

# Trends and Techniques in Unit Testing

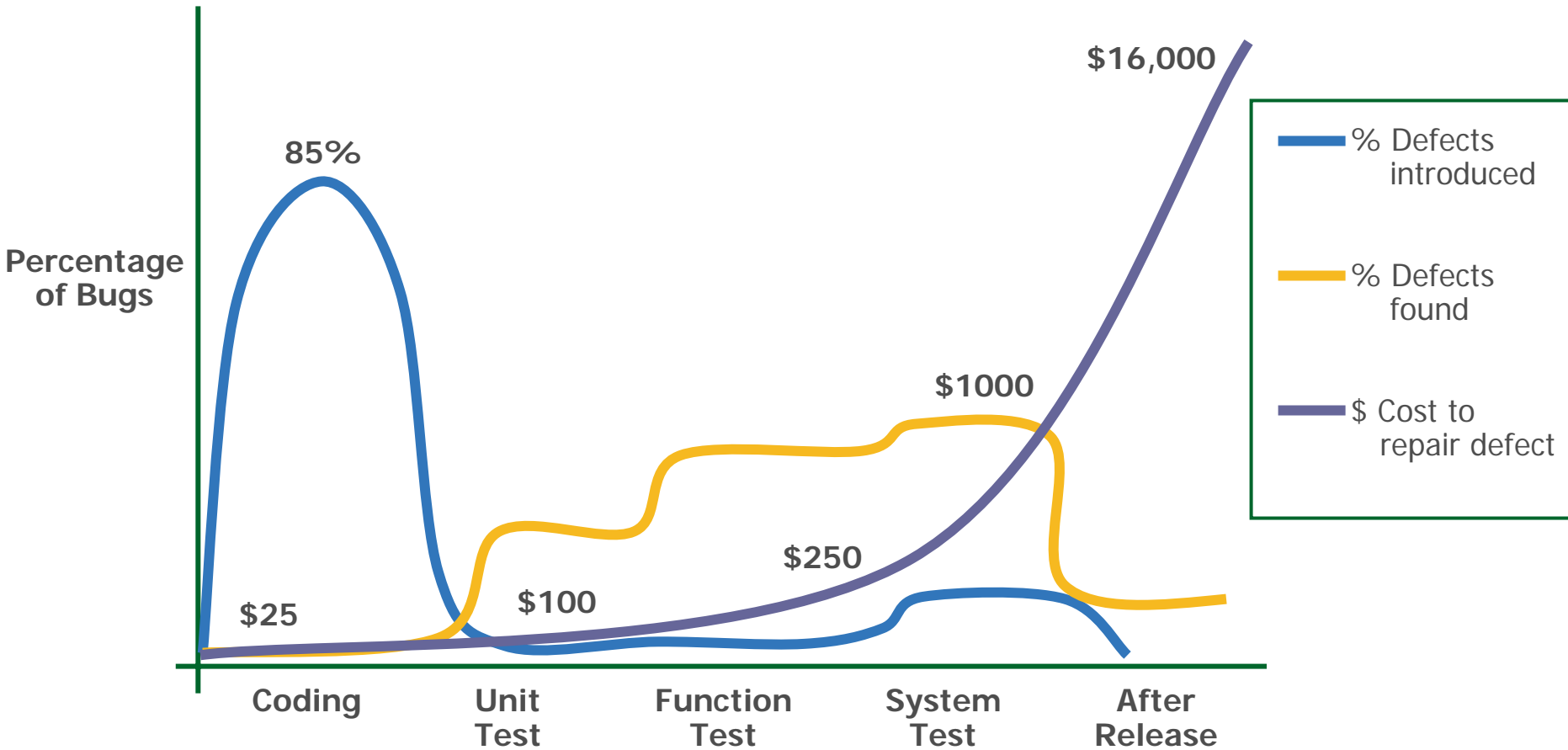
From Daikon to Agitator and Beyond

**Marat Boshernitsan**

**Agitar Software Laboratories**



# Bugs Cost Money



Source: Applied Software Measurement, Capers Jones, 1996

# Unit Testing - Effective Way to Reduce Bugs

Testing a “unit” of code before integrating with the rest of the system

Cost of fixing unit-level defects early is smaller

Enables the QA team to focus on larger defects

Performed by developers (not QA)

- Often called “developer testing”

Does not replace system or integration testing

# The Anatomy of a Unit Test

## JUnit tests

```
void testLinkedList() {
```

```
    IntegerLinkedList list = new IntegerLinkedList();
```

```
    list.addFirst(42);
```

```
    assertTrue(list.getFirst() == 42);
```

```
}
```

{P}

S

{Q}

## Partial Correctness Assertions (PCA)

- Hoare triple: {P} S {Q}

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

# Benefits of Unit Testing

Rapid feedback (on code and on design)

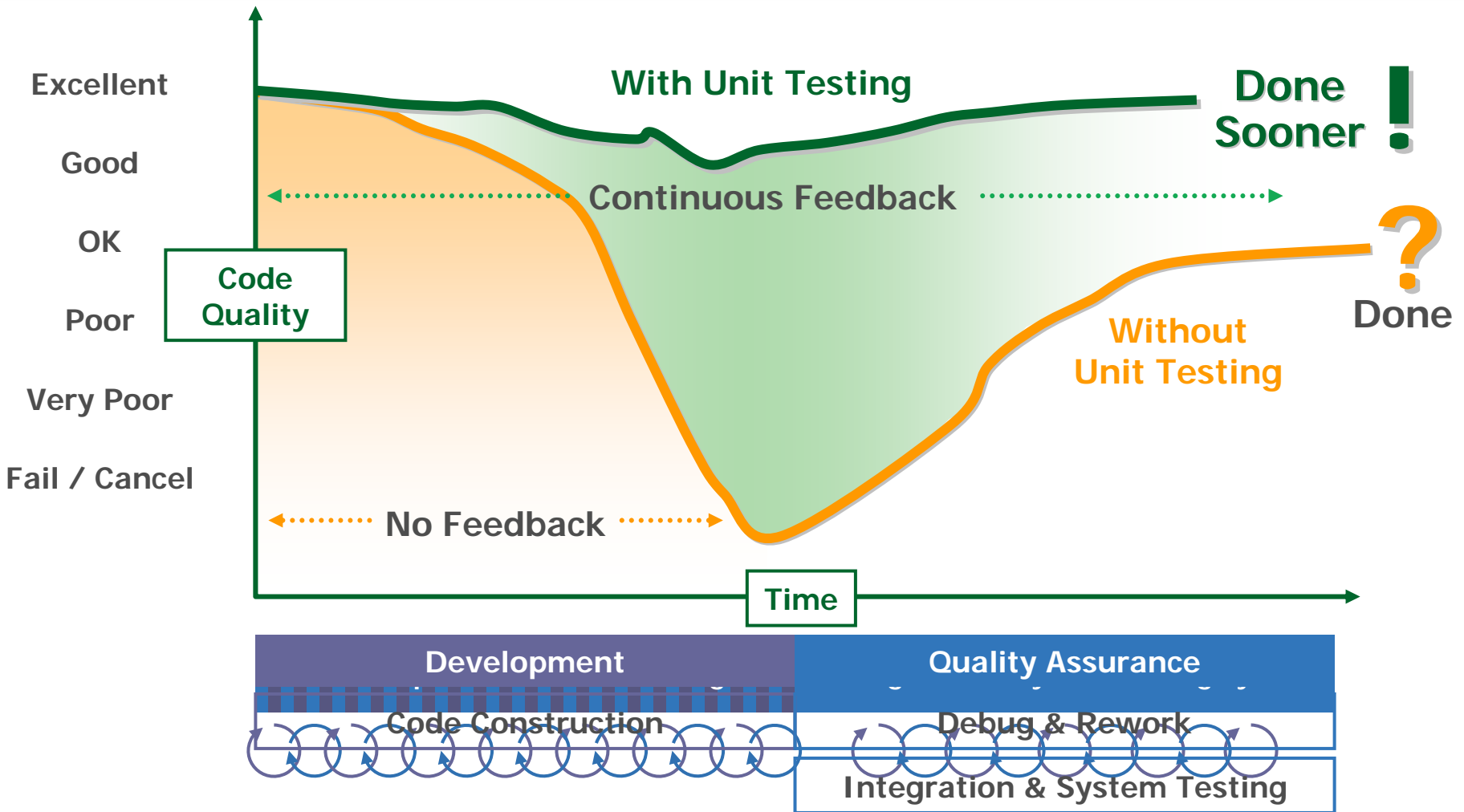
Forces to slow down and think  
(which makes progress faster overall!)

Documents code's current behavior

- For detection of future regressions
- For the benefit of other programmers
- For defense against other programmers

Reduces fear of code change

# Why Unit Testing Works



# Limited Adoption of Unit Testing

Cultural problem: “testing is for QAs”

- Not as satisfying as coding
- Hard to perceive immediate benefit
- Agile proponents are popularizing unit testing (but they are often too dogmatic)

Tools problem: “it is too hard”

- Difficult to transition (mentally) to testing
- Takes too much grunt work to write tests
- Maintaining a test suite is challenging

# There's Hope Yet

Modern unit-testing frameworks standardize test format

- Java: JUnit, TestNG
- C#: NUnit, C++: CPPUnit, etc.

Automation of test execution enables continuous monitoring of code quality

- Know when things work (and break)

...but what about writing unit tests?

# Testing vs. Formal Verification

Hoare's  $\{P\}S\{Q\}$  triples were introduced in the context of formal verification

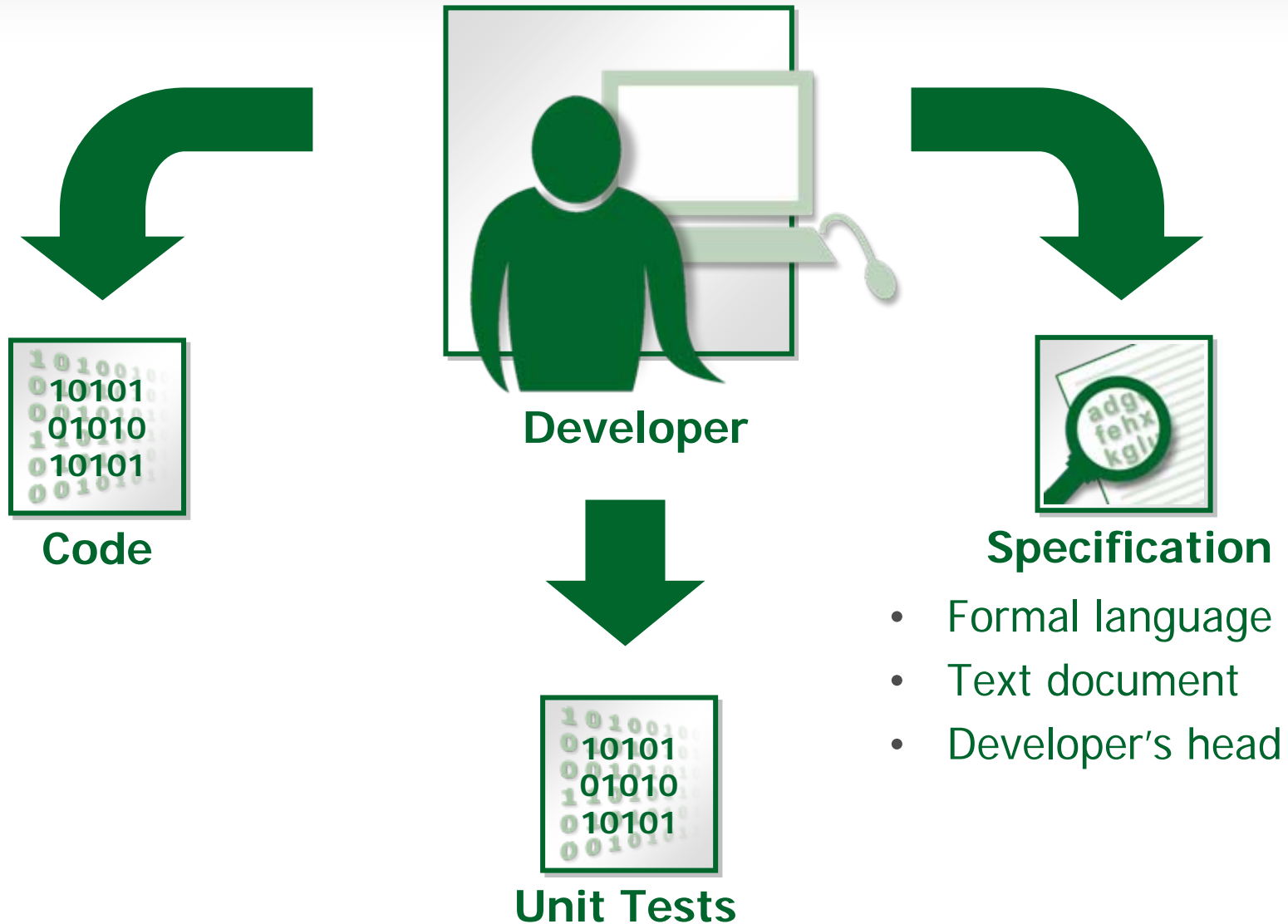
Testing only guarantees that your program works in a few specific cases

*Testing can be used to show the presence of bugs, but never to show their absence!*  
(E. W. Dijkstra)

Rigorous unit testing requires variation of Ps and strong Qs (test assertions)

- This is challenging!

# How Unit Tests Get Written



- Formal language
- Text document
- Developer's head

# Minding Ps and Qs: Test Data

Challenge: cover as many various code paths as possible

- ...including all corner cases
- ...including all error cases

Example: testing a HashMap

- Did you test `put(k,v)` into an empty map?
- ...one element map?
- ...when two keys hash to the same bucket?
- ...when a `put()` causes internal resize?

# Minding Ps and Qs: Assertions

Challenge: assertions must reflect how the code should behave

- ...does the specification exist?
- ...are the assertions strong enough?  
`assertTrue(size > 0)` vs. `assertTrue(size == 5)`

Developers find it difficult to switch mental context from coding to testing

- Coding: mostly constructive
- Testing: mostly destructive

# Tools from Research Ideas

## Test data:

- Use static and dynamic source code analysis to create test data to cover as many paths in the code as possible

## Assertions:

- Help the user by computing some possible invariants, but rely on their expertise to decide which ones correspond to the spec

# In the beginning there was Daikon...

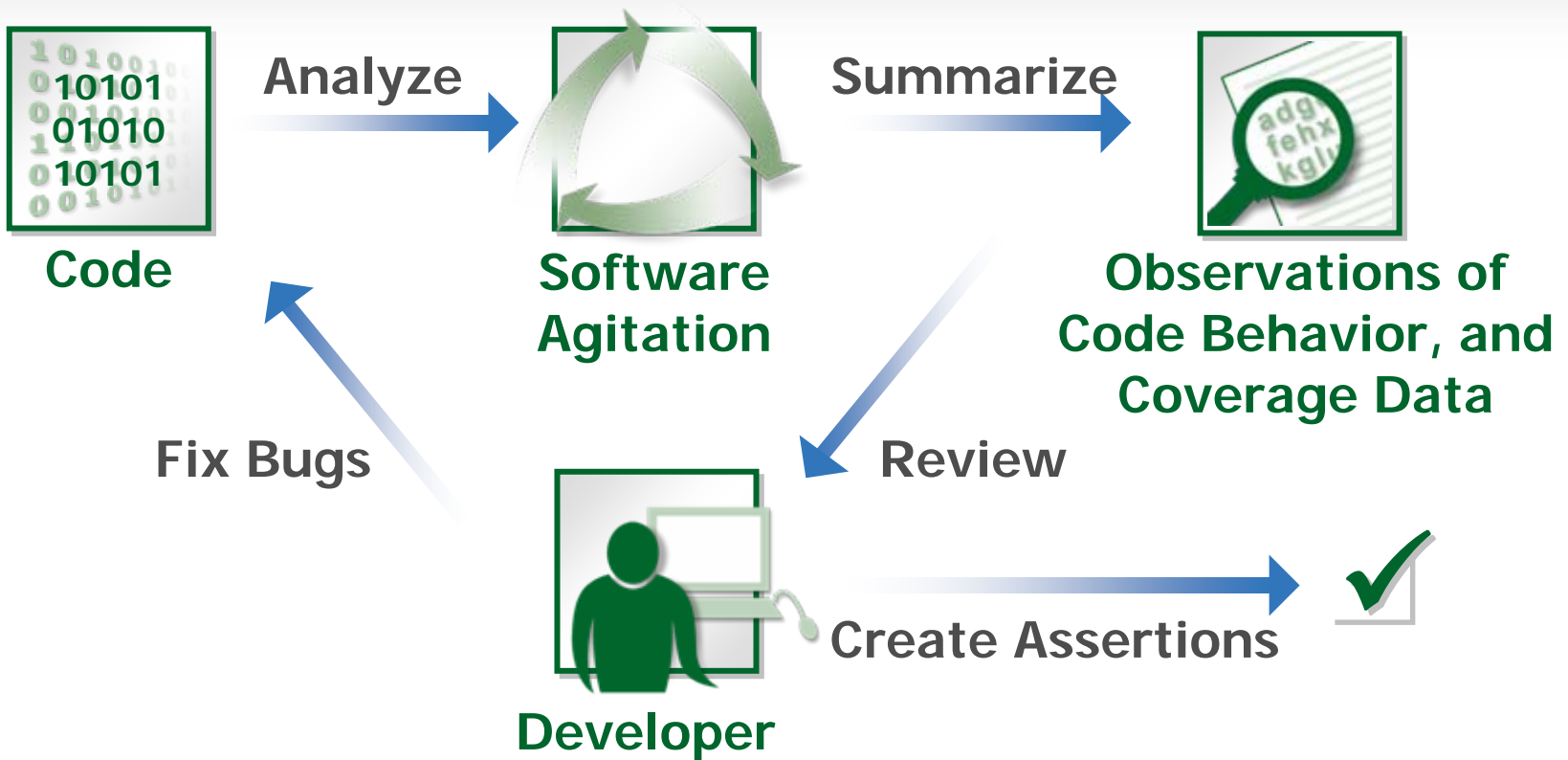
Daikon: dynamic detection of likely invariants  
(Ernst et al., In ICSE'99)

- Applies dynamic analysis to infer invariant properties of program
- Example:  $j \leq i < k$ ,  $a = b + 42$
- ... but uses existing tests to exercise code

Idea #1: Can combine invariant inference with test-input generation (+ heuristics for scalability)

Idea #2: Can avoid “the oracle problem” by putting developer in control

# Software Agitation



# Program Analysis for Test-Input Generation

Agitation goal: basis path coverage

Static analysis:

- Path-constraint solving for param. values
- Type-specialized solvers provide values for common non-primitives (String, Collections)

Dynamic analysis:

- Observes execution and tries to force alternative paths
- Uses heuristics to guess values and objects that get coverage

# Program Analysis: It's OK to Cheat!

Agitation iterates: no need to achieve coverage on first attempt

- Also, invariant detection needs multiple passes to get various data points

Heuristics help performance:

- Only consider partial paths (up to size  $X$ )
- Try common values (-1, 0, 1, constants from class files, mutate at random, etc.)

# Dynamic Invariant Detection

Find observable values at a given point

- Parameters, fields accessed in method
- Properties of simple data types (string len.)
- Getters for objects in scope

Assume all possible relationships

- Boolean expressions in source code
- Relational, ranges, linear, etc.

Discard relationships as (sufficiently many) counterexamples are found

# From Research to Practice: Applicability and Scalability

Handle real-world code

- Poorly-structured (2k-line methods!)
- Highly interdependent
- Java native methods inhibit analysis
- ...etc.

Support continuously evolving code

Scale to modern-size applications

# From Research to Practice: Usability

Speak the developer's language

- ✗ Displaying observations in first order logic
- ✗ Designing new specification language
- ✓ Displaying observations in Java-like syntax

Make no assumptions about expertise

Integrate into existing workflow

# Beyond Software Agitation: Working Effectively With Legacy Code\*

Software agitation enables *exploratory* testing of code

- Works well for green-field projects
- But what if you have 1M lines of code with few (or no) tests?

Need a way to capture behavior of the code to detect future regressions

# Characterization Tests: Minding Only Your Ps

Writing assertions requires knowing the “correct” behavior

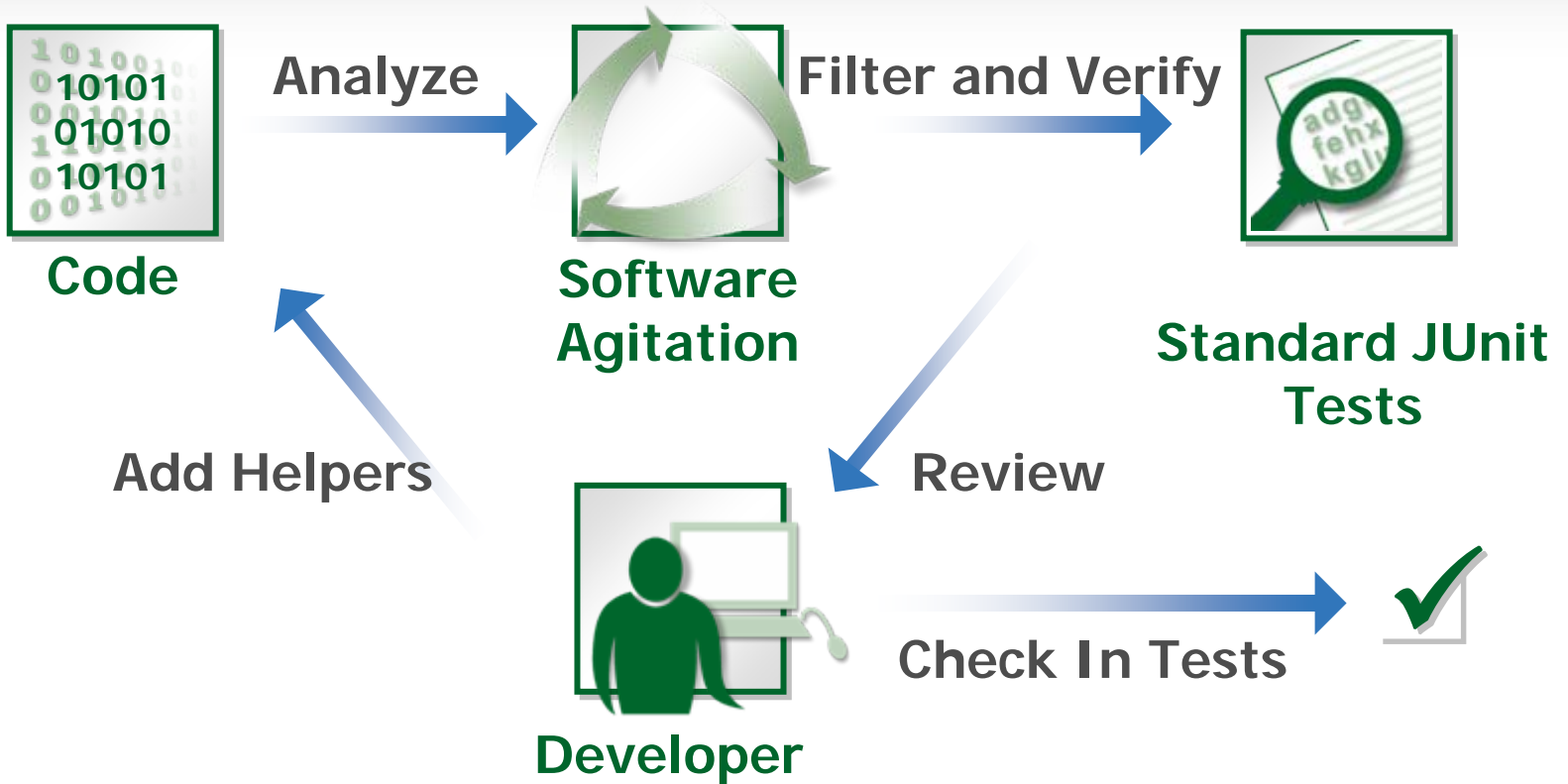
- This is known as the *oracle problem*

Software agitation enables the developer to act as the test oracle

Generation of characterization tests uses program’s existing behavior as the oracle

- Can generate assertions by observing interesting state at the end of a test

# JUnit Generation



# Technology Behind JUnit Generation

Coverage-targeted test-input generation provided by the Agitator engine

Instead of capturing invariants, test generation records method call sequences

Test sequence generator extracts call sequences for desired method outcomes

# JUnit Generation: From Sequences To Tests

Shortens sequences to remove irrelevant steps

Beautifies test input values for better readability and determinism

Removes volatile call sequences

Generates assertions to capture “interesting” program states

# JUnit Generation: Aggressive Mocking

Unit-testing challenge: it is often hard to achieve unit-level isolation

- Accessing a database
- Accessing a file system
- Difficult to instantiate objects
- ...etc.

Static analysis and aggressive mocking allow us to isolate a single execution path

Agitar's JUnit generator can achieve 80% to 100% test coverage in most situations

# Selling Agitar's Technology to Developers (and their managers)

Many Java developers lack the motivation and/or skills to test their own code

- Even automated tools require some effort
- Developer education is often a problem

Features and schedules sometimes trump keeping tests “green” and coverage high

JUnit generation proved to be a great way for many organizations to get started

# Tools as Amplifiers of Developer's Effort

Automated unit testing tools are not a silver bullet of software development

Even the best of tools require some effort

- Set up for a particular project
- Integration into build process
- Improvements to code coverage
- Checking of observations

Benefits afforded by a tool must exceed the effort required to use it

# Research Challenges: Technology

Better coverage (on real-world code)

- Good techniques exist, but do not scale

Better assertions (without specification)

- Improving mutation adequacy
- Capture higher-level behavior

Intelligent unit isolation through mocking

- “Test factoring” (Saff et al.)
- “Test carving” (Elbaum et al.)

# Research Challenges: Human Factors

Many barriers to wider adoption of unit testing are not technical

Motivating developers to change habits

- Match each developer's expertise
- Encourage “best practices” workflows

Fitting well with the way developers work

- Present information when it is needed
- Adapt to each developer's needs
- Play well with development teams

# Technology Transfer is a Contact Sport\*

People, not papers transfer technology!

Active pursuit of relationships helps

- Industry seminars
- Industrial research grants
- Student internships

Relationships must be built over time

- Short-term funding rarely results in transfer

Tech transfer is a bottom-up effort

\*Jim Foley, *Technology Transfer from University to Industry*, In CACM 96

# Research and Development at Agitar

Using Modern Portfolio Theory to achieve better and more consistent software quality (ISSTA'07)

Assessing test adequacy by observing the flow of test data through code under test (FSE'07)

Improving regression testing adequacy and assisting test repair with test. gen tools (FSE'07)

Combining benefits of design-by-contract with test-driven development (in JUnit 4.4, publication in progress)

Using structural metrics and machine learning to help developers use testing tools more effectively (publication in progress)

# Concluding Thoughts

Code quality is a huge problem

- Developer testing helps significantly
- Automated unit-test generation helps with legacy code

Unit-testing offers good potential for automation

Academic research can inspire interesting tools

- Need to combines research with practical tradeoffs
- Scalability, reliability, usability, performance are essential (and interesting!)

Questions?

Marat Boshernitsan ([marat@agitar.com](mailto:marat@agitar.com))