# Testing Object-Oriented Software

# Problems in object-oriented testing [Binder]

Each *level in the class hierarchy* creates a *new context* for inherited features:

$\Rightarrow$ correctness of superclass does not guarantee that of subclass

Q: Do superclass methods work correctly within context of subclass ?

For `B` inheriting method `m` from `A`, we should know:

1. can we completely skip re-testing `B.m` ?
2. are the test cases for `A.m` enough ?
3. or do we need new test cases ? which ?

# Liskov Substitution Principle

subclass can be used anywhere instead of superclass

$$\text{pre}(m, \text{Class}) \Rightarrow \text{pre}(m, \text{SubClass})$$

$$\text{post}(m, \text{SubClass}) \Rightarrow \text{post}(m, \text{Class})$$

$$\text{inv}(\text{SubClass}) \Rightarrow \text{inv}(\text{Class})$$

But: we must *know* invariants to check them
At the minimum, we analyze *which fields are modified*

# The classic example: rectangle and square

```
public class Rectangle {
  private int height; private int width;
  public void setHeight(int value) { this.height = value; }
  public void setWidth(int value) { this.width = value; }
  public int getArea() { return this.height * this.width; }
}

public class Square extends Rectangle {
  public void setHeight(int value) { super.setHeight(value);
                                     super.setWidth(value); }
  public void setWidth(int value) { super.setWidth(value);
                                    super.setHeight(value); }
}
```

# More object-oriented testing problems

Interactions between *method calls* and *object state* are complex

Are there undesired interactions between methods ?

# More object-oriented testing problems

Interactions between *method calls* and *object state* are complex

Are there undesired interactions between methods ?

*Polymorphism* and dynamic binding increase number of execution paths

make static analysis more difficult

```
void foo(A obj) { obj.m(); }
```

could call method `m` for any subclass of `A`

# More object-oriented testing problems

Interactions between *method calls* and *object state* are complex

Are there undesired interactions between methods ?

*Polymorphism* and dynamic binding increase number of execution paths

make static analysis more difficult

```
void foo(A obj) { obj.m(); }
```

could call method `m` for any subclass of `A`

Encapsulation limits state *observability* when testing

# More object-oriented testing problems

Interactions between *method calls* and *object state* are complex

Are there undesired interactions between methods ?

*Polymorphism* and dynamic binding increase number of execution paths

make static analysis more difficult

```
void foo(A obj) { obj.m(); }
```

could call method `m` for any subclass of `A`

Encapsulation limits state *observability* when testing

*Dynamic binding* increases potential for misunderstanding and error

# More object-oriented testing problems

Interactions between *method calls* and *object state* are complex

Are there undesired interactions between methods ?

*Polymorphism* and dynamic binding increase number of execution paths

make static analysis more difficult

```
void foo(A obj) { obj.m(); }
```

could call method `m` for any subclass of `A`

Encapsulation limits state *observability* when testing

*Dynamic binding* increases potential for misunderstanding and error

*Interface errors* more likely due to many small components

# More object-oriented testing problems

Interactions between *method calls* and *object state* are complex

Are there undesired interactions between methods ?

*Polymorphism* and dynamic binding increase number of execution paths

make static analysis more difficult

```
void foo(A obj) { obj.m(); }
```

could call method `m` for any subclass of `A`

Encapsulation limits state *observability* when testing

*Dynamic binding* increases potential for misunderstanding and error

*Interface errors* more likely due to many small components

Control of *object state* is difficult: *distributed* throughout program

# Specific problems in OO testing (cont.)

[McGregor&Sykes] Due to fundamental language constructs

# Specific problems in OO testing (cont.)

[McGregor&Sykes] Due to fundamental language constructs

*Objects*

information hiding ⇒ harder to observe state in testing
have persistent state ⇒ inconsistency can cause errors later
have a lifetime ⇒ errors if constructed/destructed at wrong time

# Specific problems in OO testing (cont.)

[McGregor&Sykes] Due to fundamental language constructs

*Objects*

information hiding ⇒ harder to observe state in testing
have persistent state ⇒ inconsistency can cause errors later
have a lifetime ⇒ errors if constructed/destructed at wrong time

*Methods/messages* ⇒ important for testing object *interactions*

may be called in improper object state
have parameters (used/updated): are those in the right state?
do they correctly implement their interfaces? (subtyping errors)

# Specific problems in OO testing (cont.)

*Interface* = behavioral specification

Preconditions for correct behavior may be handled in two ways:

*contract-based*: assumed

*defensive programming*: checked

# Specific problems in OO testing (cont.)

*Interface* = behavioral specification

Preconditions for correct behavior may be handled in two ways:

*contract-based*: assumed

*defensive programming*: checked

⇒ influences complexity of implementation and testing

simplifies/complicates class/integration testing

Note: defensive programming should also check results! (although in practice, often receiver is considered trustworthy, only caller not)

# Specific problems in OO testing (cont.)

*Class*

*specification*: method pre/postconditions, class invariants $\Rightarrow$ tested!
Specification must also be *validated* !

*implementation*: error opportunities through
Constructors/destructors (incorrect initialization/deallocation)
Inter-class collabor.: members or object param. may have errors
Do clients have the means to check preconditions? (hidden state?)

# Specific problems in OO testing (cont.)

*Inheritance*

May propagate errors to descendants $\Rightarrow$ stopped by timely testing

Typical OO code style: short methods, little processing, many calls
$\Rightarrow$ code/decision coverage loses relevance

Offers a mechanism for test reuse, from super- to subclass

Testing may detect inheritance just for code reuse
without inheriting specification

# Specific problems in OO testing (cont.)

*Polymorphism*

Testing must check observing the substitution principle

# Specific problems in OO testing (cont.)

*Polymorphism*

Testing must check observing the substitution principle

From the perspective of *observable states* in program/testing:

- Subclass keeps all observable states and transitions among them
- May add transitions (supplementary behavior)
- May add observable states (sub-states of initial ones)

# Specific problems in OO testing (cont.)

*Polymorphism*

Testing must check observing the substitution principle

From the perspective of *observable states* in program/testing:

Subclass keeps all observable states and transitions among them
May add transitions (supplementary behavior)
May add observable states (sub-states of initial ones)

*Yo-yo problem*: difficulty of understanding/testing sequence of calls
⇒ likely error: call wrong method implementation from hierarchy

# Specific problems in OO testing (cont.)

*Polymorphism*

Testing must check observing the substitution principle

From the perspective of *observable states* in program/testing:

Subclass keeps all observable states and transitions among them
May add transitions (supplementary behavior)
May add observable states (sub-states of initial ones)

*Yo-yo problem*: difficulty of understanding/testing sequence of calls
$\Rightarrow$ likely error: call wrong method implementation from hierarchy

*Abstraction* in class hierachy reflected in tests (general $\rightarrow$ specific)

# Testing axioms

[Weyuker '86,'88], reformulated for OO by [Perry & Kaiser '90]

*Antiextensionality*:

Different implementations to same functionality need different tests.

1) A redefined method needs other/more tests (depending on code)

2) The *same* method when inherited needs different class-based tests
e.g.: `A: +m(), +n()`
`B: +m()`
`C: +n()`  `m` calls `n()`

$\Rightarrow$ `C::m` inherits `B::m` but calls *another* `n()` $\Rightarrow$ different tests!

# Testing axioms (cont.)

*Antidecomposition*:

A test set adequate for a program need not be adequate for one of its components

(it could be exercised in a different context to that program)

$\Rightarrow$ Adequate testing for a client is insufficient for a library
(client could use only part of the functionality)

$\Rightarrow$ If deriving from a tested class, must still test inherited methods
(code added may interact with the state $\Rightarrow$ with inherited methods)

# Testing axioms (cont.)

*Anticomposition*:

A test set adequate for components need not be adequate for their combination.

brief argument for sequential combination:
$p$ program paths in $P$ and $q$ paths $Q \Rightarrow p \cdot q > p + q$ paths $P; Q$
even more when execution alternates between $P$ and $Q$

$\Rightarrow$ Unit/module testing cannot replace integration testing!

$\Rightarrow$ A method tested in the base class is not tested sufficiently in the derived class (it may be composed in different ways).

*General Multiple Change*

Programs with the same control flow but different operations/values need different test suites.

# Error examples: Encapsulation

Set class with methods:

```
add(element)  // precondition: element not in set
              // raise Duplicate exception otherwise
remove(element)
```

Testing: two consecutive `add(x)` raise exception
    but element might still be added a second time

$\Rightarrow$ error discovered only with 2 $\times$ `add`, 2 $\times$ `remove`
    harder to test than with directly observable object state

# Error examples: Inheritance

Problem: implementing a class requires understanding details and representation conditions of all base classes to be sure of correct implementation

⇒ *Inheritance weakens encapsulation*

Two main classes of problems:

1) initialization
   forgetting correct initialization of superclass

2) forgetting redefinition of method accounting for class specifics
   copy methods or `isEqual`

# Coverage in object-oriented testing

Q: what are relevant object/method combinations to consider ?

*target-methods criterion*: all callable method implementations

*receiver-classes criterion*: all possible receiver classes

Example [Rountev, Milanova, Ryder 2004]

```
class A { public void m() { ... } }
class B extends A { public void m() { ... } }
class C extends A { ... }
A a;
...
a.m();
```

*target-methods*: test calls to la `A.m()`, `B.m()`
*receiver-classes* (more comprehensive): test a of type A, B, C

# Fault patterns in OO testing [Offutt]

*Inconsistent type use*: *Deriv* used inconsistently also as *Base*
e.g.: *Stack* (access at one end) derived from *Vector* (indexed access)
using `Vector::removeAt(idx)` on *Stack* violates class invariant
Cause: design error. Detection: test class invariants

# Fault patterns in OO testing [Offutt]

*Inconsistent type use*: *Deriv* used inconsistently also as *Base*
e.g.: *Stack* (access at one end) derived from *Vector* (indexed access)
using `Vector::removeAt(idx)` on *Stack* violates class invariant
Cause: design error. Detection: test class invariants

*State definition errors*

1) Overriden methods interact differently with object state
   Detection: check that methods define/use same members
2) Local redefinition of a member (hides inherited member)
   inherited methods still access the old member ⇒ inconsistency
3) Redefined method computes same member differently
   ⇒ state inconsistency with respect to the (inherited) specification

# Fault patterns in OO testing [Offutt]

*Inconsistent type use*: *Deriv* used inconsistently also as *Base*
e.g.: *Stack* (access at one end) derived from *Vector* (indexed access)
using `Vector::removeAt(idx)` on *Stack* violates class invariant
Cause: design error. Detection: test class invariants

*State definition errors*

1) Overriden methods interact differently with object state
   Detection: check that methods define/use same members
2) Local redefinition of a member (hides inherited member)
   inherited methods still access the old member ⇒ inconsistency
3) Redefined method computes same member differently
   ⇒ state inconsistency with respect to the (inherited) specification

*Constructor errors*: calling non-final method
(overridden in subclass, method has thus access to uninitialized state)

*Visibility anomalies*

# Specifics of OO testing

*Testing levels*: intra- and inter-method, intra- and inter-class

# Specifics of OO testing

*Testing levels*: intra- and inter-method, intra- and inter-class

*Visibility problem* (caused by encapsulation):

- explicit flattening of class hierarchy
- better: allowing data access by testing framework
- or: use getter methods to access state

# Specifics of OO testing

*Testing levels*: intra- and inter-method, intra- and inter-class

*Visibility problem* (caused by encapsulation):

explicit flattening of class hierarchy

better: allowing data access by testing framework

or: use getter methods to access state

*Polymorphism*: tests need to instantiate all possible subtypes for an object declared as a base type

static analysis to find all possibilities (class hierarchy analysis)

# Specifics of OO testing

*Testing levels*: intra- and inter-method, intra- and inter-class

*Visibility problem* (caused by encapsulation):

explicit flattening of class hierarchy

better: allowing data access by testing framework

or: use getter methods to access state

*Polymorphism*: tests need to instantiate all possible subtypes for an object declared as a base type

static analysis to find all possibilities (class hierarchy analysis)

*Dataflow* testing

Data and changed state are important;
line/branch coverage gives little info on small method bodies

*Coupling*: defined by *def-use* pairs b/w methods

i.e. a member defined(written) in m1() and used(read) by m2()

used to select methods that are tested together

# Testing class hierarchies

Distinguish: tests starting from *specification* or *implementation* (code)

S: new tests for old methods, when specification changes

S: new postconditions/invariants for old tests in derived classes

I: new tests for new methods, depending on desired coverage

# Testing class hierarchies

Distinguish: tests starting from *specification* or *implementation* (code)
  S: new tests for old methods, when specification changes
  S: new postconditions/invariants for old tests in derived classes
  I: new tests for new methods, depending on desired coverage

Examples:

Change a method `m()`: retest methods that interact:
  methods calling `m` and that have *coupling* with `m`

Change `m()` in superclass: re-test `m()` + interacting methods;
re-test `m()` in context of subclass(es)

Overwrite `m()`: augment tests of `Base::m` for adequate coverage

Overwrite `m()` used by `Base::n`: test `n` in subclass

Change of interface (abstract class): re-test whole hierarchy

# OO testing patterns [Binder]

# OO testing patterns [Binder]

At *method* level

- Category/Partition (I/O analysis, partitioning/equivalence)
- Combinational Function Test (condition coverage)
- Recursive Function Test
- Polymorphic Message Test (client of a polymorphic server)

# OO testing patterns [Binder]

At *method* level

- Category/Partition (I/O analysis, partitioning/equivalence)
- Combinational Function Test (condition coverage)
- Recursive Function Test
- Polymorphic Message Test (client of a polymorphic server)

At *class* level

- Invariant Boundaries (valid/invalid values for class invariant)
- Nonmodal Class Test (class w/o sequencing constraints)
- Modal Class Test (class with sequencing constraints)
- Quasi-Modal Class Test (constraints dependent on state)

# OO testing patterns [Binder]

At *method* level

- Category/Partition (I/O analysis, partitioning/equivalence)
- Combinational Function Test (condition coverage)
- Recursive Function Test
- Polymorphic Message Test (client of a polymorphic server)

At *class* level

- Invariant Boundaries (valid/invalid values for class invariant)
- Nonmodal Class Test (class w/o sequencing constraints)
- Modal Class Test (class with sequencing constraints)
- Quasi-Modal Class Test (constraints dependent on state)

For *reusable components*

- Abstract Class Test (interface)
- Generic Class Test (parameterized)
- New Framework Test
- Popular Framework Test (changes in an API)

# Example: Polymorphic Message Test

For a virtual method call (in a client), test all possible classes to which the call could be made

Need to deal with / potential errors:
– incorrect preconditions on call for some subclasses
– call to unintended class (reference to unintended type)
– change of class hierarchy (affects code/tests)

Dynamic binding is similar to (multi-way) branch in code
$\Rightarrow$ covering all instances $\simeq$ branch coverage

# Nonmodal Class Test

Nonmodal class: accepts any method call in any state
e.g. DateTime accepts any sequence of get/set (use/def)

Types of test behavior

– define-operation: set to valid input / check answer
– define-exception: set to invalid input / check answer
– define-exception-corruption: state not corrupt after exception

– use-exception-test: normal return after use
– use-correct-return: return with correct value after use
– use-corruption: object not corrupt after use

# (Quasi-)Modal Class Test

*Modal Class Test*:
class with fixed constraints on operation order create a *model* with object state and transitions between them

Problems:
– missing transition: an operation is rejected in a valid state
– incorrect action / response for a method in a given state
– invalid resulting state: method causes transition to wrong state
– corrupt resulting state
– message accepted when it should be rejected

# (Quasi-)Modal Class Test

*Modal Class Test*:
class with fixed constraints on operation order create a *model* with object state and transitions between them

Problems:
– missing transition: an operation is rejected in a valid state
– incorrect action / response for a method in a given state
– invalid resulting state: method causes transition to wrong state
– corrupt resulting state
– message accepted when it should be rejected

*Quasi-modal class test*
method order constraints change depending on state
e.g. container / collection classes (full/empty), etc.

Typically, we'd like N+ coverage (any method in any state)

# Testing at class level

*Small Pop* approach
– write class, write tests, run (no other details/intermediate steps)
– good for simple classes in stable contexts

*Alpha-Omega* approach
– run object from creation to destruction through all methods
  – constructors
  – accesors (get)
  – predicates
  – modifiers (set)
  – iterators
  – destructors