SYSC 4101 / 5105

Inheritance/Specialization and Testing

Class vs. Procedure Testing

Different notion of unit under test

- Procedural programming
  - basic component: function (procedure)
    - sometimes notion of state (e.g., global variables)
  - testing method:
    - based on input/output relation of functions in isolation
    - groups of functions with functional testing (FSM, input domain…)
- Object-oriented programming
  - basic component: class = data members (attributes) + set of operations
    - objects (instances of classes) are tested
    - correctness cannot simply be defined as an input/output relation,
      - must also include the object state (data members/attributes = state).
  - testing method:
    - methods must be considered together (more than in isolation)
    - their joint reaction and effect on state (state often important)

Class vs. Procedure Testing

Different notion of unit under test

- Procedural programming
  - Unit testing = testing a function
  - Integration testing = testing interacting functions
- Object-oriented programming = new abstraction level
  - Unit testing = class (object) testing
  - Integration testing = testing interacting classes (objects)
  - Older notions:
    - Unit testing = testing methods in isolation
    - Intra-class testing (notion of integration) = testing interactions of methods of a class
    - Inter-class testing (notion of integration) = testing interactions of methods from different classes

Class vs. Procedure Testing

Observability & Controlability issues more prevalent in OO

- Procedural programming
  - state, when there is one
  - global variables easily accessible (no encapsulation)
- Object-oriented programming
  - because of private/protected attributes (encapsulation)
  - may be accessed using public class methods (setAge(), getAge())
  - sometimes there is no setXXX() or getXXX() methods.
    - these methods break encapsulation

Example I

encapsulated state

Method behaviour depends on shared state

```
public class BufferedOutputStream extends FilterOutputStream {
  protected byte buf[]; // internal buffer
  protected int count; // number of bytes currently in buffer
  public BufferedOutputStream(OutputStream out) { this(out, 8192); }
  public BufferedOutputStream(OutputStream out, int size) {
    super(out);
    if (size <= 0) { throw new IllegalArgumentException("Buffer size <= 0"); }
    buf = new byte[size]; // creating the internal buffer
  }
  private void flushBuffer() throws IOException { // writing buffer to out
    if (count > 0) {
      out.write(buf, 0, count);
      count = 0;
    }
  }
  public synchronized void write(int b) throws IOException {
    if (count >= buf.length) { flushBuffer(); }
    buf[count++] = (byte)b;
  }
  public synchronized void write(byte b[], int off, int len) throws IOException {
    if (len >= buf.length) {
      flushBuffer();
      out.write(b, off, len);
      return;
    }
    if (len > buf.length - count) {
      flushBuffer();
    }
    System.arraycopy(b, off, buf, count, len);
    count += len;
  }
  public synchronized void flush() throws IOException {
    flushBuffer();
    out.flush();
  }
}
```

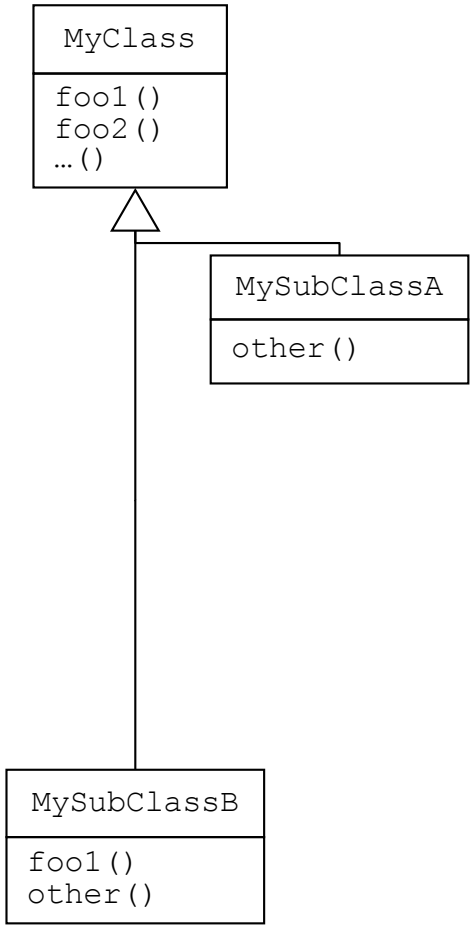
Structural Testing in OO Context

- In OO systems,
  - Most operations / methods contain a few LOCs
  - Complexity lies in method interactions
- Impact on structural testing
  - 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

New Faults Models

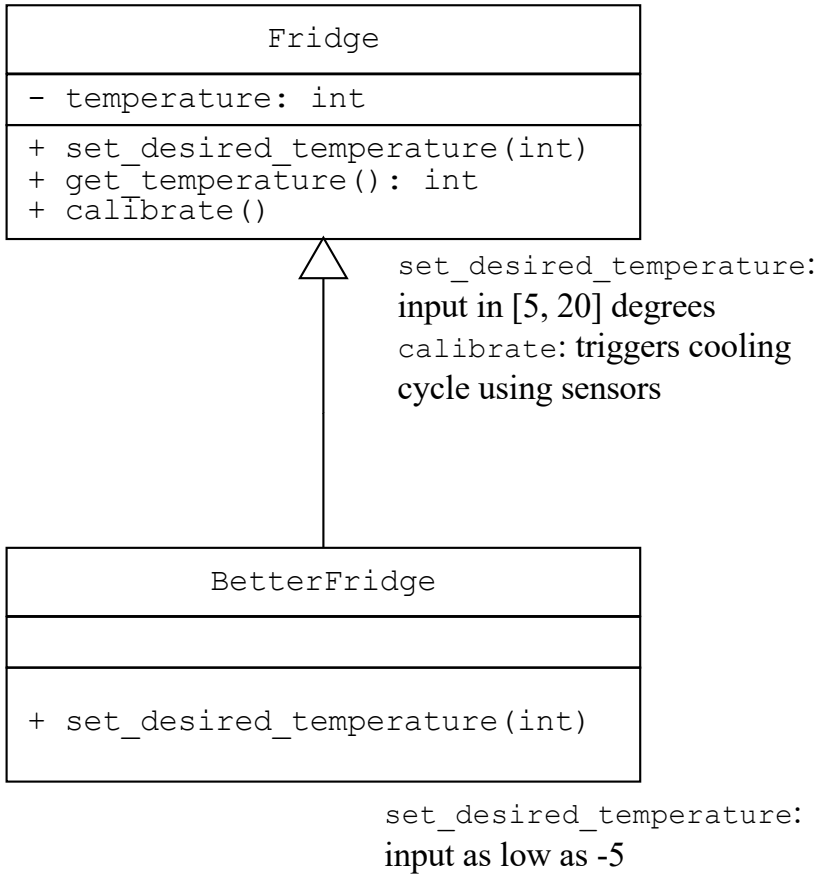
- OO specific faults
  - Wrong instance of method inherited in the presence of multiple inheritance
  - Wrong redefinition of an attribute / data member
  - Wrong instance of the operation called due to dynamic binding and type errors
  - We lack statistical information on frequency of errors and costs of detection and removal.
  - New fault models are vital for defining testing methods and techniques targeting OO specific faults
- Traditional fault taxonomies, on which are based control and data flow testing techniques, do not include faults due to object-oriented features

Testing and Inheritance



- One class to test, the parent
  - Unit test
    - Structural testing, Functional testing
  - Possible integration with service classes
- Subclassing—pure extension
  - Pure extension ?
    - New members but no interactions with inherited ones
    - No redefinitions of inherited members
  - Same as above for everything that is new
    - Unit test
      - Structural testing, Functional testing
    - Possible integration with service classes
- Subclassing—not pure extension
  - May have to retest inherited members already tested in parent class
  - Reason: subclasses provide new context for the inherited methods

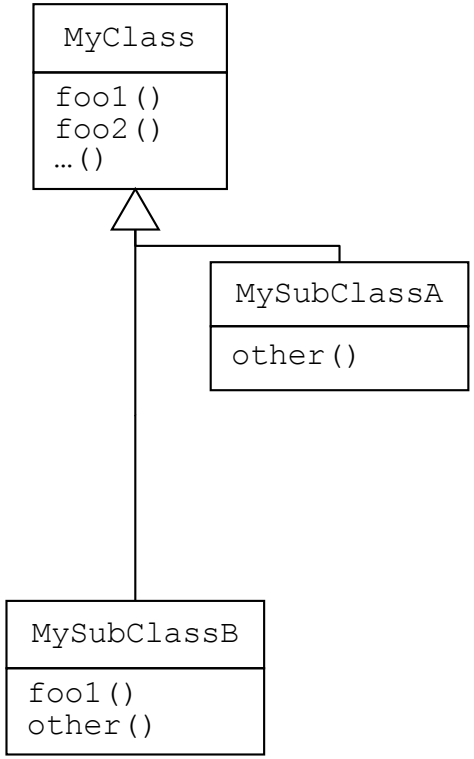
Inheritance: Example II



- Unit test class Refrigerator
  - Test its methods
  - Test method interactions
  - Test integration with services

- Should we retest `calibrate()` ?
  - The implementation has not changed!
- Yes:
  - Subclass provides new context of execution (new range of values for `temperature`)
  - `temperature` could be zero, resulting in division by zero in `calibrate()`.

Testing and Inheritance

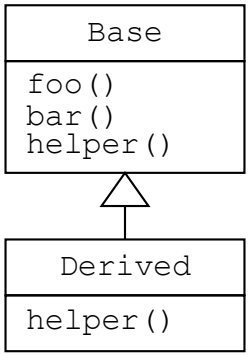


- One class to test, the parent
  - TestSuiteA – testing `foo1()`
  - TestSuiteB – testing `foo2()`
  - …
- Subclassing—pure extension
  - TestSuiteC – testing `other()`
- Subclassing—not pure extension
  - Reuse TestSuiteA – testing `foo1()`
    - Functional tests should pass
      - more functional tests may be needed if specification has changed
        (Liskov: weaker precondition, stronger postcondition)
    - Structural tests should pass
      - omplement for structural coverage if need be (not the same implementation)
    - Reusing: driver, stubs, oracles
  - TestSuiteD – testing `other()`
    - Similar, perhaps identical to TestSuiteC

Inheritance brings additional testing costs But we can reuse

Example III

```
class Base {
public:
  void foo() { … helper(); …}
  void bar() { … helper(); …}
private:
  void helper() {…}
};
```




```
class Derived : public Base {
private:
  void helper() {…}
};
```

Inheritance brings additional testing costs But we can reuse

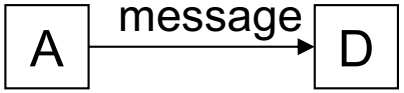
```
void test_driver() {
   Base base;
   Derived derived;
   base.foo();          // TC1
   derived.bar();       // TC2
}
```

```
void better_test_driver() {
   Base base;
   Derived derived;
   base.foo();          // TC1
   derived.foo();       // TC3
   base.bar();          // TC4
   derived.bar();       // TC2
}
```

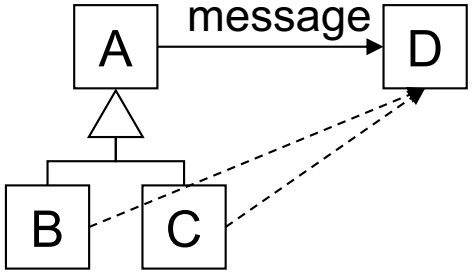
- TC1: invokes `Base::foo()` which in turns call `Base::helper()`
- TC2: invokes `Base::bar()` is invoked on the derived object, which calls `helper()` on the derived object, invoking `Derived::helper()`
- Assuming all methods have a linear control flow, do the test cases fully exercise the code of both `Base` and `Derived`?
- Traditional coverage measures (e.g., statements, control flow) would answer *yes*

Integration and Polymorphism

- Impact of polymorphism on class integration
- We assume no class is abstract

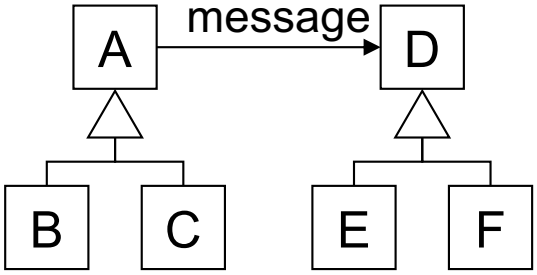


1 test set

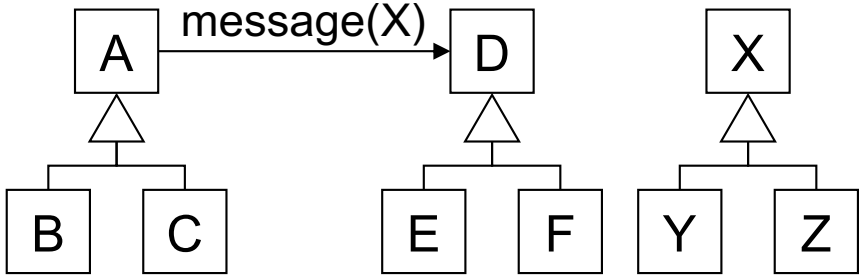


3 test sets

Inheritance brings additional testing costs But we can reuse



9 test sets



27 test sets

Hierarchical Incremental Testing

Aims at testing inheritance hierarchies

- *Step 1*: Test in a first context (parent)
  - Test all methods fully in the context of a particular class
    - base class
    - derived class for abstract base classes
- *Step 2:* Interaction coverage:
  - Any inherited method which interacts with any re-defined method should be re-tested in the context of the derived class
- Re-run all the base class test cases in the context of the derived class by which it is inherited
  - Reduces test cost for inherited methods (test drivers, including oracles, are already defined)
  - This helps check the conformance of inheritance hierarchies to the *Liskov substitution principle*

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
- For any method, *valid* contexts depend on where the method is overridden (if at all)

- 100% inheritance context coverage for all-edges means
  - Satisfying all-edges in parent
  - Satisfying all-edges in child: separate (new) context (re-work)
  - …
  - Satisfying all-edges in descendant