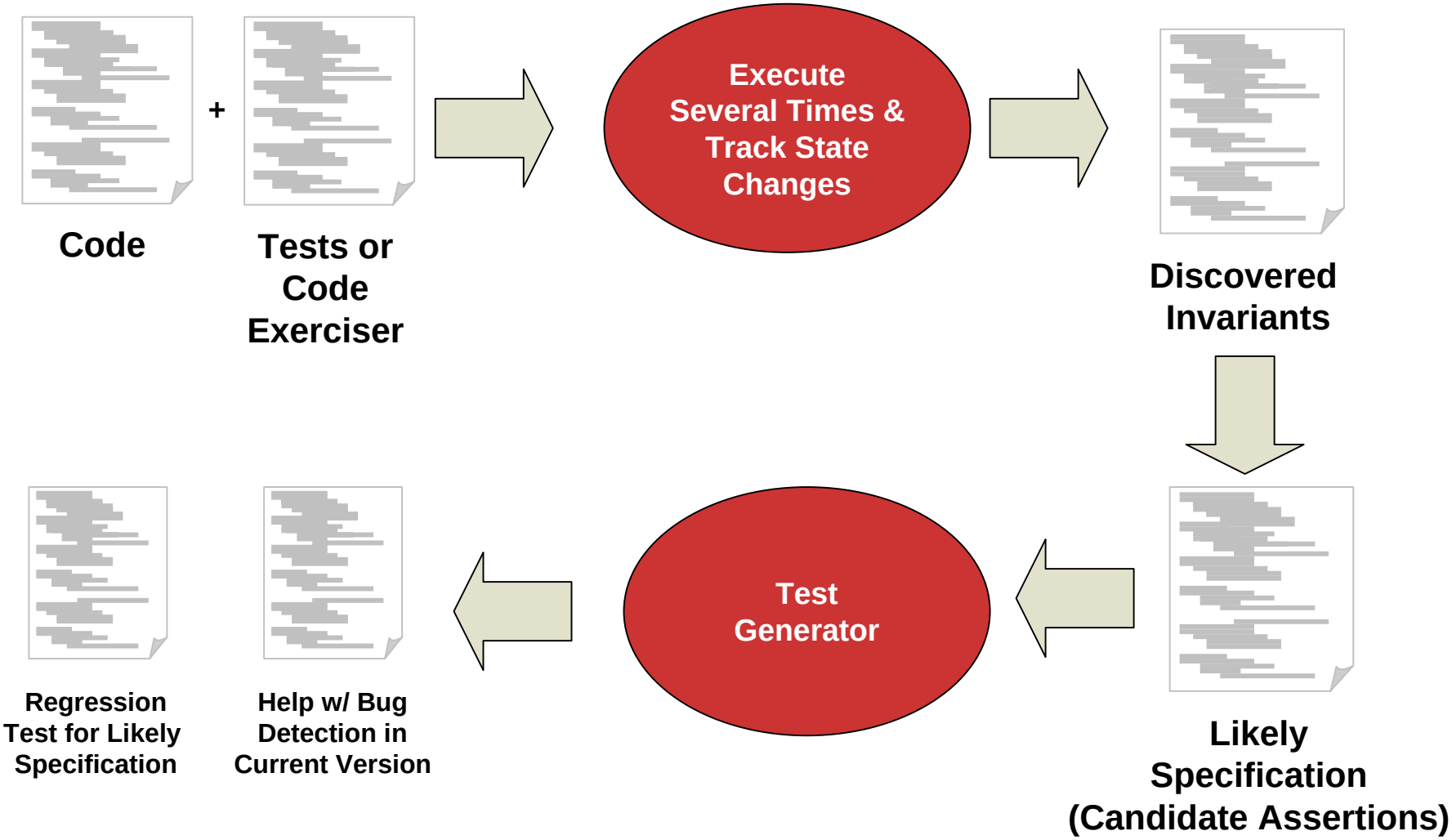
Unit Testing Tools

- Testing Frameworks (e.g. JUnit)
  - Simplify test creation
  - Automate test execution and reporting
- Automated Test Generators
  - Automate and accelerate test creation
  - Enable exploratory testing
    - Use unexpected test data
    - Discover method and class invariants
  - Automate test execution and reporting

Modern Automated Test Generation

- Historically
  - ATG → Automated Test Data Generation
- More recently
  - ATG → Automated Test Data Generation **&** Automated Assertion Generation

Automated Assertion Generation



Execution → Spec → Tests (via invariants discovery)

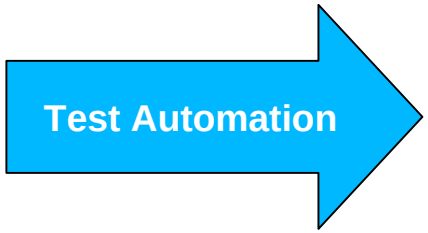
- Michael Ernst - MIT
- David Notkin, Tao Xie – University of Washington, North Carolina State University
- Johannes Henkel, Amer Diwan – University of Colorado at Boulder

```
TYPE IntStack

AXIOMS
forall s : IntStack, i : int
 pop(push(s,i).state).retval = i
 pop(push(s,i).state).state = s
 pop(IntStack().state).retval ~> EmptyStackException

SAMPLE TEST
    Stack s = new Stack();
    s.push(7);
    s.push(9);
    s.pop();
    assertTrue(s.pop(), 7);
```

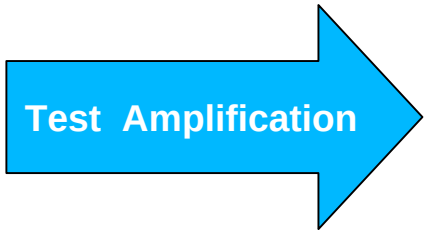
Too Much Automation?

- There is such a thing as too much test automation
- Developer out of the loop → tests that verify that the code does what the code does
- The developer should be involved in reviewing, approving, and, if necessary, modify and augment the tests

My Automated Test Generation Philosophy



Automate everything that can and ***should*** be automated



Make everything else as efficient as possible

Software Agitation: Making Invariants Detection Idea Practical

- Automated invariants detection is a brilliant idea but:
  - Need an automated way to execute/drive the code
  - Must use developer-friendly language/expressions
  - Must be integrated with IDE to maximize adoption
  - The developer needs to be involved somewhere in the cycle
- Software Agitation combines invariants detection with
  - Test data generation
  - Automated execution
  - Invariants in programmer-friendly syntax
  - IDE integration

Execution → Observations → Asserts

Code

Observations
on code behavior

If an observation
reveals a bug, fix it

If it describes desired behavior,
click to promote it to an assertion

Agitator Demo

Conclusion

- Early stage developer/unit testing is critical to software quality
- Trend
  - Old/Current: pass the buck to QA
  - Future: pass the buck to QA **after** unit testing
- Unit testing can be as challenging, and often more challenging, that development – lots of interesting theory and problems
- New test automation technology can developer/unit testing easier, faster, and more thorough

Final Word

Good test automation should not replace
human intelligence, creativity, and
developer participation in the testing process.

Instead, it should help developers focus on the activities
that require human intelligence, creativity, and insight.

Q&A

Contact: alberto@agitar.com