Testing Object-Oriented Software

Class Testing

Class Testing

- Introduction
- Accounting for Inheritance
- Testing Method Sequences
- State-Based Testing
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Motivations

- Object-orientation helps analysis and design of large systems
- But, based on existing data, it seems that more, not less, testing is needed for OO software
- OO software has specific constructs that we need to account for during testing
- Unit/Component and Integration testing are especially affected as they are more driven by the structure of the software under test

Class vs. Procedure Testing

- Procedural programming
  - basic component: function (procedure)
  - testing method: based on input/output relation
- Object-oriented programming
  - basic component: class = data members (state) + set of operations
  - objects (instances of classes) are tested
  - correctness cannot simply be defined as an input/output relation, but must also include the object state.
- The state may not be directly accessible, but can normally be accessed using public class operations

Example

```
class Watcher {
  private:
    …
    int status;
    …
  public:
    void checkPressure() {
      …
      if (status == 1)
        …
      else if (status …)
        …
    }
    …
}
```

- Testing method checkPressure() in isolation is Meaningless.
  - Generating test data
  - Measuring coverage
- Creating oracles is more difficult
  - the value produced by method *check_pressure* depends on the state of class *Watcher's* instances (variable *Status*)
  - failures due to incorrect values of variable *Status* can be revealed only with tests that have control and visibility on that variable

New Abstraction Levels

- Functions (subroutines) are the basic units in procedural software
- Classes introduce a new abstraction level:
  - *Basic unit testing*: the testing of a single operation (method) of a class (intra-method testing)
  - *Unit testing*: the testing of a class (intra-class testing)
- *Integration testing*: the testing of interactions among classes (inter-class testing), related through dependencies, i.e., associations, aggregations, specialization

New Faults Models

- Wrong instance of method inherited in the presence of multiple inheritance
- Wrong redefinition of an attribute / data member
- Wrong instance of the operation called due to dynamic binding and type errors
- We lack statistical information on frequency of errors and costs of detection and removal.
- New fault models are vital for defining testing methods and techniques targeting OO specific faults

Structural Testing in OO Context

- In OO systems, most methods contain a few LOCs – complexity lies in method interactions
- Method behavior is meaningless unless analyzed in relation to other operations and their joint effect on a shared state (data member values)
- It is claimed that any significant unit to be tested cannot be smaller than the instantiation of one class

Testing and Inheritance

- Modifying a superclass
  - We have to retest its subclasses (expected)
- Add a subclass (or modify an existing subclass)
  - We may have to retest the methods inherited from each if its ancestor superclasses
- Reason: Subclasses provide new context for the inherited methods
- No problems if the new subclass is a pure extension of the superclass

  Pure Extension of superclasses:
  - It adds new instance variables and methods and there are no interactions in either directions between the new instance variables and methods and any inherited instance variables and methods
  - Example of interaction: a superclass and one of its subclass initialize a variable to different values in two distinct methods, one in the superclass and one in the subclass

Inheritance: Example I (1)

```
class refrigerator {
public:
    void set_desired_temperature(int temp);
    int get_temperature();
    void calibrate();
private:
    int temperature;
};
```

- `set_desired_temperature` allows the temperature to be between 5 C and 20 C centigrade.
- `calibrate` puts the actual refrigerator through cooling cycles and uses sensor readings to calibrate the cooling unit.

Inheritance: Example I (2)

- A new more capable model of refrigerator is created and can cool to – 5 C centigrade
- Class `better_refrigerator` and a new version of `set_desired_temperature`
- Method `calibrate` is unchanged
- Should `better_refrigerator::calibrate` be re-tested? It has the exact same code!

Inheritance: Example I (3)

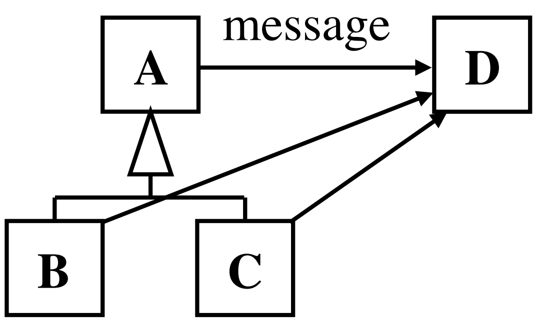
- Yes, it has to be re-tested
- Suppose that calibrate works by dividing sensor readings by temperature
- What if `temperature` = 0?
- That's possible in `better_refrigerator`
- Will cause a divide by 0 failure which cannot happen in `refrigerator`

Overriding of Methods

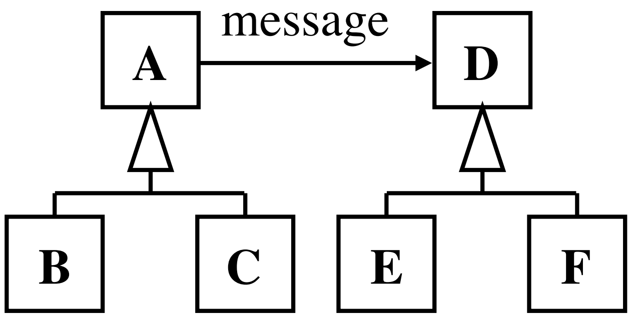
- OO languages allow a subclass to replace an inherited method with a method of the same name
- The overriding subclass method has to be tested
- But *different* test sets are needed! (though the intersection may be large)
- *Reason 1:* If test cases are derived from program structure (data and control flow), the structure of the overriding method may be different
- *Reason 2:* The overriding method behavior is also likely to be different

Integration and Polymorphism

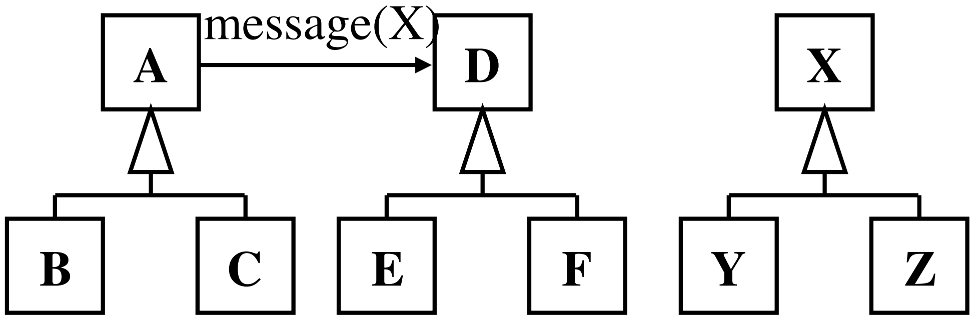
1 test set



3 test sets



9 test sets



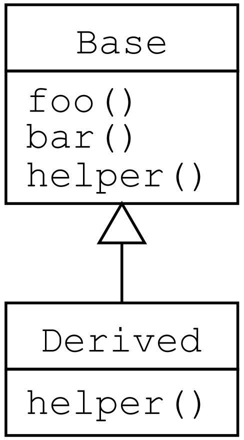
27 test sets

Class Testing

- Introduction
- **Accounting for Inheritance**
- Testing Method Sequences
- State-Based Testing
- Testability for State-based Testing
- Test Drivers, Oracles, and Stubs

Example II: Code

```
class Base    {
   public:
     void foo()         { … helper(); …}
     void bar()         { … helper(); …}
   private:
     virtual void helper()    {…}
};
class Derived : public Base {
   private:
     virtual void helper()    {…}
};
void test_driver()  {
     Base base;
     Derived derived;
     base.foo();        // Test case 1
     derived.bar(); // Test case 2
}
```



© Lionel Briand 2010

Example II: Discussion

- Test case 1: Invokes `Base::foo()` which in turns call `Base::helper()`
- Test case 2: The inherited method `Base::bar()` is invoked on the derived object, which in turns calls `helper()` on the derived object, invoking `Derived::helper()`
- Assuming all methods contain linear control flow only, do the test cases fully exercise the code of both `Base` and `Derived`?
- Traditional coverage measures (e.g., statements, control flow) would answer *yes*

Example II: Missed anything?

- We have not fully tested interactions between `Base` and `Derived`
  - `Base::bar()` and `Base::helper()`
  - `Base::foo()` and `Derived::helper()`
- It is not because `Base::foo()` works with `Base::helper()` that it will automatically with `Derived::helper()`
- We need to exercise `foo()` and `bar()` for both the base and derived class

Example II: New Test Driver

```
void better_test_driver()      {
    Base base;
    Derived derived;
    base.foo();
    derived.foo();
    base.bar();
    derived.bar();
}
```

You can see why inheritance has to be used with care – it leads to more testing!

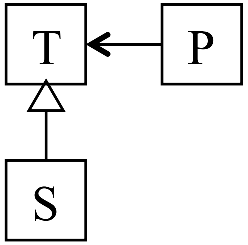
Hierarchical Incremental Testing

- Aims at testing inheritance hierarchies (Harrold, McGreggor, IEEE ICSE proceedings,1992)
- *Step 1*: Test all methods fully in the context of a particular class (base class or a derived class for abstract base classes)
- *Step 2*, Interaction coverage: Any methods which are inherited by a derived class and which interact with any re-defined methods (or new methods through inherited attributes) should be re-tested in the context of the derived class
- Re-run all the base class test cases (e.g., based on 100% edge coverage requirements) in the context of the derived class by which it is inherited
- This reduces the cost of testing inherited methods in several contexts and help check the conformance of inheritance hierarchies to the *Liskov substitution principle*

Liskov Substitution Principle

- This principle defines the notions of generalization / specialization in a formal manner
- Class S is correctly defined as a specialization of class T if the following is true:

> for each object s of class S there is an object t of class T such that the behavior of any program P defined in terms of T is unchanged if s is substituted for t.



- S is a said to be a *subtype* of T
- All instances of a subclass can stand for instances of a superclass without any effect on client classes
- Any future extension (new subclasses) will not affect existing clients.

Lack of Substitutability

```
class Rectangle : public Shape {
private: int w, h;
public:
  virtual void set_width(int wi) {
    w=wi;
  }
  virtual void set_height(int he) {
    h=he;
  }
}
```

```
class Square : public Rectangle {
public:
  void set_width(int w) {
    Rectangle::set_height(w);
    Rectangle::set_width(w);
  }
  void set_height(int h) {
    set_width(h);
  }
}
```

```
void foo(Rectangle *r) { // This is the client
    r->set_width(5);
    r->set_height(4);
    assert((r->get_width()*r->get_height()) == 20); // Oracle
}
```

- If r is instantiated at run time with instance of square, behavior observed by client is different (width*height == 16)
- May lead to problems
- Square should be defined as subclass of Shape, not Rectangle

Rules

- *Signature Rule:* The subtypes must have all the methods of the supertype, and the signatures of the subtypes methods must be *compatible* with the signatures of the corresponding supertypes methods
- In Java, this is enforced as the subtype must have all the supertype methods, with identical signatures except that a subtype method can have fewer exceptions (compatibility stricter than necessary here)
- *Method Rule*: Calls on these subtype methods must "behave like" calls to the corresponding supertype methods.
- *Properties Rule*: The subtype must preserve the properties (invariant) of the supertype.

Contracts - Definitions

- Goals: Specify operations so that caller/client and callee/ server operations share the same assumptions
- A contract specifies constraints that the caller must meet before using the class as well as the constraints that are ensured by the callee when used.
- Three types of constraints involved in contracts: Invariant (class), Precondition, postcondition (operations)
- Contracts should be specified, for known operations, at the Analysis & design stages
- In UML, a language has been defined for that purpose: The Object Constraint Language (OCL)
- JML is available to define contracts within Java programs that can be checked at run time (http://en.wikipedia.org/wiki/Java_Modeling_Language)

Class Invariant

- Condition that must always be met by all instances of a class
- Described using that an expression that evaluates to true if the invariant is met
- Invariants must be true all the time, except during the execution of an operation where the invariant can be temporarily violated.
- A violated invariant suggests an illegal system state

SavingsAccount

balance
{balance>0 and
balance<250000}

```
Context SavingsAccount inv:
self.balance > 0 and self.balance < 25000
```

Operation Pre and Post Conditions

- Pre-condition: What must be true before executing an operation
- Post-condition: Assuming the pre-condition is true, what should be true about the system state and the changes that occurred after the execution of the operation
- These conditions have to be written as logical (Boolean) expressions
- Thus, operations are treated as black boxes. Nothing is said about operations' intermediate states and algorithmic details
- If the pre- and post-conditions are satisfied, then the class invariant must be preserved

Before

After

Precondition
(what must be true before)

Postcondition
(change that has occurred)

Design by Contract

```
Contractor :: put (element: T, key: STRING)
-- insert element x with given key
```

	Obligations	Benefits
Client	Call put only on a non-full table	Get modified table in which x is associated with key
Contractor	Insert x so that it may be retrieved through key	No need to deal with the case in which the table is full before insertion

Specifying Contracts

- Specify the requirements of system operation in terms of inputs and system state (Pre-condition)
- Specify the effects of system operations in terms of state changes and output (Post-condition)
- The state of the system is represented by the state of objects and the relationships (links) between them
- A system operation may
    - create a new instance of a class or delete an existing one
    - change an attribute value of an existing object
    - add or delete links between objects
    - send an event/message to an object

Method Rule

Rule can be expressed in pre- and post-conditions

- The precondition is *weakened*
  - Weakening the precondition implies that the subtype method requires less from the caller
  - If methods `T::m()` and `S::m()` (overriding) have preconditions `PrC1` and `PrC2`, respectively, $\mathtt{PrC1} \Rightarrow \mathtt{PrC2}$
- The postcondition is *strengthened*
  - Strengthening means the subtype method returns more than the supertype method
  - If methods `T::m()` and `S::m()` (overriding) have postconditions `PoC1` and `PoC2`, respectively, $(\mathtt{PrC1} \wedge \mathtt{PoC2}) \Rightarrow \mathtt{PoC1}$

➢ The calling code depends on the postcondition of the supertype method, but only if the precondition is satisfied

IntSet

```
public class IntSet {
private Vector els; /// the elements
public IntSet() {…}
   // Post: Initializes this to be empty
public void insert (int x) {…}
   // Post: Adds x to the elements of this
public void remove (int x) {…}
   // Post: Remove x from the elements of this
public boolean isIn (int x) {…}
   //Post: If x is in this returns true else returns false
public int size () {…}
   //Post: Returns the cardinality of this
public boolean subset (IntSet s) {…}
   //Post: Returns true if this is a subset of s  else returns false
}
```

Postconditions: MaxIntSet

```
public class MaxIntSet extends IntSet {
private int biggest; // biggest element if set not empty
public maxIntSet () {…} // call super()
public max () throws EmptyException {…} // new method
public void insert (int x) {…}
   // overrides InSet::insert()
   //Additional Post: update biggest with x if x > biggest
public void remove (int x) {…}
   // overrides InSet::remove()
   //Additional Post: update biggest with next biggest
   element in this if x = biggest
}
```

Preconditions: LinkedList & Set

```
public class LinkedList {
  ...
  /** Adds an element to the end of the list
  * PRE:  element != null
  * POST: this.getLength() == old.getLength() + 1
  *       && this.contains(element) == true
  */
  public void addElement(Object element) { ... }
  ...
}

public class Set extends LinkedList {
  ...
  /** Adds element, provided element is not already in the set
  * PRE:  element != null && this.contains(element) == false
  * POST: this.getLength() == old.getLength() + 1
  *       && this.contains(element) == true
  */
  public void addElement(Object element) { ... }
  ...
}
```

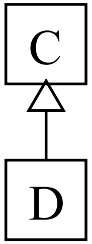
Properties Rule

- All methods of the subtype must preserve the invariant of the supertype
- The invariant of the subtype must imply the invariant of the supertype
- Assume `FatSet` is a set of integers whose size is always at least 1. The constructor and remove methods ensure this.
- `ThinSet` is also a set of integers but can be empty and therefore cannot be a legal subtype of `FatSet`

InSet, MaxInSet

- Invariant of `IntSet`, for any instance `i` :
  `i.els != null` and
  all elements of `i.els` are `Integer`s and
  there are no duplicates in `i.els`
- Invariant of `MaxIntSet`, for any instance `i` :
  invariant of `InSet` and
  `i.size > 0` and
  for all integers `x` in `els`, `x <= i.biggest`
- The invariant of MaxInSet includes the invariant of InSet and therefore implies it.
- We comply with the property rule.

Hierarchical Incremental Testing (II)



- Assuming `C` is the base class and `D` a subclass of `C`
- Override in `D` a method in `C` but no change in specification
  - Reuse all the inherited specification-based test cases
  - But will need to review implementation-based test cases to meet the test criterion for coverage
- Change in `D` the specification of an operation in `C`:
  - Additional test cases to exercise new input conditions (weakened precondition) and check new expected results (strengthened postcondition)
  - Test cases for *C* still apply
  - Refine oracle (strengthened postcondition)
- New operations introduce new functionality and code to test
- New attributes are added in connection with new or overridden operations – this may lead to re-testing inherited methods
- New class invariant: All test cases need to be rerun to verify the new invariant holds

Inheritance Context Coverage

- Extend the interpretation of traditional structural coverage measures
- Consider the level of coverage in the context of each class as *separate* measurements
- 100% inheritance context coverage requires the code must be *fully* exercised (for any selected criteria, e.g., all edges) in *each* appropriate context
- Appropriate contexts can be determined using the HIT principles seen before