Software Testing

Regression Testing

Beat Fluri

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Regression Testing

A new or modified component can fail when used with unchanged components

- Failures by generating side-effects or feature interactions

When a new or modified component causes a system failure, the system is said to *regress*

A **regression test case** is a test case that the *baseline* version of a component has passed and which is expected to pass when run on a *delta* version

Regression Testing

Regression test suite is a set of existing test cases

Does not contain tests for new or changed functionality

Plays an important role in revealing bugs

- The software failure in the Ariane 5 rocket controller was in large part due to an assumption that previously tested code would work when it was reused, obviating the need for regression testing.

Regression Testing - When and How

When a new subclass has been developed

Rerun the superclass test cases on the subclass, beside new test cases for the new subclass

When a superclass is changed

Rerun the superclass test cases on the superclass and on its subclasses. Rerun the subclass test cases

When a server class is changed

Rerun test cases for the clients of the changed class plus testing the changed class

When a bug fix has been completed

Rerun test cases on any part of the software that depend on the changed code to reveal unintended side effects of the bug fix

Regression Faults

Two components *B* and *D* pass individually their test suite, but fail when are integrated together

D has a side effect on *B*

D changes an entity used by *B*

D is a client of *B*

D sends a msg that violates *B*'s invariant or a precondition

D is a server of *B*

B sends a msg to *D*. *D*'s postconditions have changed or are buggy. The return value may cause a failure of *B*

Regression Faults

D is a new subclass of a polymorphic hierarchy that is not LSP-compliant

- *B* is a client of *D* and expects the new subtype to be compatible

D is incompatible with *B*

- Newer version of a library

Reducing a Test Suite

Full regression test can be too time consuming (setup or run)

Reduced regression test: Select a set of test cases for a specific test scenario

- In other words, which test cases can be skipped?

Safe reduction (likelihood of finding regression faults is not reduced)

Safe Reduction

Inclusiveness

Percentage of baseline test cases that may show regression faults; a safe regression test suite is 100 percent inclusive

Precision

Percentage of baseline test cases in a reduced test suite that *cannot* reveal regression faults and that are *not* selected for the reduced test suite

Efficiency

Cost of identifying a reduced regression test suite

Generality

The range of application for the selection strategy

Regression Test Patterns

Patterns describe which baseline test cases should be selected

Each pattern has a “safe reduction” evaluation with the predicates *inclusiveness*, *precision*, *efficiency*, and *generality*

Patterns

- Retest All
- Retest Risky Use Cases
- Retest Changed Code

Rest All

Rerun the entire baseline test suite on a delta version

Just run all existing tests

Safe reduction evaluation

Inclusiveness: All baseline tests are selected; => is safe

Precision: No tests that could be skipped are skipped; => the least precise

Efficiency: Lowest cost of analysis and setup. Highest run cost

Generality: Can always be applied. Easy to set up, no analysis or special development is necessary

Retest Risky Use Cases

Use risk-based heuristics to select a partial regression test suite. Run this subset on a delta version

Suspicious use cases: Select use cases that depend on components that

- are individually unstable or unproven
- have not been shown to work together before
- implement complex business rules
- have a complex implementation
- were subject to high churn during development (many changes spread all over the system)

Retest Risky Use Cases

Critical use cases: Select tests for use cases that are necessary for safe, effective operation

Safe reduction evaluation

Inclusiveness: Unsafe; test cases are not selected by analysis of dependencies

Precision: Some tests that could be skipped may be repeated

Efficiency: Setup cost are low: no code analysis is necessary; run cost may be high

Generality: Can always be applied. Easy to set up, no analysis or special development is necessary

Retest Changed Code

Use code change analysis to select a partial regression test suite. Run this subset on a delta version

Procedure

1. Use Cobertura to check which test case cover which source code parts
2. Use CVS/SVN `diff` information to see which source code parts changed
3. Rerun test cases that cover *deleted* or *changed* source code parts

Retest Changed Code

Safe reduction evaluation

Inclusiveness: All baseline test cases that can produce a different result are selected; => is safe

Precision: Most precise of the white box regression test strategies

Efficiency: If test case selection can be automated, the setup cost is low; in worst case (all baseline test case have to be selected) the total cost of this pattern can be higher than Retest All pattern

Generality: Class scope; => analysis knowledge and special development is necessary

Regression Testing and CI

Recommendation: make a separate Maven module and Hudson job

Retest all

- Copy all existing tests to the regression-testing module

Retest risky use cases

- Create a test suite with the risky use cases and configure maven

Retest changed code

- Write a script...