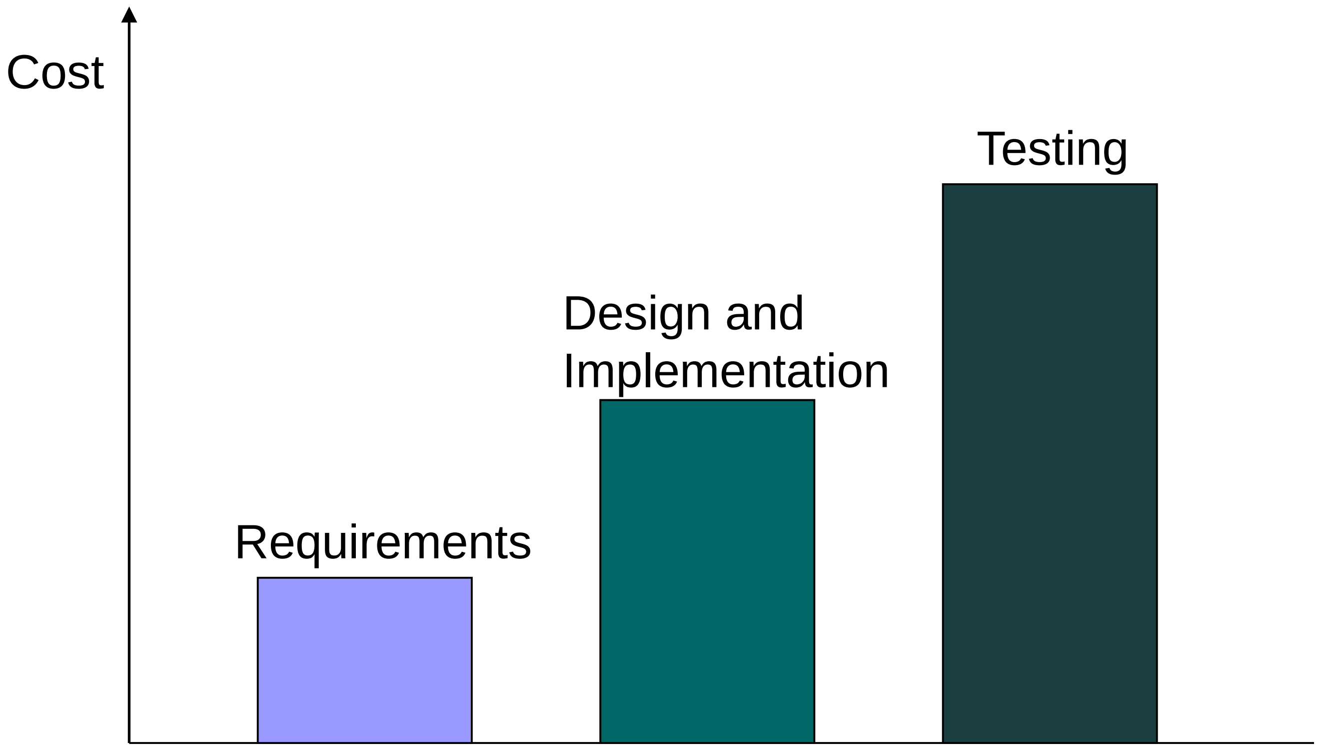
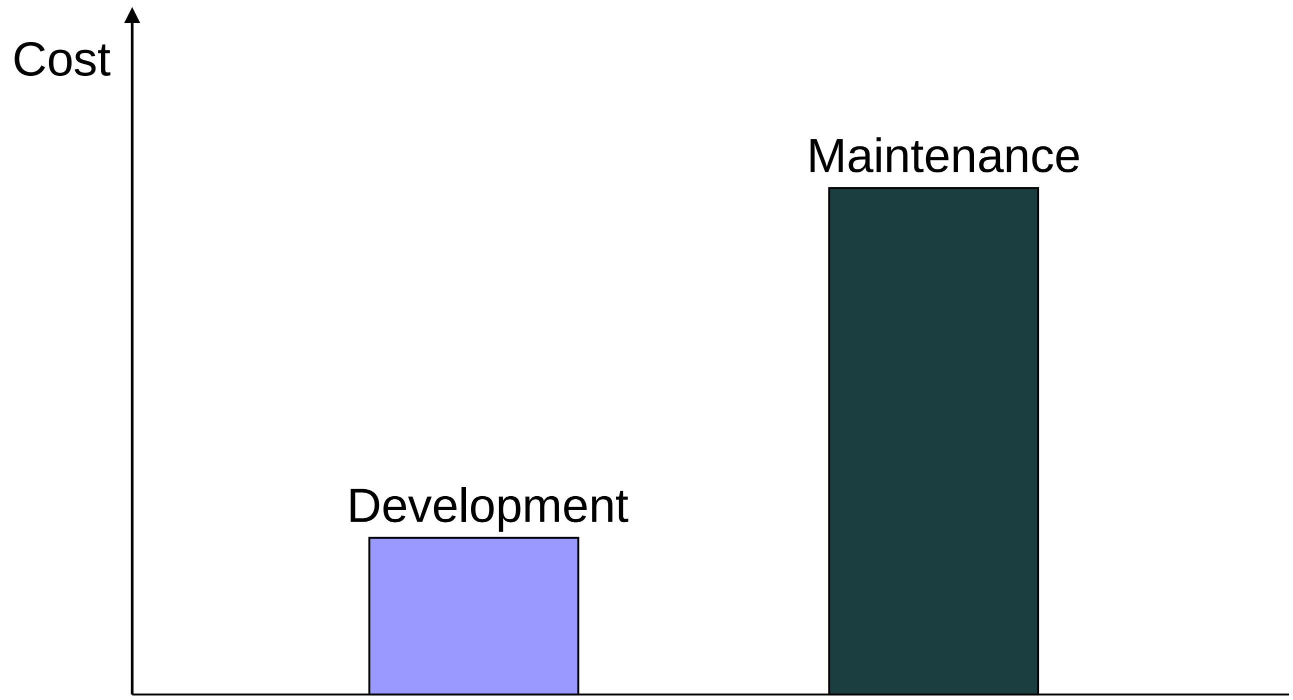
Regression Testing

Software development costs (1)



Software development cost

Software development costs (2)



Development versus Maintenance

Regress, regression fault

Modified component used with unchanged components can cause failure → system is said to **regress**

Baseline version – has passed a test suite

Delta version of the component

Regression fault

Bad fixes

Bad fix injection ranges from 2% to 20% ; on average about 7% (IBM)

Myers:

Experience shows that modifying an existing program is more error-prone process (in terms of errors per statement written) than writing a new program

Failure to test changes

Examples

Ariane-5 rocket (Ariane -4 was its predecessor)
AT & T outage

Disaster scenario:

Failure to test changes – “ *the change is so small that it does not have to be tested* ”

Problem formulation

1. Develop P
2. Test P with test set T
3. Release P

1. Modify P into P'
2. Test P' for new functionalities
3. Perform regression testing on P' the ensure that the code carried over from P behaves:

-reusing tests derived for P
-identifying T'

4. Release P'

Regression testing- testing (1)

Availability of test plan

-testing starts with a specification and a test plan. All is new
-regression testing starts with a possibly modified specification, a modified program, and an old test plan

Scope of test

-testing oriented on the overall system
-regression testing aims to check (modified) parts of the software

Time allocation

-testing time is budgeted before the development (development costs)
-regression testing time accounted for in the planning of a new release of a product

Regression testing- testing (2)

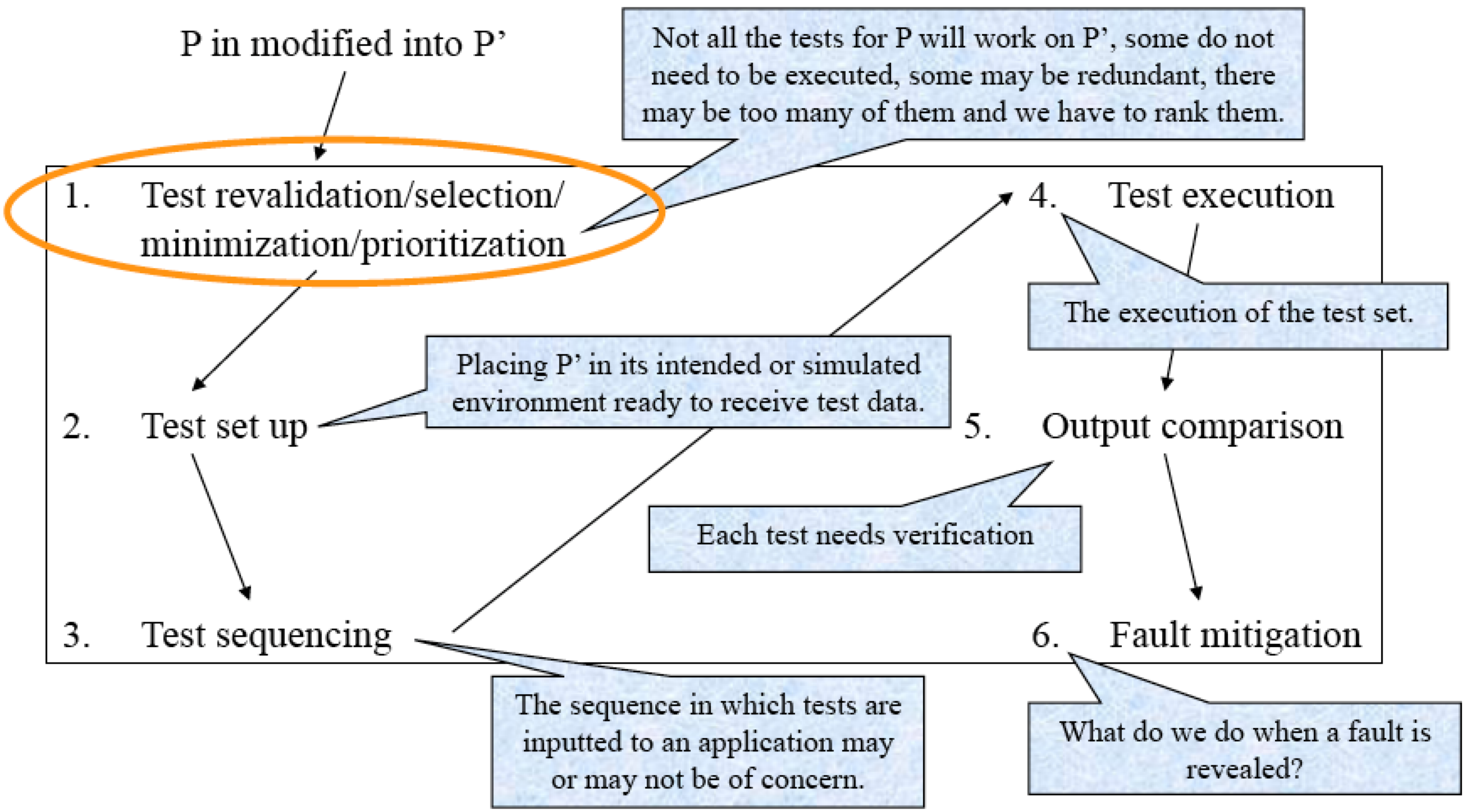
Completion time

-completion time for regression testing should be much shorter that for the testing (parts are to be tested)

Frequency

-testing occurs during system development
-regression testing occurs many times through the life of the system, after every modification made to it

Test revalidation, selection, minimization/prioritization



Test revalidation

A test case may become *obsolete*:

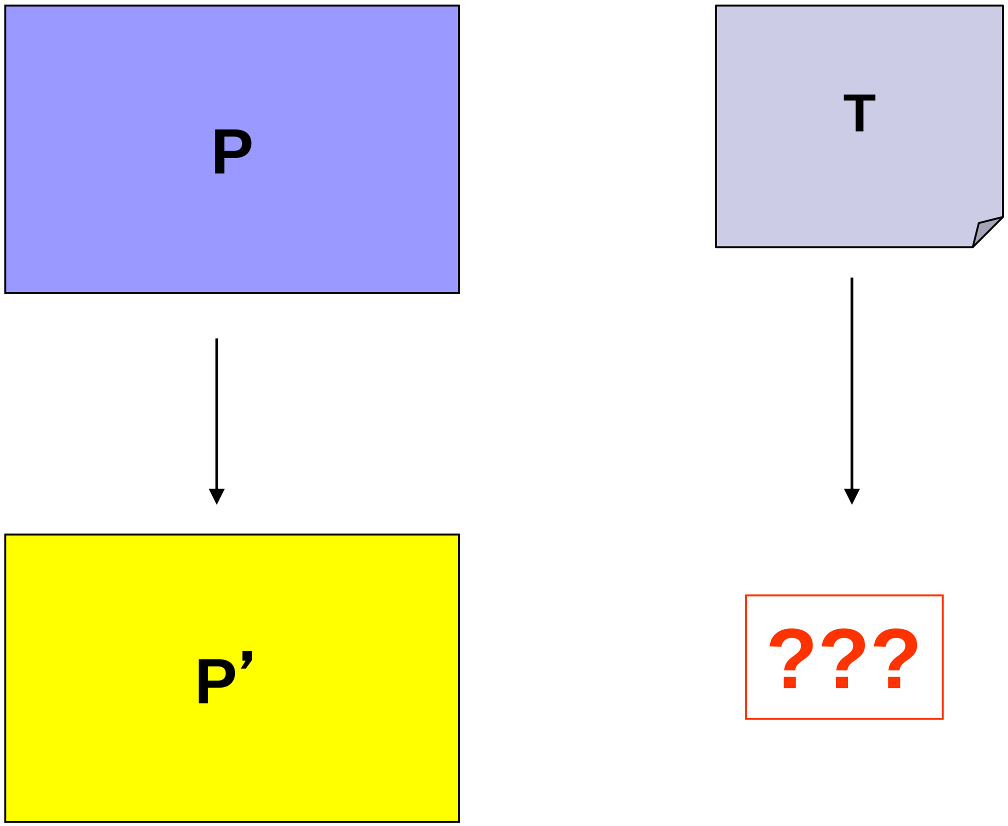
Program modification leads to modified **input-output relationship**

P modified, some test cases may correctly specify the input-output relationship but may not be testing the **same construct**

Modification from P to P' changes equivalence classes.
Test t used to exercise a boundary (which is no longer a boundary).
Test t is obsolete

A structural test t may no longer contribute to **the structural coverage** of the program

Regression testing problem



Re-testing

Re-testing the software after some modifications being made to the previous version

Before a new version is released, some (all) old test cases are run against the new version of the software, new test cases may be required.

Testing is expensive. How to choose test cases?

New version includes changes to the previous version; they could introduce faults

Re - testing

Retest whatever is affected by changes; not the entire software

E.g.,

Given the relationships between software modules

P1 calls P2, P3 is not called by P1 or P2
When changing P1, we need to re-test P1 and P2 but not P3

P1 calls P2 and P2 calls P3
When changing P1, we need to re-test P2, and P3

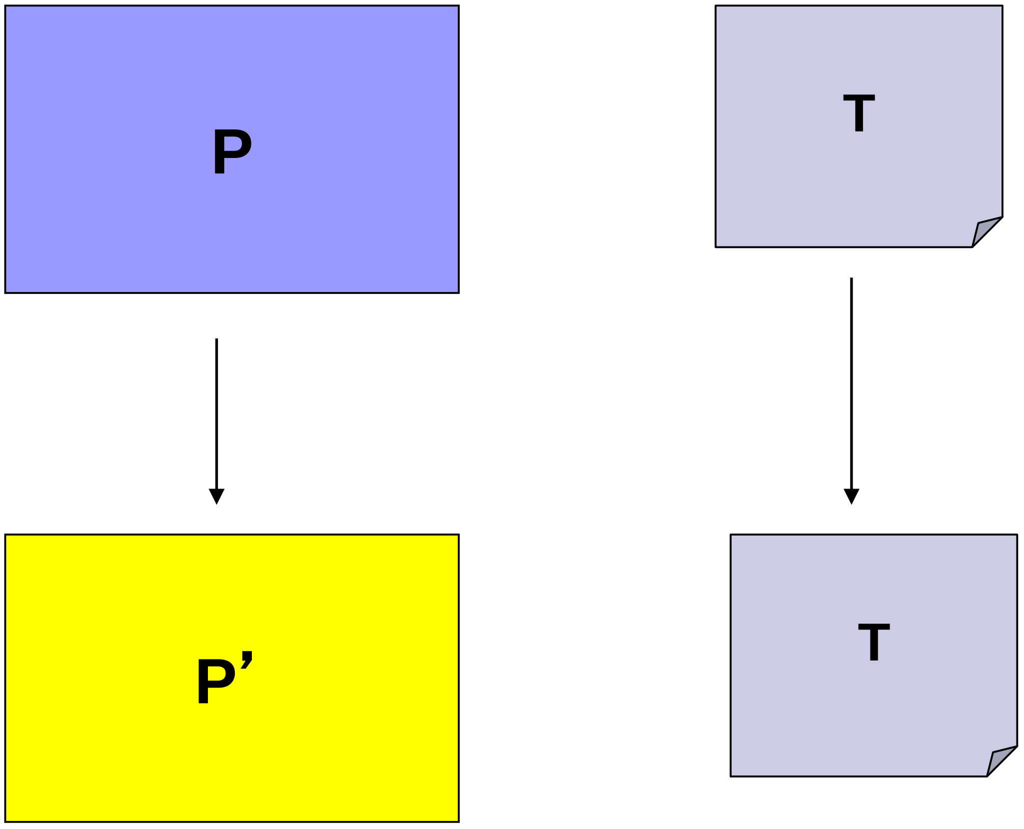
Testing scenarios

Retest-all

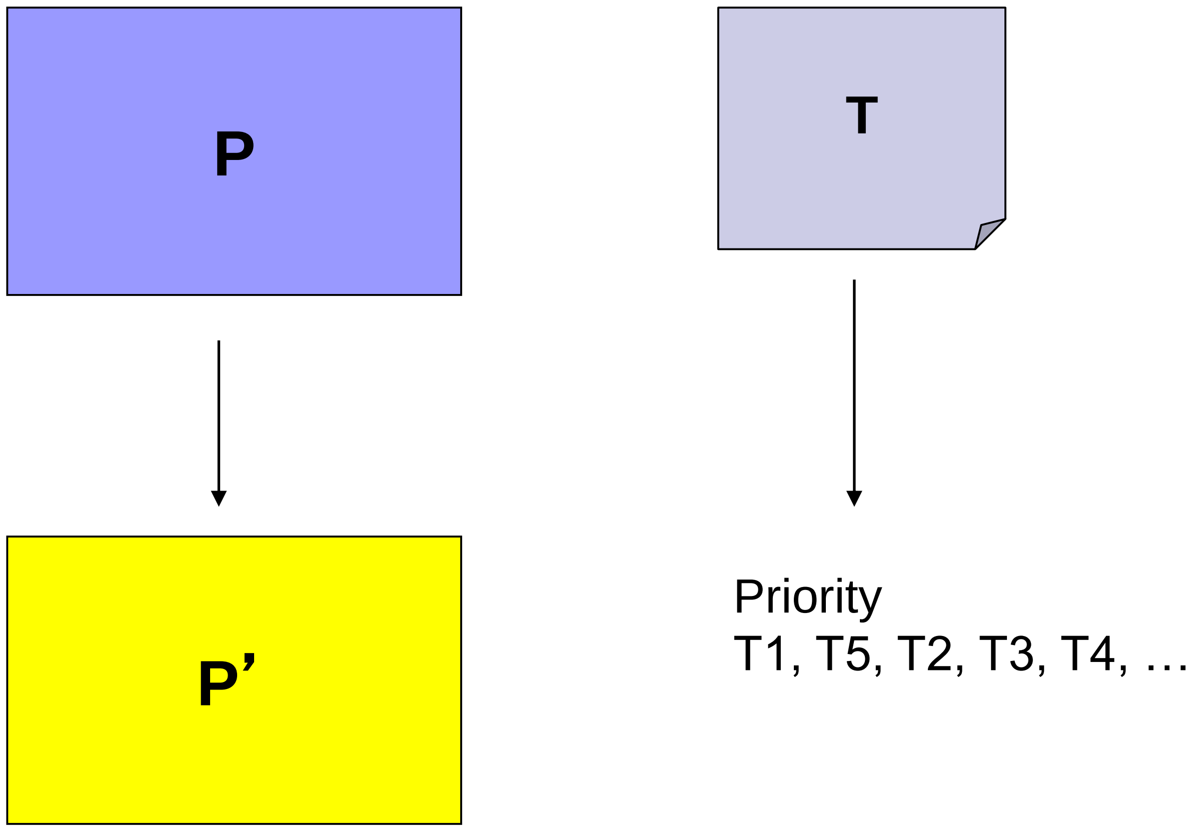
Test case prioritization

Regression testing

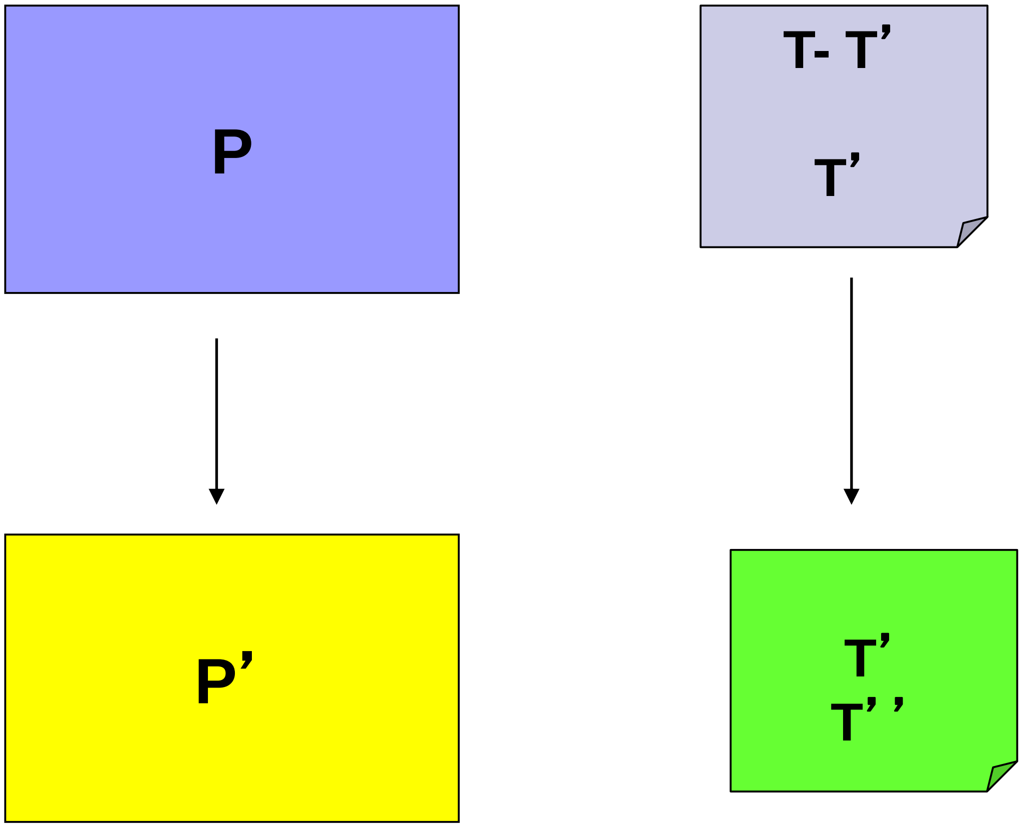
Retest All



Test Case Prioritization



Regression Test



Regression Testing

- Two main steps in regression testing
  - Identify the set of affected components of the modified program
    - Identify the subset of the original test suite that covers the affected components
    - We want to <u>maximize</u> the subset
  - Identify untested components of the modified program that must be covered
    - Generate a new test suite for the uncovered components
    - We want to <u>minimize</u> the new test suite

Regression Testing: main categories

Unit testing

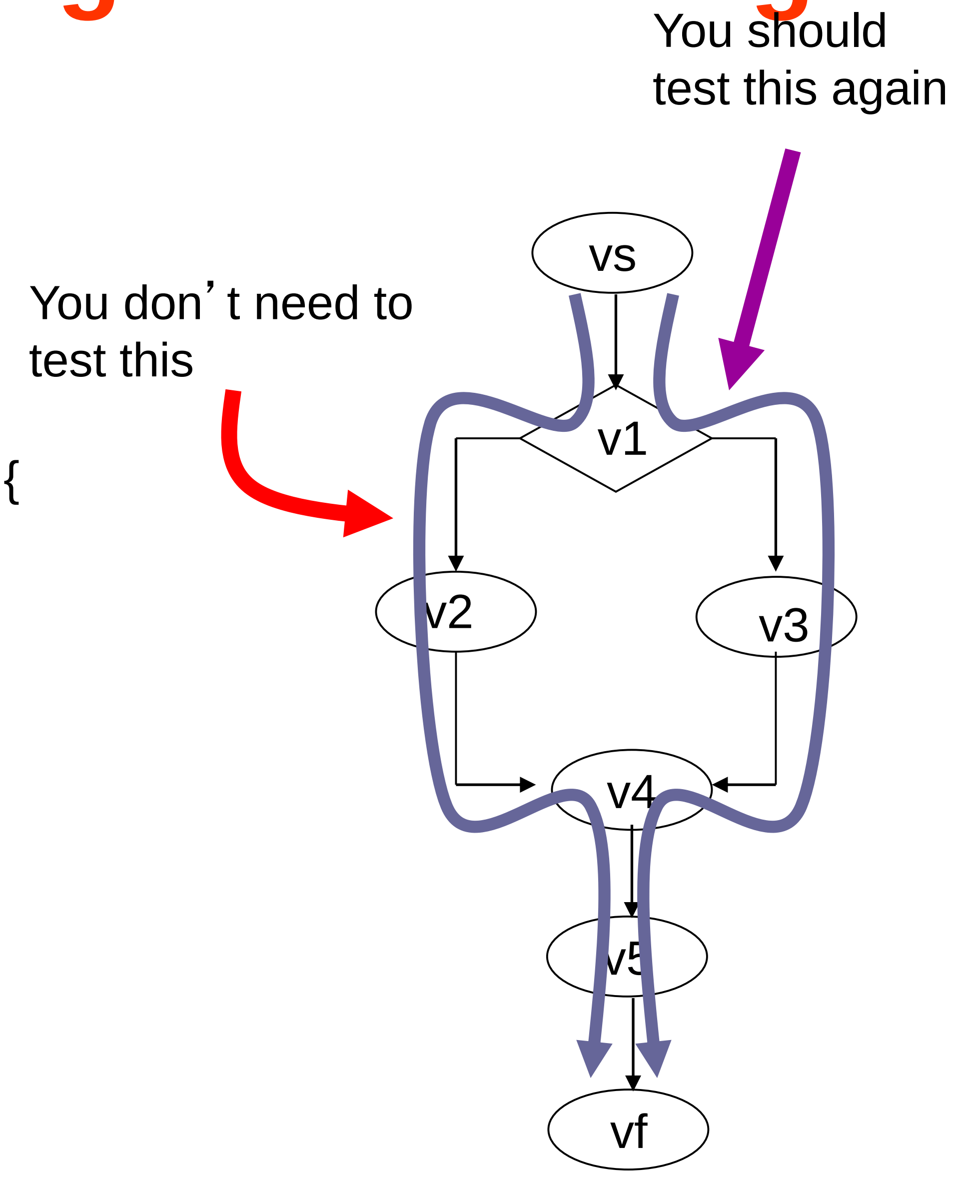
Integration testing (incremental testing)

System testing

Regression Testing – unit testing

```
vs:   int getMax(int x, int y, int z) {
              int max;

v1:           if (x > y)
v2:                  max = x;
              else
v3:                  max = y + 1;
v4:           max = max * z;
v5:           return max;
vf:   }
```



Regression testing: Prioritizing Test Cases

Assisting with regression testing:

Useful if the time required to re-run test suite is long

No test cases discarded

The problem

Given

T- a test suite,
PT – the set of permutations of T
f: PT → R+ //objective (performance) function

Find T' in PT such that for all T'' in PT where T'' is different from T'

$$f(T') > f(T'')$$

Prioritizing Test Cases: Approaches

Total statement coverage prioritization

Additional statement coverage prioritization

Total branch coverage prioritization

Additional branch coverage prioritization

Total fault exposing potential (FEP) prioritization

Total statement coverage prioritization

Prioritize test cases in terms of the total number of statements they cover by counting the number of these statements and sorting the test cases in descending order

Total statement coverage prioritization

```
   procedure P
1. s1
2. while (c1) do
3.   if (c2) then
4.     exit
     else
5.     s3
     endif
6.   s4
   endwhile
7. s5
8. s6
9. s7
```

statement	test case 1	test case 2	test case 3
1	X	X	X
2	X	X	X
3		X	X
4		X	
5			X
6			X
7	X		X
8	X		X
9	X		X

Priority: 3-1-2

Total statement coverage prioritization: complexity

m - test cases, n – number of statements.

Complexity O(mn + mlogm)

could be approximated as O(mn) as m is far smaller than n

Additional statement coverage prioritization

In subsequent testing execute statements that have not been yet covered. Iteratively select a test case that yields the highest statement coverage, then adjusts the coverage information on all remaining test cases to indicate their coverage of statements not yet covered and repeats the process until all statements have been covered.

Complexity

m – test cases   n- statements

$O(m^2n)$

Additional statement coverage prioritization

```
   procedure P
1. s1
2. while (c1) do
3.   if (c2) then
4.      exit
     else
5.      s3
     endif
6.   s4
   endwhile
7. s5
8. s6
9. s7
```

statement	test case 1	test case 2	test case 3
1	X	X	X
2	X	X	X
3		X	X
4		X	
5			X
6			X
7	X		X
8	X		X
9	X		X

Priority: 3-2-1 (originally 3-1-2)

Branch coverage prioritization

Coverage of each possible outcome of a (possibly compound) condition in a predicate

```
   procedure P
1. s1
2. while (c1) do
3.   if (c2) then
4.     exit
     else
5.     s3
     endif
6.   s4
   endwhile
7. s5
8. s6
9. s7
```

BRANCH COVERAGE

	test case 1	test case 2	test case 3
entry	X	X	X
2-true		X	X
2-false	X		X
3-true		X	
3-false			X

test case order: 3-2-1

Additional branch coverage prioritization

Fault exposing - potential prioritization

fault exposing potential (FEP)

Depends on the probability that the fault in the statement will cause a failure

Statement `s` with fault

`s`

Infection probability - probability that a change in `s` causes a change in the program state

propagation probability – probability that the change in state propagates to the output

Fault exposing - potential prioritization

Mutation analysis to estimate combined propagation-infection effect

Fault exposing potential (FEP)

FEP(s_j, t_i) – a ratio of mutants of s_j killed by t_i to the total number of mutants of s_j. If t_i does not execute s_j this ratio is set to zero

test-i:

$$\boxed{\mathrm{FEP}(t_i) = \sum_j \mathrm{FEP}(s_j, t_i)}$$

Complexity (m-test cases, n-statements) **O(mn+ m log(m));** **O(mn) as n>>m**

FEP - example

$$FEP(t_i) = \sum_j FEP(s_j, t_i)$$

```
   procedure P
1. s1
2. while (c1) do
3.    if (c2) then
4.       exit
      else
5.       s3
      endif
6.    s4
   endwhile
7. s5
8. s6
9. s7
```

statement	test case 1	test case 2	test case 3
1	.4	.5	.3
2	.5	.9	.4
3		.01	.4
4		1.0	
5			.4
6			.1
7	.5		.2
8	.6		.3
9	.3		.1

test-1: 2.3 **test-2:** 2.41 **test-3:** 2.2

Retesting with the use of operational profiles

- **Assumes operational profile**
- **Unsafe**
- **Include critical use cases**
- **Include other use cases by frequency in operational profile**

Risky cases

- **Suspicious use cases**
  - **Use cases that depend on components, objects, middleware, resources that are**
    - Unstable, unproven
    - Have not been shown to work together before
    - Implement complex business rules
    - Are complex
    - Were fault-prone during development
- **Critical use cases**

Test selection for regression testing

Inclusiveness
Expresses the extent a technique selects test from T that reveal faults in P
100% inclusive technique is safe

Efficiency
Measures the space and time requirements of the technique

Generality
Measures the ability of a technique to function in a practical and sufficiently wide range of situations

Regression testing in industrial environment

In particular useful for companies with one or more of the following characteristics:

•Development a family of similar products by reusing products or test cases they had developed before

•Development of mission-critical, safety-critical, real-time systems

•Companies maintaining large software systems over extended period of time; regression testing used as a sanity check

•Companies developing software under constant evolution as the market and technology change

•Companies that do not use software inspection as one of their quality assurance techniques

A. Onoma, Regression testing; *Comm of the ACM* [USA, Japan- Hitachi]