CS 130 SOFTWARE ENGINEERING

DISCUSSION:
TESTING AND CODE REVIEW

Professor Miryung Kim
UCLA Computer Science
Based on Materials from Miryung Kim

AGENDA

- Discussion: Why study Hoare Logic?
- Discussion: Modern Code Review Practices
- Remaining Question on Loop Invariant
  - Strengthening a loop invariant
  - Recap Exercise: Power

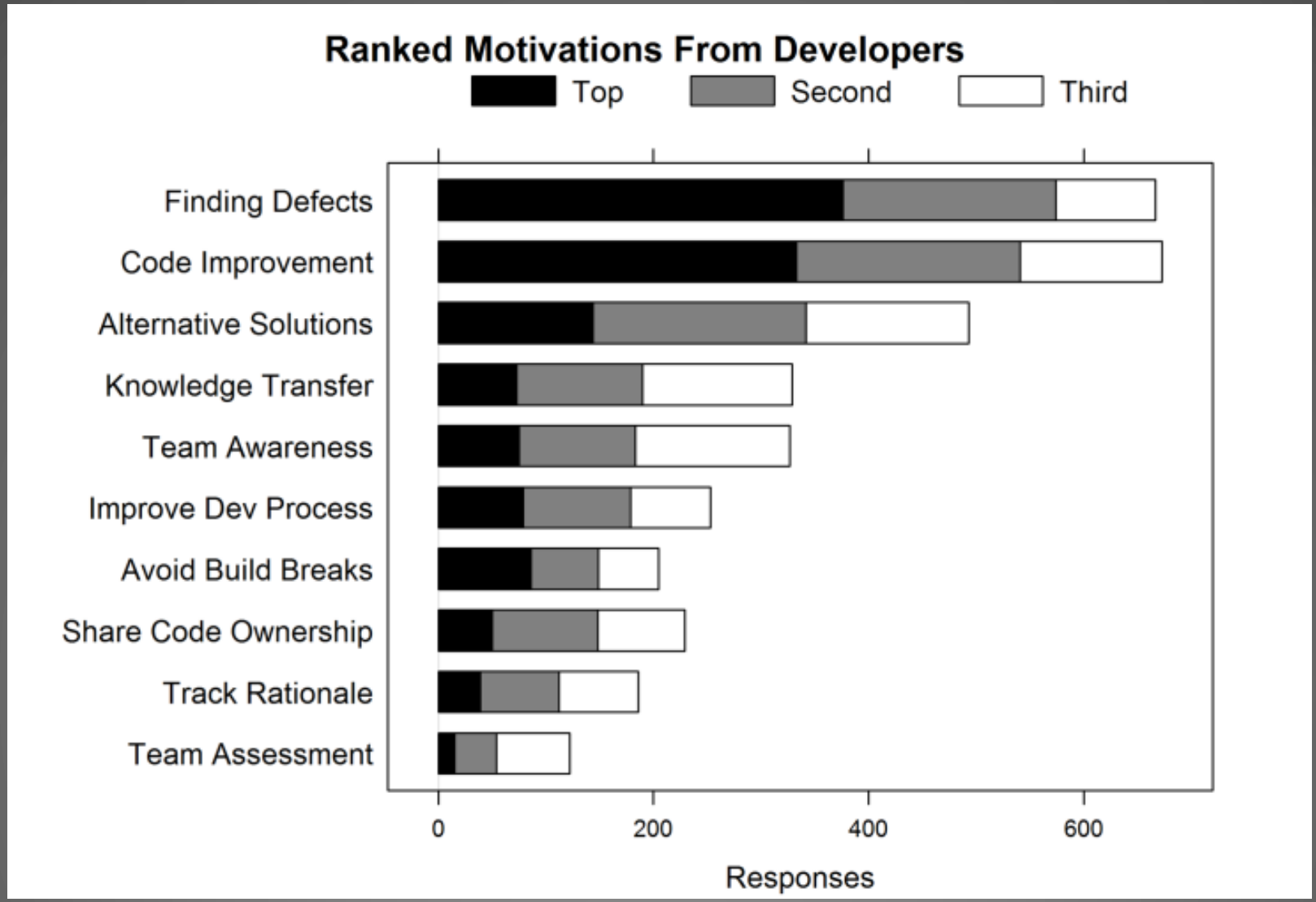
AGENDA

- Discussion on Practical Testing:
  - Systematic and Random Testing
  - Model Based Testing
  - Mutation Testing

MODERN CODE REVIEW PRACTICES

WHY CODE REVIEW DO NOT FIND BUGS. HOW CURRENT CODE REVIEW PRACTICES SLOWS US DOWN.

- *ICSE 2015, Jacek Czerwonka et al. Microsoft*
  - They mined code review tool usage data
  - CodeFlow is open 5 or 6 hours per developer per day.
  - 30 minutes of interaction time per developer per day.
  - **Primary goals:** Find defects, improve maintainability, share knowledge, broadcast progress.



EXPECTATIONS, OUTCOMES, AND CHALLENGES OF MODERN CODE REVIEW, ICSE 2013, BACCHELLI AND BIRD

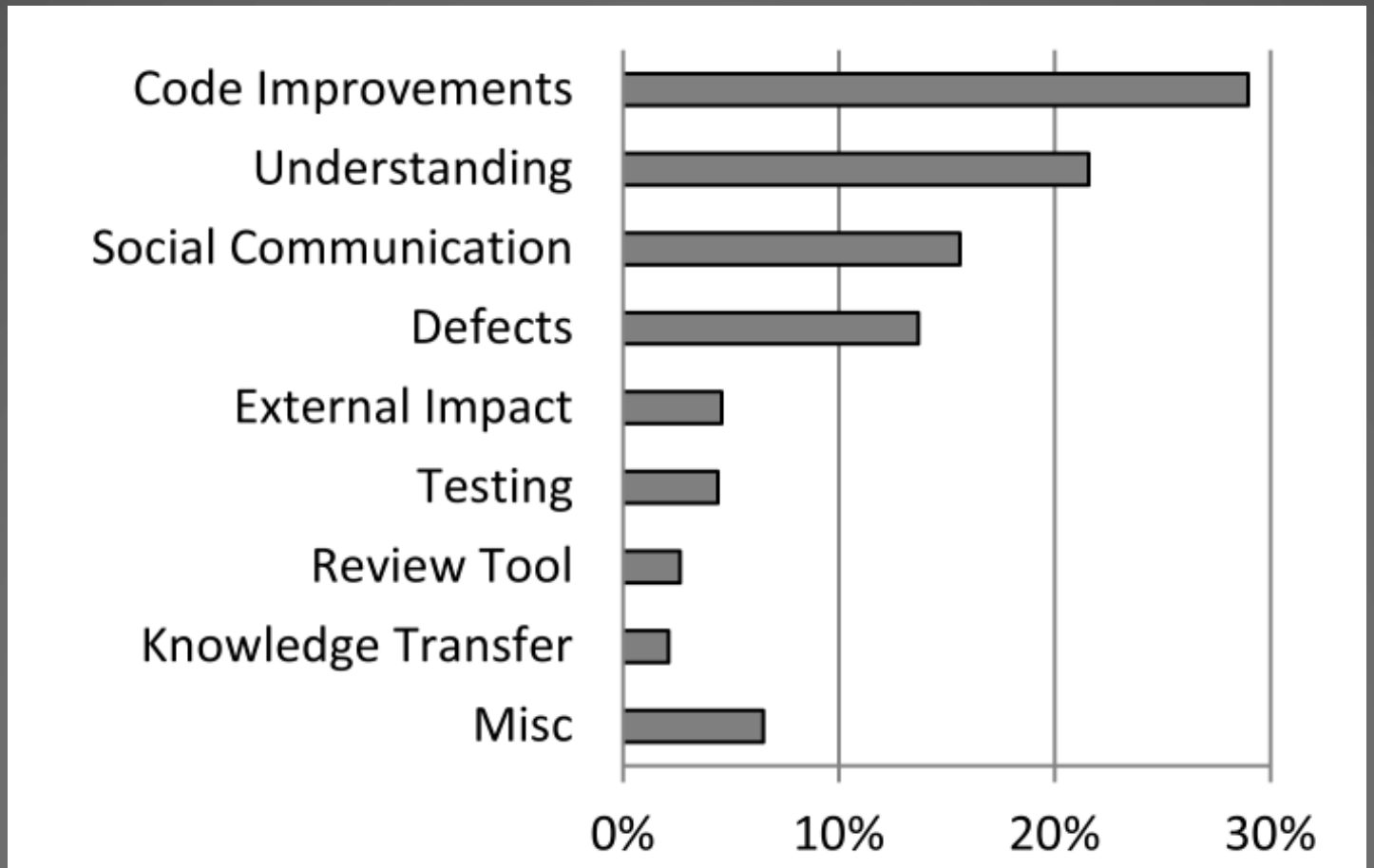


Figure 4. Frequency of comments by card sort category.

EXPECTATIONS, OUTCOMES, AND CHALLENGES OF MODERN CODE REVIEW, ICSE 2013, BACCHELLI AND BIRD

WHY CODE REVIEW DO NOT FIND BUGS. HOW CURRENT CODE REVIEW PRACTICES SLOWS US DOWN.

- *Top 3 categories are long term maintenance issues*
  - *Comments, naming, and styles (22%)*
  - *Organization of code (16%)*
  - *Alternative solutions for long term maintenance (9%)*
- *Only 15% of comments provided by reviewers indicate a possible defect.*
- *Long term maintainability issues are a much larger portion of comments (50%)*
- *Only 33% of comments are deemed useful by the author.*

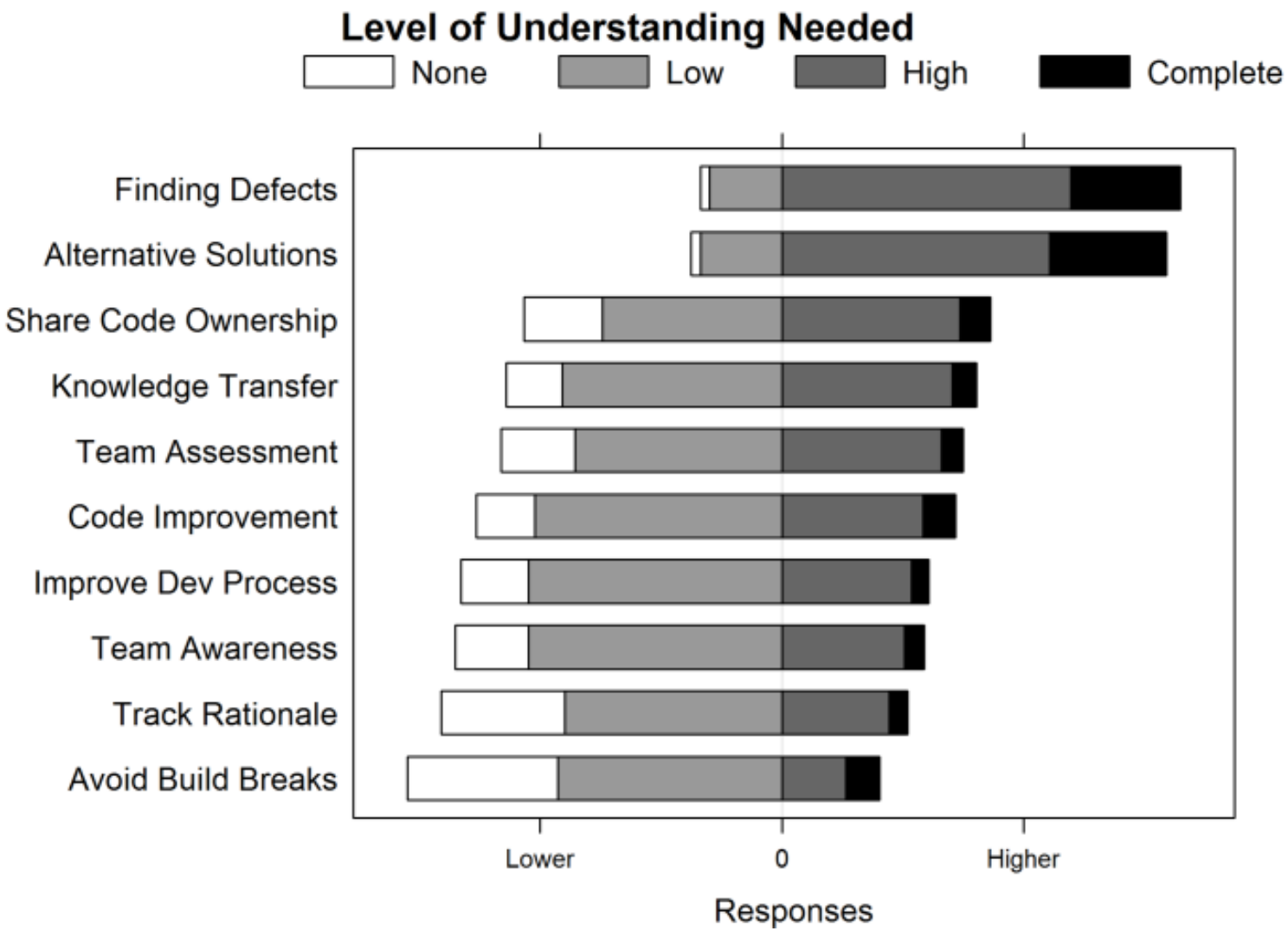


Figure 5. Developers' responses in surveys of the amount of code understanding for code review outcomes.

MODERN CODE REVIEW PRACTICES

- *New reviewers learn fast but need at least 6-12 months to be productive as the rest of the team*
- *People do not actually find bugs – mostly shallow feedback on syntax and style conformance.*
- *We need to have "rigorous criteria" for code reviews to make them effective.*
- Developers need to have "critical eyes" for reviewing code changes, thinking about corner cases.

LOOP INVARIANT EXERCISE

THINK-PAIR-SHARE

- Why do we care to learn "loop invariants" and what are the implications?
  - Software verification tools automate the proof if you write pre/post conditions with invariants.
  - Writing contract is a very good practice!

BEYOND HOARE LOGIC

- For higher-order functions, abstractions, and recursion - does Hoare logic still apply?
- For an OO language, how can you reason about side effects?
- This is an active research topic within software verification and validation.
- E.g. “How side effects are handled in program languages" is called **Shape Analysis**.

BEYOND HOARE LOGIC

- Static shape analysis reasons about the effect of program statements on the heap state.
- Dynamic shape analysis uses tests / runtime profile to provide an accurate representation of the heap state--used to identify memory leaks, relevant to garbage collection.
- Recursions are handled through code inlining and but they use a similar kind of "bounded" analysis to inline code up to a certain depth.

CLASS EXERCISE: POWER

```
public static int power(int x, int n) {
  int p = 1, i = 0;
  while (i < n) {
      p = p * x;
      i = i + 1;
  }
  return p;
}
```

- Suppose that Sheryl's manager wants a proof that after the execution of the loop, the return value p is equal to x^n.
- What is a loop invariant that must hold given a precondition n>=0?

The loop invariant J is {p=x^i and 0<=i<=n}

```
public static int power(int x, int n) {
  int p = 1, i = 0;
  while (i < n) {
      p = p * x;
      i = i + 1;
  }
  return p;
}
```

- Suppose that Sheryl's manager wants a proof that after the execution of the loop, the return value p is equal to x^n.
  - Guess J={ p=x^i AND 0=<i<=n } based on the post condition

(I) P=>J

```
{n>=0}
p = 1, i = 0;
J={ p=x^i AND  0=<i<=n }
{n=>0} implies wp(p:=1; i:=0; p=x^i
AND  0=<i<=n )
==wp (p:=1; p=x^0 AND  0=<0<=n)
=={1=x^0 AND  0=<0<=n}
=={0<=n}
{n>=0} implies {0<=n}? YES
  while (i < n) {
        p = p * x;
        i = i + 1;
  }
```

(2) {J AND B} S{ J}

```
p = 1, i = 0;
  while (i < n) {
        p = p * x;
        i = i + 1;
  }
```

To show {J AND B} S{ J}, {J AND B} =>wp(S,J)

J^B={ p=x^i AND 0=<i<n}

Wp (p = p * x;i = i + 1;p=x^i AND 0=<i<=n)

==Wp (p = p * x;p=x^(i+1) AND 0=<i+1<=n)

=={p*x=x^(i+1) AND 0=<i+1<=n}

=={p=x^i AND 0=<i+1<=n}

(3) J AND NOT B =>Q

```
p = 1, i = 0;
  while (i < n) {
        p = p * x;
        i = i + 1;
  }
```

J^(NOT B)= p=x^i AND 0=<i<=n AND i>=n}

=={p=x^i AND i=n}

=={p=x^n AND i=n} implies {p=x^n}

EXERCISE AT HOME: QUOTIENT

```
public static int quotient(int n, int d) {
  int q = 0, r = n;
  while (r >= d) {
      r = r - d;
      q = q + 1;
  }
  return q;
}
```

This code performs integer division by repeated subtraction. It divides numerator n by divisor d, returning quotient q and also remainder r. What is a loop invariant that satisfies the expected post-condition, `r<d  AND n = q*d +r`?

The loop invariant J is {n=qd+r}

- Guess a loop invariant
- n= qd+r

(I) P=>J

```
int q = 0, r = n;
J={n=qd+r}
  while (r >= d) {
        r = r - d;
        q = q + 1;
  }
```

(2) {J AND B} S{ J}

```
int q = 0, r = n;
  while (r >= d) {
        r = r - d;
        q = q + 1;
  }
```

J={n=qd+r}

(3) J AND NOT B=>Q

```
int q = 0, r = n;
  while (r >= d) {
        r = r - d;
        q = q + 1;
  }
```

J={n=qd+r}

HOW DO WE APPLY WHAT WE LEARNED JUST NOW?

- Add assertions at a right point during debugging.
- You can later remove them if performance is an issue.
- Create test cases that satisfy weakest preconditions and expect to pass.
- Also create test cases that violate weakest preconditions and expect to fail.

RECAP: PATH COVERAGE AND INFEASIBLE PATH

```
public static int how_many_path_is_here(int x) {
        if (x < 4) { return 0;}
        int value = 0;
        int y = 3 * x + 1;
        if (x * x > y) { value = value + 1;
        } else { value = value – 1;}
        for (int i=2; i<=n; i++) {
                for (int j=0; j< i*3 +4 ; j++) {
                        if (a[i]>k) {

    System.out.println("HelloCS130Student");
                        }
                }
        }
        return value;
}
```

- First we have to estimate the number of branch executions, but we also need to watch out for "fixed branched executions" since all branch executions are not independent.

- So this question covers multiple concepts.
- (1) infeasible path
- (2) the number of bounded iterations.
- (3) the number of paths by counting the number of independent branch executions.

HOW MANY TIMES DO YOU EVALUATE THE BRANCH PREDICATE "A[I]>K"?

- You may assume that that an array a exists, n and k are arbitrary inputs, and a[i] never throws an array out of bound exception.

HOW MANY TIMES DO YOU EVALUATE THE BRANCH PREDICATE "A[I]>K"?

- First count the number of path leading to the loop
- Path 1: X<4 (and early return)
- Path 2: X>=4 AND X^2 >3X+1 (Always True)
- Path 3: X>=4 AND X^2<=3X+1 (Never True)
- So there's a single path reaching the loop (Path 2 above.) We now compute how many branch executions exist.

HOW MANY TIMES DO YOU EVALUATE THE BRANCH PREDICATE "A[I]>K"?

- J(i) to be the number of branch executions for i_th's iteration
- J(2)=2*3 +4
- J(3)=3*3 +4
- …
- J(n)=n*3 +4
- T(n) is the total number of branch executions.
- T(n)= J(2)+..J(n)= 3(2+… n) +4n-4 =3(2+n)*(n-1)/2+4n-4

$= 3(2+ 3+ \ldots N) + 4(N-1)$

$= 3(2+N)(N-1)/2 + 4(N-1)$

- Let's call this

  $S1 = 3(2+ 3+ \ldots n)$

  $S1 = 3(n+n-1+..2)$ // just reordering

- Now sum up the above two equations.

$2S1 = 3((2+n)+(2+n)+\ldots (n+2)) = 3(2+n)(n-1)$ // there are n-1 appearances of (2+n)

- So $2S1 = 3(n+2)(n-1)$
- That means $S1 = 3(n+2)(n-1)/2$

FOR THE PROGRAM ABOVE, HOW MANY ***FEASIBLE*** PATHS DO EXIST IN THE BELOW FUNCTION?

- The branch always execute the same way within the nested loop. So basically there are n branches and one early exist path.
- There are 2^(n-1) +1 paths.
- If the condition was actually (new Random().nextInt()%2 ==0), then the branch will execute 3(2+n)*(n-1)/2+4n-4. Therefore 2^(3(2+n)*(n-1)/2+4n-4)+1.

- if the condition was, i>5 (not a[i]>k), the number of "independent" branch executions that could evaluate both true or false will be 0, as the branch execution is deterministic, meaning
-
- When i=2, (i>5) is false,
- When i=3, (i>5) is false,
- ...
- when i=n, n>5, i>5 is true.

THINK-PAIR-SHARE

- When n=11, there are 1025 paths. That means there are potentially 1025 sets of test inputs to exercise every path. *If I am a tester, how should I deal with such state explosion problem? What if I don't have time to create 1025 test cases.*

THINK-PAIR-SHARE

- Systematic enumeration is expensive. How can you overcome the challenges of systematic testing in practice?

SYSTEMATIC VS. RANDOM TESTING

- Spaces are very large e.g. a system with 2 32-bit integers as inputs has 2^64 possible test cases (i.e. approx 1020).
- Relative to this number of potential tests, the number of tests we can apply is always tiny.

SYSTEMATIC VS. RANDOM TESTING

- Random sampling means we can automate and apply a very large number of tests but even then the coverage will remain very small (particularly for complex problems).
- For example, in the case of buffer overrun failure, the likelihood of adding a very long sequence of elements is very small.
- So faults with small profiles and the size of input spaces force a hybrid where we must consider some systematic testing – possibly reinforced with randomised testing.

SYSTEMATIC VS. RANDOM TESTING

- Key take away:
  - Be systematic in exploring search space but randomize to explore variation.
  - Prioritize boundary conditions first

Some lecture notes are based on Stuart Anderson and Conrad Hughes

PRACTICAL TIPS ON MODERN TESTING

WHITE BOX VS. BLACK BOX

- When we consider details of the implementation, it's known as "white box testing"
- When we work from external descriptions, treating the implementation as an opaque artifact with inputs and outputs: it's called "black box testing."

THINK PAIR SHARE

- Which testing method is better, white-box vs. black-box testing?

STRUCTURAL TESTING

- Testing that is based on the structure of the program.
- Usually better for finding defects than for exploring the behavior of the system as a black-box.
- Fundamental idea is that of "basic block" and flow graph – most work is defined in those terms.

TWO KINDS OF STRUCTURAL TESTING

- **Control oriented:** how much of the control aspect of the code has been explored?
  - This is what we studied: Statement coverage, Branch coverage, Path Coverage
- **Data oriented:** how much of the definition/use relationship between data elements has been explored.

RECAP: STRUCTURAL COVERAGE

- **Statement adequacy**
  - All statements have been executed by at least one test.
- **Branch Condition adequacy**
  - Let T be a test suite for program P. T covers all the basic conditions of P iff each basic condition of P evaluates to true under some test in T and evaluates to false under some test in T.
- **Path adequacy**
  - Let T be a test suite for program P. T satisfies the path adequacy criterion for P iff for each path p of P there exists at least one testcase in T that causes the execution of p.

MODEL BASED TESTING

THINK-PAIR-SHARE

- We learned white-box structural testing criteria (e.g., statement, branch, path coverage). How can you test programs as a black box?

- Enumerate test search space based on **specifications** and **abstract models.**

INDEPENDENT TESTABLE FEATURE

- **Independently Testable Feature (ITF):** depends on the control and observation that is available in the interface to the system – design for testability will focus on providing the means to test important elements independently.

MOTIVATION FOR MODEL-BASED TESTING

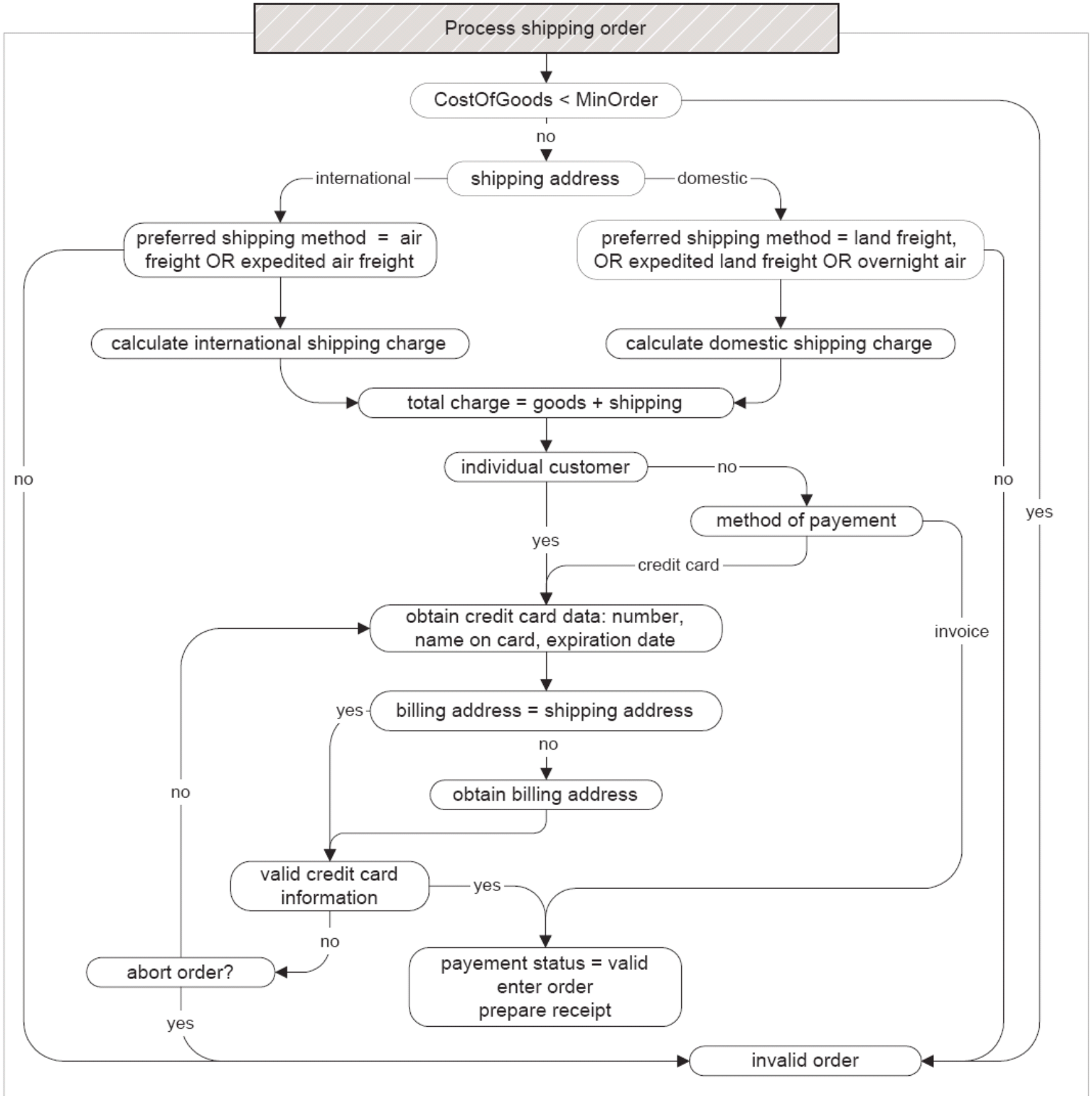
- Generally enumerating all possible combinations is exhaustive but probably infeasible given cost constraints.
- Alternative is to choose some systematic way of reducing the space.

MODEL BASED TESTING

- We have some model of the system and use that to decide how to exercise the system. Typical examples of models include:
  - Decision trees/graphs
  - Workflows
  - Finite State Machines
  - Grammars
- All of these models provide **some kind of abstraction** of the system's behavior.

TYPE 1. UML ACTIVITY DIAGRAMS

- Often specify the human process the system is intended to support.
- Can be used to represent both "normal" and "erroneous" behaviours (and recovery behaviour).
- Abstract away from internal representations.
- Focus on interactions with the system
- Similar to Control Flow Graph


Process shipping order
CostOfGoods < MinOrder
no
yes
shipping address
international
domestic
preferred shipping method = air freight OR expedited air freight
preferred shipping method = land freight, OR expedited land freight OR overnight air
calculate international shipping charge
calculate domestic shipping charge
total charge = goods + shipping
individual customer
no
yes
method of payement
credit card
invoice
obtain credit card data: number, name on card, expiration date
billing address = shipping address
yes
no
obtain billing address
valid credit card information
yes
no
abort order?
yes
payement status = valid
enter order
prepare receipt
invalid order

THINK-PAIR-SHARE

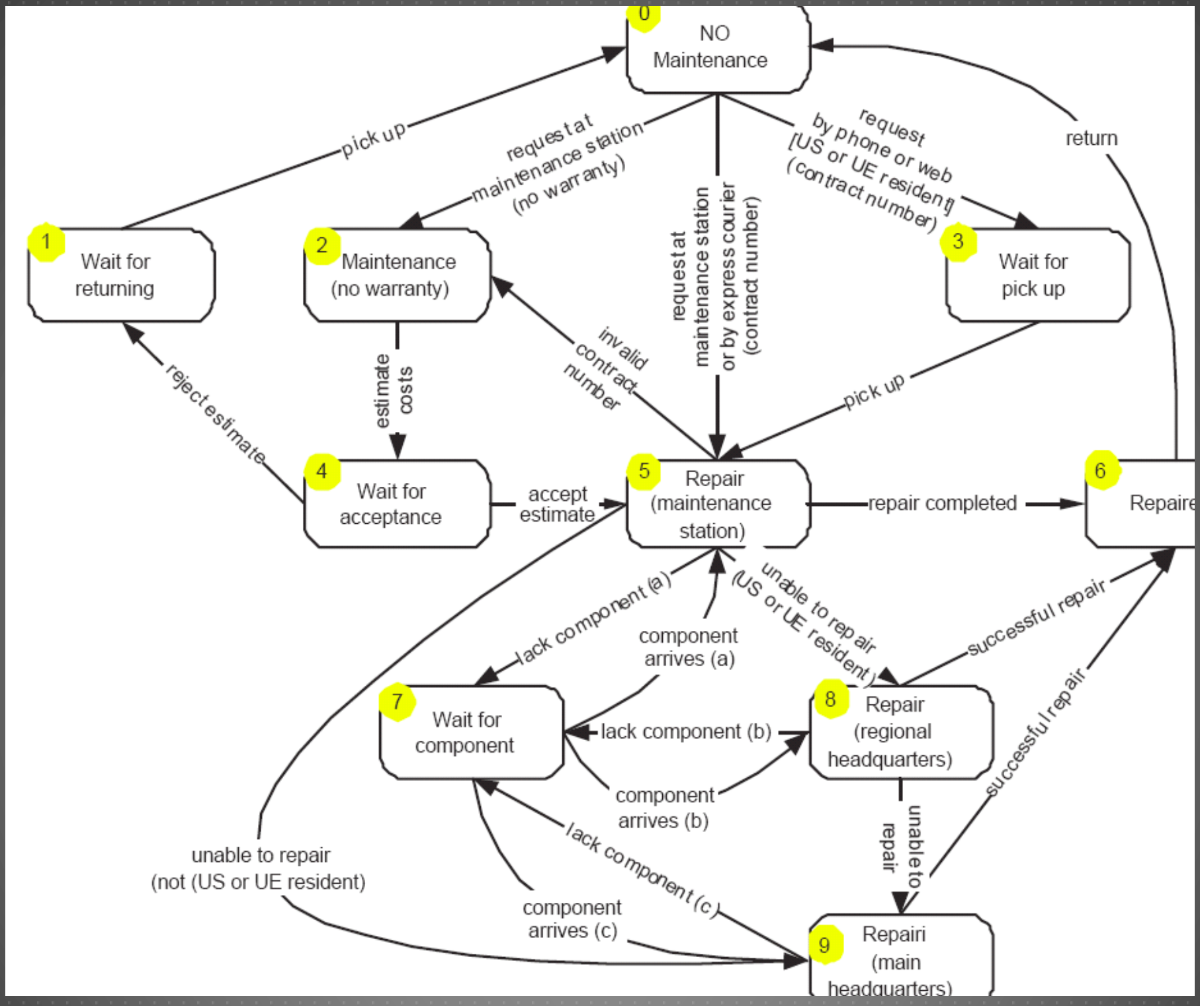
- If your manager gives you specifications / requirements of a program in terms of UML activity diagrams, how would you generate tests?

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- **Node coverage** – ensure that test cases cover all the nodes in the activity diagram.
- **Branch coverage** – ensure we branch in both directions at each decision node.
- **Mutations** – we might also consider introducing mutations where the user does not follow the activity diagram

TYPE 2: FINITE STATE MACHINE

- Good at describing interactions in systems with a small number of modes.
- Good examples are systems like communication protocols or many classes of control systems (e.g. automated braking, flight control systems).


0
NO Maintenance
pick up
request at maintenance station (no warranty)
request at maintenance station or by express courier (contract number)
request by phone or web [US or UE resident] (contract number)
return
1
Wait for returning
2
Maintenance (no warranty)
3
Wait for pick up
reject estimate
estimate costs
invalid contract number
pick up
4
Wait for acceptance
accept estimate
5
Repair (maintenance station)
repair completed
6
Repaire
lack component (a)
component arrives (a)
unable to repair (US or UE resident)
successful repair
7
Wait for component
lack component (b)
component arrives (b)
8
Repair (regional headquarters)
lack component (c)
component arrives (c)
unable to repair
successful repair
unable to repair (not (US or UE resident)
9
Repairi (main headquarters)

THINK-PAIR-SHARE

- If your manager gives you specifications / requirements of a program in terms of UML statecharts, how would you generate tests?

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- Single state path coverage: collection of paths that cover the states:
- Single transition path coverage: collection of paths that cover all transitions.
- Boundary interior loop coverage: criterion on number of times loops are exercised.

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- We can consider **mutation** to discover how the system responds to unexpected inputs.
- We can use **probabilistic automata** to represent distributions of inputs if we want to do randomized testing.

TYPE 3: GRAMMAR BASED TESTING

- Grammars are used to describe well-formed inputs to systems.
- We can use grammars to generate sample inputs.
- We can use coverage criteria on a test set to see that all constructs are covered.

GRAMMAR BASED TESTING

$$
\begin{aligned}
\langle search \rangle &::= \langle search \rangle\ \langle binop \rangle\ \langle term \rangle \mid \boxed{\texttt{not}}\ \langle search \rangle \mid \langle term \rangle \\
\langle binop \rangle &::= \boxed{\texttt{and}} \mid \boxed{\texttt{or}} \\
\langle term \rangle &::= \langle regexp \rangle \mid \boxed{\texttt{(}}\ \langle search \rangle\ \boxed{\texttt{)}} \\
\langle regexp \rangle &::= Char\ \langle regexp \rangle \mid Char \mid \boxed{\texttt{\{}}\ \langle choices \rangle\ \boxed{\texttt{\}}} \mid \boxed{\texttt{*}} \\
\langle choices \rangle &::= \langle regexp \rangle \mid \langle regexp \rangle\ \boxed{\texttt{,}}\ \langle choices \rangle
\end{aligned}
$$

THINK-PAIR-SHARE

- If your manager gives you specifications of search terms in terms of context free grammar, how would you generate tests?

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- Every production at least once
- Boundary conditions on recursive productions – 0, 1, many
- Probabilistic CFGs allow us to prioritize heavily used constructs.
- Probabilistic CFGs can be used to capture and abstract real world data.

MUTATION TESTING

THINK-PAIR-SHARE

- In practice, suppose that you cannot generate tests exhaustively. So you did your best to be systematic and random, carefully thinking about corner cases.
- How would you test your own confidence about the adequacy of tests you have written?
- You still don't know what kinds of faults are likely to happen in practice

MUTATION TESTING

- Mutation testing is a structural testing method, i.e. we use the structure of the code to guide the test process.
- A **mutation** is a small change in a program.
- Such small changes are intended to model low level defects that arise in the process of coding systems.

WHAT IS MUTATION TESTING?

- **Mutation testing** is a structural testing method aimed at assessing/improving the **adequacy** of test suites, and estimating the number of faults present in systems under test.

WHAT IS MUTATION TESTING?

- The process, given program P and test suite T, is as follows:
  - We systematically apply mutations to the program P to obtain a sequence P1, P2,... Pn of mutants of P. Each mutant is derived by applying a single mutation operation to P.
  - We run the test suite T on each of the mutants, T is said to kill mutant Pj if it detects an error.
  - If we kill k out of n mutants the adequacy of T is measured by the quotient k/n. T is mutation adequate if k=n.
- One goal of mutation testing is to assess or improve the **efficacy** of test suites in discovering defects.

KINDS OF MUTATIONS

- **Value Mutations:** these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being one out on the start or finish is a very common error.
- **Decision Mutations**: this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs. E.g. a typical mutation might be replacing a > by a < in a comparison.
- **Statement Mutations:** these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

VALUE MUTATION

- Here we attempt to change values to reflect errors in reasoning about programs.
- Typical examples are:
  - Changing values to one larger or smaller (or similar for real numbers).
  - Swapping values in initializations.
- The commonest approach is to change constants by one in an attempt to generate a one-off error (particularly common in accessing arrays).

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t length 1 and have l < t[0] < u, then the program would fail to count t[0] and return 0 rather than 1 as a result

DECISION MUTATION

- Modeling "one-off" errors by changing < to <= or vice versa (this is common in checking loop bounds).
- Modeling confusion about larger and smaller, so changing > to < or vice versa.
- Getting parenthesis wrong in logical expressions e.g. mistaking precedence between && and ||

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to t[i]>u will cause miscounting

We can model “one-off” errors in the loop bound by changing this condition to i<=t.length - provided array bounds are checked exactly this will provoke an error on every execution.

STATEMENT MUTATION

- Typical examples include:
  - Deleting a line of code
  - Duplicating a line of code
  - Permuting the order of statements.
- Coverage Criterion: We might consider applying this procedure to each statement in the program (or all blocks of code up to and including a given small number of lines).

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k)
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

BLACK BOX TESTING AND MUTATION TESTING

- Black-box test sets are poorer at killing mutants – we'd expect this because black-box tests are driven more by the operational profile than by the need to cover statements.
- We could see mutation testing as a way of forcing more diversity on the development of test sets if we use a black-box approach as our primary test development approach.

THINK-PAIR-SHARE

- Is "mutation testing" an appropriate tool for designing test experiments?

THINK-PAIR-SHARE

- *Is mutation an appropriate tool for testing experiments? ICSE 2005, Andrews J.H et al.*
  - The most influential paper award at ICSE 2015
- They did a systematic study and found that mutation faults can effectively emulate real world faults.
- *Are Mutants a Valid Substitute for Real Faults in Software Testing? FSE 2014, Just et al.*

INDEX CARD FEEDBACK

- "what is the most important thing you have learned in this class session?"
- "what questions do you still have?"

RECAP (1)

- Be systematic in exploring search space but randomize to explore variations.
- Generally enumerating all possible combinations is exhaustive but probably infeasible given cost constraints.
- Model based testing uses abstract models to effectively explore test search space.

RECAP (2)

- Mutations model low level errors in the mechanical production process.
- Mutation testing is a structural testing method aimed at assessing/improving the adequacy of test suites, and estimating the number of faults present in systems under test

QUESTIONS?