



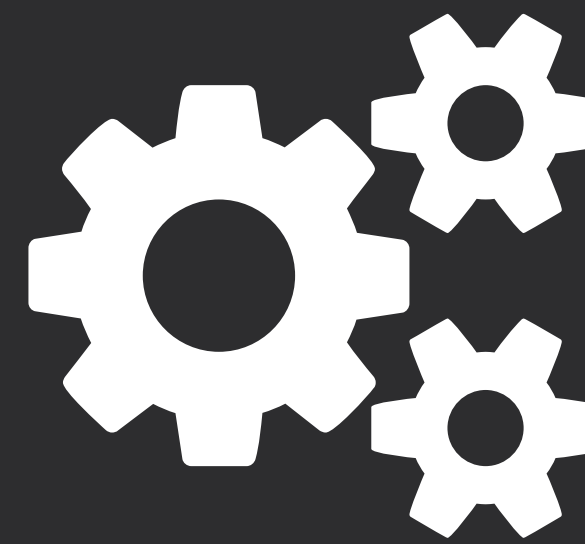
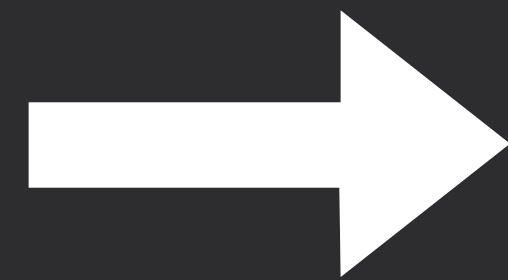
Master's Thesis: Regression Testing

Viktor Lövgren

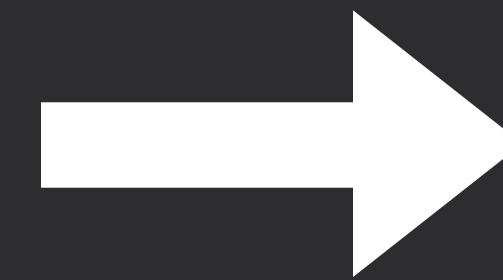




Current System



Make Changes



Next Version

Software Evolution

Did anything break?

Software Testing

Write tests to verify functionality.

- The system works if all tests pass. ✓
- The system is broken if any test fail. ✗

Also used to judge quality or acceptability.

Software Testing

Manual tests are run by humans.

- Spontaneously, or using checklists.
- Effort proportional to number of tests.

Automated tests are run by software.

- Requires effort in writing and maintaining.

Software Testing

Different abstraction levels for testing.

- Unit testing — for the smallest components.
- Integration testing — between components.
- System testing — for the system as a whole.

Many other abstraction levels are used.

Agile Software Development

Current development practices focus on:

- Iterative, incremental development.
- Delivery of most important requirements.
- Rapid software evolution.
- Receiving quick feedback.

Feedback Cycles

A feedback cycle means that we:

1. Change something (e.g. source code).
2. Find out how it went (e.g. using tests).
3. Learn from it (e.g. which tests failed).

Feedback Cycles

There are many different feedback cycles.

- Pair-programming (shortest cycle).
- Software- and regression testing.
- Scrum sprint cycle (longest cycle).

Feedback Cycles

Agile strategies try to shorten feedback cycles.

- Test-Driven Development (TDD).
- Extreme Programming (XP).

Try to keep them as short as possible.

Regression Testing

Agile strategies involve software testing.

- Automatic tests — no or few manual tests.
- Use tests for regression testing purposes.
- Number of tests are continually increasing.

Regression Testing

Run tests before and after changes.

- The system works if all tests pass. ✓
- The system is broken if any test fail. ✗

Assume tests passed before the changes.

Not concerned with test case correctness.

Regression Testing



More tests means increased execution time.

- It takes more time to verify the system.
- Prolonged feedback cycle times.
- Problem when it takes too much time.

Idea #1: Select some test cases for execution.

Test Case Selection

Takes too long time to execute all tests.

- Select some of all tests for execution.
- The system works if all tests pass. 
- The system is broken if any test fail. 

Problem: how to perform the selection?

Test Case Selection

Use a test case selection technique.

- Desirable to exclude passing tests.
- Desirable to include failing tests.

A safe technique only excludes passing tests.

Test Case Selection



Actual Status



Safe Technique



Unsafe Technique

Test Case Selection

Most techniques are modification-aware.

- Identify which parts have been changed.
- Select test cases relevant to the changes.

Common choice is to use static code analysis.

Test Case Selection

Metrics for evaluating test case selection.

- Inclusiveness - tests with different output.
- Precision - avoid tests with same output.
- Efficiency - computational cost of selection.
- Generality - handling arbitrary changes.

Test Suite Minimization

Idea #2: Eliminate redundant test cases.

- Surely, there are overlap in test cases.
- Some redundant tests could be removed.
- Either permanent or temporary reduction.

Sometimes also called *test suite reduction*.

Test Case Redundancy

When is a test case redundant?

The ideal definition would be:

- Test cases that execute with the same status, independently of code changes.

However, it is difficult to prove this property.

Test Case Redundancy

An alternative definition uses code coverage.

- Coverage means executed by test case.
- For example, method coverage criterion.
- Redundant if overlap in method coverage.

Also: statement coverage, class coverage, ...

Test Suite Minimization

Differences compared to *test case selection*:

- Minimization often seen as permanent.
- Selection is usually only temporary.
- Minimization independent of changes.
- Selection performed based on changes.

Test Suite Minimization

Metrics for evaluating test suite minimization.

- Inclusiveness - should not be affected.
- Precision - should also not be affected.
- Efficiency - cost of performing minimization.
- Generality - handle arbitrary complex code.

Test Case Prioritization

Idea #3: Prioritize the order of test execution.

- For example, try to detect faults earlier.
- Run failing tests as early as possible.
- Helps to shorten the feedback cycle.

Dependencies could affect the ordering.

Test Case Prioritization



Actual Ordering



Prioritized Ordering



Prioritized (Ideal)

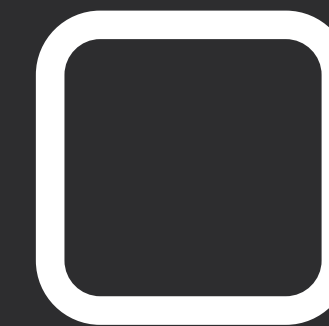
Test Case Prioritization (With Selection)



Actual Execution



Prioritized/selected



Ideal Execution

Test Case Prioritization

Problem: How to find the failing test cases?

- Executing the tests is not an option.
- Use some metric for each test case.
- Typically involves static code analysis.

For example, use number of covered methods.

Test Case Prioritization

Evaluating test case prioritization techniques.

- Depends on prioritization criteria.
- Fault detection: estimate the detection rate.
- Compare against without the technique.

Current Testing Practice

Survey by Engström and Runeson (2010).

- Software process improvement network.
- Utilized focus groups and questionnaires.
- 46 software engineers, 38 organizations.

Current Testing Practice

Some notable findings of the survey.

- Regression testing based on experience.
- Not utilizing researched testing techniques.
- Difficult to select regression test cases.
- Prioritizing test cases seen difficult.

Researched Techniques

First research papers on testing techniques.

- Selection (Fischer, 1977).
- Minimization (Harrold et. al, 1993).
- Prioritization (Rothermel et. al, 1999).

Many ideas, techniques, and tools are available.

Researched Techniques

Active research area for more than 37 years.

- Usage seems to be limited in industry.
- Why is the research not being used?
- Is it possible to use the research?

Thesis Approach

Attempt to reduce unit testing feedback cycles.

- Determine problems in a software project.
- Select a researched regression technique.
- Implement with the project context in mind.
- Evaluate the technique's performance.

Testing Feedback Cycles

The testing feedback cycle is when we:

1. Make a code change to the software.
2. Execute unit/integration/system tests.
3. See status, make additional changes.

Status can be seen as test cases execute.

Project Context

Software project at Valtech in Stockholm.

- Joint website for healthcare in Sweden.
- In development for 4 consecutive years.
- 4-6 developers are working full time.
- Agile approach (Scrum, XP, TDD, ...).

Project Context

Project with regression testing problems.

- 50 000 lines of code (40% is test-related).
- Code written in C#; NUnit used for unit tests.
- Mainly unit, integration, and system testing.
- Continuous integration execute tests.

Project Context

Meeting/interview with developer in project.

- Execution time perceived as problematic.
 - Risk for test cases not being executed.
- No test case selection efforts made.
- Unaware of prioritization techniques.

Selecting Technique

Four papers evaluated using four criteria.

1. Suitability for project context.
2. Implementation realization potential.
3. Partial implementation ability.
4. Potential for reducing feedback cycles.

Selecting Technique

Technique by Mansour and Statieh (2009).

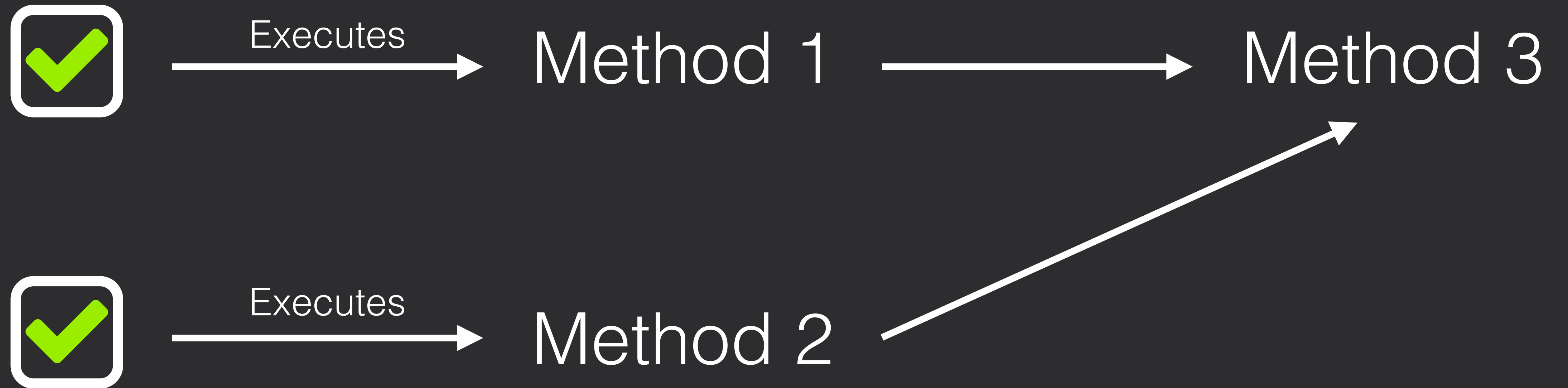
- Modification-based selection technique.
- Suggested metric for test case prioritization.
- Developed specifically for the C# language.
- Much pseudo-code, algorithm descriptions.

Technique Details

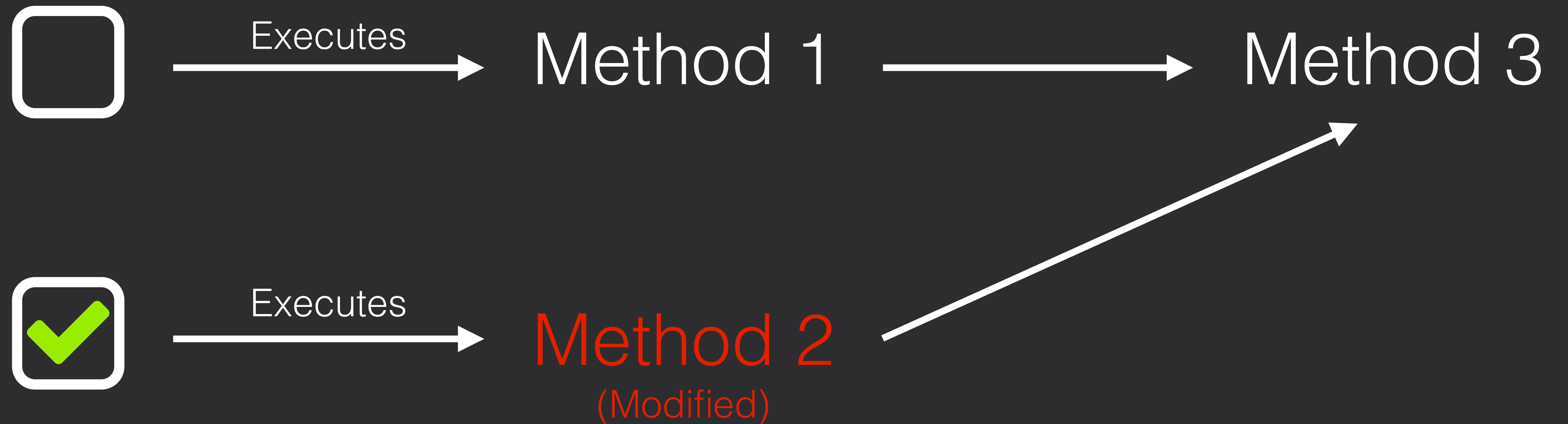
Technique consists of three phases.

1. Modification-based selection (methods).
2. Modification-based selection (instructions).
3. Further reduction and prioritization.

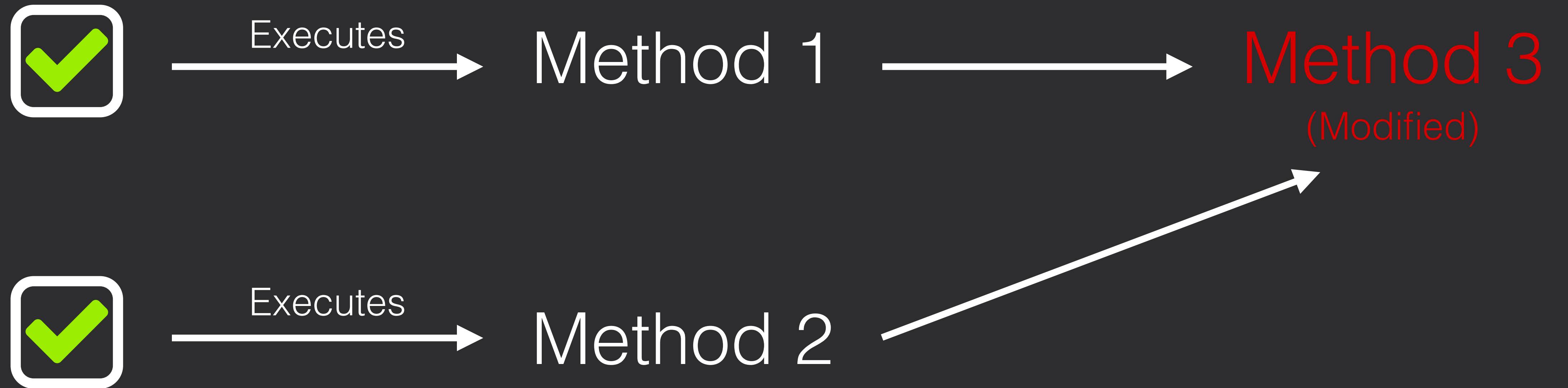
Phase 1: Modification-based selection



Phase 1: Modification-based selection



Phase 1: Modification-based selection



Phase 1: Modification-based selection

Steps for test case selection.

1. Find classes with modified/deleted methods.
2. Generate a class diagram for the program.
3. Determine test case-method coverage.
4. Select test cases based on previous steps.

Phase 1: Modification-based selection

1. Find classes with modified/deleted methods

- Save previous version of the program.
- Determine differences between versions.
 - Uses the intermediary format (IL code).
 - The Mono.Cecil library is used for this.

Phase 1: Modification-based selection

2. Generate a class diagram for the program

- Need to determine subclass relationships.
- Changes in superclass can affect subclass.
- Mono.Cecil supports the generation.

Phase 1: Modification-based selection

3. Determine test case-method coverage.

- Need to track which methods are executed.
- Attach trace attribute to every method.
 - Aspect-oriented programming helps.
 - PostSharp allows attachment with 1 line.

Phase 2: Modification-based selection



`Calc(5) == 10`



`Calc(6) == 6`

```
function Calc(int n) {  
    if(n <= 5)  
        return 2 * n;  
  
    return n;  
}
```


Phase 2: Modification-based selection

Using only the first phase.



`Calc(5) == 10`

```
function Calc(int n) {  
    if(n <= 5)  
        return 3 * n;  
    return n;  
}
```



`Calc(6) == 6`

```
    return n;  
}
```

Phase 2: Modification-based selection

Using the first and second phases.



`Calc(5) == 10`

```
function Calc(int n) {  
    if(n <= 5)  
        return 3 * n;  
    return n;  
}
```



`Calc(6) == 6`

```
    return n;  
}
```

Phase 2: Modification-based selection

Requires execution flow analysis.

- How to perform such analysis?
- Extensive tracing, storage/performance.
- This phase has not been implemented.

Phase 3: Further reduction and prioritization

Suggestions for the third phase include:

- Select one test case covering method.
- Use prioritization metric based on:
 - number of covered affected methods.
 - classes covering affected methods.

Phase 3: Further reduction and prioritization

Implementation for the third phase.

- Technique is kept safe; keep all tests.
- Use prioritization metric based on:
 - Covered affected classes/methods.
- Prioritization not easily done in NUnit.

Implementation Structure

Implemented as a NUnit addin.

- Seamless integration with NUnit.
- PostSharp required as dependency.
- One line of code setup per project.

Practical Considerations

- Requires storage space for coverage data.
- Handles changing test suites as well.
- Always re-execute failed test cases.

Practical Limitations

- Does not handle external dependencies.
- No efforts to detect simple renaming.
- Performance limited due to disk storage.
- Does not consider time/space requirements.
- Assumes only code changes affect status.

Robustness Evaluation

Implementation evaluated for robustness.

- Five technique-tailored evaluation cases.
- Provides some confidence for technique.
- Difficult to verify if all cases are handled.

Empirical Evaluation

Evaluation of the technique in the project.

- The project has 3230 unit tests.
- Reference execution time is 1m 19s.
- Seven evaluation cases were used.

Empirical Evaluation

Average savings using the selection technique.

- Read/write coverage to disk: $\sim 9\%$ (7s).
- Exclude writing coverage: $\sim 29\%$ (23s).
- Coverage kept in memory: $\sim 60\%$ (47s).

Empirical Evaluation

Prioritization savings (coverage in memory).

- For a case with 1 failing test: ~ 60% (24s).
- For a case with 18 failing tests:
 - Executing all failing tests: ~ 66% (51s).
 - Might be slower for fewer failing tests.

Discussion

Generality considerations.

- Limited to C# and the intermediary format.
- Requires Mono.Cecil and PostSharp.
- No external dependencies considered.
- Possible to support other test frameworks.

Discussion

Implementation efficiency.

- Storing coverage data to disk inefficient.
- Improvements of second phase excluded.
- Comparisons against uniform distribution.
- Evaluate the efficiency before using.

Discussion

Alternative approaches not considered.

- Parallelization of test case execution.
- Minimization techniques - remove tests.

Future Work

Suggestions for future work.

- Improve implementation performance.
- Extend NUnit to allow for prioritization.
- Continue to evaluate robustness.
- Consider time/space requirements.

Conclusions

- Possible to implement technique.
- Possible to handle test suite changes.
- Reduced feedback cycles ($\sim 66\%$, 50s).
 - Selection technique reduces the most.
- Evaluate suitability on a per-project basis.

Thanks!