Development testing of Object-Oriented software

Erkki Pöyhönen, erkki.poyhonen@nokia.com
Nokia, Technology Platforms
Product Creation Services and Tools

Few slides by Roland Müller & Tero Laine, Nokia

Introduction

• Background
  • OO Testing takes mostly place in lower abstraction levels
  • Development testing (Unit, Component, Module, Program testing) has a big effect but it is badly managed activity
  • Overlooked by traditional testing approaches, testing professionals and project management

• Objectives
  • To highlight how different practices support each other
  • Bring awareness for both development and testing roles
Contents: Development testing of OO SW

“If there are two or more ways to do something, and one of those ways can result in a catastrophic results, then someone will do it.”
Edward A. Murphy, Jr.

1. Objects are different - in good and bad
   • What has changed in OO SW organizations

2. It is still software
   • Still no silver bullet

3. Models are helping
   • Being early has its benefits in many ways

4. Focus, focus, focus
   • Strategy for winning the fight

1/4  Objects are different - in good and bad

• Testability (visibility and controllability) in low level will suffer
   • Because of the information hiding
   • Because of the active objects

• Improved support for managed unit testing in class or component levels
   • Test harnesses (drivers and stubs) flexible to implement using available programming tools and skills
   • Low level testing parallel with development means fast feedback

• Models provide an effective way for preventing software faults by testing the abstractions
  = inspecting SW models at early stage
Object benefits

• Object-Oriented methods have a great potential in normal case over older methods
• It takes a lot more than using an object-oriented language to really achieve the potential benefits
  • Agreed analysis and design practices help to achieve SW asset (models/documents & code) transferability across organization
  • Proper OO competence development with agreed modeling methods (UML with style guide) help to achieve better quality models and code
  • Well-managed system and application architecture models is the basis for SW asset reuse and testing
• Remember: large and complex problems are large and complex - no matter which software engineering approach is used
  • Software is now bigger and more complex, but quality has not improved (Les Hatton)

Different concerns, different models

• Different model types are used in different project phases and for different modeling targets (static/structural to dynamic/behavior)
• But: different abstraction levels can share model types
  • Example: class models used in levels from whole system to few classes
One good source

- Tells how to test Object-Oriented Software (surprise 😊)
- What is special about OO SW for testing?
- Collection of tens of OO testing and integration patterns
- Binder speaks about:
  - **Models** that are necessary for test design
  - **Patterns** expressing best practices for test design
  - **Tools** assisting in the implementation of the test design

Deriving Tests from UML Diagrams

- **Use Case**
  - Information from UML use cases is typically not specific enough for testing:
  - Therefore testing should be done with scenarios or extended use cases
- **State Machines**
  - Sequence of transitions
- **Class Diagram**
  - Relationships between objects (1:1, 1:n, life cycle)
- **Sequence Diagram**
  - Sequence of messages and triggered responses
- **Activity Diagram**
  - Action 1 is followed by Action 2
- **OCL = Object Constraint Language**
  - Straightforward usage for invariants, preconditions, postconditions
Model-based testing

- Required Behavior
- Observed Behavior
- Responsibility-based Test model
- Implementation-based Test model
- Component Representation
- Component Implementation
- Meta-Model
- Completeness Checking
- Consistency Checking
- Responsibility-based Testing
- Implementation-based Testing
- Validation

An OO Testing manifesto [BIND00, ch. 4.5]
Technical and process issues

- Reuse does **not** reduce the amount of OO testing
- **Negative impact of OO features** on testing
  - Test case design, testability and coverage analysis all suffer from Inheritance, polymorphism, late binding and encapsulation
- OO development is **iterative and incremental**:
  - Thus, also test planning, design and execution must be iterative
- **Regression testing** must be automated and is an essential part of OO tester toolbox
An OO Testing manifesto [BIND00, ch. 4.5]
Effective testing of OO software

• Unique bug hazards in OO SW means also using established test design practices
• Application and domain specific test tools
  • must be object-oriented
  • and work against the testability problems of OO SW
• Testing process must adapt to iterative and incremental development and small-scale modularity.
• Test design must consider method, class and component levels simultaneously

Specific OO Hazards

• The use of OO languages may reduce some kinds of errors, but increase the chance of others
• Methods often are small
  ⇒ control-flow bugs are less likely
• OO developers are not more immune to errors
  ⇒ Coding errors (misspelling, misnaming, wrong syntax) are as likely as ever
• Dynamic binding and complex inheritance structures
  ⇒ faults due to unanticipated bindings or misinterpretation of correct usage
• Many small components
  ⇒ more interfaces to have programming and usage errors
• Object state control is distributed over the whole program
  ⇒ state control errors are likely
Especially dangerous OO constructs

- Classes that send relatively more messages to instance variable and message parameter objects are more likely to be buggy
- Classes with greater depth (number of superclasses) and higher specialization (number of new and overridden methods defined in a class) are more likely to be buggy

But

- Most literature agrees that biggest threat to programs is not dangerous coding but ambiguous requirements and sloppy design
  - Normally bad programming constructs are a sign of bad design or delayed code restructuring (compare to refactoring rules in XP practice)

00 Testing at different levels 1/2

- Use Cases & Scenarios
  - Abstraction level similar to functional specification
  - Possible to create tests on subsystems and components
  - Useful as functional acceptance tests, complementing module tests
    - On SW team level each iteration could end at use case – based tests of team-developed subsystem
    - Development team could test the subsystem / component, produced and already module tested by a partner
  - Extended Use Case pattern
00 Testing at different levels 2/2

- Class clusters
  - PACT (Parallel Architecture for Class Testing) makes test suites maintainable and modular
- Class
  - State-based (modal) test methods (FREE pattern)
  - N-switch coverage
- Method
  - Classical Black box techniques
  - Code-based coverage measured
  - Category partition pattern

00 Testing above component level

- Integration
  - Component interfaces (structural aspect)
  - Subsystem use-cases (functional aspect)
- Architecture
  - Testing frameworks by developing test applications that exercise the framework on a determined way
  - Many different kind of architecture validation techniques to assess success in achieving different architecture objectives
- OO Middleware
  - Typically CORBA in Nokia
  - Also DCOM & .NET elsewhere
  - Standard interface description languages
  - Support for 3rd party tools
  - Visibility on system-level use-case operation
System Testing and Use Cases

- System tests are based on use cases
- Use cases are normally connected with functional requirements
- Use cases, as defined by UML, are not sufficient for system testing test cases
  - requires more describing information
- You have to also collect information from other sources
- Several attributes are needed for prioritization between use cases:
  - Risk
  - Frequency = density of enquiries (measurable)
  - Performance = answering time to an enquiry (measurable)
  - Load = allowed load generated to system and is the load constant or varying (measurable)
- General technique is to establish three levels:
  - use case (can be in several layers)
  - scenarios (instances of the use case)
  - test cases (based on scenarios with additional details)

Lessons learned from current OO design & testing practice

- Caper Jones analyzed 600 projects from 150 organizations depending on OO methods and found out that
  1. OO learning curve is very steep and causes many first-use errors
  2. OO analysis and design seem to have higher defect potential than older design methods
  3. Defect removal efficiency against **OO design problems** seems lower than against older design methods
     - **OO projects** typically produce programs of increased volume and complexity against more complex requirements
     - **OO model inspection** is even more important than with older design methods
  4. OO programming languages seem to have lower defect potentials than procedural programming languages
  5. Defect removal efficiency against **programming errors** is roughly equal to or somewhat better than removal efficiency against older procedural language errors

   [JONES97]
It is still software

- The software must still be tested, regardless of the programming paradigm used
- Testing just the software and not the models is an expensive way to identify quality problems – “Quality retrofit”
- Development testing provides the most important guidance for lower fault density and also to SW reliability
  - Low fault density is the essential basis for good reliability
  - On very buggy software the advanced reliability methods provide only a marginal effect: user/usage profiling, fault tolerance, parallel programming, ...
  - The promising statistical reliability methods will fail because of too much noise
- Even development testing of OO SW is still first based on black box methods and focused in higher level risk planning
  - Product risks and specific component level risks

Development Testing (DT):
Process vs. Development Approach

Two different aspects overlap when speaking about development testing:

1. Organizational aspect / Formal Process
   - DT as entry criteria for next phase
   - Reporting, approving of planning artifacts
   - Organizational roles, collaboration
   - DT final run as exit criteria is one step, but prepared over time

2. Module Testing as part of developer’s daily work, SW Development Culture
   - How work is conducted in development group or by developer
   - Agile method: MT done all the time, after each implementation step (TDD, XP)
Fundamental Assumptions of Testing

- Testing depends on these assumptions (which are described in more detail in chapter 1 in [MARI95]).

1. Most errors are not very creative. Since errors are clichéd, methodical checklist-based approaches will have a high payoff.
2. Faults of omission, those caused by a failure to anticipate special cases, are the most important and most difficult type.
3. Specification faults, especially omissions, are more dangerous than code faults. They’re also harder to find.
4. At every stage of testing, mistakes are inevitable. Later stages should compensate for them.
5. How thoroughly the tests exercise the code (code coverage) is a good approximate measure of their quality. Because it’s an approximation, coverage must be used with extreme care.

3/4 Models are helping

- Models are not yet software, they are abstractions
  - Abstraction is always a selection of relevance for the solution you are modeling
  - Abstraction means that we can concentrate on the essential
    - No trivia
    - Timely concerns for the risks of each project phase
- This is not different than in other paradigms:

  Only way to prevent SW bugs is to have good customer understanding and design models / documents, and good code!
General questions [MCGR01]

• Syntax
  • Does the model/diagram follow the rules defined by UML?
  • Does the model/diagram follow the program/project rules?

• Traceability
  • Can the content of UML diagram be traced back correctly and completely to its predecessor/to other diagrams?
  • Is everything in the predecessor/in other diagrams reflected correctly and completely to this diagram?

• Domain
  • Is the diagram correct in terms of the “real world”?
  • Is there something missing?

• Check 3C's:
  • complete, correct, consistent

Traceability between different diagrams

• Use case diagram <-> sequence diagram
  • is each use case represented by one or more sequence diagrams
  • does each actor appear on at least in one sequence diagram

• Sequence diagram <-> class diagram
  • is each object in the sequence diagrams presented in class diagrams
  • is every message in the sequence diagram presented as a method in a class diagram
  • if object “a” of class A sends a message "m" to an object "b" of class B in sequence diagram, it means that "m" must be either an operation in class B or if B is an active class "m" must be a signal recognized by the B
Traceability between different diagrams

- Sequence diagram <-> state diagram
  - assume: there is an object "c" of class C in sequence diagram and C has a state diagram
  - "c"s vertical line in sequence diagram describes one path in state diagram
  - every incoming message in sequence diagram must have a corresponding item in state diagram (outgoing message from some state)
  - every outgoing message from sequence diagram's vertical line must have a corresponding item in state diagram (either transition action, entry, exit or do action)
  - note that messages are ordered in sequence diagram and this order remain in one execution path of state diagram

4/4 Focus, focus, focus

- Focus changes along the project path in both development and testing
  - from technical integration
    (getting the product build working)
  - to functional and structural coverage
    (making and testing the full product)
  - to suitability to use
    (non-functional and business aspects)
- Testers can be of help to development if they can help with current development concerns
  - Big part of testing-development communication problems come from unrealistic or out-of-sync testing expectations
  - Example: Early involvement of testing as highlighted by TPI model may be misinterpreted as testing to drive the development focus – not likely to success
Test strategy

- The most important principle (Grove consultants) states the focus of the (development) testing strategy

  Plan your testing (continuously) so that whenever you have to stop, you have done the best possible testing in the time allowed

- I added the word “continuously”, as we seldom can implement projects as planned
  - More about the learning effect than really just changed requirements
  - Use risk to drive your focus to current risks

Use Cases and Operational Profiles

- The use case diagram is a way of representing the intended uses of a system
- Operation:
  - major system task
- Operational profile:
  - complete set of operations with probabilities of occurrence
- These two can be combined to create a perspective for the prioritization of test cases.
Myths about OO testing

- Myth: Testing is unnecessary
  - Reality: Human error is as likely as ever
  - Myth: Testing gets in the way
  - Reality: Testing is a complementary, integral part of the development
  - Myth: Testing is an idea tied to the waterfall life-cycle
  - Reality: Testing can and should be incremental and iterative
  - Myth: Testing is trivial
  - Reality: Hunches about testing completeness are notoriously optimistic
  - Myth: (Automated) GUI testing is sufficient
  - Reality: GUI-based tests may be little more than automated testing-by-poking around
  - Myth: If programmers were more careful, testing would be unnecessary
  - Reality: Many bugs only surface during integration

Myths about OO testing (contd.)

- Myth: Testing is inconsistent with a commitment to quality
  - Reality: Reliable software can not be obtained without testing
  - Myth: Testing is too expensive – we don’t have time
  - Reality: Pay me now, or pay me much later
  - Myth: Testing is the same as it is with conventional software
  - Reality: OO code structure matters
  - Myth: Conventional testing is useless for objects
  - Reality: Conventional testing techniques can be adapted
  - Myth: Inheritance means never having to say you’re sorry
  - Reality: Subclasses create new ways to misuse inherited features
  - Myth: Reuse means never having to say you’re sorry
  - Reality: Every new usage provides ways to misuse a server
  - Myth: Black box testing is sufficient
  - Reality: OO structure matters, again
References