THE CHALLENGE OF OO TESTING (. . . *THE GURUS ARE SPEAKING*)

- THE FAIRY TALE OF THE EARLY BIRDS:

  "Both testing and maintenance are simplified by an oo approach . . ."

  [Rumbaugh 91]

- OPTIMISM ALL OVER:

  ". . . the use of oo design doesn't change any basic testing principles; what does change is the granularity of the units tested."

  [Booch 94]

  . . .

- THE BIG DISCOVERY:

  " . . . we have uncovered a flaw
  in the general wisdom about oo languages -
  that "proven" (that is well-understood, well-tested, and well-used)
  classes can be reused as superclasses
  without retesting the inherited code."

  [Perry 90]

4.1 What are the special challenges of systematic testing of object-oriented software?

- PESSIMISM FIGHTS BACK:

  " . . . it costs a lot more to test oo software
  than to test ordinary software - perhaps four or five times as much . . .

  Inheritance, dynamic binding, and polymorphism create
  testing problems that might exact a testing cost so high
  that it obviates the advantages."

  [Beizer 94]

SOME DIFFERENCES (I)

- increasing modularization
  - -> decreasing module size
  - -> more inter-module dependencies
    (if methods depend on methods of other classes)

- project is divided into oo (data structure-oriented) work packages
  - -> instead of function-oriented work packages
  - -> functionality may depend on classes developed by co-workers
  - -> increasing dependencies among co-workers
  - -> dependencies require coordination
  - -> coordination requires time = money
  - -> coordination may result into misunderstanding
  - -> misunderstanding results into errors

- functionality - collaboration among objects
  - -> collaboration requires interfaces -> public methods
  - -> interfaces tend to be complex
  - -> interfaces require coordination
  - -> coordination <see above>

- general purpose classes
  - -> reuse beyond the current project
  - -> higher degree of potential applications
  - -> public methods may be used by any method of any other class
  - -> testing of all (currently) relevant states
    requires anticipation of user profile

SOME DIFFERENCES (II)

- program structure does not reflect program functionality
  - -> functionality is realized by a subset of methods
  - -> new instrumentation technique to check functional test coverage
  - -> user profile oriented instrumentation
- object methods communicate by shared object attributes
  - -> the object state produced by a former method (in a sequence) may influence the behaviour of the latter method
  - -> the method behaviour is influenced by
    method parameters AND
    object state
  - -> exhaustive testing =
    all possible state transitions in all possible states
- methods call often other methods of the same class
  - -> procedural coupling among methods

4.1 What are the special challenges of systematic testing of object-oriented software?

- oo software is not only harder to test,
  there is even a richer set of potential errors
  - -> dedicated oo test techniques required

STATE OF THE ART (LATEST NEWS FROM CASE STUDIES)

- oo software exhibits an higher fault rate
- inaccurate classes in inheritance hierarchies
  - -> three times more bound to be erroneous
    than classes without inheritance
- concise code results into higher fault density
- oo analysis and design faults
  - -> greater influence
    than faults in classical analysis and design techniques
- the real fault causes are harder to detect
  - -> difficult debugging
- insufficient oo analysis/design/programming skills
  - -> avoidable faults
- BUT:
  reused classes produce generally less faults
  - -> higher dependability seems to be possible

THE MOST IMPORTANT TROUBLEMAKERS

4.1 What are the special challenges of systematic testing of object-oriented software?

- encapsulation
  - -> restricts visibility of object states
  - -> restrictes observability of intermediate test results
  - -> code adaption for test purposes, e.g. “friendly” methods
  - -> fault discovery more difficult
- inheritance
  - -> the oo goto statement
  - -> invisible dependencies between super/sub-classes
  - -> reduced code redundancy = increased code dependencies
  - -> erroneous functionality is inherited too
  - -> a subclass can’t be tested without its superclasses
  - -> abstract classes can’t be tested at all
- polymorphism & dynamic binding
  - -> static program structure /= dynamic behaviour
  - -> all possible bindings have to be tested
  - -> explosion of potential execution paths
  - -> explosion of potential errors

(CURRENT ?) CONCLUSIONS

- high dependability demands
  - -> avoid oo

    [Sneed 2002]
  - -> “Currently, at the time of developing this standard, it is not clear whether object-oriented languages are to be preferred to other conventional ones.”

    [IEC 61508-7, p. 169]
- to promote oo
  - -> developed skills in sophisticated oo testing techniques
  - -> testing costs may be much higher than developing costs

4.2 Why does procedure testing differ from method testing, and why does module testing differ from class testing?

- lessons learnt
  - -> method test /= procedure test
  - -> class test /= module test

4.1 What are the special challenges of systematic testing of object-oriented software?

- oo testing
  - -> class test - a challenge
  - -> integration test - a challenge
  - -> system test - reuse of conventional test strategies

REFERENCES

- Arbeitskreis
  GI-FG 2.1.7 “Test objektorientierter Programme”

- Binder, Robert V.:
  Testing Object-oriented Systems; Models, Patterns, and Tools;
  Addison-Wesley, 4th edition, 2003

- IEC 61508
  Functional safety of electrical/electronic/programmable electronic safety-related systems, part 7 Overview of techniques and measures, IEC, first edition August 2002

- Sneed, H. M.; Winter, M.:
  Testen objektorientierter Software; Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme;
  Hanser 2002

- Vigenschow, U.:
  Objektorientiertes Testen und Testautomatisierung in der Praxis; Konzepte, Techniken und Verfahren;
  dpunkt.verlag 2005 (2. Auflage 2010), www.oo-testen.de