Integration Testing

Conrad Hughes
School of Informatics
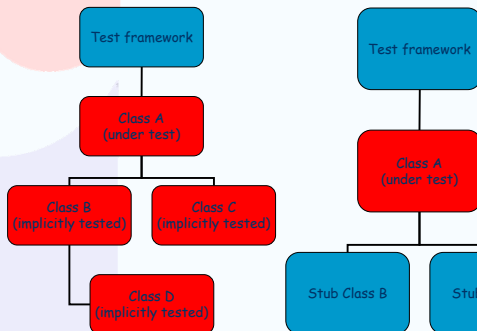
Slides thanks to Stuart Anderson

informatics

17 February 2009   Software Testing: Lecture 10   1

Unit Test vs Integration Testing 1

- 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.

17 February 2009   Software Testing: Lecture 10   2

Unit testing: isolation, stub/mock objects

17 February 2009   Software Testing: Lecture 10   3


Unit Testing vs Integration Testing 2

- 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.
- One interesting study of critical failures is:
  - Robyn R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proceedings of the IEEE International Symposium on Requirements Engineering*. IEEE Computer Society Press, Jan, 1993.
- Lutz concludes that interface failures contribute significantly to the overall fault density in a system.

17 February 2009   Software Testing: Lecture 10   4

Lutz on Software Errors

- What is a software error?
  - Discrepancy between a computed value and theoretically correct value.
- What is a safety-related software error?
  - Software errors that cost human lives
- The following slides are Lutz's on her analysis of errors in the Voyager and Galileo space probes...

17 February 2009   Software Testing: Lecture 10   5

Error Distributions in Embedded Software
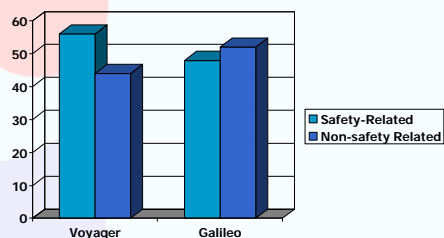
- What's the Goal?
  Making the system secure by removing safety-related errors.
- How?
  Find the source of the problem

17 February 2009   Software Testing: Lecture 10   6

Slide 7

Lutz' 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:

Documented Software Error → Human Error → Process Flaws (development level)

Slide 8

CASE STUDY ON VOYAGER AND GALILEO

Usually corrected before system testing → •Internal Faults (syntax errors)

Documented Software Error

•Functional Faults (most common)

Behavioral faults
-Incorrect behavior by programs
-About 50% of functional faults

Conditional faults
-**73% are safety related errors**
-Values are critical for decision making
(triggering right response at the right time)

Operating faults
-Omission of operation, failure to perform tasks.

•Interface Faults
(incorrect interaction with other system components)

Safety related
36% Voyager, 19% Galileo

Slide 9

CASE STUDY ON VOYAGER AND GALILEO

•Coding or Editing Errors

•Communication Errors Within a Team
(misunderstanding S/W interface specs)

•Communication Errors Between Teams
(misunderstanding H/W interface specs or Other team's S/W interface specs)

Human Error

•Errors in Recognizing Requirements
(misunderstanding specs or problem domain)

•Errors in Deploying Requirements
(problem with implementing requirements to the design)

Slide 10

CASE STUDY ON VOYAGER AND GALILEO

•Inadequate Code Inspection and Testing Methods

•Inadequate Communication
(among S/W developers and between S/W and H/W developers)

•Inadequate Interface Specifications

Process Flaws

•Incomplete Documentation

•Requirements Not Identified or Understood

•Inadequate Design

Slide 11

CASE STUDY ON VOYAGER AND GALILEO

Source of the Problem
Inadequate Interface Specification
+
Inadequate Communication between **S/W** and **H/W Developers**
(didn't document interface specs)

Source of the Problem
Requirements Not Identified
+
Incomplete Documentation
(missing requirements)

Process Flaws

Communication Errors Between Teams
(93% on Voyager, 72% on Galileo)
•Misunderstanding other team's
•S/W specs
•Misunderstanding hardware specs

Errors in Recognizing Requirements
(62% on Voyager, 79% on Galileo)

Human Errors

Safety-Related Interface Faults

Safety-Related Functional Faults

Documented Software Errors

Slide 12

Recommendations for Solving the Problem

Use formal specification techniques in addition to natural-language software requirement specifications.

Add requirement for "Defensive design"

Promote informal communication among teams

Communicate requirement changes

Focus on interface between H/W and S/W

Identify safety-critical hazards early in the requirement analysis

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?

Test framework

Integration

Class A

Integration

Integration

Class B

Class C

Class D

Integration
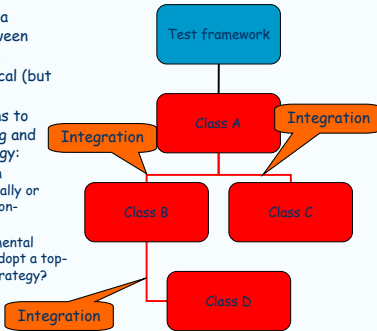
(Non-)Incremental strategies – Comparison (Meyers)

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 versus 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.
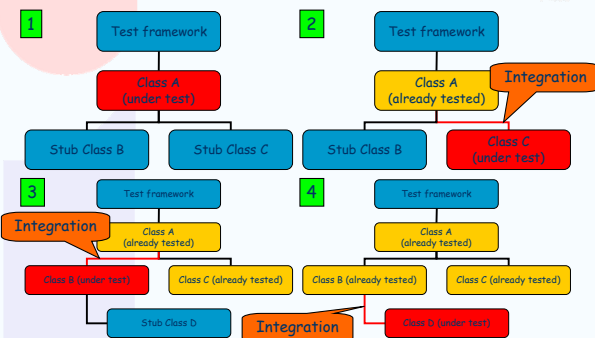

Top-down Incremental Test

- 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.
- 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

1. Test framework — Class A (under test) — Stub Class B — Stub Class C

2. Test framework — Class A (already tested) — Stub Class B — Integration — Class C (under test)

3. Test framework — Class A (already tested) — Integration — Class B (under test) — Class C (already tested) — Stub Class D

4. Test framework — Class A (already tested) — Class B (already tested) — Class C (already tested) — Class D (under test) — Integration

---

Advantages and disadvantages of 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).
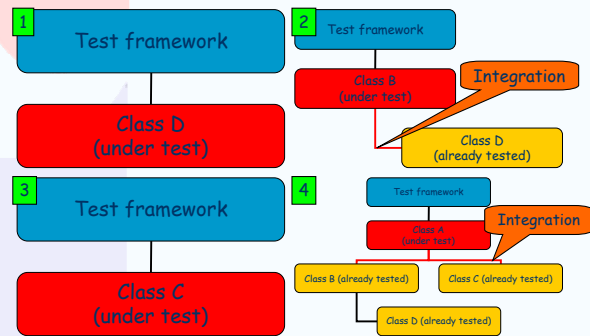
3

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

Advantages and Disadvantages of Bottom-up testing

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

- Disadvantages:
  1. Need to create driver modules (but arguably this is easier than creating stub code – and tools like JUnit help).
  2. 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

- An Analysis Tool for Coupling-based Integration Testing, Jeff Offutt, Aynur Abdurazik and Roger T. Alexander. *The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 172--178, Tokyo Japan, September 2000
- Kinds of Coupling:
  - *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).

4

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.

Coupling–based Coverage Basics: Examples

- **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.

```
class A {
  caller(…) {
    x = 1; // last-def A1
    if(…) // A-if1
      … // Doesn't 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-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).

Coupling–based Coverage: all-coupling-paths
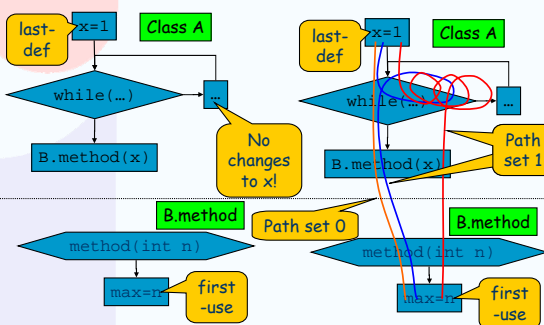
Summary

- 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.

Slide 31 — CouplingDemo1 / Tools

CouplingDemo1   **Tools**

```
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);
}

}
```

```
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;
}

}
```

17 February 2009   Software Testing: Lecture 10   31

Slide 32 — CouplingDemo1 / Tools (annotated)

CouplingDemo1   **Tools**

Annotations: Last-def, Call, First-use, First-use, Last-def, Return, Last-def?, First-use

```
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);
}

}
```

```
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;
}

}
```

17 February 2009   Software Testing: Lecture 10   32

Slide 33 — CouplingDemo2 / Tools

CouplingDemo2   **Tools**

```
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);
}

}
```

```
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;
}

}
```

17 February 2009   Software Testing: Lecture 10   33

Slide 34 — CouplingDemo2 / Tools (annotated)

CouplingDemo2   **Tools**

Annotation: First-use impossible!

```
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);
}

}
```

```
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;
}

}
```

17 February 2009   Software Testing: Lecture 10   34