Testing of Object-Oriented Applications

Meenakshi D'Souza

International Institute of Information Technology
Bangalore.

Outline

We (will) cover the following topics in testing of object-oriented software.

- Overview of features of object-oriented software.
    - Some of them already done.
- Mutation testing for OO integration: Already done.
- Errors in object-oriented software.
- Testing of object-oriented software.
    - Yo-yo graph.
    - OO-call coverage criteria.

Note: Unit testing of object-oriented applications can be done using any relevant coverage criteria that is applicable.

Object-oriented features

Testing of OO-software concentrates on faults that occur due to the following features:

- Abstraction: Focusses on how the components are *connected*.
- Inheritance and polymorphism: Defines new connections that are different from other languages.

Hence, testing focusses on how we connect the software components (integration testing) instead of unit testing.

Abstraction in OO programs

OO languages use

- Classes to represent data abstraction.
- In addition inheritance, polymorphism and dynamic binding support abstraction.
- New type created by inheritance are descendents of the existing type.
- Extension and refinement:
    - A class extends its parent class if it introduces a new method name and does not override any methods in an ancestor class.
    - A class refines the parent class if it provides new behaviour not present in the overridden method, does not call the overridden method and its behaviour is semantically consistent with that of the overridden method.

Inheritance in OO programs

Two types of inheritance are used:

- Sub-type inheritance: Famously called the substitution principle. If class `B` uses sub-type inheritance from class `A`, it is possible to freely substitute any instance of `B` for `A` and still satisfy an arbitrary client of `A`. `B` has an *is-a* relationship with `A`.
- Sub-class inheritance: This allows descendant classes to reuse methods and variables from ancestor classes without necessarily ensuring that instances of the descendants meet the specifications of the anscestor type.

Polymorphic methods

- If class `B` inherits from class `A` and both `A` and `B` define a method `m()` then, `m()` is called a polymorphic method.
- If an object `x` is *declared* to be of type `A` then during execution `x` can have either the *actual* type `A` or `B`.
  - The version that is executed depends on the *current* actual type of the object.
- The collection of methods that can be executed is called the polymorphic call set. For this e.g., it is {`A:m()`, `B:m()`}.

Four levels of class testing

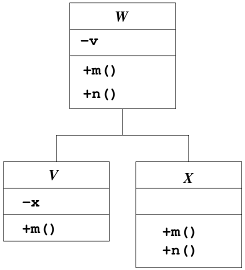
With respect to testing that is unique to OO software, a class is usually regarded as the basic unit of testing.
There are four levels of testing classes.

- Intra-method testing: Tests are constructed for individual methods (traditional unit testing).
- Inter-method testing: Multiple methods within a class are tested in concert (traditional module testing).
- Intra-class testing: Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- Inter-class testing: More than one class is tested at the same time, usually to see how they interact (kind of integration testing).

Visualizing OO interactions

- We assume that a class encapsulates state information in a collection of *state variables*. The behaviors of a class are implemented by methods that use the state variables.
- The interactions between the various classes and methods within/outside a class that occur in the presence of inheritance, polymorphisn and dynamic binding are complex and difficult to *visualize*.
- We will go through some examples to illustrate the issues.

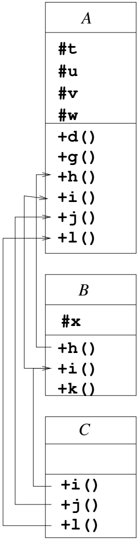
Class hierarchy and method overriding: Example



```
1.  void f(boolean b)
2.  {
3.    W o;
4.    ...
5.    if(b)
6.      o = new V();
7.    else
8.      o = new W();
9.    ...
10.   o.m();
11. }
```

- V and X extend W, V overrides method `m()` and X overrides methods `m()` and `n()`.
- $-$: attributes are private, $+$: attributes are non-private.
- The declared type of o is W, but at line 10, the actual type can be either V or W.
- Since V overrides `m()`, which version of `m()` is executed depends on the input flag to the method.

Data flow anamolies with polymorphism: Example



Method	Defs	Uses
A::h	{A::u,A::w}	
A::i		{A::u}
A::j	{A::v}	{A::w}
A::l		{A::v}
B::h	{B::x}	
B::i		{B::x}
C::i	{C::y}	
C::j		{C::y}
C::l		{A::v}

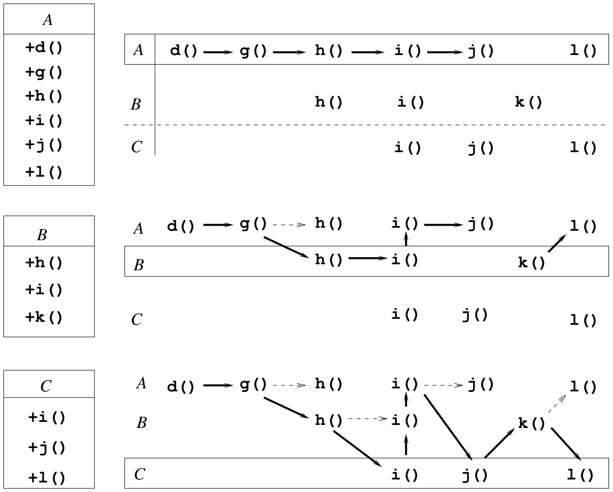
Data flow anamolies with polymorphism: Example, contd.

- The root class *A* has four state variables and six methods, two descendents *B* and *C*. Methods of *A* are called in sequence.
- The state variables of *A* are protected, i.e., available to *B* and *C*.
- *B* declares one state variable and three methods, *C* declares three methods.
- `B::h()` overrides `A::h()`, `B::i()` overrides `A::i()`, `C::i()` overrides `B::i()`, `C::j()` overrides `A::j()` and `C::l()` overrides `A::l()`.
- Data flow anamoly: Suppose an instance of *B* is bound to an object *o* and a call to `d()` is made. *B*'s version of `h` and `i` are called, `A::u` and `A::w` are not given values, and thus the call to `A::j` can result in a data flow anamoly.

Visualizing polymorphism with the Yo-Yo graph

- Understanding which version of a method *will* be executed and which versions *can* be executed is very difficult.
- Execution can *bounce* up and down among levels of inheritance.
- The yo-yo graph is defined on an inheritance hierarchy.
    - It has a root and descendants.
    - Nodes are methods: new, inherited and overridden methods for each descendent.
    - Edges are method calls as given in the source: directed edge is from caller to callee.
- In addition,
    - Each class is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type of that level. These are depicted by *bold arrows*.
    - *Dashed arrows* are calls that cannot be made due to overriding.

Yo-Yo graph: Example



Yo-Yo graph: Example

- *A*'s implementation: `d()` calls `g()`, `g()` calls `h()`, `h()` calls `i()` and `i()` calls `j()`.
- *B*'s implementation: `h()` calls `i()`, `i()` calls its parent's (*A*'s) version of `i()` and `k()` calls `l()`.
- *C*'s implementation: `i()` calls its parent's (*B*'s) version of `i()` and `j()` calls `k()`.

Yo-Yo graph: Example

- Top level: A call is made to method `d()` through an object of actual type A.
  - This sequence of calls is simple and straightforward.
- Second level: Object is of actual type B. When `g()` calls `h()`, the version of `h()` defined in B is executed. The control then continues to `i()` in B, `i()` in A and then to `j()` in A.
- Third level: Object is of actual type C. Control proceeds from `g()` in A, to `h()` in B to `i()` in C, then, to `i()` in A and B etc— exhibiting a *yo-yo* effect.

Potential for faults in OO programs

- Complexity is relocated to the connections among components.
- Less *static determinism*; many faults can now only be detected at runtime.
- Inheritance and Polymorphism yield *vertical* and *dynamic* integration.
- Aggregation and use relationships are more complex.
- Designers do not carefully consider visibility of data and methods.

OO faults

- Faults related to inheritance, polymorphism, constructors and visibility are listed.
- Assumption: Anamoly/fault is manifested through polymorphism in a context that uses an instance of the ancestor. i.e., instances of descendant classes can be substituted for instances of the ancestor.
- Faults described are *language independent*.

Categories of faults/anomalies

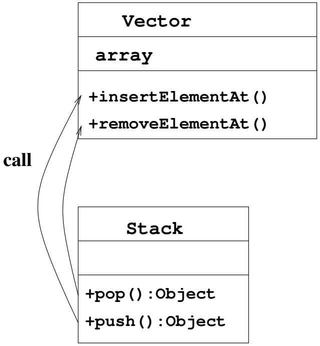
Acronym	**Fault/Anomaly**
ITU	Inconsistent type use
SDA	State definition anomaly
SDIH	State definition inconsistency
SDI	State defined incorrectly
IISD	Indirect inconsistent state definition
ACB1	Anomalous construction behaviour (1)
ACB2	Anomalous construction behaviour (2)
IC	Incomplete construction
SVA	State visibility anomaly

Inconsistent type use fault: ITU

- A descendent class does not override any inherited method—hence, no polymorphic behaviour.
- Class C extends class T, and C adds new methods (extension).
- An object is used “as a C”, then as a T, then as a C.
- Methods in T can put object in state that is inconsistent for C.

Inconsistent type use fault: Example

- Class `Vector` is a sequential data structure that supports direct access to its elements.
- Class `Stack` uses methods inherited from `Vector` to implement the stack.



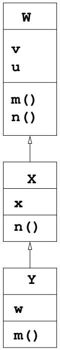
```
s.push("string1");
s.push("string2");
s.push("string3");
dumb(s);
pop();
pop();
pop(); //Stack is empty

void dumb(Vector v)
{
  v.removeElementAt(v.size()-1);
}
```

State definition anomaly: SDA

- The state interactions of a descendant are not consistent with those of its ancestor.
- The refining methods fail to define some variables and hence required definitions might not be available.
- For e.g., let us say that a class `X` extends class `W` and `X` overrides some methods of `W`.
- The overriding methods in `X` fail to define some variables that the overridden methods in `W` defined.

State definition anomaly: Example



W::m() defines v and W::n() uses v

X::n() uses v

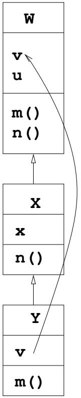
Y::m() does not define v

For an object of actual type Y, a data flow anomaly exists and results in a fault if m() is called, then n().

State definition inconsistency anomaly (SDIH)

- A local variable is introduced to a class definition and the name of the variable is the same as an inherited variable `v`.
    - The inherited variable is hidden from the scope of the descendent (unless explicitly qualified, as in `super.v`).
- A reference to `v` will refer to the descendent's `v`.
- Anomaly exists if a method that normally defines the inherited `v` is overridden in a descendant when an inherited state variable is hidden by a local definition.

State definition inconsistency: Example



`Y overrides W's v.`

`Y::m defines Y::v.`

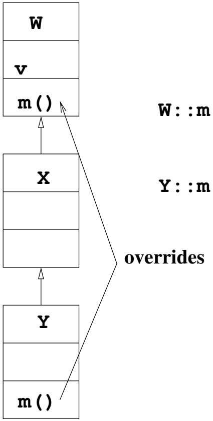
`X::n uses v, getting W's version of v.`

For an object of actual type `Y`, a data flow anomaly exists and results in a fault if `m()` is called, then `n()`.

State visibility anomaly (SVA)

- Consider a class `W`, which is an ancestor `X` and Y: `X` extends `W` and `Y` extends `X`.
- `W` declares a private variable `v`, `v` is defined by `W::m`.
- `Y` overrides `m()`, and calls `W::m()` to define `v`.
- This causes a data flow anomaly as `v` is private for `W`.

State visibility anomaly: Example



`W::m() defines v`

`Y::m() cannot call W::m()`

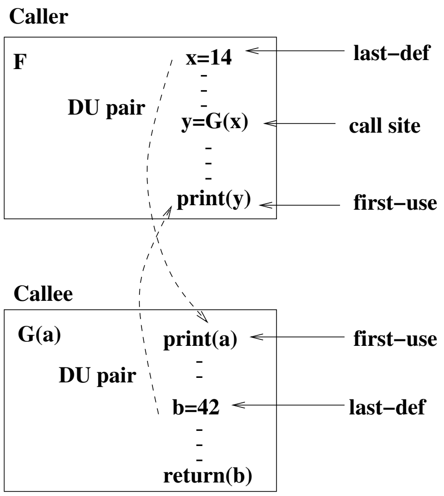
Other faults

- State defined incorrectly: An overriding method defines the same state variable that the overridden method defines. In the absence of identical computations by the two methods, this could result in a behavior anomaly.
- Indirect inconsistent state definition fault: A descendent adds an extension method that defines an inherited state variable.

Inter-procedural DU pairs: From an earlier lecture

- Coupling variables: Variables defined in one unit and used in another unit.
- Last-def: The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module.
    - Can be from caller to callee (parameter or shared variable) or from callee to caller (return value).
- First-use: The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.
- Coupling du-path: A du-path from a last definition to a first use.

Coupling du-pairs example: From an earlier lecture



Coupling Sequences

- Pairs of method calls within body of method under test.
  - Made through a common *instance context*.
  - With respect to a set of state variables that are commonly referenced by both methods.
  - Consists of at least one coupling path between the two method calls with respect to a particular state variable.
- Represent potential *state space interactions* between the called methods with respect to calling method.
- Used to identify points of integration and testing requirements.

Some definitions

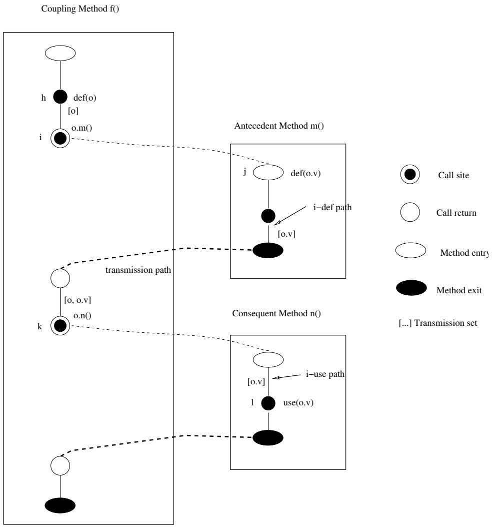
In the definitions below, `o` is an identifier whose type is a *reference to an instance* of an object pointing to a memory location that contains an instance (value) of some type.

- A reference `o` can refer to instances whose actual instantiated types are either the base type of `o` or a descendent of `o`'s type.
  - For e.g., in Java, `A o = new B();`, `o`'s base or declared type is `A` and its instantiated or actual type is `B`.
- `o` is considered to be defined when one of the state variables `v` of `o` is defined.

Polymorphic call set

- Definitions and uses in OO-applications for coupling variables can be indirect.
    - Indirect definitions and uses occur when a method defines/references a particular value `v`.
- In the presence of indirect definitions and uses, we have to consider all the methods that can potentially execute.
- Polymorphic call set or Satisfying set: Set of methods that can potentially execute as result of a method call through a particular instance context.

Typical control flow coupling sequence



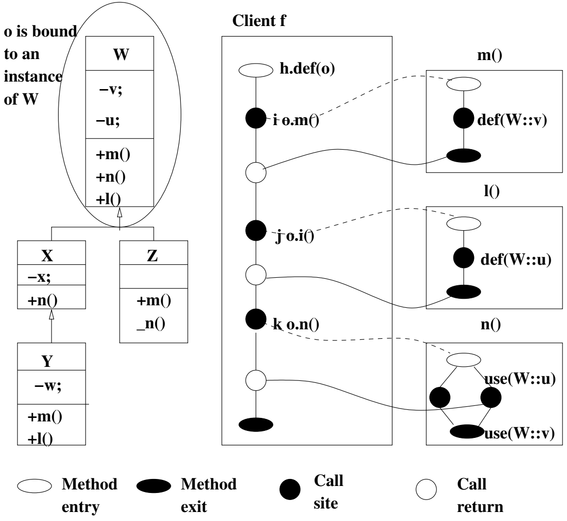
More definitions

- Coupling sequence: A pair of nodes that call two methods in sequence. The first method defines a variable, the second method uses it.
- The calling method is the coupling method `f()`, it calls `m()`, the antecedent method, to define a variable, and `n()`, the consequent method, to use the variable.
- Transmission set: Variables for which a path is def-clear.
- In the figure, the path from `h` to `i` to `k` and finally to `l` forms a transmission path with respect to `o.v`. The object `o` is the context variable.

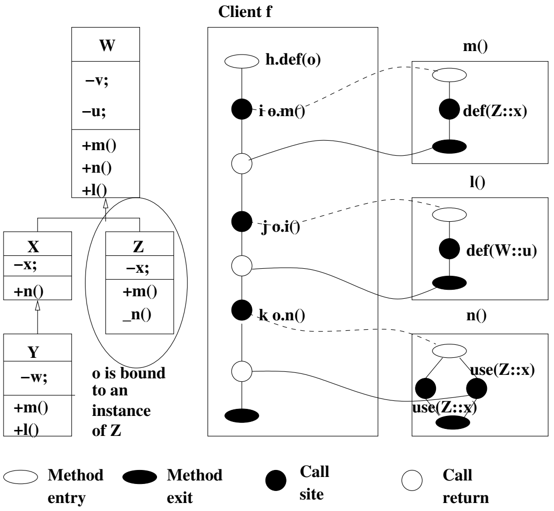
Definitions, contd.

- $S_{j,k}$: Coupling sequence.
- $\Theta^{t}_{S_{j,k}}$: Coupling set of $S_{j,k}$ is the intersection of the variables defined by `m()`, used by `n()`, through the instance context provided by a context variable `o` that is bound to an instance of `t`.
- As usual, the versions of `m()` and `n()` that run are determined by the actual type `t` of the instance bound to `o`.

Coupling sequence: Example



Coupling sequence: Example



OO coupling: Testing goals

- We want to test how a method can interact with instance bound to object o.
    - Interactions occur through the coupling sequences.
- Need to consider the set of interactions that can occur.
    - What types can be bound to o?
    - Which methods can actually execute? (polymorphic call sets).
- Test all couplings with all type bindings possible.

All-Coupling Sequences

- All-Coupling-Sequences (ACS): For every coupling sequence S_j in $f()$, there is at least one test case t such that there is a coupling path induced by $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- At least one coupling path must be executed.
- This does not consider inheritance and polymorphism.

All Poly Classes

- All-Poly-Classes (APC): For every coupling sequence $S_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $S_{j,k}$, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the set of coupling paths of $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- Includes instance contexts of calls.
- At least one test for every type the object can bind to.
- Test with every possible type substitution.

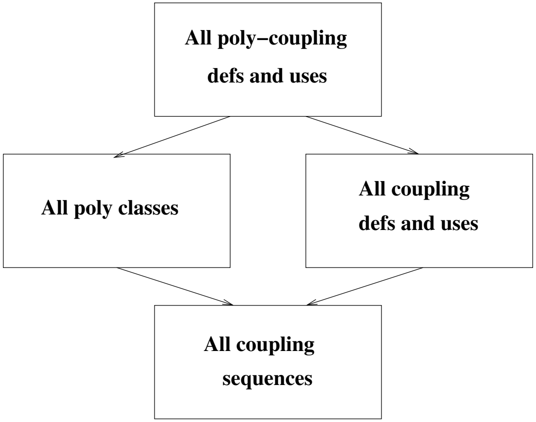
All coupling defs-uses

- All-Coupling-Defs-Uses (ACDU): For every coupling variable v in each coupling $S_{j,k}$ of t, there is a coupling path induced by $S_{j,k}$ such that p is a sub-path of the execution trace of $f(t)$ for at least one test case t.
- Every last definition of a coupling variable reaches every first use.
- Does not consider inheritance and polymorphism.

All poly coupling defs and uses

- All-Poly-Coupling-Defs-and-Uses (APDU): For every coupling sequence $S_{j,k}$ in $f()$, for every class in the family of types defined by the context of $S_{j,k}$, for every coupling variable v of $S_{j,k}$, for every node m that has a last definition of v and every node n that has a first-use of v, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the coupling paths of $S_{j,k}$ that is a sub-path of the trace of $f()$.
- Every last definition of a coupling variable reaches every first use for every type binding.
- Combines previous criteria.
- Handles inheritance and polymorphism.
- Takes definitions and uses of variables into account.

OO coupling coverage criteria subsumption



Conclusions

We presented the following about OO applications testing.

- A model for understanding and analyzing faults that occur as a result of inheritance and polymorphism.
  - Yo-yo graph
  - Defs and Uses of state variables
  - Polymorphic call set
- Technique for identifying data flow anomalies in class hierarchies.
- A fault model and specific faults that are common in OO software.
- Specific test criteria for detecting such faults.