A Fault Model for Subtype Inheritance and Polymorphism

Jeff Offutt
Information & Software Engineering
George Mason University
Fairfax, VA USA
www.ise.gmu.edu/faculty/ofut/
ofut@ise.gmu.edu

Joint research with: Roger T. Alexander, Colorado State University
Ye Wu, George Mason University
Quansheng Xiao , George Mason University
Chuck Hutchinson , George Mason University

Supported by NSF and NIST.

Inheritance

Allows common features of many classes to be defined in one class

A derived class has everything its parent has, plus it can:

- **<u>Enhance</u> derived features (overriding)**
- **<u>Restrict</u> derived features**
- **<u>Add</u> new features (extension)**

Inheritance (2)

<u>Declared type</u>: The type given when an object reference is declared
Clock w1; // declared type Clock

<u>Actual type</u>: The type of the current object
w1 = new Watch(); // actual type Watch

In Java, the method that is executed is the <u>lowest</u> version of the method defined between the actual and root types in the inheritance hierarchy

A
↑
B
↑
C

Polymorphism

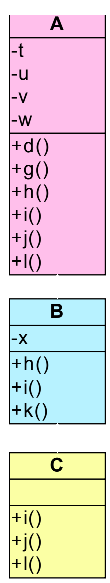
- **The same variable can have different types depending on the program execution**
- **If *B* inherits from *A*, then an object of type *B* can be used when an object of type *A* is expected**
- **If both *A* and *B* define the same method *m()***
  (*B* <u>overrides</u> *A*), then the same statement will sometimes call *A*'s version of *M*, and sometimes *B*'s version

Subtype and Subclass Inheritance

- **<u>Subtype Inheritance</u>: If B inherits from A, any object of type B can be <u>substituted</u> for an object of type A**
  - A *laptop* “is a” special type of *computer*
  - Called *substitutability*
- **<u>Subclass Inheritance</u>: Objects of type B may <u>not</u> be substituted for objects of type A**
  - Objects of B may not be “type compatible”

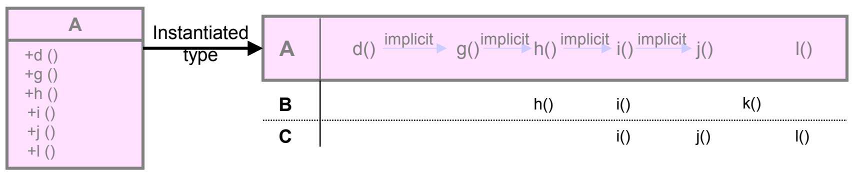
This talk assumes subtype inheritance, subclass inheritance will be addressed later.

Example



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

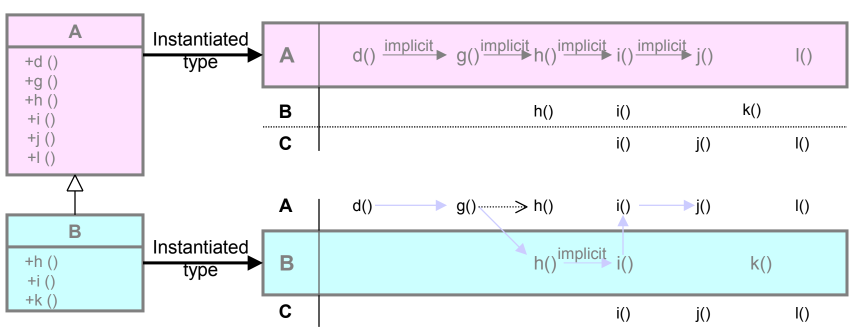
Polymorphism Headaches (Yo-Yo)



Object is of type A
A::d ()

7

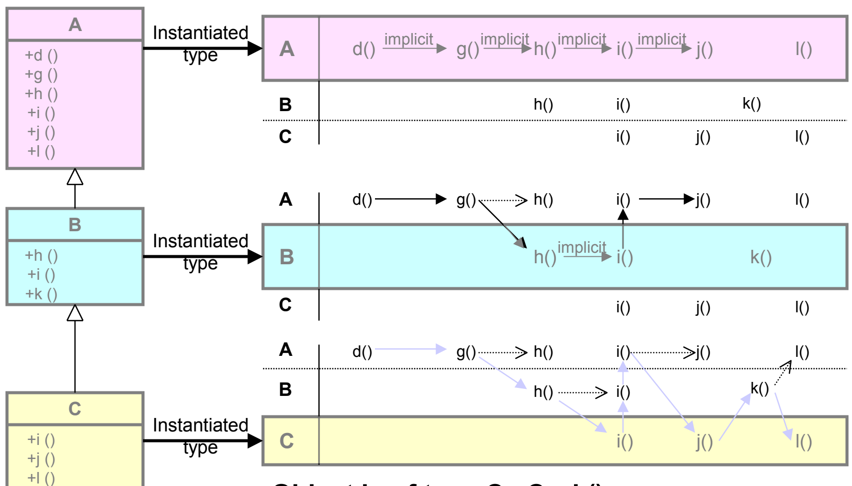
Polymorphism Headaches (Yo-Yo)



Object is of type B
B::d ()

8

Polymorphism Headaches (Yo-Yo)



Object is of type C, C::d ()

Potential for Faults in OO Programs

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

A Fault-Failure Model for OO Programs

1. **Reachability: The faulty location must be reached**
   1. class *D* extends *T*
   2. *D* has a method *m()* that overrides *T*'s *m()*
   3. There is an object *o* of type *T* (*T o;*)
   4. The actual type of *o* is *D* (*o = new D();*)
   5. *m()* is called in the context of *o* (*o.m();*)
2. **Infection: Program state must be incorrect**
   – *T::m()* and *D::m()* modify different variables; both are available to T (*T::m()* defines *x* and *D::m()* defines *y*)
3. **Propagation: Program output must be incorrect**
   – A path exists from the definition of the variable in *T::m()* or *D::m()* to a use
   – The use of the variable affects the output state of the program

Object-oriented Faults

- **Only consider faults that arise as a direct result of OO language features:**
  – inheritance
  – polymorphism
  – constructors
  – visibility
- **Language independent (as much as possible)**

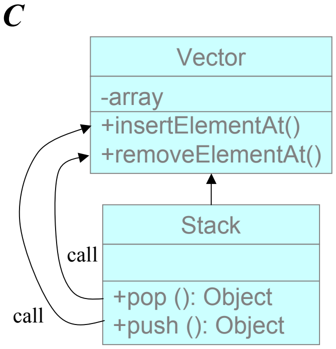
OO Faults and Anomalies

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 Behavior (1)
ACB2	Anomalous Construction Behavior (2)
IC	Incomplete Construction
SVA	State Visibility Anomaly

 13

Inconsistent Type Use (ITU)

- **No overriding (no polymorphism)**
- ***C* extends *T*, and *C* adds new methods (extension)**
- **An object is used "*as a T*", then as a *C*, then as a *T***
- **Methods in *T* can put object in a state that is <u>inconsistent</u> for *C***



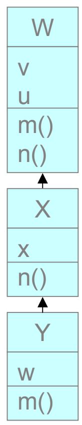

```
s.push ("steffi");
s.push ("joyce");
s.push ("chelsea");
dumb (s);
s.pop();
s.pop();
s.pop();   // Stack is empty!

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

 14

State Definition Anomaly (SDA)

- ***C*** **extends** ***T*****, and** ***C*** **<u>overrides</u> some methods**
- **The overriding methods in** ***C*** **<u>fail to define</u> some variables that the overridden methods in** ***T*** **defined**



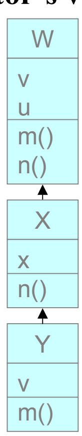
- *W::m ()* defines *v* and *W::n()* uses *v*
- *X::n ()* uses *v*
- *Y::m ()* does <u>not</u> define *v*

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

15

State Definition Inconsistency (SDIH)

- **Overriding a variable, possibly accidentally**
- **If the descendant's version of the variable is defined, the ancestor's version may not**



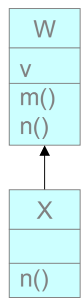
- *Y* overrides *W*'s version of *v*
- *Y::m()* defines *Y::v*
- *X::n()* uses *v*

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

16

State Defined Incorrectly (SDI)

- **Overriding a method *m()* that defines a variable *v***
- **The overriding method may define *v* incorrectly**

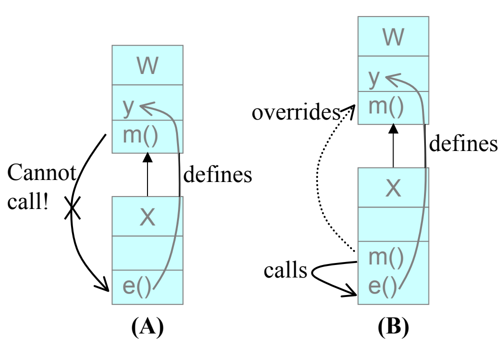


- *W::n()* defines *v*
- *X::n()* also defines v, but incorrectly

For an object of type *X*, a behavioral error exists occurs if *W::m()* uses v and assumes it has a value as given in W::n()

Indirect Inconsistent State Definition (IISD)

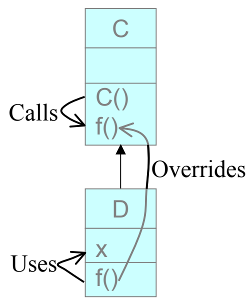
- **A method is added that defines an inherited state variable**
- **Method puts the ancestor in an inconsistent state**



- *W::m()* cannot call *X::e()*
- *X::m()* calls *X::e()*, which defines *W::y* incorrectly …

Anomalous Construction Behavior (ACB1)

- **Constructor of *C* calls a method *f()***
- **Child of *C*, *D*, overrides *f()***
- ***D::f()* uses variables that should be defined by *D*'s constructor, but are not**

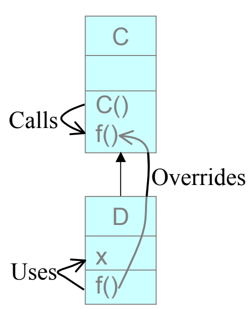


When an object of type *D* is constructed, *C()* is run before *D()*.

When *C()* calls *D::f()*, *x* is used, but has <u>not</u> yet been given a value!

Anomalous Construction Behavior (ACB2)

- **Constructor of *C* calls a method *f()***
- **Child of *C*, *D*, overrides *f()***
- ***D::f()* uses variables that are defined by *C*'s constructor**



The author of *C* cannot know anything about *D::f()*

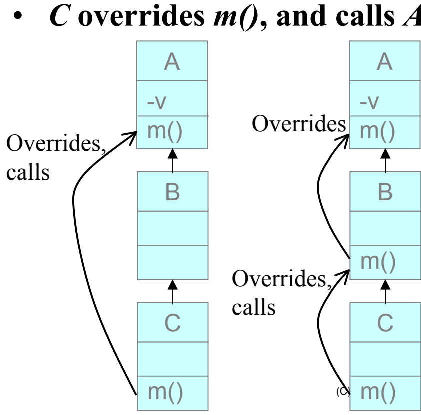
If *D::f()* uses a variable *x* that *C()* defines, <u>and</u> the definition is after the call to *f()*, *x* has no value in *D::f()*

Incomplete Construction (IC)

- **Constructors should give all variables "reasonable" values**
- **In C++, the variables have no values by default!**
- **Two possible faults:**
  1. Wrong value assigned to a variable
  2. No value assigned to a variable (more dangerous in C++)

State Visibility Anomaly (SVA)

- **A private variable *v* is defined in ancestor *A*, and *v* is defined by *A::m()***
- ***B* extends *A* and *C* extends *B***
- ***C* overrides *m()*, and calls *A::m()* to define *v***



B::m() is added later

C:m() can no longer call *A::m()*!

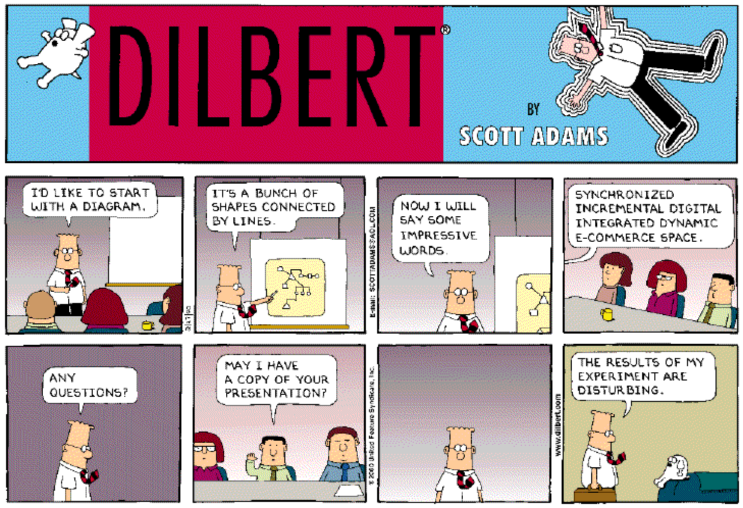
Conclusions

- **A model for understanding and analyzing faults that occur as a result of inheritance and polymorphism**
- **Technique for identifying data flow anomalies in class hierarchies**
- **Guidelines for proper use of inheritance, polymorphism, and constructors**

Future Work

- **How often do these faults occur in practice**
- **Fault injection techniques for OO experimentation**
- **Guidelines for developing safe inheritance hierarchies**
- **Guidelines or standards for safe use of polymorphism**
  - It is unsafe for constructors to call polymorphic methods
- **Mutation operators for Java**

Summary of Talk



www.ise.gmu.edu/faculty/ofut/

