CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

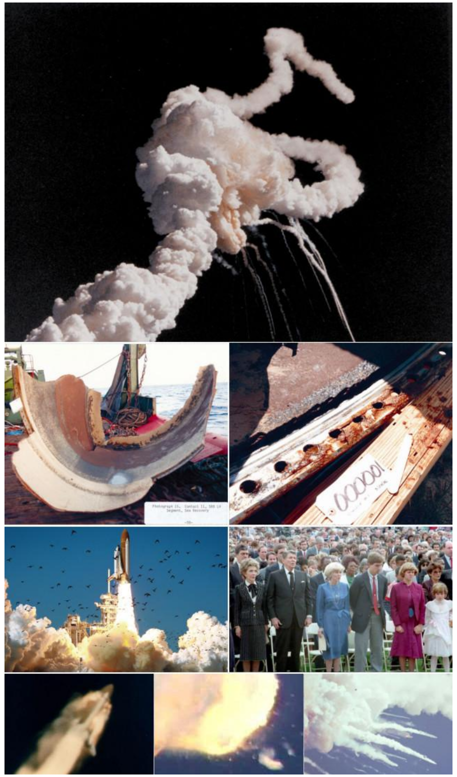
Lecture 11: Mutation Testing

Gregory Gay
DIT636/DAT560 - February 23, 2026

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Space Shuttle Challenger

- Seal failure in rocket booster causes explosion, killing seven astronauts.
- Investigation found technical and organizational issues.
  - Became a case example studied in many forms of engineering.
  - **Learn from your failures.**



Fault-Based Testing

- By studying faults in previous designs, we can prevent similar faults in new designs.
- Many testing techniques based on what we ***think should happen***.
- We can also design tests based on knowledge of ***what has gone wrong in other programs***.

Implemented 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
  - Prevent malformed value use in input or computations.

Fault-Based Testing

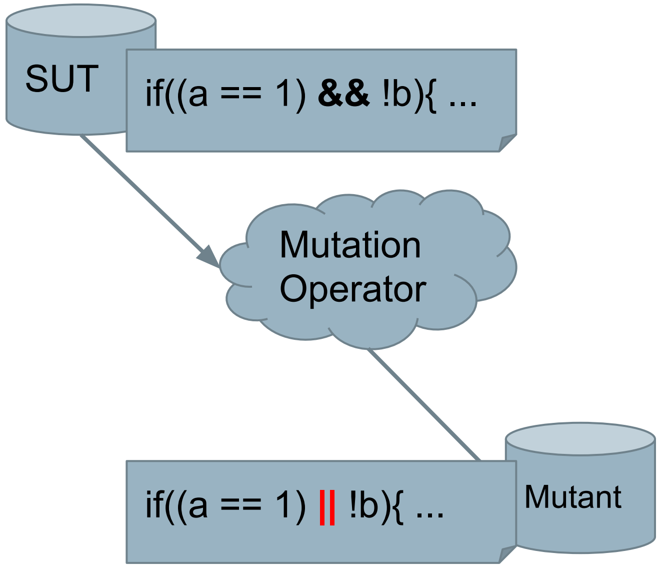
- Consider the types of faults we expect to see.
  - Create **mutated** versions of the program.
  - See if tests fail for those mutated versions.
- **Fault Seeding**
  - Deliberately creating programs with faults to see if our tests are good enough to detect them.
  - May help us find new faults in the unmutated program.

Uses of Fault Seeding

- Fault seeding can be used to:
  - Judge the adequacy of a test suite.
    - **Alternative to code coverage**.
  - Design test cases to augment a suite.
- Provides evidence that we have done a good job.
  - If our tests have not found faults, are there no more major issues, or are they bad tests?

Mutation Testing

- Encode common faults as **mutation operators**.
  - Insert the modeled fault into program statements.
- Produces a **mutant**.
  - A clone of the program with a seeded fault.



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.

Mutation Operators

```
public class MyCode {
    …
    public void myFunction (...) {
        Object x = (a + ((b - x[1]) / 3));
        Object y = this.y;
        ~~Object z = …;~~
    }
    …
}
```

Object-Oriented Mutations

Operand Mutations

Expression Mutations

Language-Specific Mutations

Statement Mutations

Operand Modifications

```
public class MyCode {
    …
    public void myFunction (...) {
        Object x = (a + ((b - c[1]) / 3));
        Object y = this.z;
        ~~Object z = …;~~
    }
    …
}
```

Replace constant *C1* with constant *C2*.

3 -> 15

Replace constant *C1* with variable *S*.

3 -> a

Replace variable *S* with constant *C1*.

a -> 10

Replace variable *S1* with variable *S2*.

z -> x

Operand Modifications

```
public class MyCode {
    …
    public void myFunction (...) {
        Object x = (a + ((b - c[1]) / 3));
        Object y = this.z;
        ~~Object z = …;~~
    }
    …
}
```

Replace variable or constant with array reference A[i].

3 -> c[5]

Replace array reference A[i] with variable or constant.

c[1] -> a

Replace array reference A1[i] with array reference A2[y].

(another array or another index in same array)

c[1] -> c[5]

Expression - Arithmetic Operators

```
public class MyCode {
    …
    public void myFunction (...) {
        Object x = (a + ((b - c[1]) / 3));
        Object y = this.z;
        ~~Object z = …;~~
    }
    …
}
```

Replace one arithmetic operator with another

```
(b - c[1]) -> (b + c[1])
```

Replace one shortcut operator with another

```
(b++) -> (b--)          x += y -> x /= y
```

Insert an arithmetic operator (and operand)

```
Object x = (a + ((b - c[1]) / 3)) / 5;
```

Insert a shortcut operator.

```
Object x = (++a + ((b - c[1]) / 3));
```

Delete an arithmetic operator (and operand)

```
Object x = (a + ((b ~~- c[1])~~ / 3));
```

Delete a shortcut operator.

```
(b++) -> (b)
```

Expression - Relational Operators

```
public void myFunction (...) {
    int x = (a + ((b - c[1]) / 3));
    if (x >= 5) {
        Boolean y = ((m && n) || o);
    }
}
```

Replace one relational operator with another

`(x >= 5) -> (x != 5)`

Replace one boolean operator with another

`((m && n) || o) -> ((m || n) || o)`

Insert or delete relational and boolean operators.

`((m && n) || o) -> ((m && n) || o) && p`

((m && n) || o) -> ((m ~~&& n~~) || o)

Expression Modifications

- Absolute Value Insertion
  - Replace a subexpression with *abs(e)*.
    - `int Z = X + Y; -> int Z = abs(X + Y);`
- Constant for Predicate Replacement
  - Replace boolean predicate with a constant value *(T/F)*.
    - `bool Z = (A || B) && C; -> bool Z = (A || true) && C;`

Statement Modifications

```
public class MyCode {
    …
    public void myFunction (...) {
        Object x = (a + ((b - c[1]) / 3));
        Object y = this.z;
        ~~Object z = …;~~
    }
    …
}
```

Delete a random statement.

~~Object z = …;~~

Replace labels in a switch statement.

case **1:** -> case **2:**

Move closing brace up or down one line.

```
    Object z = …;    ->    }
}                              Object z = …;
```

Encapsulation/Inheritance

- Access Modifier Change
  - Change a modifier to *(public/protected/private)*
  - `public void DoThis(int x)` -> `private void DoThis(int x)`

Inheritance Modifications

- Overriding Method Deletion
  - Delete an overriden method from a subclass.
  - References call the version inherited from a parent.

  - `Class Child implements Parent { …`
    ~~`@Override public int doThis(){ .. } …`~~
    `int X = doThis(); }`

Inheritance Modifications

- Super Keyword Insertion/Deletion
  - Inserts or deletes the `super()` keyword.

  -
    ```
    @Override
    public void doSomething(){
        super(); … } ->
    @Override
    public void doSomething(){
        … }
    ```

Inheritance Modifications

- Super Calling Position Change
  - Moves calls to the parent version to other positions.

  -
    ```
    @Override
    public int doThis(){
        int x = super(); int y = 5; ...  }   ->

        int y = 5; ... int x = super();  }
    ```

Inheritance Modifications

- Explicit Parent Constructor Call Deletion
  - Deletes *super()* call in a **constructor**.
  - To detect, tests must detect an incorrect initial state.

  - Class Child implements Parent {
        int x;
        public Child () { **super();** ... } } ->
    Class Child implements Parent {
        int x;
        public Child () { ... } }

Polymorphism Modifications

- Declaration with Child Class Type
  - Replace a declaration with a valid child instance.
    - `Parent a = new Parent();` -> `Parent a = new Child();`
- Declaration With Parent Class Type
  - Change the declared type of a variable to its parent.
    - `Child a = new Child();` -> `Parent a = new Child();`
    - `boolean equals(Child c){..}` -> `boolean equals(Parent c){..}`

Polymorphism Modifications

- Type Cast Operator Insertion/Deletion
  - Cast the type of an object reference to the parent or child of the original type.
    - `p.toString() -> ((Child) p).toString()`
  - Or delete a type cast operator.
    - `((Child) p).toString()-> p.toString()`

- Cast Type Change
  - Changes a cast to another valid data type.
  - `((SomeChild) c).toString() -> ((OtherChild) c).toString()`

Language-Specific Modifications

- Mutation operators written for a particular language.
- Java:
  - *this* insertion/deletion
  - Static modifier insertion/deletion
  - Member variable initialization deletion
  - Default constructor deletion

Mutation Testing

Mutation Testing

- Select mutation operators.
- Generate mutants by applying mutation operators.
- Execute tests against original class and 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

- Mutation operators reflect small syntactic mistakes.
  - **Programmers do make such mistakes!**
- However, many faults are ***conceptual*** mistakes.
  - Mistaken assumptions about requirements.
  - Forgotten requirements.
- **Is mutation testing a reasonable technique for judging test adequacy?**

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 small textual changes.
  - 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 syntax errors.
- Complex faults 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 is a series of small changes.
  - If one change not covered up by others, a test that exposes it can also detect a more complex change.
- Mutation testing effective if **both** competent programmer and coupling effect hypotheses hold.

Judging Test Sensitivity

- Mutants are often simpler than real faults.
- Mutation is still good at judging **sensitivity of your tests to minor changes in the code**.
  - If tests can distinguish mutants from the real code, then your tests execute the code *thoroughly*.
  - If you miss mutants, you can add new tests to detect them and make your suite more sensitive.

Mutant Quality

To be used in testing, mutants must be:

- Syntactically correct (***valid***)
  - Mutants must compile and execute.
- Plausible (***useful***)
  - Must provide valuable information on how the system works for testers working to improve the system.
- **A mutant can be valid, but not useful.**
  - All or almost all tests fail.

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.

Mutant Type Summary

	Valid	**Invalid**
Useful	Few Tests Detect Mutant	Does Not Compile
Not Useful	Almost All Tests Detect Mutant	Does Not Compile
Equivalent	Output Always Same As Original Program	Does Not Compile

Let’s Take a Break

Mutation Coverage

Adequacy of suite can be measured as:

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

- Helps ensure that the test suite is robust against the modeled mutation types.
    - Ensures that suite is sensitive to small changes in code.

Practical Considerations

Mutation testing is **expensive**.

- Must run *all* tests against *all* mutants.
- Many mutants typically generated.
  - One mutation operator applied per mutant.
  - May be dozens - hundreds per class.
- Can randomly choose X mutants or operators.

Statistical Mutation Testing

- A test suite that kills *some* mutants may be as effective as one that kills *all* mutants.
- 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.

Mutation Testing at Google

- Very large codebase, so using all mutants or using mutants often impractical.
  - Skip lines not covered by tests.
  - Skip “uninteresting” lines.
    - Logging, testing, timing, loop conditions.
- Used during code reviews.
  - Present undetected mutants to suggest new tests or potential code mistakes.

Activity

1. How many mutations are possible for Relational Operator Replacement, Constant-for-Constant Replacement
2. Apply relational operator replacement operation to line 4, choose input that will show different output from original.
3. Design an equivalent mutant.
4. Design a valid, but not useful 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:
    - Two lines can be mutated
      - `for(int i=0; i < a.length; i++){`
      - `if(a[i] < threshold){`

Activity - Solution

- How many mutations are possible:
  - Constant-for-Constant Replacement
    - Two lines can be mutated.
      - `int threshold = 0;`
      - `for(int i=0; i < a.length; i++){`

Activity - Solution

- Apply the relational operator replacement operation to statement 4:
  - `if(a[i] < threshold){    ->`
  - `if(a[i] == threshold){`
- Choose test input 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.
  - Does not help us test, as the fault cannot cause a failure.

Activity - Solution

- **Design a valid, but not useful mutant.**
  - Compiles, but trivially fails.
  - Apply relational operator replacement 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 not useful.

Activity

- **Valid-but-useful?**
  - Compiles, but is subtle and hard to detect.
  - Valuable when testing - we need the right test to detect.

  - `int threshold = 2;`
    - Constant-for-constant
    - Only detected if the input array contains 1 in it.
    - If we check boundary values, we might catch this, but otherwise could miss it!

PITest Demo (https://pitest.org/)

We Have Learned

- Mutation testing inserts faults to judge test suite sensitivity and adequacy.
- Mutation operators automatically create faulty versions of a program.
  - Operators model expected syntactic faults.
- Tests are judged according to their ability to detect faults - useful sensitivity analysis.

Next Time

- Model-Based Testing
- Exercise Session: Mutation Testing

- Assignment 3 due March 1.

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY