Performance Testing & Stress Testing
A Brief Introduction

SEDep

IT Faculty, HCMUNS

2006

Question ?

- What are types of Performance Testing ?
- What is tested ?

Two Types of Performance Testing

- Product
  - Focus on algorithms
    - Searches, loops, recomputations
    - Order of algorithms (ex. Sort algorithms)
- System
  - Focus on deployment
    - How many servers
    - Database structure
    - Caching
    - Network connectivity

What is tested ?

- Throughput (lưu lượng)
  - How many and how large “things” can I process in a given time?
- Number of users
  - Especially web sites
  - Victoria’s Secret
- Response time
  - How long does the user need to wait?
- Availability
  - Is it available to the user at all?
- Slow enough is not distinguishable from broken
  - Status helps, but only to a degree
- Stress Testing (next slides)

Stress Testing – Concept

- Tag line
  - *“Overwhelm the product.”*
- Fundamental question or goal
  - Learn about the **capabilities** and **weaknesses** of the product by driving it through failure and beyond. What does failure at extremes tell us about changes needed in the program’s handling of normal cases?
- Paradigmatic case(s)
  - Buffer overflow bugs
  - High volumes of data, device connections, long transaction chains
  - Low memory conditions, device failures, viruses, other crises.

Stress Testing – Strengths & Blind spots

- Strengths
  - Expose weaknesses that will arise in the field.
  - Expose security risks.
  - Perhaps good for assessing performance, reliability, or efficiency.
- Blind spots
  - Weaknesses that are not made more visible by stress.

Stress Testing – How ?

- Look for functions or sub-systems of the product that may be vulnerable (có thể bị tổn thương) to failure due to challenging input or constrained resources.
- Identify input or resources related to those functions or sub-systems.
- Select or generate challenging data and platform configurations to test with: e.g., large or complex data structures, high loads, long test runs, many test cases, limited memory, etc.
- *Force the program to fail, watch how it fails, report the vulnerability*

Test cases

- Need to be built on realistic assumptions
  - Volumes: consider the problems at Altona and with Mars Pathfinder
  - Mix of functions: searches versus creations
  - Benchmarks
  - Operational profiles
- What can you economically test?
  - Modeling of the function
  - Simulators for large numbers of users

Web Testing

- Simulator to create lot of users
  - Figures out capacity
  - Doesn’t create the diversity (đa dạng) needed
    - Location
    - Connectivity
    - Client systems
- Major service offerings
  - Find real clients for you
  - On major events, reporters used to hire to be able to report bad results

Are numbers enough?

- First step in performance
  - Throughput numbers with different size inputs
    - XML parsing
    - Benchmarks
  - Number of users
    - Simultaneous
      - Response time
    - Registered
      - Database size
- But…

Need the trends

- Algorithm complexity
- Bottlenecks
- Inflection points
- Stone walls

Really good performance testers

identify the **specific bottlenecks**

AND

propose **solutions**

References

Introductory material: Reiner Dumke et al. (2001), "Performance Engineering, State of the Art and Current Trends", Springer Verlag

Mathematical models: Boudewijn Haverkort et al. (2001), "Performability Modeling, Techniques and Tools", John Wiley & Sons.

GUI Automation Testing & Regresstion Testing

Regresstion

"Repeat testing after changes."

Regresstion Testing – What ?

- Tag line: *"Repeat testing after changes."*
  - But scratch the surface and you find three fundamentally different visions:
    1. *Procedural:* Run the same tests again
    2. *Risk-oriented:* Expose errors caused by change
    3. *Refactoring support*: Help the programmer discover implications of her code changes.
- Fundamental question or goal
  - Good regression testing gives clients confidence that they can change the product (or product environment).

Why Are Regression Testing Weak?

- Does the same thing over and over
- Most defects are found during test creation
- Software doesn’t break or wear out
- Any other test is equally likely to stumble over unexpected side effects
- Automation reduces test variability
- Only verifies things programmed into the test

1.Procedural regression testing

- Tag line
  - *"Run the same tests again"*
- Paradigmatic cases
  - Manual, scripted regression testing
  - Automated GUI regression testing
  - Smoke testing (manual or automated)

Procedural RT - Benefits

- The tests exist already (no need for new design, or new implementation, but there will be maintenance cost)
- Many regulators and process inspectors like them
- Because we are investing in re-use, we can afford to take the time to craft each test carefully, making it more likely to be powerful in future use
- This is the dominant paradigm for automated testing, so it is relatively easy to justify and there are lots of commercial tools
- Implementation of automated tests is often relatively quick and easy (though maintenance can be a nightmare)

Procedural regression testing

- Blind spots / weaknesses
  - Anything not covered in the regression series.
  - Repeating the same tests means not looking for the bugs that can be found by other tests.
  - Pesticide paradox
  - Low yield from automated regression tests
  - Maintenance of this standard list can be costly and distracting from the search for defects.

2.Risk-oriented regression testing

- Tag line
  - *“Test after changes.”*
- Fundamental question or goal
  - Manage the risks that

    (a) a bug fix didn’t fix the bug or

    (b) the fix (or other change) had a side effect.
- Paradigmatic case(s)
  - Bug regression (Show that a bug was not fixed)
  - Old fix regression (Show that an old bug fix was broken)
  - General functional regression (Show that a change caused a working area to break.)

Risk-oriented regression testing (cont.)

- In this approach, we might **re-use old tests** or **create new ones**.
- Often, we **retest an area** or function with tests of **increasing power** (perhaps by combining them with other functions).
- The focus of the technique is on testing for **side effects of change**, not the inventory (tóm tắt) of old tests.

Risk-oriented RT - Examples

- Here are examples of a few common ways to test a program more harshly while retesting in the same area:
  - Do more iterations (one test hits the same function many times).
  - Do more combinations (interactions among variables, including the function under test's variables).
  - Do more things (sequences of functions that include the function under test).

Risk-oriented RT – Examples (cont.)

- Methodically cover the code (all N-length sequences that include the function under test; all N-wide combinations that include the function under test's variables and variables that are expected to interact with or constrain or be constrained by the function under test).
- Look for specific errors (such as similar products' problems) that involve the function under test.
- Try other types of tests, such as scenarios, that involve the function under test.
- Try to break it (take a perverse view, get creative).

3.Refactoring support: Change detectors

- Tag line
  - *"Change detectors"*
- Fundamental question or goal
  - Support refactoring: Help the programmer discover implications of her code changes.
- Paradigmatic case(s)
  - ***Test-driven development using glass-box testing tools like junit, httpunit, and fit…***

Note

- The programmer creates these tests and runs them every time she compiles the code.
- If a test suite takes more than 2 minutes, the programmer might split tests into 2 groups (tests run at every compile and tests run every few hours or overnight).
- The intent of the tests is to exercise every function in interesting ways, so that when the programmer refactors code, she can quickly see
  (a) what would break if she made a change to a given variable, data item or function or
  (b) what she did break by making the change.

Refactoring support: How ?

- In the unit test situation, the programmer (not an independent tester) writes the tests, typically before she writes the code. The testing focuses the programming, yielding better code in the first place.

Refactoring support: How ?

- In the unit test case, when the programmer makes a change that has a side-effect, she immediately discovers the break and fixes it → There is no communication cost.
- You don't have (as you would in black box testing) a tester who discovers a bug, replicates it, reports it, and then a project manager who reads the report, maybe a triage team who study the bug and agree it should be fixed, a programmer who has to read the report, troubleshoot the problem, fix it, file the fix, a tester who has to retest the bug to determine that the fix really fixed the problem and then close the bug. All labor-hours considered, this can easily cost 4 hours of processing time, compared to a few minutes to fix a bug discovered at the unit level.

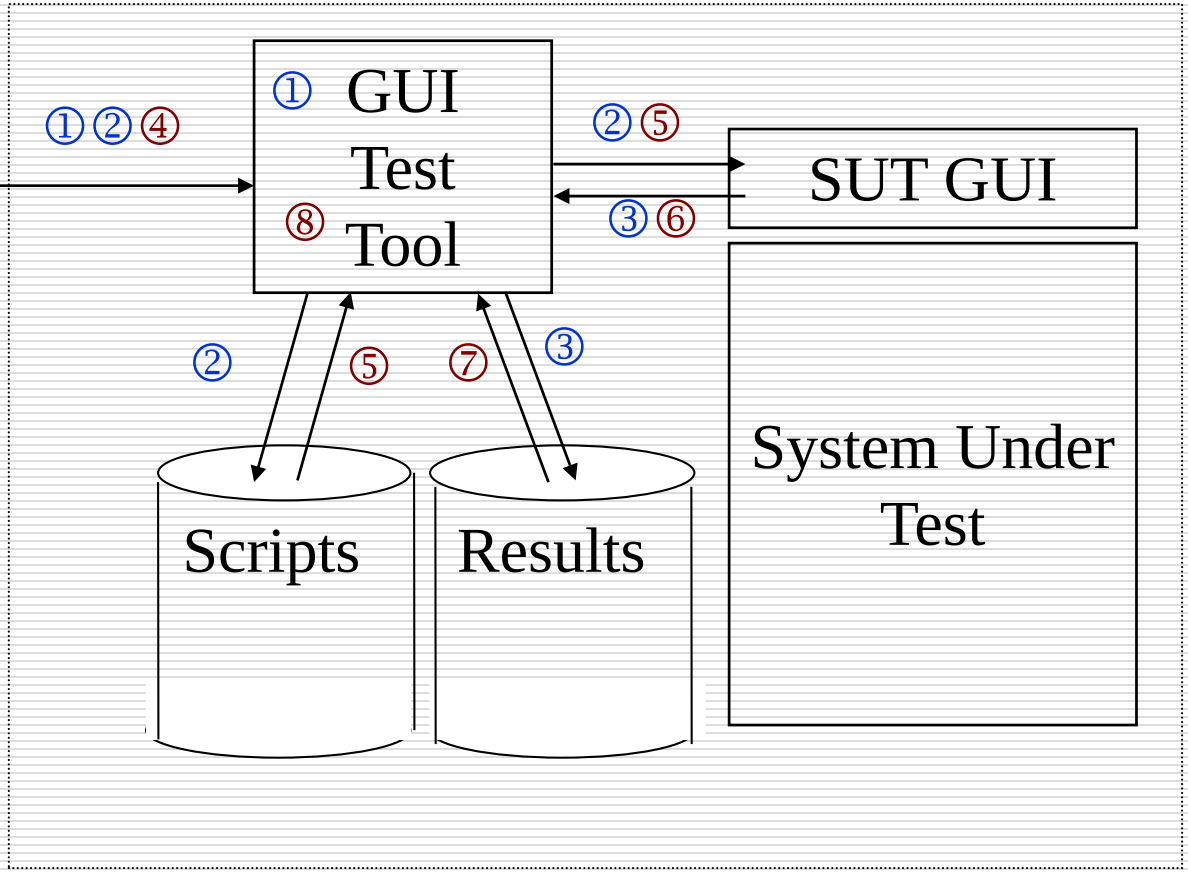
Automating GUI regression testing

Good or bad ?

Automating GUI regression testing

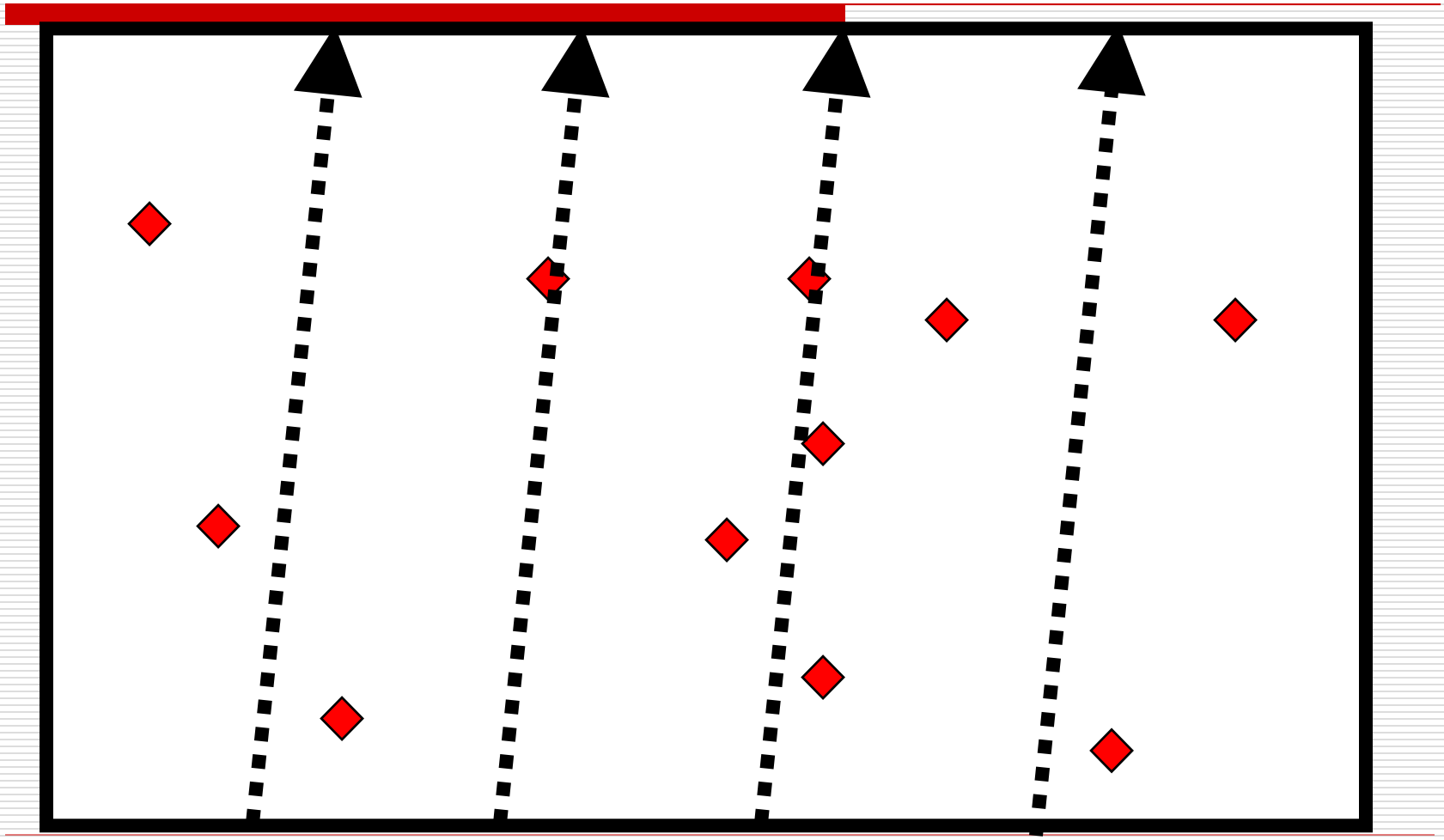
- This is the most commonly discussed automation approach:
  1. conceive and create a test case
  2. run it and inspect the output results
  3. if the program fails, report a bug and try again later
  4. if the program passes the test, save the resulting outputs
  5. in future tests, run the program and compare the output to the saved results
  6. report an exception whenever the current output and the saved output don’t match

A GUI Regression Test Model



① Launch tool
② Test; tool captures script
③ Test; capture result
④ Launch automated run
⑤ Play script
⑥ Capture SUT response
⑦ Read recorded results
⑧ Compare and report

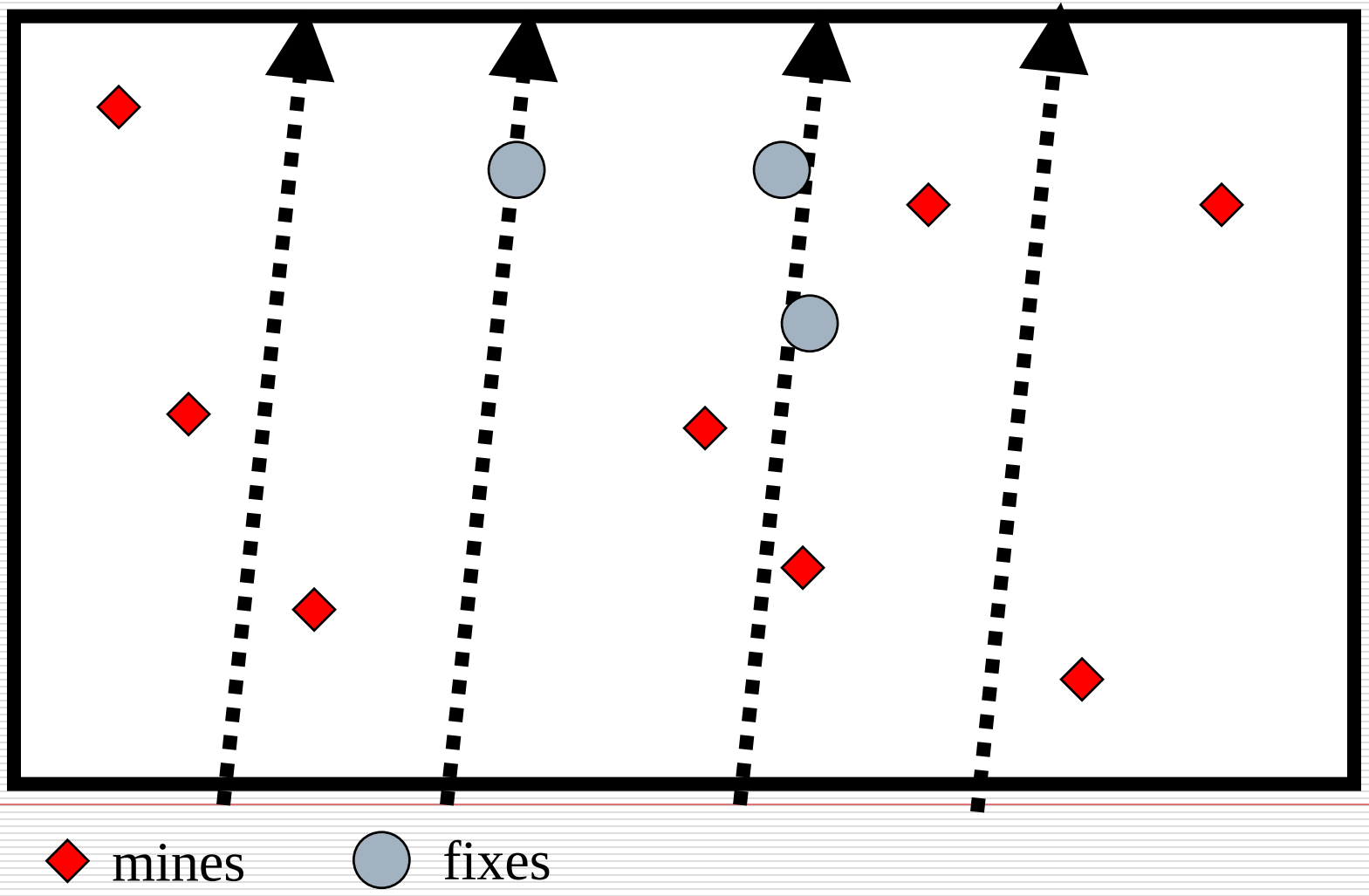
An analogy: Clearing mines



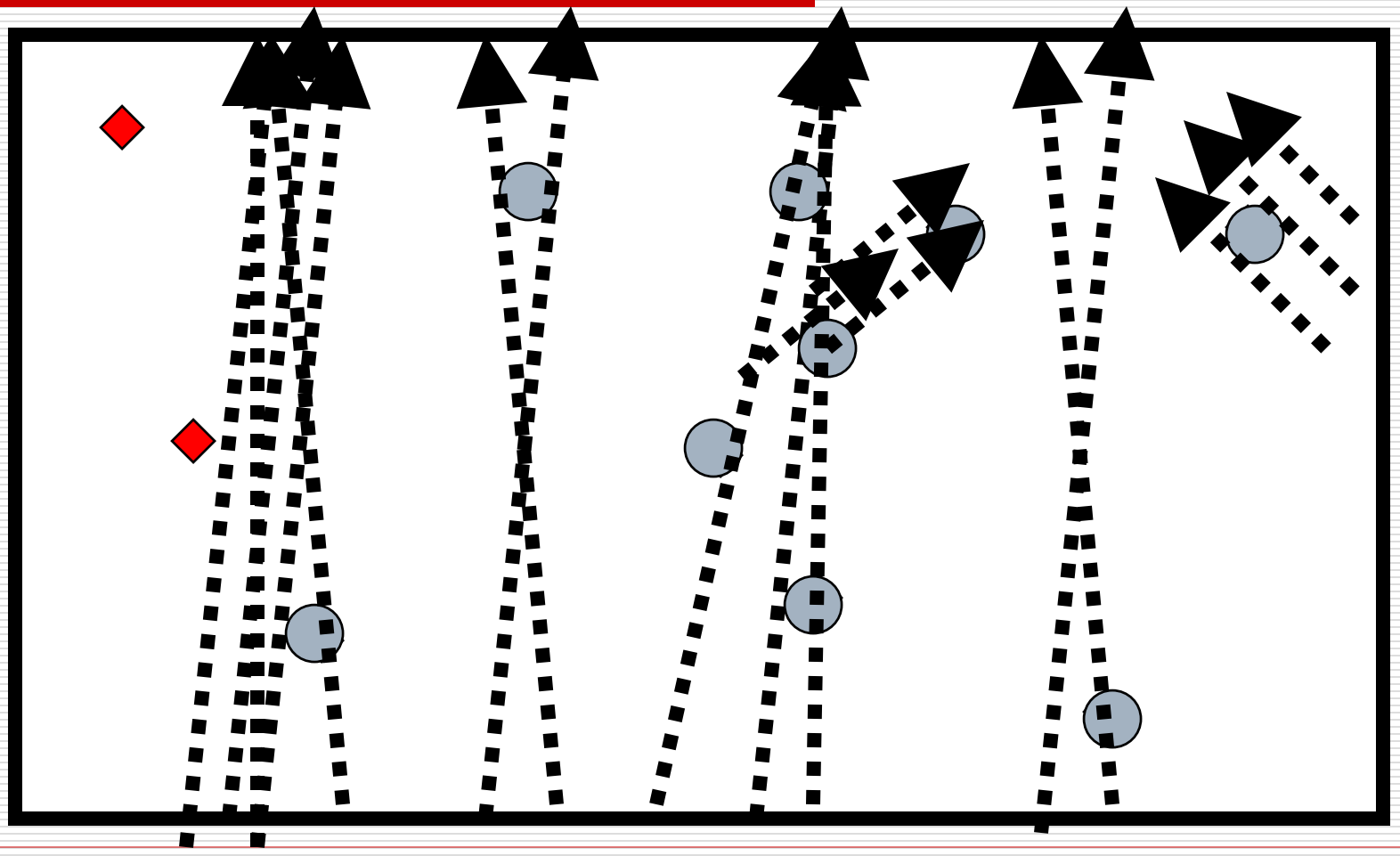
mines

This analogy was first presented by Brian Marick.
These slides are from James Bach..

Totally repeatable tests won't clear the minefield



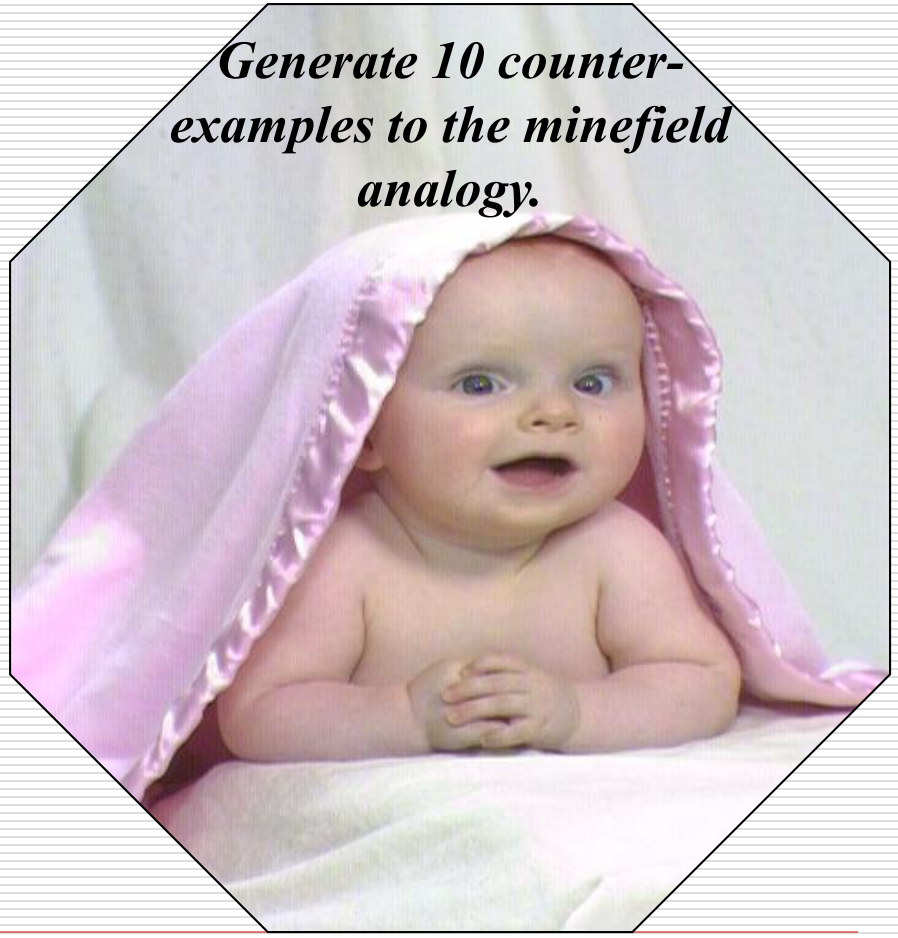
Variable Tests are Often More Effective



mines fixes

Automated GUI regression testing

- Look back at the minefield analogy
- Are you convinced that variable tests will find more bugs under all circumstances?
  - ***If so, why would people do repeated tests?***



Is this really automation?

- Analyze product -- human
- Design test -- human
- Run test 1st time -- human
- Evaluate results -- human
- Report 1st bug -- human
- Save code -- human
- Save result -- human
- Document test -- human
- **Re-run the test -- MACHINE**
- **Evaluate result -- MACHINE plus human if there's any mismatch**
- Maintain result -- human

GUI automation is expensive

- Test case creation is expensive. Estimates run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (LAWST).
- You usually have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation
- Automation can delay testing, adding even more cost (albeit hidden cost.)
- Excessive reliance leads to the 20 questions problem. \

GUI automation pays off late

- GUI changes force maintenance of tests
  - May need to wait for GUI stabilization
  - Most early test failures are due to GUI changes
- Regression testing has low power
  - Rerunning old tests that the program has passed is less powerful than running new tests
  - Old tests do not address new features
- Maintainability is a core issue because our main payback is usually in the next release, not this one

Test automation is programming

- Win NT 4 had 6 million lines of code, and 12 million lines of test code
- Common (and often vendor-recommended) design and programming practices for automated testing are appalling (kinh khủng):
  - **Embedded constants**
  - No modularity
  - ***No source control***
  - No documentation
  - `No requirements analysis`
  - **No wonder we fail.**

Common mistakes in GUI automation

- Don’t underestimate the cost of automation.
- Don’t underestimate the need for staff training.
- Don’t expect to be more productive over the short term.
- Don’t spend so much time and effort on regression testing.
- Don’t use instability of the code as an excuse.
- Don’t put off finding bugs in order to write test cases.
- Don’t write simplistic test cases.
- Don’t shoot for “100% automation.”
- Don’t use capture/replay to create tests.
- Don’t write isolated scripts in your spare time

Common mistakes in GUI automation (cont.)

- Don’t create test scripts that won’t be easy to maintain over the long term.
- Don’t make the code machine-specific.
- Don’t fail to treat this as a genuine programming project.
- Don’t “forget” to document your work.
- Don’t deal unthinkingly with ancestral code.
- Don’t give the high-skill work to outsiders.
- Don’t insist that all of your testers be programmers.
- Don’t put up with bugs and crappy support for the test tool.
- Don’t forget to clear up the fantasies that have been spoonfed to your management.

Think about:

- Automation is software development.
- Regression automation is expensive and can be inefficient.
- Automation need not be regression--you can run new tests instead of old ones.
- Maintainability is essential.
- Design to your requirements.
- Set management expectations with care.

Some Simple Automation Approaches

Getting Started With Automation of Software Testing

Six Sometimes-Successful "Simple" Automation Architectures

- Quick & dirty
- Equivalence testing
- Frameworks
- Real-time simulator with event logs
- Simple Data-driven
- Application-independent data-driven

Quick & Dirty

- Smoke tests
- Configuration tests
- Variations on a theme
- Stress, load, or life testing

Equivalence Testing

- A/B comparison
- Random tests using an oracle (Function Equivalence Testing)
- Regression testing is the weakest form

Framework-Based Architecture

- Frameworks are code libraries that separate routine calls from designed tests.
  - modularity
  - reuse of components
  - hide design evolution of UI or tool commands
  - partial salvation from the custom control problem
  - independence of application (the test case) from user interface details (execute using keyboard? Mouse? API?)
  - important utilities, such as error recovery
- For more on frameworks, see Linda Hayes' book on automated testing, Tom Arnold's book on Visual Test, and Mark Fewster & Dorothy Graham's excellent new book "Software Test Automation."

Real-time Simulator

- Test embodies rules for activities
- Stochastic process
- Possible monitors
  - Code assertions
  - Event logs
  - State transition maps
  - Oracles

Data-Driven Architecture

- In test automation, there are (at least) three interesting programs:
  - The software under test (SUT)
  - The automation tool that executes the automated test code
  - The test code (test scripts) that define the individual tests
- From the point of view of the automation software, we can assume
  - The SUT's variables are data
  - The SUT's commands are data
  - The SUT's UI is data
  - The SUT's state is data
  - The test language syntax is data
- Therefore it is entirely fair game to treat these implementation details of the SUT as values assigned to variables of the automation software.
- Additionally, we can think of the externally determined (e.g. determined by you) test inputs and expected test results as data.
- Additionally, if the automation tool's syntax is subject to change, we might rationally treat the command set as variable data as well.

Data-Driven Architecture

- In general, we can benefit from separating the treatment of one type of data from another with an eye to:
  - optimizing the maintainability of each
  - optimizing the understandability (to the test case creator or maintainer) of the link between the data and whatever inspired those choices of values of the data
  - minimizing churn that comes from changes in the UI, the underlying features, the test tool, or the overlying requirements
- You store and display the different data can be in whatever way is most convenient for you