

# Testing Object-Oriented Software: a Survey (part I)

---

J.K. Vis

Friday, September 18th 2009



# Contents

---

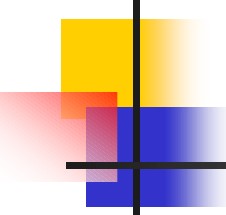
- Introduction
- Abstract Data Type Testing
- Testing Theory
  - Inheritance
  - Polymorphism
  - C++
  - Adequacy and Coverage



# Introduction

---

- Survey of publications up to the end of December 1994
- Testing is necessary to produce highly reliable systems
- Only a small proportion of developers practice adequate testing



# Views on O-O Testing

---

- **Optimistic view:**
  - Technology reduces the need of testing
- **Minimalist view:**
  - Testing remains necessary, but it will be easier and less costly
- **Expansive view:**
  - O-O systems require new approaches



# Common (and personal) View

---

- Different types and proportions of errors due to:
  - Inheritance, Polymorphism and complex inter-class communication
- Encapsulation acts twofold:
  - Reduces global data faults
  - No direct access to an object's state



# Issues in Testing O-O Systems

---

1. **Fault hypothesis:**

Are some facets of O-O systems more likely than others to contain faults?
2. **Test case design:**

What models are useful for reasoning about probable sources of error?
3. **Test automation issues**
4. **Test process issues**



# 1. Fault hypothesis

---

- What kind of faults are likely?
- How can revealing test cases be constructed?
- What effect does the scope of a test have on the change of revealing a fault?
- To what extent are fault models specific to applications, programming languages and development environments?



## 2. Test case design

---

- How much testing is sufficient for methods, classes, class clusters, etc.?
- To what extent should inherited features be retested?
- How should abstract and generic classes be tested?
- What models for understanding collaboration patterns are useful for testing?



## 2. Test case design (cont.)

---

- How can state-dependent behavior be verified?
- How can dynamic binding (polymorphism) be tested?



## 3. Test automation issues

---

- How should test cases be represented?
- How should test cases be applied?
- How can test results be represented and evaluated?
- What is an effective strategy for integration testing?
- What role can built-in test play?



## 4. Test process issues

---

- When should testing begin?
- Who should perform testing?
- How can testing techniques help in preventing errors?
- How can testing facilitate reuse?
- What is an effective test and integration strategy given the iterative approach for O-O development?



# Abstract Data Type Testing

---

- An abstract data type is a software module that encapsulates data and operations
- Encapsulated data is only accessible by operations from the ADT interface
- Used as a method for formal verification



## ADT vs. O-O

---

- Both rely on encapsulation of data (information hiding)
- Both support effective modularity
- Classes are usually constructed by inheritance while ADTs do not provide inheritance
- Also usually no polymorphism in ADTs



# Testing Theory

---

- Two basic questions:
  - What is likely to go wrong?
  - To what extent should testing be performed?
- Testing is based on a *fault hypothesis*  
(an assumption about where faults are likely to be found)



# The role of a fault hypothesis

---

Two general fault hypotheses correspond to two basic testing strategies:

- Conformance-directed testing, which seek to establish conformance to requirements
- Fault-directed testing, which seek to reveal implementation faults



# Conformance-directed testing

---

- Constructing tests that are sufficiently representative of the essential features of a system
- Non-specific fault hypothesis: any fault will do to prevent conformance
- Should exercise all specified features:  
*feature efficient*



# Fault-directed testing

---

- Motivated by observing that conformance can be demonstrated in a system that contains faults
- Specific fault hypothesis: a strong suspicion or evidence to direct the search
- *Fault efficient*: a very high probability of revealing a fault



# O-O Fault Hypotheses

---

- Inheritance
- Polymorphism
- C++



# Inheritance

---

Can contribute to errors:

- Weakens encapsulation
- Fault hazards similar to global data in conventional languages
- Unusually powerful macro substitution for programmer convenience



## Inheritance (cont.)

---

- Forgotten methods

Methods like `copy` and `isEqual` are easily forgotten and overridden

- Incorrect initialization

Variables of superclasses are visible within subclasses, which poses fault hazards similar to global data access in conventional languages



# Inheritance (cont.)

---

- Semantic mismatch
  - Simple inheritance with overriding imply different semantics of the subclass
  - Multiple inheritance adds even more opportunities for misnaming and semantic mismatches
- Test reduction

The optimistic view argues that reduced testing is necessary in derived classes



# Testing axioms

---

- **Antiextensionality axiom:**
  - The same function but different implementations requires different test suites
- **General multiple change axiom:**
  - The same shape (flowgraph) but different data domains requires different test suites



## Testing axioms (cont.)

---

- Antidecomposition axiom:
  - Invocation or used by different callers requires different test suites for each context
- Anticomposition axiom:
  - Tests which are individually adequate for units of a program are not collectively adequate for the assembled units



# Inheritance

---

- When a class is changed, it and all its dependants should be (re)tested
- When a superclass is changed, it and all its subclasses should be (re)tested
- When a subclass is changed or added, all inherited methods should be (re)tested in the context of the subclass



# Polymorphism

---

Polymorphism replaces explicit compile time binding and checking by runtime binding and checking

Can contribute to errors:

- 'yo-yo' problem
- Polymorphism in combination with inheritance

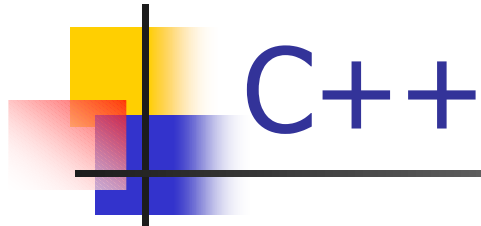


# The 'yo-yo' problem

---

The combination of polymorphism and method refinement (inheritance) make understanding of the lower level classes difficult

Often we get the feeling of riding a yo-yo when we try to understand the message trees



- C++ inherits some deficiencies of C
- Common problems include:
  - Pointer problems
  - Memory problems
  - Uninitialized values
  - Improper casts
  - Improper union usage



# C++ example

---

```
class Base {  
public:  
};  
class Derived:public Base {  
public:  
    int member_function() {  
        clear(); // calls Global clear  
    }  
};
```



# Adequacy and Coverage

---

- An adequacy criterion defines a set of components to be exercised
- A test suite is adequate with respect to some fault hypothesis if all of the component tests are called for by the testing model have been produced
- Adequacy does not imply 'fitness for use'



# Coverage

---

Coverage provides a simple operational definition for an adequacy criterion e.g.:

- Statement coverage: percentage of statements exercised
- Decision coverage: each branch of each conditional statement is exercised
- Dataflow coverage: exercise paths from object definition to its use



# Discussion

---

Thank you for your attention