Integration Testing

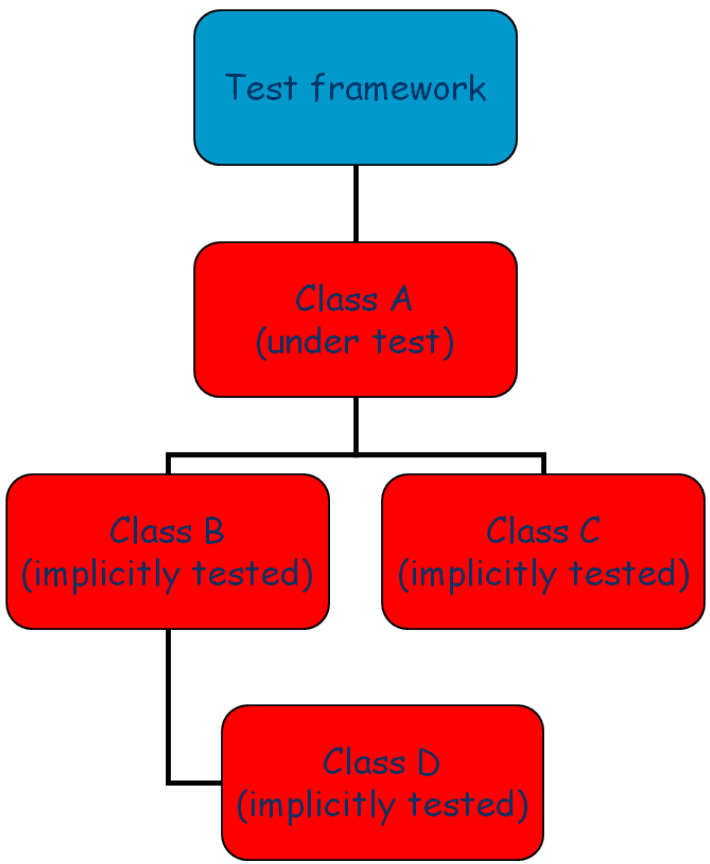
Stuart Anderson

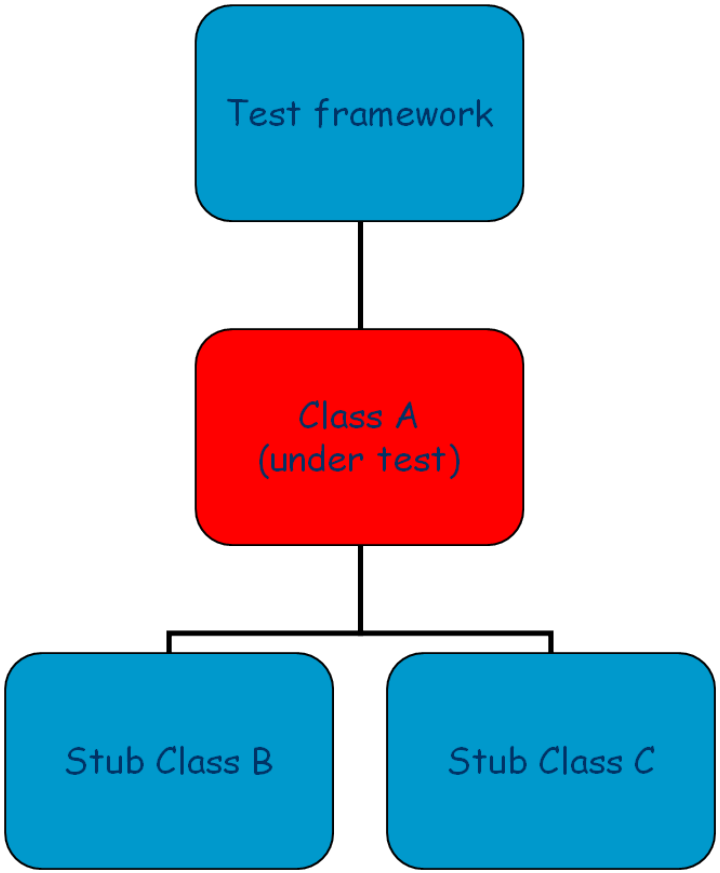
School of informatics

Unit Test vs Integration Testing

- The ideal in unit testing is to isolate a single code unit and test it against its behavioural specification.
- This may involve the construction of extensive 'scaffolding' code that allows the tests to proceed. In particular the tester needs to construct:
  - Driver code to drive the code unit. This is usually contained in the individual JUnit tests.
  - Stub code to take the place of functions or objects that are used by the code unit in providing functionality. Often the stub code is standing in for as yet unwritten code and the stub has limited functionality using lookup to return a value rather than compute it.
- Unit test depends on having some kind of specification for the code units.
- Unit tests often expend effort on testing functionality that is never exercised in the system for which the code module has been constructed.

Unit Test vs Integration Testing





Unit Test vs Integration Testing

- Integration or Incremental testing aims to reduce the need for scaffolding code by using the actual code modules as they are developed to provide the scaffolding for testing.

- Integration or Incremental test provides a means of carrying out unit tests but at the same time it tests the integration of the system across module boundaries.

What are Software Errors?

- What are software errors?

  Discrepancies between computed values and theoretically correct values.

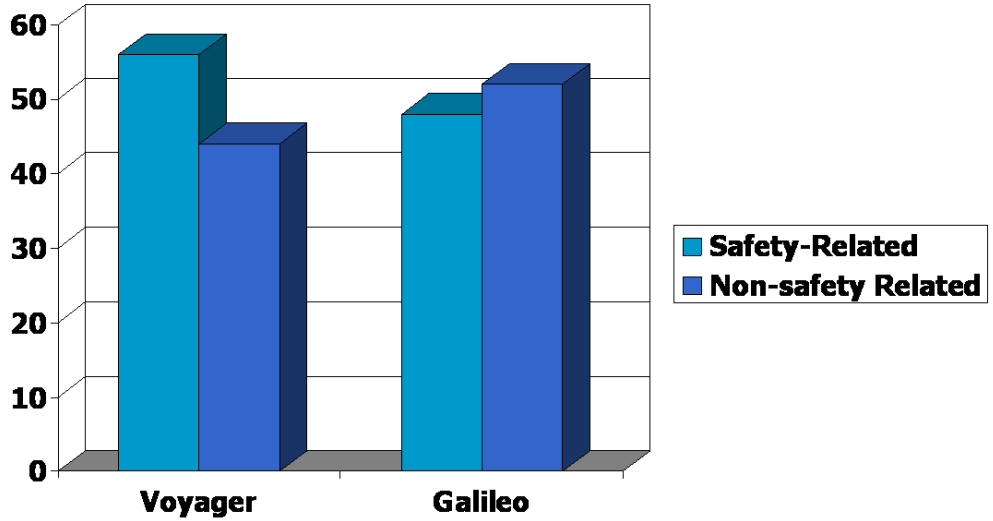
- What are safety-related software errors?

  Software errors that cost human lives

- In an interesting study of critical failures, Lutz concludes that interface failures contribute significantly to the overall fault density in a system.

The following slides are Lutz's on her analysis of errors in the Voyager and Galileo space probes...

Case Study on Voyager and Galileo

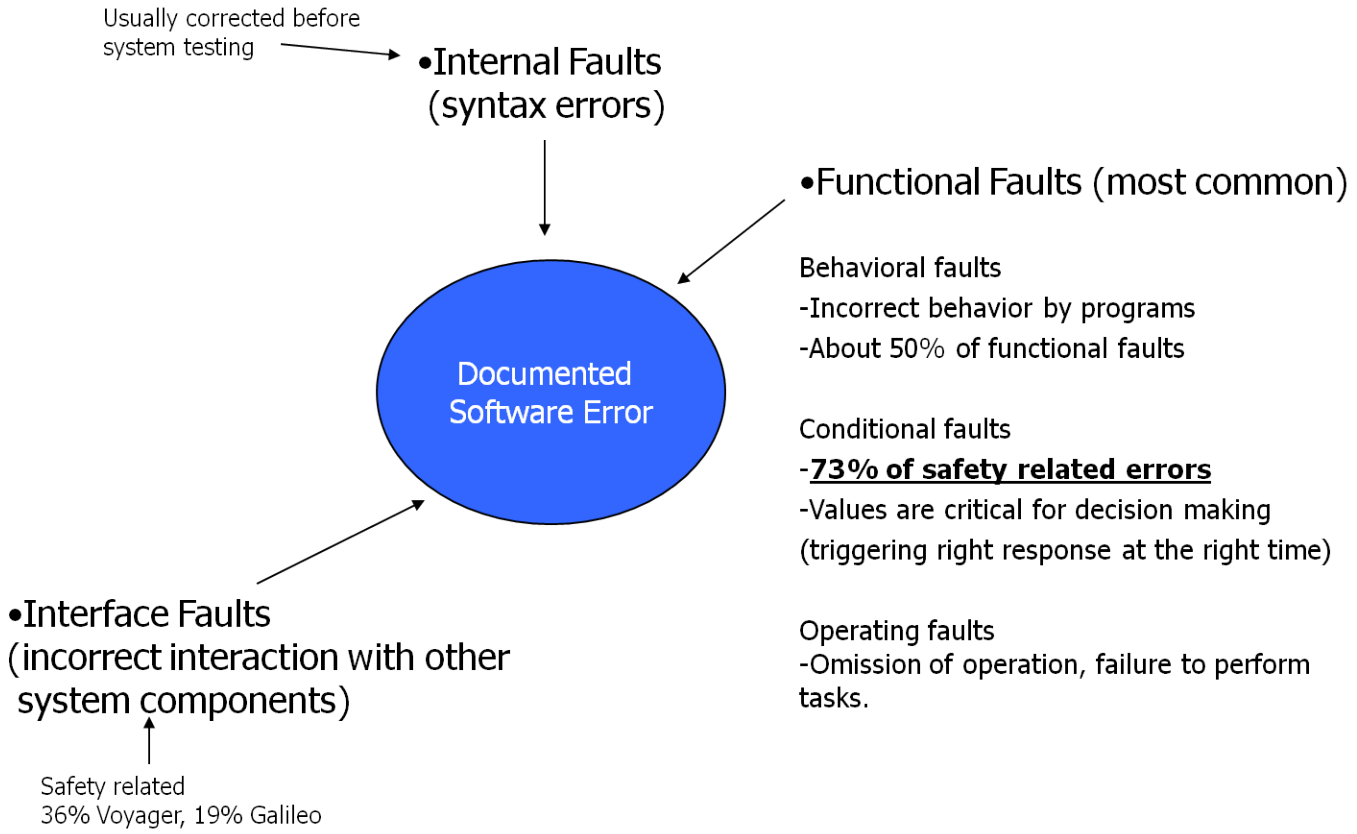


- What is the Goal?
  Making the system secure by removing safety-related errors.
- How?
  Find the source of the problem

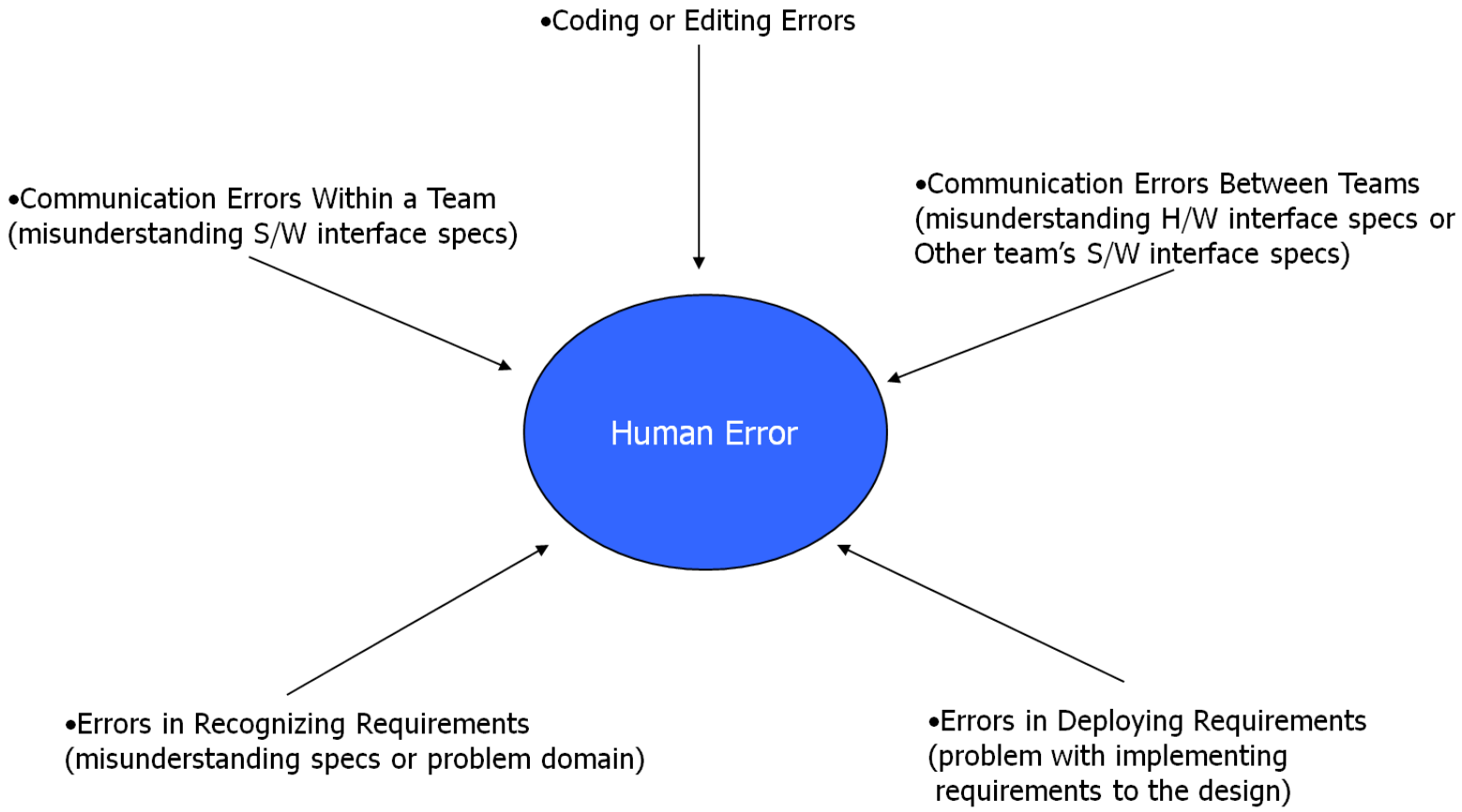
Case Study on Voyager and Galileo

- Lutz's Methodology is to attempt to find root causes
- Nakajo and Kume's classification scheme leads backwards in time from the evident software error to an analysis of its root cause

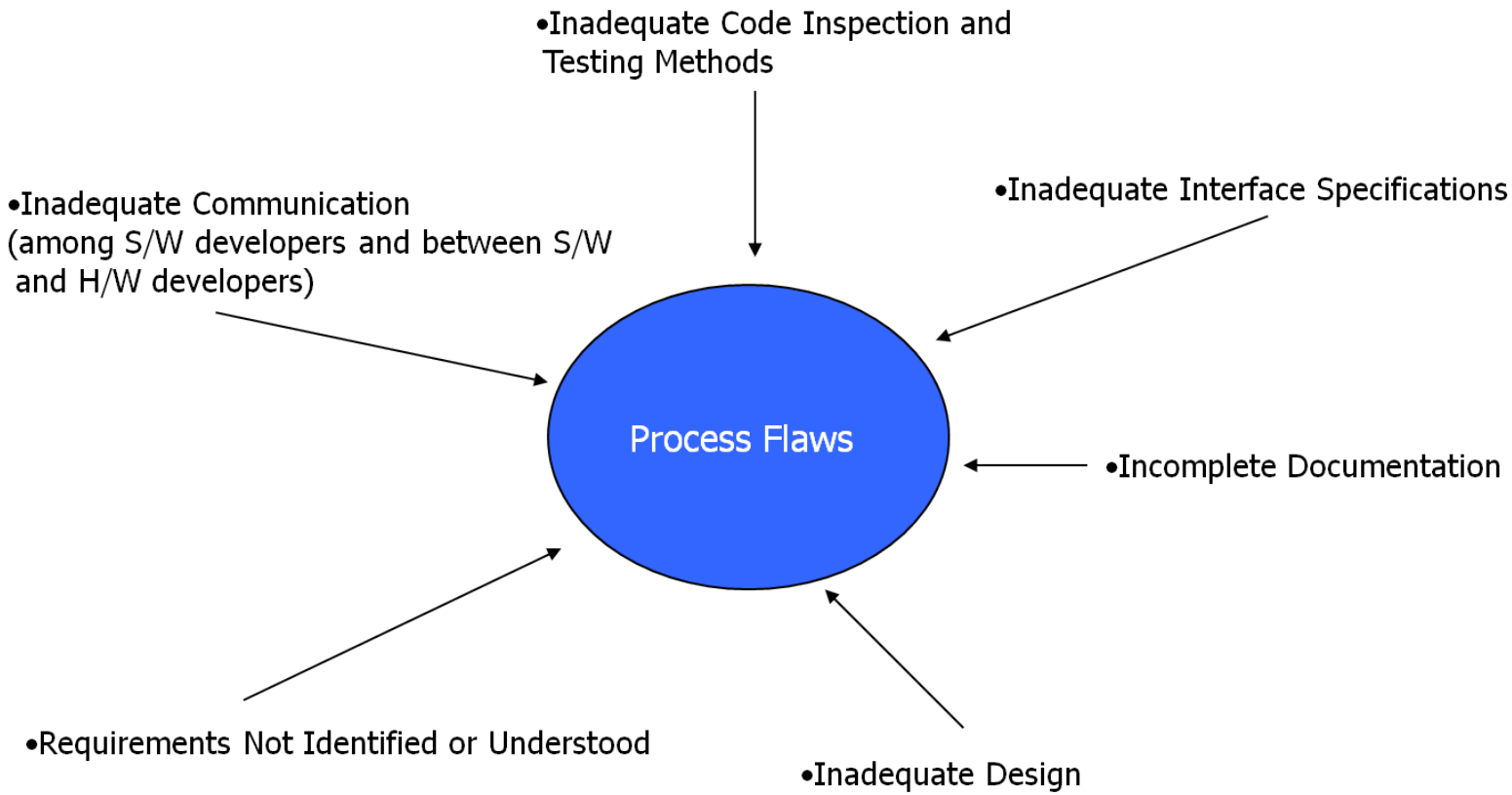
Case Study on Voyager and Galileo



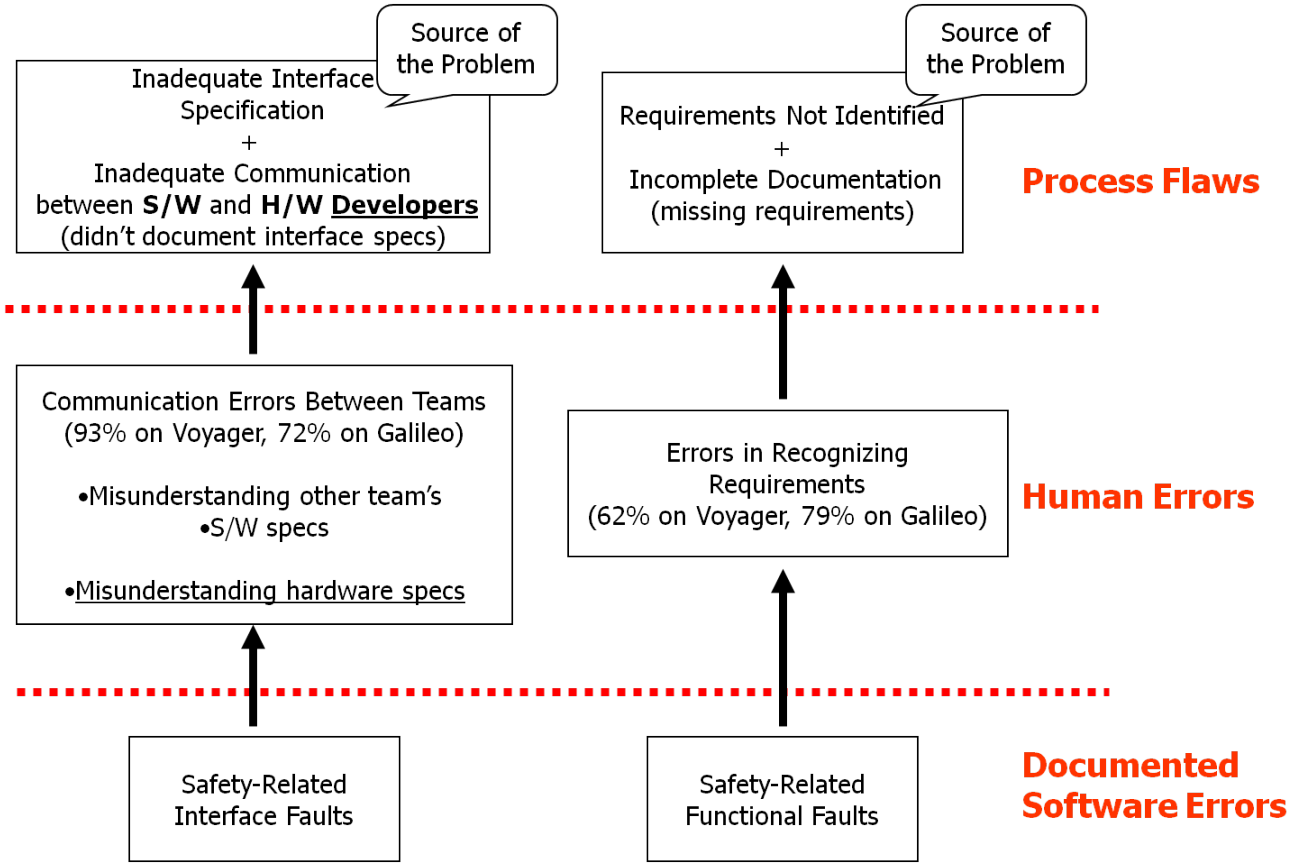
Case Study on Voyager and Galileo



Case Study on Voyager and Galileo



Case Study on Voyager and Galileo



Recommendations for Solving the Problem

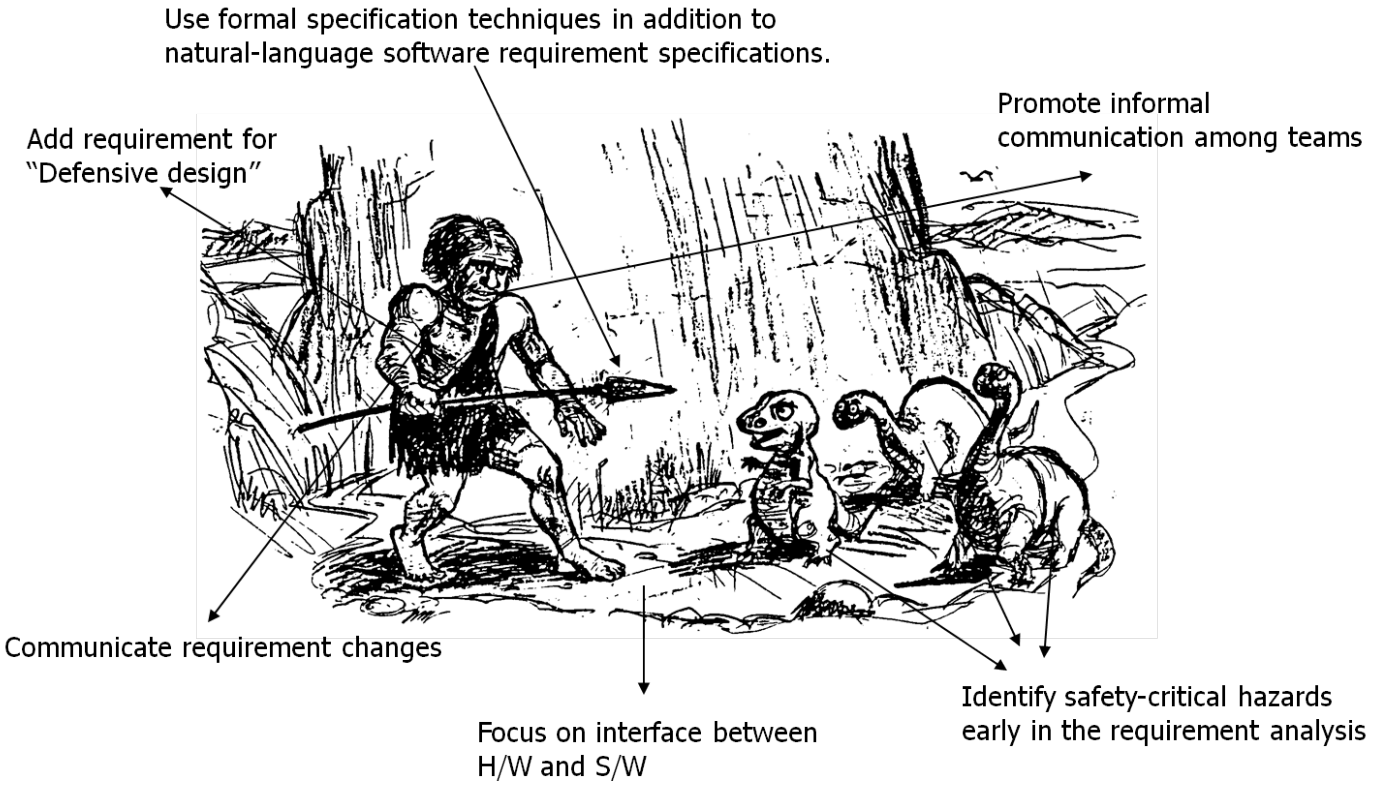
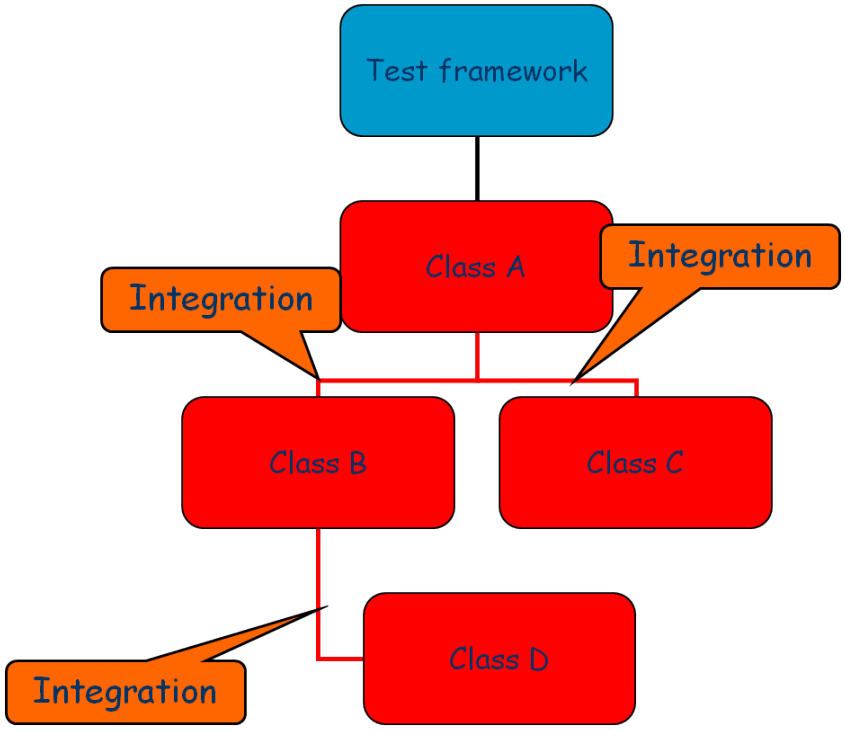


Image by ©USC-CSE University of Southern California **Center for Software Engineering**

How to Approach Integration Testing

- In any system we have a dependency graph between modules of the system.
- Often this is hierarchical (but not necessarily).
- We have two dimensions to consider in constructing and integration test strategy:
  - Whether we approach integration incrementally or whether we adopt a non-incremental strategy.
  - If we adopt an incremental strategy, should we adopt a top-down or bottom-up strategy?



(Non-)Incremental Strategies

1. Non-incremental testing requires the creation of more scaffolding. In particular if we test incrementally bottom-up we require fewer stub programs.
2. Incremental testing reveals errors and misunderstandings across interfaces earlier than non-incremental approaches.
3. Incremental testing should lead to earlier identification of problems and easier debugging.
4. Incremental testing should be more thorough since each increment fully tests some behavioural specification of a sub-component of the system (whereas non-incremental testing tests just the overall functionality).
5. Non-incremental may make more effective use of test effort since it focuses on the system behaviour.
6. Non-incremental test might encourage more concurrency in doing the testing.

Top-down vs Bottom-up Incremental Test

- This choice is dependent on the particular structure of the system under test.
- Architecture is a key element:
  - Layered architectures (e.g. operating system, protocol stack) lend themselves to bottom-up test.
  - Top-down approaches can be useful in testing software that is intended to be generic e.g. components in product lines that provide a service on top of system-specific infrastructure.

Top-down Incremental Test

- Test commences with the top module in the system and tests in layers descending through the dependency graph for the system.

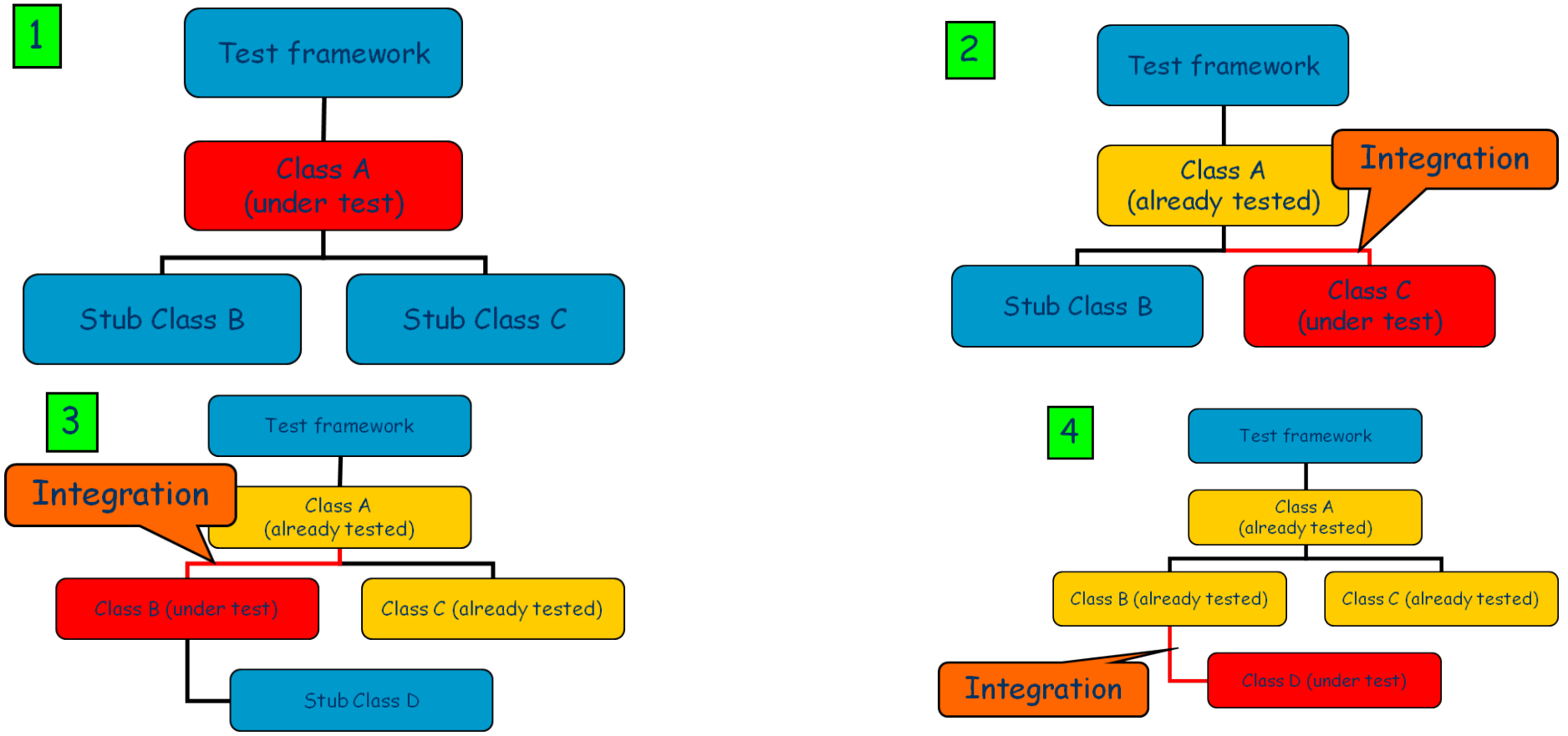
- This may require successive layers of 'stub' modules that replace modules lower in the dependency graph.

Top-down Incremental Test

The complexity of the stub modules is an issue for top-down test:

- Initially stub modules may be quite simple, just indicating that a particular method or procedure call has taken place.
- This may not be adequate in later rounds of testing; one approach is to write modules that always return the same sequence of results for a sequence of calls i.e. those results that we expect for a particular test – but we also need to check the calls have the expected parameters.
- Stubs are eventually replaced by real modules we might want to check the behaviour of the real module is consistent with the behaviour of the stub.
- As stubs become more deeply embedded in a system, determining their behaviour becomes increasingly difficult.
- Deciding the balance between different increments is difficult – e.g. do we want a stub just to check if a module is used in the expected manner?

Top-down Incremental Integration Testing



Top-down Testing

Advantages

- If major defects are more likely at the top level modules top-down is beneficial.
- Getting I/O functions in early can ease test writing.
- Early demonstration of the main functionality can be helpful in highlighting requirements issues and in boosting morale

Disadvantages

- Too much effort on stubs.
- Stub complexity can introduce errors.
- Defining stubs can be difficult if some code is yet to be written.
- It may be impossible accurately to reproduce test conditions.
- Some observations may be impossible to make.
- Encourages the idea that test and development can overlap.
- Encourages deferring full testing of modules (until lower level modules are complete).

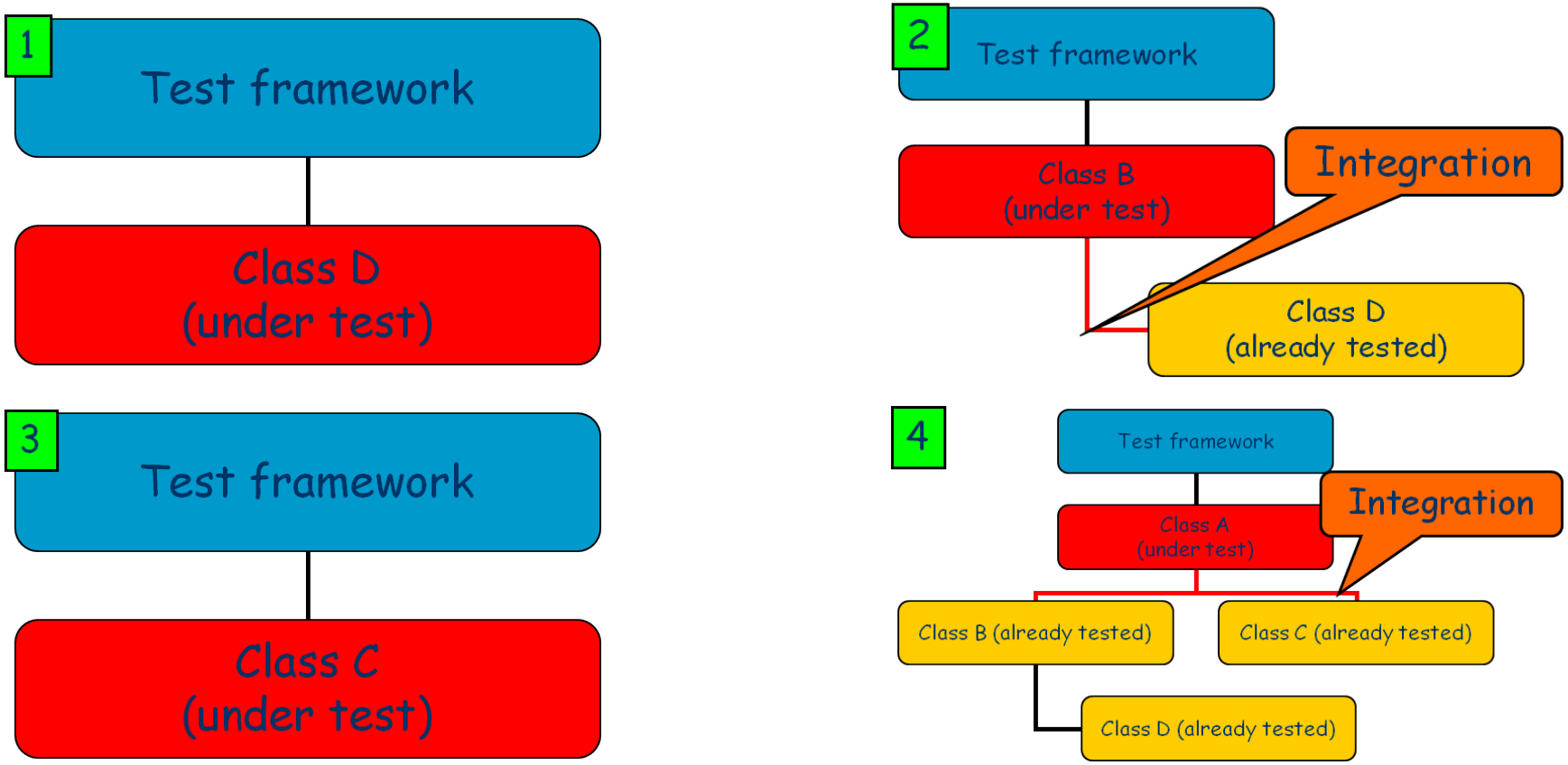
Bottom-up Testing

- Initiate testing with unit tests for the bottom modules in the dependency graph.

- Candidates for inclusion in the next batch of tests depend on the dependency structure – a module can be included if all the modules it depends on have been tested (issue about potential circularity need to consider connected components).

- Prioritisation of modules for inclusion in the test sequence should include their 'criticality' to the correct operation of the system.

Bottom-up Incremental Integration Testing



Bottom-up Testing

Advantages

- Helpful if errors are likely deep down in the dependency structure (e.g. in hardware specific code).
- Test conditions are easier to create.
- Observation of test results is reasonably easy.
- Reduced effort in creating stub modules.

Disadvantages

- Need to create driver modules (but arguably this is easier than creating stub code – and tools like JUnit help).
- The entire system is subjected to the smallest amount of test (because the top modules are included in the tests at the final stage).

Hybrid Strategies

- It is clear that judicious combination of stubs and drivers can be used to integrate in a middle-out approach.
- Also for some groups of modules we may want to take a non-iterative approach and just consider testing them all at once (this means we choose a bigger granularity for our integration steps).
- Using such approaches there are a range of potential criteria for deciding how to group modules:
  - **Criticality:** decide on groups of modules that provide the most critical functionality and choose to integrate those first.
  - **Cost:** look for collections of modules with few dependencies on code lower in the dependency graph and choose to integrate there first. The goal here is to reduce the cost of creating stub code.

Adequacy criteria

- Recall the definitions of *coupling* and *cohesion* from earlier software engineering courses. Both are qualitative measures of the properties of dependencies and module structure in programs. They are used to assess the quality of the modular structure of a system:
  - **Cohesion:** is a measure of how strongly elements in a module relate to one another. Generally we expect to see elements in a module having high cohesion with one another and lower level of relatedness to objects outside the module.
  - **Coupling:** is a measure of relatedness of the objects in a module to other modules. Generally we expect to see low coupling to other modules.
- If we identify elements in a system that contribute to coupling (this is a white-box measure) then we might be able to define coverage criteria for integration tests.

Coupling-based Integration Test Adequacy

- **Call coupling:** component A calls another component B without passing parameters, and A and B do not share any common variable references, or common references to external media.

- **Parameter coupling:** A calls B and passes one or more data items as a parameter.

- **Shared data coupling:** A calls B and they both refer to the same data object (either globally or non-locally).

- **External device coupling:** A calls B and they both access the same external medium (for example, a file or sensor or actuator).

Coupling-based Coverage: Basics

- Coupling-based testing requires that the program execute from definitions of actual parameters through calls to uses of the formal parameters.
- A coupling path is a sequence of statements that, when executed, proceed from a definition of a variable, through a call to a method or a return from a method, to a use of that variable.
- A statement that contains a definition of a variable that can reach a call-site or a return is called a last-def.
- When a value is transmitted into or out of a method (through a parameter, a **return value**, or a **non-local variable reference**), the first time it is used on an execution path after the method is entered or **exited** is called a first-use.
- Note that there can be more than one last-def and first-use of a given variable and call-site.

Coupling-based Coverage Criteria 1

Assume that there is a call from component C1 to component C2, and x is an actual parameter in C1 that maps to a formal parameter y in C2, and the program is tested with a set of test cases T. Then coverage criteria are:

- **Call coupling** requires that the set of paths executed by the test set T covers all call-sites in the system.

- **All-coupling-defs** requires that for each last-def of each actual parameter x in C1, the set of paths executed by the test set T contains at least one coupling path to at least one first-use of y in C2.

- **All-coupling-uses** requires that for each last-def of x in C1, the set of paths executed by the test set T contains at least one coupling path to each first-use of y in C2.

Couplingbased Coverage Basics

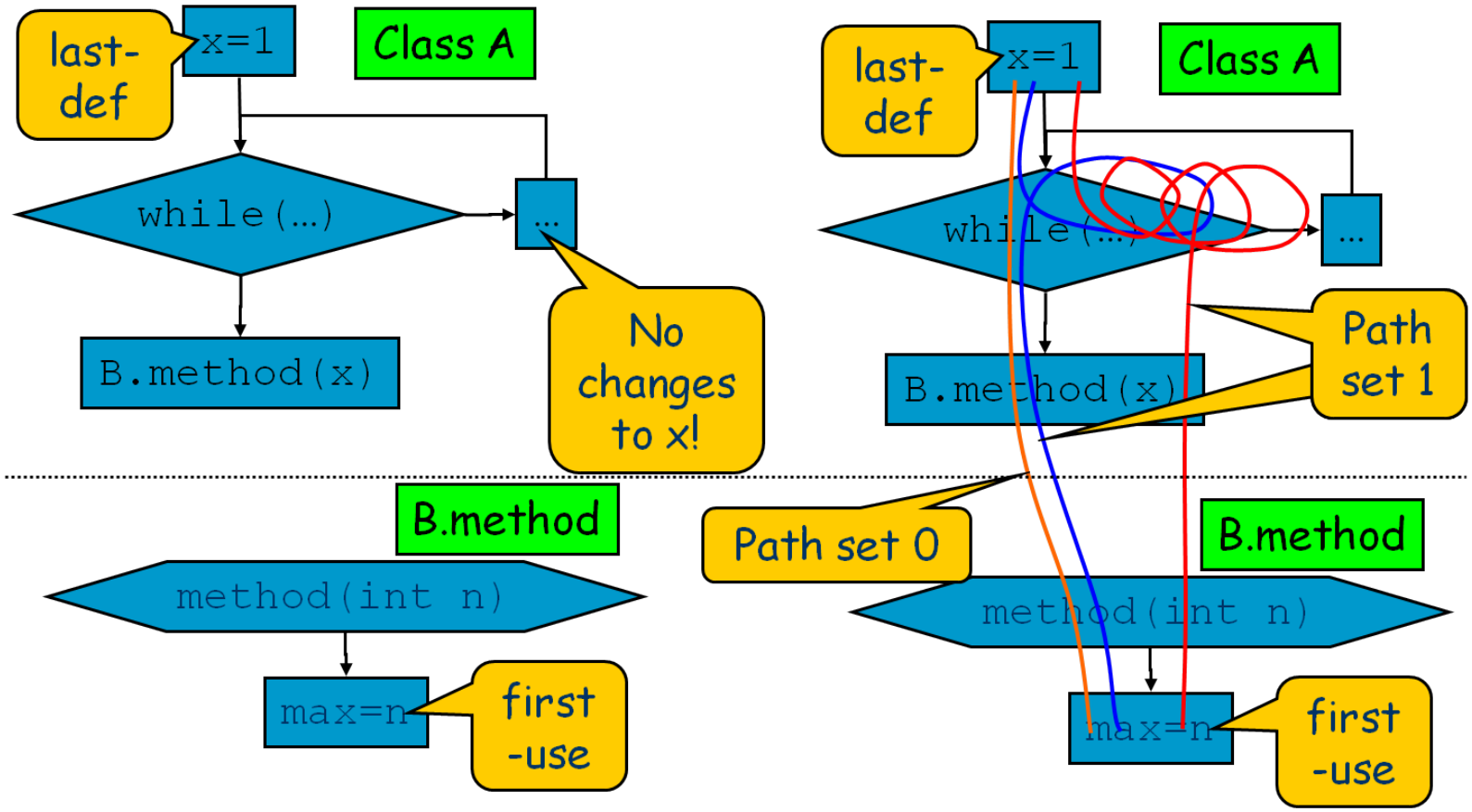
```
class A {
   caller(...) {
      x = 1; // last-def A1
      if(...) // A-if1
         ...// Doesnt change x
      if(...) // A-if2
         x = 2; // last-def A2
      someB.doSomething(x);
...

class B {
   doSomething(int n) {
      if() { // B-if
         y = n; // first-use B1
      } else {
         y = -n; // first-use B2
      }
```

- **Coupling paths** from A to B here are:
  - A1-B1 (A-if1 false, A-if2 false)
  - A1-B1 (A-if1 true, A-if2 false)
  - A1-B2 (A-if1 false , A-if2 false)
  - A1-B2 (A-if1 true , A-if2 false)
  - A2-B1
  - A2-B2
- **All-coupling-defs** would be satisfied by a test which included, e.g. A1-B1 (A-if1 false, A-if2 false) and A2-B1.
- **All-coupling-uses** would be satisfied by a test which included A1-B1 (A-if1+, A-if2-), A1-B2 (A-if1-, A-if2-), A2-B1, and A2-B2.

Coupling-based Coverage Criteria 2

- All-coupling-paths:
  - A **subpath set** to be the set of nodes on some subpath. There is a many-to-one mapping between subpaths and subpath sets; that is, if there is a loop within the subpath, the associated subpath set is the same no matter how many iterations of the loop are taken.
  - A **coupling path set** is the set of nodes on a coupling path.
- For each definition of x, the set of paths executed by T contains all **coupling path sets** from the definition to all reachable uses.
- Note that if there is a loop involved, all-coupling-paths requires two test cases; one for the case when the loop body is not executed at all, and another that executes the loop body some arbitrary number of times (c.f. Boundary Interior Loop criterion).

Couplingbased Coverage: all-coupling-paths



Summary

- We can choose a range of strategies involving non-incremental and incremental approaches. Incremental approaches need to decide on the direction, sequence and granularity of the steps. There is a wide range of choice in integration steps.
- Integration testing has been relatively little studied and there are very few good support tools.
- One approach to providing guidance on the adequacy of integration testing is to use coupling-based coverage measures. This has been the subject of recent research.
- We should view the sequence and scaffolding code as part of the system release.
- We can use data collected on faults detected in the field to modify the integration sequence to provide more adequate testing of error-prone parts of the system.

Required Readings

- **Textbook (Pezzè and Young):** Chapter 21, Integration and Component-based Software Testing
- **Textbook (Pezzè and Young):** (beginning of) Chapter 15, Testing Object-Oriented Software
- Robyn R. Lutz, Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems, In Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press, Jan, 1993. http://dx.doi.org/10.1109/ISRE.1993.324825
- Jeff Offutt, Aynur Abdurazik and Roger T. Alexander (2000). An Analysis Tool for Coupling-based Integration Testing, The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), pp 172–178. http://dx.doi.org/10.1109/ICECCS.2000.873942

Example

CouplingDemo1

```
package st.coupling;

import java.util.Locale;

public class CouplingDemo {

public static void main(String[] args) {
  Locale l = Locale.getDefault();
  if(l.getLanguage().equals("en")) {
    System.out.println("Hi there!  I " +
        "speak English too!");
  }
  String d = Tools.localisedDistance(l,
      200);
  System.out.println("Distance: " + d);
}
}
```

Tools

```
package st.coupling;

import java.util.Locale;

public class Tools {

public static String localisedDistance(
    Locale l, double metres)
{
  String ld;
  if(! l.getLanguage().equals("en"))
    System.out.println("Warning: " +
        "non-English, assuming km");
  if(l.getLanguage().equals("en"))
    ld = (metres / 1600) + "mi";
  else
    ld = (metres / 1000) + "km";
  return ld;
}
}
```

Example

CouplingDemo2

```
package st.coupling;

import java.util.Locale;

public class CouplingDemo {

public static void main(String[] args) {
  Locale l = Locale.GERMAN;
  if(l.getLanguage().equals("en")) {
    System.out.println("Hi there!  I " +
        "speak English too!");
  }
  String d = Tools.localisedDistance(l,
      200);
  System.out.println("Distance: " + d);
}
}
```

Tools

```
package st.coupling;

import java.util.Locale;

public class Tools {

public static String localisedDistance(
    Locale l, double metres)
{
  String ld;
  if(! l.getLanguage().equals("en"))
    System.out.println("Warning: " +
        "non-English, assuming km");
  if(l.getLanguage().equals("en"))
    ld = (metres / 1600) + "mi";
  else
    ld = (metres / 1000) + "km";
  return ld;
}
}
```