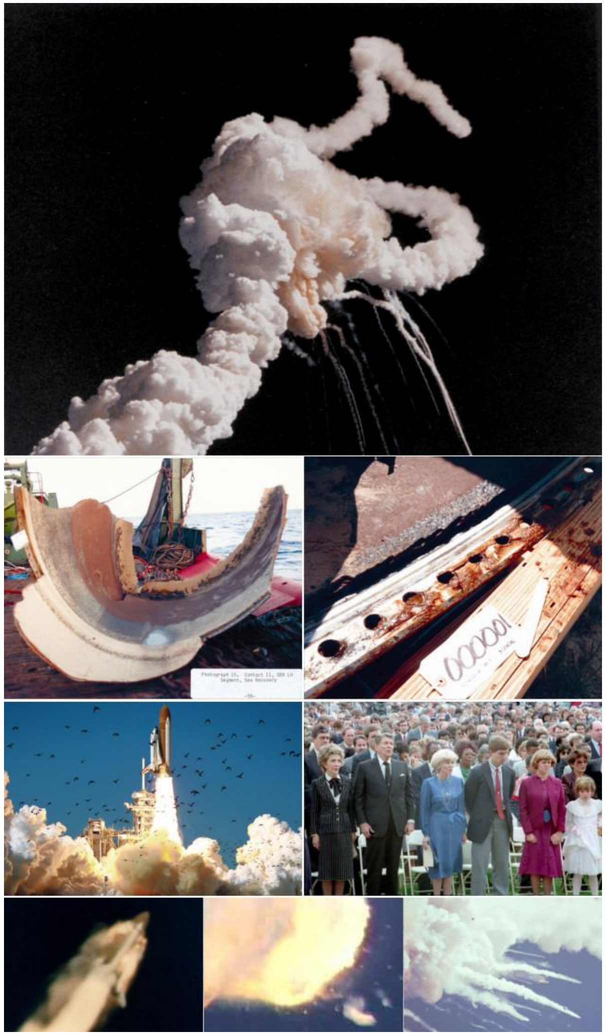
Fault-Based Testing

CSCE 747 - Lecture 14 - 02/23/2017

Space Shuttle Challenger

- January 28, 1986 - seal failure in a rocket booster causes the shuttle to explode, killing all seven astronauts.
- Three year investigation found technical and organizational issues.
- Became a case example studied in many forms of engineering.



Fault-Based Testing

By studying faults in previous designs, we can predict and prevent similar faults in future product designs.

Many testing techniques based on what we *think should happen*. We can also test based on knowledge of *what has gone wrong before.*

Used in Language Design

- Automated Garbage Collection
  - Prevents dangling pointers, memory leaks, other memory management faults.
- Automatic Array Bounds Checking
  - Does not prevent bad indexes from being used, but ensures they are noticed and limits damage.
- Type Checking
  - Prevents malformed values from being used as input or in computations.

Fault-Based Testing

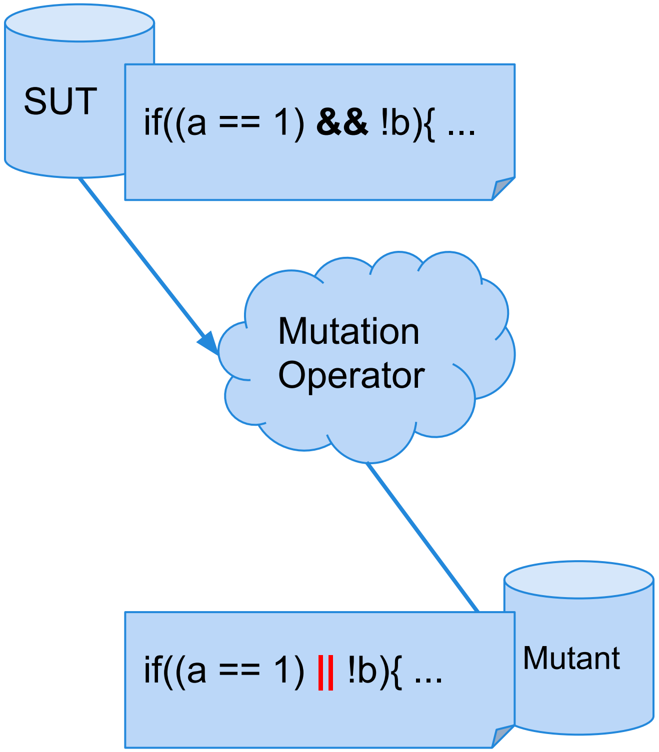
- Model the type of faults we expect to see in a program.
  - Create alternate versions of the program with those faults.
  - Design tests that distinguish the real program from the faulty program.
- Process of *fault seeding* - deliberately creating programs with faults to see if our tests can find those *intentional* faults.

Uses of Fault Seeding

- *Fault seeding* can be used to:
  - Judge the adequacy of a test suite.
  - Select test cases to augment a suite.
  - Estimate the number of faults in a program.
- Provides evidence that we have done a good job in testing.
  - If our tests have not found any new faults, have they found all major issues, or are they bad tests?
  - Fault seeding helps answer this question.
    - Can the existing tests find the seeded faults?

Mutation Testing

- Encode common syntactic faults as *mutation operators.*
  - Functions that take in candidate program statements and insert the modeled fault.
- Produces a *mutant.*
  - A clone of the program with 1+ seeded faults.



Mutation Operators

Mutation Operators

- Intended to model common types of faults.
- Designed to be applied to any type of code, without human intervention.
- Tend to be simple syntactic faults.
    - Replacing one variable reference with another.
    - Changing a comparison from < to <=.
    - Referencing a parent class instead of a child.

Operand Modifications

- X for Y replacement
  - Replace constant *C1* with constant *C2*.
  - Replace constant *C* with scalar variable *S*.
  - Replace scalar *S* for constant *C*.
  - Replace scalar *S1* with scalar *S2*.
  - Replace scalar/constant with array reference *A[I]*.
  - Replace array reference *A[I]* with scalar/constant.
  - Replace array reference with another array reference.
    - Either another array or another index in the same array.

Expression Modifications

- Arithmetic Operators
  - Binary operators: *x (+, -, *, /, %) y*
  - Unary operators: *+x, -x*
  - Shortcut operators: *x++, ++x, x--, --x*
- Arithmetic Operator Replacement
  - Replace binary/unary/shortcut operator with another.
  - Replace shortcut operator with a unary operator.
- Arithmetic Operator Insertion
  - Insert an additional operator into an expression.
- Arithmetic Operator Deletion
  - Remove an operator from an expression.

Expression Modifications

- Conditional Operators
  - Binary: *x (&&, ||, &, |, ^) y*
  - Unary: (~, *!)x*
- Relational Operators
  - *x (>, >=, <, <=, ==, !=) y*
- Shift Operators
  - *x (>>, <<, >>>>) y*
- (Conditional/Relational/Shift) Operator Replacement, Insertion Deletion

Expression Modifications

- Shortcut Operators
  - *x (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=) y*
  - Shortcut Operator Replacement
- Absolute Value Insertion
  - Replace a subexpression with *abs(e)*.
- Constant for Predicate Replacement
  - Replace a predicate *(a || b)* with a constant truth value *(true/false)*.

Statement Modifications

- Statement Deletion
  - Remove a random statement from the program.
- Switch Case Replacement
  - Replace the label of one case with another.
- End Block Shift
  - Move closing brackets to an earlier or later location.

Encapsulation/Inheritance Modifications

- Access Modifier Change
  - Change a modifier to *(public/protected/private)*
- Hiding Variable Deletion
  - Hiding variable - a variable in a subclass that has the same name and type as a variable in the parent.
  - Delete a hiding variable.
  - Causes references to that variable to access the version in the parent instead.
- Hiding Variable Insertion
  - Insert a hiding variable into a subclass.
  - Now, two variables of the same name exist.

Inheritance Modifications

- Overriding Method Deletion
  - Delete an overriden method from a subclass.
  - References call the version inherited from a parent.
- Overridden Method Calling Position Change
  - Overridden methods can call the parent method.
  - Moves calls to the parent version to other positions.
- Super Keyword Insertion/Deletion
  - Super keyword is used to access parent variables and methods within the child.
  - Inserts or deletes the keyword within methods.

Inheritance Modifications

- Overridden Method Renamed
  - Rename a method in the parent class that was overridden by the child.
  - Ensures that the overridden version is always called instead of the parent version.
- Explicit Parent Constructor Call Deletion
  - Deletes *super(parent)* constructor calls.
  - To kill, tests must cause and notice an incorrect initial state.

Polymorphism Modifications

- New Method Call with Child Class Type
  - Replace a declaration with a valid child instance.
    - *Parent a = new Parent();* becomes *Parent a = new Child();*
- Variable/Parameter Declaration With Parent Class Type
  - Change the declared type of a variable to its parent.
    - *Child a = new Child();* becomes *Parent a = new Child();*
    - *boolean equals(Child c){..}* becomes *boolean equals(Parent c){..}*

Polymorphism Modifications

- Type Case Operator Insertion/Deletion
  - Change the actual type of an object reference to the parent or child of the original type.
    - *p.toString()* becomes *((Child) p).toString()*
  - Or delete a type cast operator.
- Cast Type Change
  - *((SomeChild) c).toString()* becomes *((OtherChild) c).toString()*
- Reference Assignment with Other Compatible Type
  - Change an object reference to point to another compatible variable.
  - Object obj;
String s = “hello”;
Integer i = new Integer(4);
obj=s;

    becomes

    Object obj;
String s = “hello”;
Integer i = new Integer(4);
obj=**i**;

Polymorphism Modifications

- Overloading allows 2+ methods to have the same name if they have different signatures.
- Overloading Method Contents Change
  - Replace the body of a method with the body of another method with the same name.
- Overloading Method Deletion
  - Deletes one of the overloading methods.
- Argument of Overloading Method Change
  - Changes the order or number of arguments in an invocation, as long as there is a version that will accept the list.

Language-Specific Modifications

- Mutation operators can be written for a particular language.
- Java:
  - *this* insertion/deletion
  - Static modifier insertion/deletion
  - Member variable initialization deletion
  - Default constructor deletion
  - Getter/Setter method replacement

Mutation Testing

Mutation Testing

- **Select *mutation operators*** - code transformations that represent classes of faults that we are interested in.
- **Generate *mutants*** by applying mutation operators to the program.
- Execute the same tests against the program and mutants to ***kill* mutants**.
  - A mutant is killed if the test passes on the original program and fails on the mutant.
  - A mutant not killed is considered *live*.

Mutation Testing

- Most mutation operators reflect small syntactic mistakes.
- Programmers do make such mistakes. However, many faults are actually **conceptual** mistakes.
    - Mistaken assumptions about requirements.
    - Forgotten requirements.
- Is mutation testing a viable technique?

Viability of Mutation Testing

- Mutation testing is valid if seeded faults are *representative* of real faults.
- *Competent Programmer Hypothesis*
  - A faulty program differs from a correct program only by a small textual change.
  - If so, we only have to distinguish the program from all such small variants.
  - Assumption: the SUT is “close to” correct.

Coupling Effect

- Many faults are small syntactical errors.
- Conceptual faults often manifest as syntactical errors.
- Complex faults may result in larger textual differences.
    - However, mutation testing is still valid if test cases for simple issues can detect complex issues.
    - *Coupling Effect Hypothesis* - complex faults can be modeled as a set of small faults.

Coupling Effect

- A complex change to a program is a series of small changes.
- If one of these small changes is not masked by the effects of other changes, then a test case that can notice that change may also detect a more complex change.
- Mutation testing is effective if both the competent programmer hypothesis and coupling effect hypothesis hold.

Mutant Quality

To be used in testing, mutants must be:

- Syntactically correct (*valid*)
    - Mutants must compile and execute.
- Plausible (*useful*)
    - Must provide information on how the system works.

Can a mutant be valid, but not useful?

Mutant Quality

Mutants might remain live if:

- They are *equivalent* to the original program.
  - for(i=0; i < 10; i++)
  - for(i=0; i != 10; i++)
  - Identifying equivalency is NP-hard.
- Test suite is *inadequate* for that mutation.
  - (a <= b) and (a >= b) cannot be differentiated if a==b in the test case.

Mutation Coverage

Adequacy of the suite can be measured as:

$$\frac{\text{(\# mutants killed)}}{\text{(total mutants)}}$$

- Mutants can be equivalent when both the original and the mutant are wrong.
- Helps ensure that the test suite is *robust* against the modeled mutation types.

Mutation and Structural Coverage

Mutation coverage can subsume structural coverage metrics.

- Statement Coverage
  - Apply statement deletion to all statements.
  - To kill a mutant where statement *S* has been deleted requires executing *S* in the original program.
- Branch Coverage
  - Apply constant replacement to all predicates.
  - To kill a mutant where a predicate is set to true, a test must execute the original with a false value.

Practical Considerations

Mutation testing is expensive.

- Must run *all* tests against *all* mutants.
- Many mutants typically generated.
  - One mutation operator applied per mutant.

- If cost is an issue, use "weak" mutation testing:
  - Apply multiple mutation operators per mutant.

Weak Mutation Testing

Mutation testing is expensive.

- Must run *all* tests against *all* mutants.
- Many mutants typically generated.
  - One mutation operator applied per mutant.

- If cost is an issue:
  - “weak” mutation testing - seed multiple faults per mutants.
  - Sample from space of mutants until statistical significance is achieved.

Weak Mutation Testing

- Seed multiple faults into a single mutant.
  - Called a “meta-mutant”
- Divide the program into segments and track internal state of both original and all mutants when executing a segment.
- Kill all detected mutants when intermediate state differs instead of waiting for output.
- Decreases the number of test executions.

Statistical Mutation Testing

- A test suite that kills *some* mutants may be as effective at finding real faults as one that kills *all* mutants.
- Mutation testing can be used to obtain a statistical estimate of the ability of the suite to detect mutations.
  - Randomly generate *N* mutants.
  - Samples must be a valid statistical model of occurrence frequencies of real faults.
  - Target 100% coverage over the sample.

Estimating Number of Real Faults

- Mutants can be used to estimate the number of remaining faults in a program.

$$\frac{\textbf{Number of Seeded Faults}}{\textbf{Number of Real Faults}} = \frac{\textbf{Seeded Faults Detected}}{\textbf{Real Faults Detected}}$$

- **Be careful!**
  - We must have a reason to believe that our tes as effective as real faults as seeded faults.
  - Fault model must reflect the real program.
  - These assumptions are rarely true.

Activity

1. How many mutations are possible for Relational Operator Replacement, Arithmetic Operator Replacement
2. Apply relational operator replacement operation to statement 4, design a test that would kill that mutant.
3. Design an equivalent mutant.
4. Design a valid, but useless mutant.

```
public int[] makePositive(int[] a){
	int threshold = 0;
	for(int i=0; i < a.length; i++){
		if(a[i] < threshold){
			a[i]= -a[i];
		}
	}
	return a;
}
```

Activity - Solution

- How many mutations are possible:
  - Relational Operator Replacement:
    - `for(int i=0; i < a.length; i++){`
      - (>=, <, <=, ==, !=), 5 mutations
    - `if(a[i] < threshold){`
      - (>, >=, <=, ==, !=), 5 mutations
  - Arithmetic Operator Replacement
    - `for(int i=0; i < a.length; i++){`
      - Shortcut replacement, (++*i, i--, --i*), 3 mutations
    - `a[i]= -a[i];`
      - Unary replacement, (+a[i]), 1 mutation
      - Unary to shortcut replacement, *(a[i]++, ++a[i], a[i]--, --a[i]*), 4 mutations

Activity - Solution

- Apply the relational operator replacement operation to statement 4:
    - `if(a[i] < threshold){` becomes:
    - `if(a[i] == threshold){`
- Design a test case that would kill that mutant.
    - a[-1,0,1]
    - -1 would not become positive.

Activity - Solution

- **Design an equivalent mutant.**
  - Can do so by applying the relational operator replacement operation to statement 4:
    - `if(a[i] < threshold){` becomes:
    - `if(a[i] <= threshold){`
  - Since threshold=0, and -0 = 0, no test would detect this fault.
  - Does not help us test, as the fault cannot cause a failure.

Activity - Solution

- **Design a valid, but useless mutant.**
  - For example: mutant that compiles, but trivially fails.
  - Apply the relational operator replacement operation to statement 4:
    - `if(a[i] < threshold){` becomes:
    - `if(a[i] > threshold){`
    - Any positive numbers are made negative, all negative remain negative. Almost any test would detect this.
  - **Many** mutants are useless for detecting real faults.

We Have Learned

- Mutation testing is the process of inserting faults to help develop a test suite that can detect unknown real faults.
- Mutation operators automatically create faulty versions of a program.
  - Operators model expected fault types.
- Tests are judged according to their ability to detect faults.

Next Time

- Midterm Review
    - Practice Midterm is up on the course site.
    - Try to answer the questions - bring your answers on Tuesday.
    - Answer key will go up after next class.