Cpt S 422: Software Engineering Principles II

Object-Oriented (Class) testing – Part 1

Dr. Venera Arnaoudova

WASHINGTON STATE UNIVERSITY

Object-Oriented Testing

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

Motivation

- Object-orientation helps the analysis and design of large systems
- But, based on existing data, it seems that more testing is needed for OO software
- OO software has specific constructs that we need to account for during testing
- Unit 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
  - testing method: based on input/output relation

- Object oriented programming
  - basic component: class = data members/attributes (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.

- Problem: The state of an object might not be directly accessible (recall the **encapsulation** principle), but can only 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.
- Creating oracles is more difficult, e.g.,
  - the value produced by method `checkPressure` depends on the state of class `Watcher`'s instances (variable `status`);
  - failures due to incorrect values of attribute `status` can be revealed only with tests that have control and visibility on that variable.

New Abstraction Levels

- Classes introduce a new abstraction level for unit testing:
  - *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
- As system testing is concerned with testing the system as a whole, it does not depend on the approach used to develop the software

New Faults Models

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

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 {
  private:
     int temperature;

  public:
     void set_desired_temperature(int temp);
     int get_temperature();
     void calibrate();
};
```

- `set_desired_temperature` allows the temperature to be between 5 C and 20 C centigrade.
- `calibrate` puts the 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 centigrade
- A new version of `set_desired_temperature`
- Method `calibrate` is unchanged
- Is `better_refrigerator` a pure extension?
- Should `better_refrigerator::calibrate` be re-tested? **It has the exact same code!**

Inheritance: Example I (3)

- Not a pure extension as `set_desrired_temperature` is redefined and accesses an inherited attribute
- Yes, `better_refrigerator::calibrate` has to be re-tested
- Suppose that calibrate works by dividing sensor readings by temperature
- What happens if the `temperature` is 0?
- That's possible in `better_refrigerator` and 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 (expected)

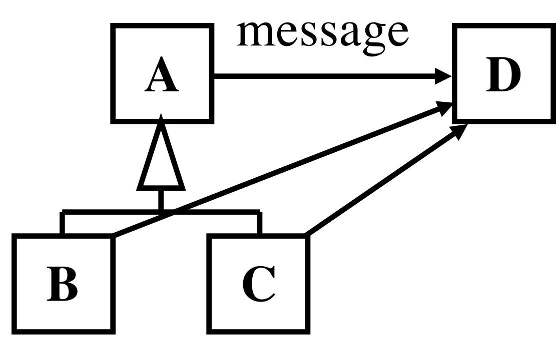
- But ***different* test sets are needed**, for two reasons:

  - *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**

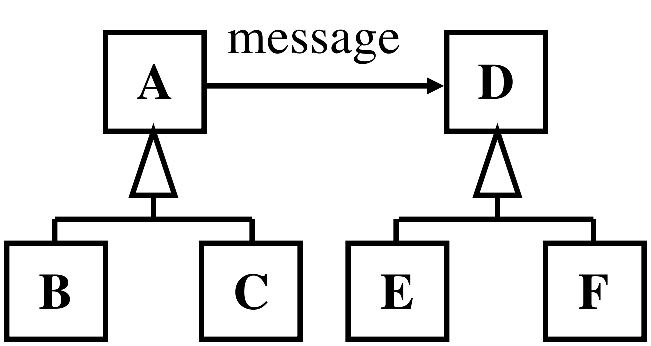
Integration Testing and Polymorphism

1 test set

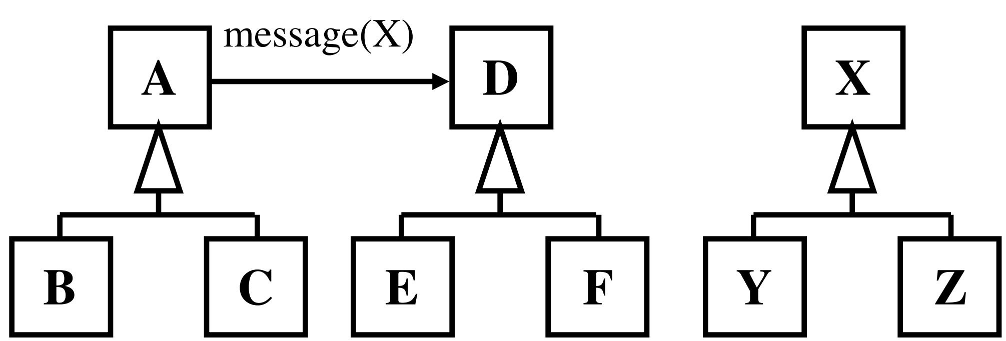


3 test sets

Assume that no class is ABSTRACT



9 test sets



27 test sets

Object-Oriented Testing

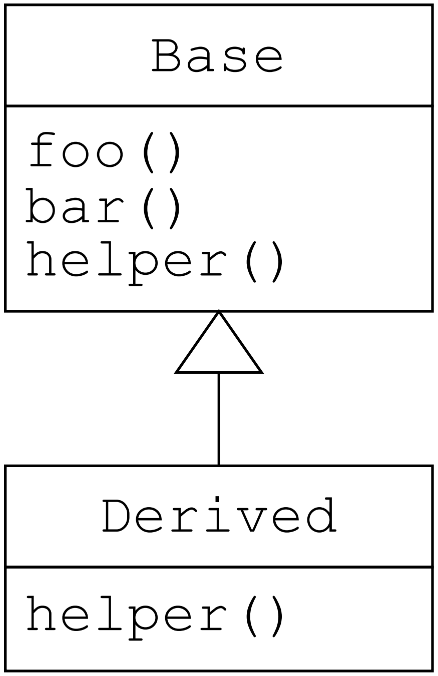
- ✓ Introduction
- ❑ Accounting for Inheritance
- ❑ Testing Method Sequences
- ❑ State-Based Testing
- ❑ Testability for State-based Testing
- ❑ Test Drivers, Oracles, and Stubs

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(); // 1
      derived.bar(); // 2
      // … assertions
}
```



Which methods are invoked?

Example II: Discussion

- 1: Invokes `Base::foo()` which in turns call `Base::helper()`
- 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`, we are missing
  - `Base::bar()` and `Base::helper()`
  - `Base::foo()` and `Derived::helper()`

- **It is not because** `Base::foo()` **works with** `Base::helper()` **that it will correctly work 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.bar();
      derived.foo();
      base.bar();
      // … assertions
}
```

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

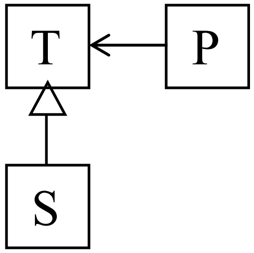
Hierarchical Incremental Testing (Harrold and McGreggor, 1992)

- Aims at testing inheritance hierarchies:
  - *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 method that is inherited by a derived class and that interacts with any re-defined method (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
- This reduces the cost of testing inherited methods in several contexts and helps to check the conformance of inheritance hierarchies to the <u>*Liskov substitution principle*</u>*:* ***In class hierarchies, it should be possible to treat a specialized object as if it were a base class object.***

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 then 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

Does the substitution principle hold?

```
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, the behavior observed by the client is different (width*height == 16)
- Square should be defined as subclass of Shape, not Rectangle

Subtype 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
- ***Method Rule***: Calls on these subtype methods must “behave like” calls to the corresponding supertype methods.
- ***Properties Rule***: The subtype must preserve the invariant of the supertype.

Contracts - Definitions

- Goals: Specify operations so that caller/client and callee/server operations share the same assumptions

- A contract specifies <u>constraints that the caller must meet</u> before using the class as well as the <u>constraints that are ensured by the callee</u> when used.

- Three types of constraints involved in contracts: Invariant (class), Precondition and 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)

Class Invariant

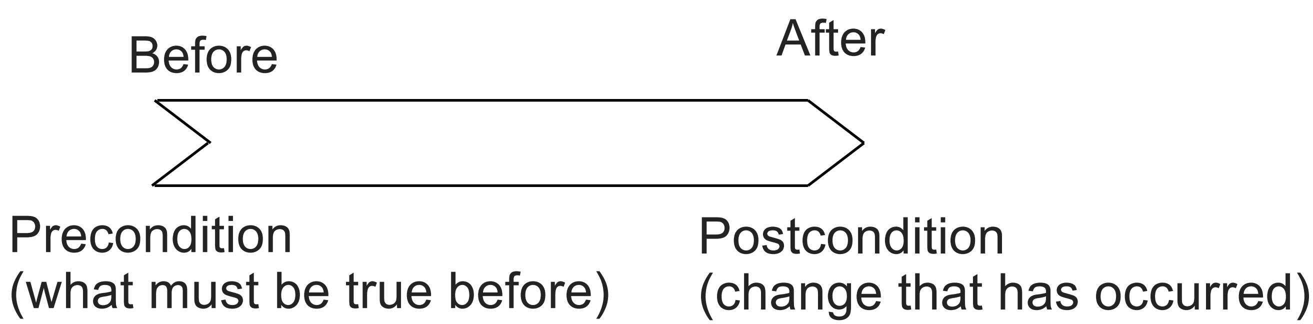
- Condition that must always be met by all instances of a class
- Described using that an expression that evaluates to true if the invariant is met
- Invariants must be true all the time, except during the execution of an operation where the invariant can be temporarily violated.
- A violated invariant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

SavingsAccount
balance {balance>0 and balance<250,000}

```
Context SavingsAccount inv:
self.balance > 0 and self.balance < 250,000
```

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, system operations are treated as **black boxes**. Nothing is said about operations' intermediate states and algorithmic details



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 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

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

Method Rule

Can be expressed in terms of pre- and post-conditions

- The precondition can be *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, `PrC1` $\Rightarrow$ `PrC2`
- The postcondition can be *strengthened*
  - Strengthening means that the subtype method returns more than the supertype method
  - If methods `T::m()` and `S::m()` (overriding) have postconditions `PoC1` and `PoC2`, respectively, `(PrC1 AND PoC2)` $\Rightarrow$ `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 els to be empty
   public void insert (int x) {…}
     // Post: Adds x to the elements of els
   public void remove (int x) {…}
     // Post: Remove x from the elements of els
   public boolean isIn (int x) {…}
     // Post: If x is in els returns true else returns false
   public int size () {…}
     // Post: Returns the cardinality of this
   public boolean subset (IntSet s) {…}
     // Post: Returns true if els is a subset of s, else returns false
}
```

Is the Liskov substitution principle satisfied?

Postconditions: MaxIntSet

```
public class MaxIntSet extends IntSet {
   private int biggest; // biggest element
                        // if the set not empty
   public maxIntSet () {…} // calls 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 els if x = biggest
}
```

Preconditions: LinkedList & Set

Is the Liskov substitution principle satisfied?

```
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
- Can `ThinSet` be a subtype of `FatSet`?
  - No. `ThinSet` is also a set of integers but can be empty and therefore cannot be a legal subtype of `FatSet`

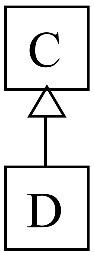
IntSet, MaxInSet

- 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`

- Is the Property Rule satisfied?
  - Yes because the invariant of MaxInSet includes the invariant of InSet and therefore implies it.

Hierarchical Incremental Testing (II)



- Assuming `C` is the base class and `D` a subclass of `C`
- Override in `D` a method of `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` of the specification of an operation of `C` :
  - Additional test cases are needed to exercise new input conditions (weakened precondition) and check new expected results (strengthened postcondition)
  - Test cases for C still apply
  - Refine oracle
  - 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 that 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 that the code must be *fully* exercised (for any selected criteria, e.g., all edges) in *each* appropriate context