LIMKOKWING UNIVERSITY OF CREATIVE TECHNOLOGY

BOTSWANA

BTECH3618 • Software Testing and Reliability

CHAPTER 6

Testing Object-Oriented Software

Motivation

- Object-Oriented software testing issues
- How to test OO software
- How to test ?? software

Basic Terminology

- Class
- Instantiate
- Object
- Identity
- Has
    - Member Functions(methods)
    - Defines
    - Behavior
    - State
    - Yields

Encapsulation and Data Abstraction

- Encapsulation

  Prevents clients from knowing about or depending on the implementation of the class.

  - Data Abstraction

  Creation of a new data type from previously existing components

Inheritance Terminalogy

- Parent class and Superclass refer to the class from which another inherits
- Child class, derived class, and subclass refer to a class that inherits from one or more classes.

Inheritance

- Allows common features of many classes to be defined in one class.
  - A child class can inherit features from a parent class.
  - A child class can also enhance, restrict derived
- features, and add new features

Polymorphism

- A subclass redefines or replaces a function
- derived from its parent

– Override – same name and signature

– Overload – same name but different signature

Impacts

- What are the difference between procedural
- programs and OO programs?
  – Traditional procedural programs allow easy access to internal state information, data information is often global, program control flow is deterministic.
  – OO programs emphasized information hiding and data encapsulation, private data is not directly accessible during testing, non deterministic.

Impacts

Can we directly adopt previous unit testing

- strategies? If we can, are they sufficient?
  - Black-box testing
  - SC,BC, and other path testing strategies
  - Data-flow testing
- What is the scope of unit testing and

integration testing

Are they sufficient?

- OO faults that do not occur in traditional
- software system
  - Encapsulation faults
  - Inheritance faults
  - Polymorphism faults
- • What shall we do?

LIMKOKWING UNIVERSITY OF CREATIVE TECHNOLOGY BOTSWANA

Encapsulation

- Involves violations of encapsulation, or
- information hiding.
  - Returning a pointer to a hidden object.
  - Memory Management Faults
  - Implicit Function Faults

Example – Inheritance Faults

```
• Class Parent {
int v;
Parent() { v = 1; }
void k() {
return 1/v;
}
}
Class Child extends {
Child() { v = 0; }
void new_funtion() {
… …
}
}
```

Unit Testing Issues for OO SW

- Based class
  - Test each function with traditional testing strategies.
- Derived classes
  - Don't forget about base class methods
- They may be available as is
- They may have been replaced in an inherited class
  - Test all overloads as separate functions
  - Inline functions are still functions
  - Watch for overloaded operators

OO Coverage Criterion

- All – binding:
- Every possible binding of each object must
- be exercised at least once when the object is
- defined or used. If a statement involves
- multiple objects, then every combination of
- a possible binding needs to be tested at least
- once.

Example

```
Shape {
double() {
perimeter /= 2;
}
Class Square extends Shape {
draw(x,y,perimeter) {… …}
}
Class Circle extends Shape {
draw(x,y,radius) {… …}
}
Class MainClass {
main() {
Read(shape,perim,x,y)
if (shape == 1)
ob = new Square(..)
else ob = new Circle(…);
ob.draw()
if (perim > 10) {
ob.double();
ob.draw()
}
}
```

Class Testing

- Base class testing
  - Exhaustive testing

    10 methods, 10! = 3628800 test cases
  - Divide and conquer

    A class can be viewed as composition of a set of ***slices***: a quantum of a class with only a single data member and a set of member functions such that each member functions can manipulate the values associated with this data member.

Testing for Sequencing Constrains

- Many errors are a result of incorrect
- sequencing of operations
- Combination of the previous two
- methodologies

Example

- File class with 3 operations
  - Open (def)
  - Close (kill)
  - Write (use)
- Must open before write or close (du)
- Must not write after close (ku)
- Should write before close (dk)
- This is an outgrowth of data flow analysis

Testing Inherited Classes

- Should you retest inherited methods?
- Can you reuse superclass tests for inherited and overridden methods?
- To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?

LIMKOKWING UNIVERSITY OF CREATIVE TECHNOLOGY BOTSWANA

Testing Inherited Classes

- No member function overloading or
- overriding, no change states of the parent
- data members.
- *Parent class member functions need not to be retested*
- • New overloading, overriding function, or
- new functions that will change states of the
- parent data members.
- *No unit testing necessary for parent class member*
- *functions, but further class testing are needed.*

Example

- Class Account {
- int amount; Vector transactions;
- int calculate() {
- amount = … …
- }
- void PrintTransaction() {
- print amount;}
- }
- Class Checking extends Account {
- void PrintCheckingTransaction()
- {… …}
- }

Class Saving extends Account {
int interest;
int calculate() { amount = …}
}
For Checking class,
Account.calculate and
Account.PrintTransaction do
need to be retested.
For Saving class,
Account.PrintTransaction needs
to be retested.

Inheriting Class Test Suites

- Can you reuse superclass method tests?
  - Inherited methods Yes
  - Override methods Probably
  - Overloaded methods No
  - New methods No
- Do you need to develop new test cases?
  - Inherited methods Maybe
  - Override methods Yes
  - Overloaded methods Yes
  - New methods Yes

Example – Step (8)

- Step 0:Derive all the def, use set
- Step 1:Initialization, make all mindef(i) = {ALL}, defclear(I)={}.
- Step 2:mindef(start) = ∩ {} = {}

defclear(start) = ∪ {} – use(start) = {}

- Step 3:mindef(1)= [mindef(start) ∪def(start)] = {}

defclear(1)= [defclear(start) ∪def(start)] – use(1) = {}

- Step 4:mindef(2)= [mindef(1) ∪def(1)] ∩ [mindef(4) ∪def(4)] ∩ [mindef(5) ∪def(5)]

={A} ∩ {ALL} ∩ {ALL} = {A}
defclear(2)=[defclear(1)∪def(1)]∪[defclear(4)∪def(4)]∪[defclear(5) ∪def(5)]- use(2)
= {A} ∪ {A} ∪ {B} – {A} = {B}

- Step 5:mindef(3) = [mindef(2) ∪ def(2)] = {A}

defclear(3)= [defclear(2) ∪ def(2)] – use(3) = {B} – {A} = {B}

- Step 6:mindef(4) = [mindef(3) ∪ def(3)] = {A}

defclear(4)= [defclear(3) ∪ def(3)] – use(4) = {B} – {A, B} = {}

- Step 7:mindef(5) = [mindef(3) ∪ def(3)] = {A}

defclear(5)= [defclear(3) ∪ def(3)] – use(5) = {B} – {B} = {}

- Step 8:mindef(6) = [mindef(2) ∪ def(2)] = {A}

defclear(6)= [defclear(2) ∪ def(2)] – use(6) = {B} – {A,B} = {}
mindef(end) = [mindef(6) ∪ def(6)] = {A}
defclear(end)= [defclear(6) ∪ def(6)] – use(end) = {} – {} = {}
When trying to calculate for the second time, all sets remain the same, stop.

Identify Faults and Data Flow Anomalies

d- : defclear(exitnode) <> {}

dd: defclear(i) ∩ def(i) <> {}

-u: use(i) - mindef(i) <> {}

Identify Faults and data Flow Anomalies

- NODE 1 dd: defclear(1) ∩ def(1) = {} ∩ {A} = {} √

-u: use(i) - mindef(i) = {} – {} = {} √

- NODE 2 dd: defclear(2) ∩ def(2) = {A} ∩ {} = {} √

-u: use(i) - mindef(i) = {A} – {A} = {} √

- NODE 3 dd: defclear(3) ∩ def(3) = {} ∩ {} = {} √

-u: use(i) - mindef(i) = {A} – {A} = {} √

- NODE 4 dd: defclear(4) ∩ def(4) = {} ∩ {A} = {} √

-u: use(i) - mindef(i) = {A,B} – {A} = {B} X -u type Fault

- NODE 5 dd: defclear(5) ∩ def(5) = {A} ∩ {B} = {} √

-u: use(i) - mindef(i) = {B} – {A} = {B} X -u type Fault

- NODE 6 dd: defclear(6) ∩ def(6) = {} ∩ {} = {} √

-u: use(i) - mindef(i) = {A,B} – {A} = {B} X -u type Fault

- EndNode d-: delclear(end) = {} √

Discussion

- Unachievable d-u pair
- Array, pointer
- Inter – procedure data-flow analysis