Analysis of Faults from 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 Alexander, Colorado State University
Ye Wu, George Mason University
Yu-Seung Ma, KAIST, Korea
Yong-Rae Kwon, KAIST, Korea

Supported by NSF and NIST.

From OO Faults to Mutation

1. **Yo-Yo graph: Models polymorphism and dynamic binding**
2. **Criteria for testing polymorphism: Based on couplings among classes**
3. **Inheritance and polymorphism faults**
4. **Java mutation operators for class testing**

© Copyright 1999-2002. All Rights Reserved.

Inheritance

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

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

- **Enhance derived features (overriding)**
- **Restrict derived features**
- **Add new features (extension)**

© Copyright 1999-2002. All Rights Reserved.

Inheritance (2)

Declared type: The type given when an object reference is declared

```
Clock w1; // declared type Clock
```

Actual type: The type of the current object

```
w1 = new Watch(); // actual type Watch
```

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

© Copyright 1999-2002. All Rights Reserved.

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* overrides *A*), then the same statement will sometimes call *A*'s version of *M*, and sometimes *B*'s version

© Copyright 1999-2002. All Rights Reserved.

Subtype and Subclass Inheritance

- **Subtype Inheritance: If B inherits from A, any object of type B can be substituted for an object of type A**
  - A *laptop* "is a" special type of *computer*
  - Called *substitutability*
- **Subclass Inheritance: Objects of type B may not 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.

© Copyright 1999-2002. All Rights Reserved.

Example

A

-t
-u
-v
-w

+d()
+g()
+h()
+i()
+j()
+l()

B

-x

+h()
+i()
+k()

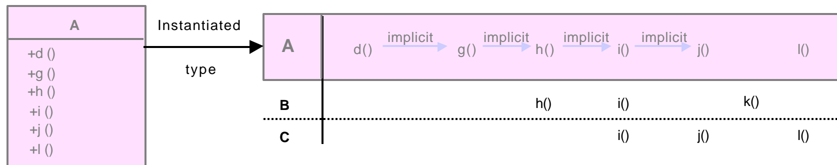
C

+i()
+j()
+l()

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}

© Copyright 1999-2002. All Rights Reserved.

Polymorphism Headaches (Yo-Yo)

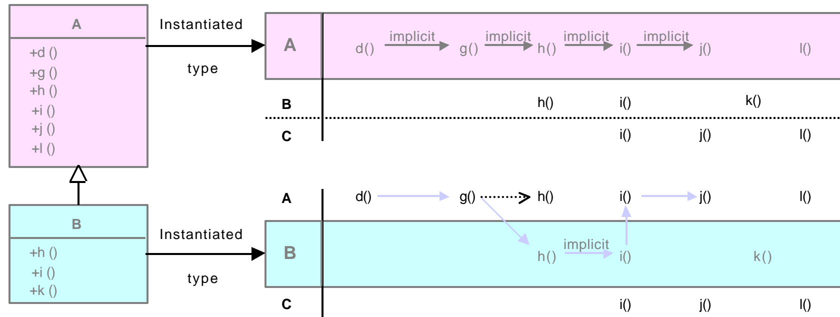


Object is of type A

A::d ()

© Copyright 1999-2002. All Rights Reserved.

Polymorphism Headaches (Yo-Yo)

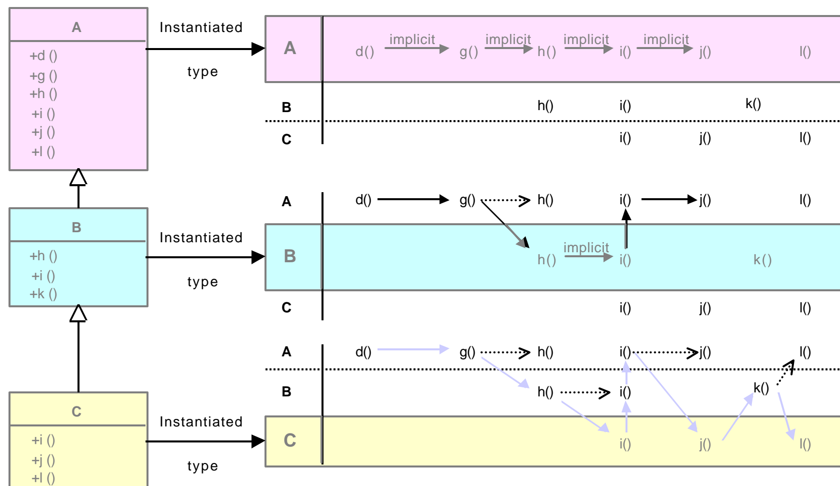


Object is of type B

B::d ()

© Copyright 1999-2002. All Rights Reserved.

Polymorphism Headaches (Yo-Yo)



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

© Copyright 1999-2002. All Rights Reserved.

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

© Copyright 1999-2002. All Rights Reserved.

Testing OO Software

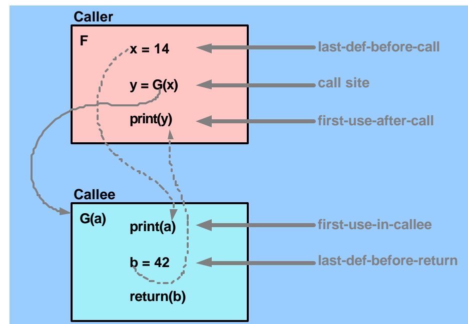
1) Intra-method testing: Testing individual methods within classes
2) Inter-method testing: Multiple methods within a class are tested in concert
3) Intra-class testing: Testing a single class, usually using sequences of calls to methods within the class
4) Inter-class testing: More than one class is tested at the same time (integration)

We are focusing on inter-class testing for polymorphism problems

© Copyright 1999-2002. All Rights Reserved.

Coupling-Based Testing

- **Derived from previous work for non-procedural programs.**
- **Based on insight that integration occurs through couplings among software artifacts.**



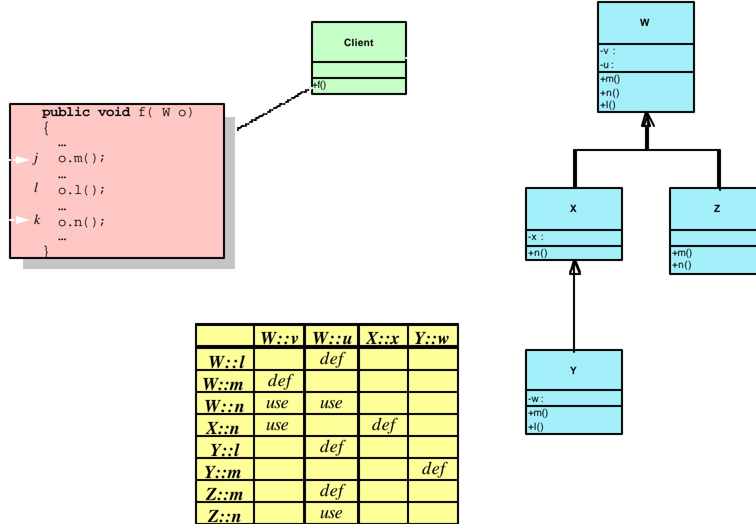
© Copyright 1999-2002. All Rights Reserved.

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**

© Copyright 1999-2002. All Rights Reserved.

Example: A Pair of Call Sites



	W::v	*W::u*	*X::x*	*Y::w*
W::l		*def*		
W::m	*def*			
W::n	*use*	*use*		
X::n	*use*		*def*	
Y::l		*def*		
Y::m				*def*
Z::m		*def*		
Z::n		*use*		

© Copyright 1999-2002. All Rights Reserved.

Polymorphic Call Set

Set of methods that can potentially execute as result of a method call through a particular instance context

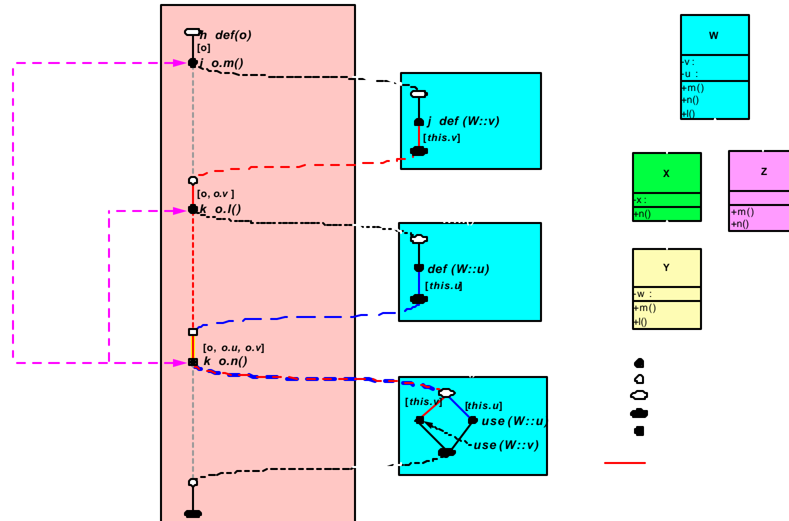
***pcs(o.m)* = {*W::m*,*Y::m*,*W::m*}**

```
public void f( W o)
{
   …
j  o.m();
   …
l  o.l();
   …
k  o.n();
   …
}
```

© Copyright 1999-2002. All Rights Reserved.

Example Coupling Sequence

o is bound to instance of *W*



© Copyright 1999-2002. All Rights Reserved. 17

Testing Requirements

- **Want to test the ways in which *f* 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?

© Copyright 1999-2002. All Rights Reserved. 18

All-Coupling-Sequences

Every <u>coupling sequence</u> in every method must be tested at least once

- **At least one coupling path must be executed**
- **Does not consider inheritance and polymorphism**

© Copyright 1999-2002. All Rights Reserved.

All-Poly-Coupling-Defs-and-Uses

Every coupling path must be executed for every member of the type family defined by the context of a coupling sequence <u>and</u> for every coupling variable in the sequence

- **Handles <u>inheritance</u> and <u>polymorphism</u>**
- **Takes <u>definitions</u> and <u>uses</u> of variables into account**

© Copyright 1999-2002. All Rights Reserved.

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)**

© Copyright 1999-2002. All Rights Reserved.

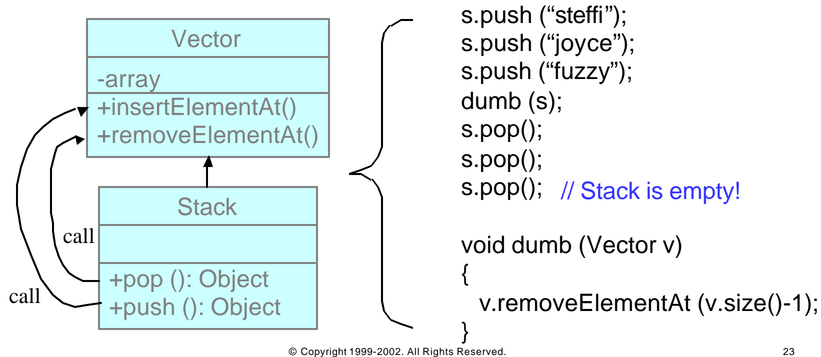
OO Faults and Anomalies

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

© Copyright 1999-2002. All Rights Reserved.

Inconsistent Type Use (ITU)

- **No overriding (no polymorphism)**
- ***C* extends *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 <u>inconsistent</u> for *C***



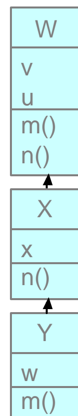

```
s.push ("steffi");
s.push ("joyce");
s.push ("fuzzy");
dumb (s);
s.pop();
s.pop();
s.pop();   // Stack is empty!

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

© Copyright 1999-2002. All Rights Reserved.

State Definition Anomaly (SDA)

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



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

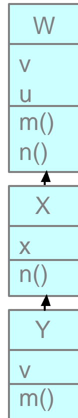
For an object of type *Y*, a data flow anomaly exists and results in a fault if *m()* is called, then *n()*

© Copyright 1999-2002. All Rights Reserved.

•

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 be**



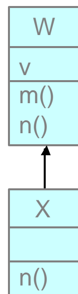
- *Y* overrides *W*'s version of *v*
- *Y::m()* defines *Y::v*
- *X::n()* uses *v*

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

© Copyright 1999-2002. All Rights Reserved.

State Defined Incorrectly (SDI)

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



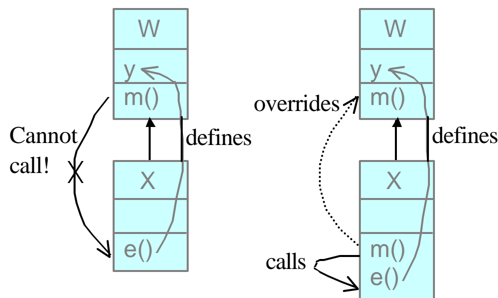
- *W::n()* defines *v*
- *X::n()* also defines v, but incorrectly

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

© Copyright 1999-2002. All Rights Reserved.

Indirect Inconsistent State Definition (IISD)

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

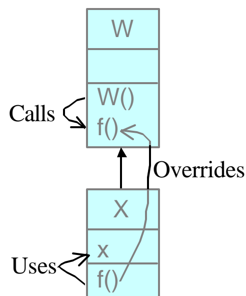


- *W::m()* cannot call *X::e()*
- *X::m()* calls *X::e()*, which defines *W::y* incorrectly …

© Copyright 1999-2002. All Rights Reserved. 27

Anomalous Construction Behavior (ACB1)

- **Constructor of *W* calls a method *f()***
- **A child of *W*, *X*, overrides *f()***
- ***X::f()* uses variables that should be defined by *X*'s constructor**



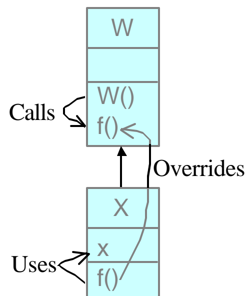
When an object of type *X* is constructed, *W()* is run before *X()*.

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

© Copyright 1999-2002. All Rights Reserved. 28

Anomalous Construction Behavior (ACB2)

- **Constructor of *W* calls a method *f()***
- **A child of *W*, *X*, overrides *f()***
- ***X::f()* uses variables that are defined by *W*'s constructor**



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

If *X::f()* uses a variable *x* that *W()* defines, and the definition is after the call to *f()*, *x* has no value in *X::f()*

© Copyright 1999-2002. All Rights Reserved. 29

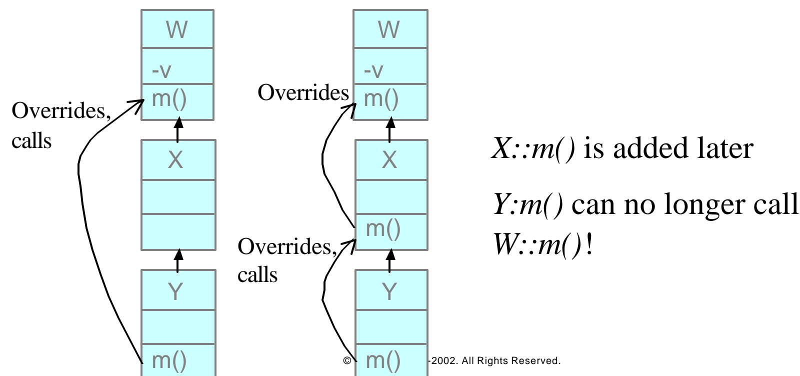
Incomplete Construction (IC)

- **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++)

© Copyright 1999-2002. All Rights Reserved. 30

State Visibility Anomaly (SVA)

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



X::m() is added later

Y:m() can no longer call *W::m()*!

© Copyright 1999-2002. All Rights Reserved.

Java Mutation Operators

- **Develop inter-class level tests by <u>mutation</u> testing**
- **Mutate various inheritance and polymorphic features**
- **Previous Java mutation operators were <u>incomplete</u>**

© Copyright 1999-2002. All Rights Reserved.

Previous Java Mutation Operators

- Kim, Clark, and McDermid: 13 operators
  - **Access levels**
  - **Variable and method overriding**
  - **Method overloading**
  - **Actual type declaration**
  - **Constructor modification**
  - **Static keyword modification**
- Chevalley and Thevenod-Fosse: 3 additional operators
  - **Reference assignments**
  - **Method call replacements**

© Copyright 1999-2002. All Rights Reserved.

Faults Not Covered

- **State visibility anomaly**
- **Anomalous constructor behavior**
- **Incomplete construction**
- ***super* keyword misuse**
- ***this* keyword misuse**
- **Incorrect overloading methods implementation**
- ***static* modifier misuse**

© Copyright 1999-2002. All Rights Reserved.

Categories of Mutation Operators

1. **Access Control**
2. **Inheritance**
3. **Polymorphism**
4. **Overloading**
5. **Java-Specific Features**
6. **Common Programming Mistakes**

© Copyright 1999-2002. All Rights Reserved.

Mutation Operators
1) Access Control

Access control is one of the most common source of mistakes in OO software. AMC changes access modifiers to force testers to find problems.

- AMC: Access modifier change

© Copyright 1999-2002. All Rights Reserved.

Java Mutation Operators
2) Inheritance

Programmers have difficulty using inheritance correctly. The inheritance operators check for improper use of inheritance.

- IHD: Hiding variable deletion
- IHI: Hiding variable insertion
- IOD: Overriding method deletion
- IOP: Overridden method calling position change
- IOR: Overridden method rename
- ISK: *super* keyword deletion
- IPC: Explicit call of a parent's constructor deletion

© Copyright 1999-2002. All Rights Reserved.

Java Mutation Operators
3) Polymorphism

Although a powerful abstraction mechanism, polymorphism and dynamic binding are hard to use correctly.

- IPC: Explicit call of a parent's constructor deletion
- PNC: *new* method call with child class type
- PMD: Instance variable declaration with parent class type
- PPD: Parameter variable declaration with child class type
- PRV: Reference assignment with other compatible type

© Copyright 1999-2002. All Rights Reserved.

Java Mutation Operators
4) Overloading

Method overloading makes it easy to call the incorrect method, usually by getting the parameters wrong.

- OMR: Overloading method contents change
- OMD: Overloading method deletion
- OAO: Argument order change
- OAD: Argument number decrease
- OAI: Argument number increase

© Copyright 1999-2002. All Rights Reserved.

Java Mutation Operators
5) Java-Specific Features

The previous operators are mostly language-independent. These operators focus on language features that are specific to Java.

- JTD: *this* keyword deletion
- JSD: *static* modifier deletion
- JID: Member variable initialization deletion
- JDC: Java-supported default constructor create

© Copyright 1999-2002. All Rights Reserved.

Java Mutation Operators
6) Common Programming Mistakes

Chevalley found a number of mistakes that programmers commonly make when using OO languages.

- EOA: Reference assignment and content assignment replacement
- EOC: Reference comparison and content comparison replacement
- EAM: Accessor method change
- EMM: Modifier method change

© Copyright 1999-2002. All Rights Reserved.

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**
- **A fault model and specific faults that are common in OO software**
- **Fault type-based mutation operators for Java**

© Copyright 1999-2002. All Rights Reserved.

Future Work

- **Mutation system for Java**
- **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

© Copyright 1999-2002. All Rights Reserved.

Relevant Papers on my Web Site

- Class Mutation Operators for Java, Ye-Seung Ma, Yong-Rae Kwon and Jeff Offutt. *Submitted*. February 2002.
- A Fault Model for Subtype Inheritance and Polymorphism, Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao and Chuck Hutchinson. *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, pages 84--95, Hong Kong, PRC, November 2001.
- Criteria for Testing Polymorphic Relationships, Roger Alexander and Jeff Offutt. *The Eleventh IEEE International Symposium on Software Reliability Engineering (ISSRE '00)*, pages 15–23, San Jose CA, October 2000.
- Analysis Techniques for Testing Polymorphic Relationships, Roger Alexander and Jeff Offutt. *Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104-114, Santa Barbara CA, August 1999.
- Coupling-based Criteria for Integration Testing, Zhenyi Jin and Jeff Offutt. *The Journal of Software Testing, Verification, and Reliability,* 8(3):133-154, September 1998.
- A Software Metric System for Module Coupling, Jeff Offutt, Mary Jean Harrold, and P. Kolte. *The Journal of Systems and Software*, 20(3):295-308, March 1993.

www.ise.gmu.edu/ofut/rsrch/abstracts/integ.html

© Copyright 1999-2002. All Rights Reserved.