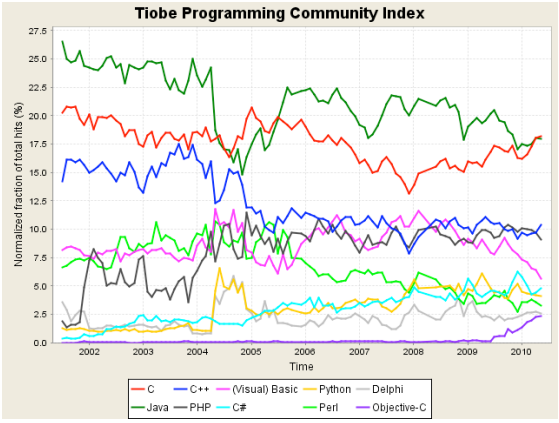
Structural Testing and Object Oriented Code

Software Engineering

Gordon Fraser • Saarland University

Based on slides from Mauro Pezzè & Michal Young, and from Paul Ammann and Jeff Offutt



What's different in OO?

What's different

- Less complexity in procedures
  Short methods
- Complexity is relocated
  to the connections among components
- Less problems
  based on intra-procedural and control flow
- More problems
  related to interaction between classes
- Less static determinism
  many faults can now only be detected at runtime

State dependent behavior

```
public class Model extends Orders.CompositeItem
{
  private Slot[] slots;

  // ...

  private void checkConfiguration() {
    legalConfig = true;
    for(int i=0; i<slots.length; ++i) {
      Slot slot = slots[i];
      if(slot.required && !slot.isBound()) {
        legalConfig = false;
      }
    }
  }
}
```

Outcome of method depends on state (slots)

Encapsulation

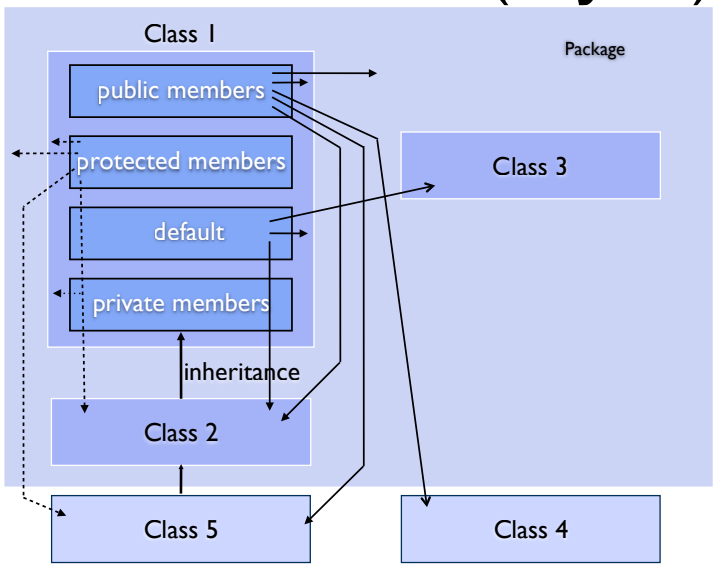
```
public class Model extends Orders.CompositeItem
{
  private Slot[] slots;

  // ...

  private void checkConfiguration() {
    legalConfig = true;
    for(int i=0; i<slots.length; ++i) {
      Slot slot = slots[i];
      if(slot.required && !slot.isBound()) {
        legalConfig = false;
      }
    }
  }
}
```

Private methods not externally accessible

Access Control (in Java)



Exception Handling

- Exception handling is integral part of OO programming
- Test where exceptions are thrown
- Test where exceptions are handled
- Test for unhandled exceptions

Abstract Classes

```
public class Model extends Orders.CompositeItem
{
  private Slot[] slots;

  // ...

  private void c
    legalConfig
    for(int i=0; i<slots.length; ++i) {
      Slot slot = slots[i];
      if(slot.required && !slot.isBound()) {
        legalConfig = false;
      }
    }
  }
}
```

How to test abstract classes?

Inheritance

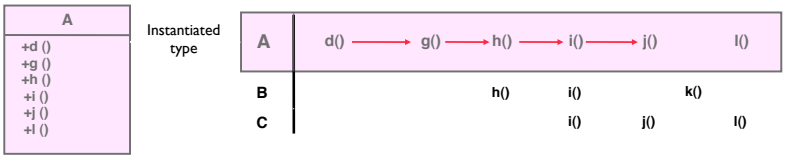
```
public class Model extends Orders.CompositeItem
{
  private Slot[] slots;

  // ...

  private void c
    legalConfig
    for(int i=0;
      Slot slot = slots[i];
      if(slot.required && !slot.isBound()) {
        legalConfig = false;
      }
    }
  }
}
```

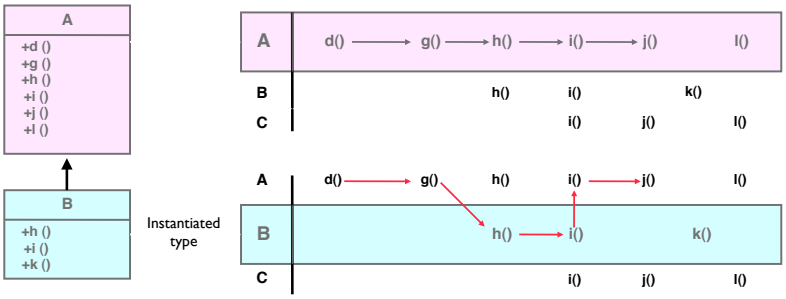
When to test overridden methods?

Polymorphism Headaches



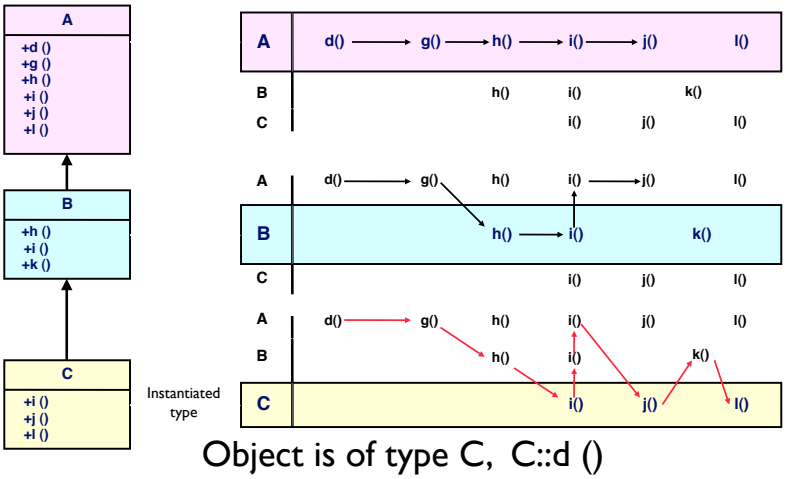
Object is of type A
A::d ()

Polymorphism Headaches



Object is of type B
B::d ()

Polymorphism Headaches



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

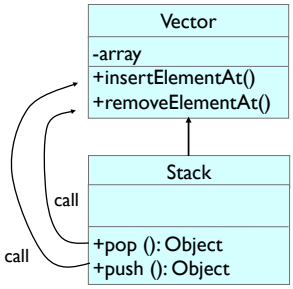
Inconsistent Type Use

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 inconsistent for *C*




```
s.push ("Steffi");
s.push ("Joyce");
s.push ("Andrew");
dumb (s);
s.pop();
s.pop();
s.pop(); // Stack is empty!

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

Dynamic binding

```
abstract class Credit {
 // ...
 abstract boolean validateCredit(Account a, int amt, CreditCard c);
 // ...
}
```

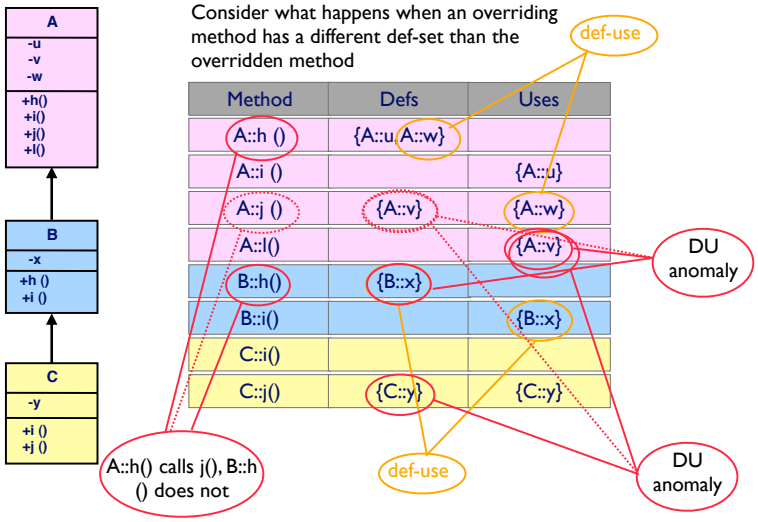
EduCredit
BizCredit
IndividualCredit

USAccount
UKAccount
EUAccount
JPAccount
OtherAccount

VISACard
AmExpCard
StoreCard

The combinatorial problem: 3 x 5 x 3 = 45 possible combinations of dynamic bindings (just for this one method!)

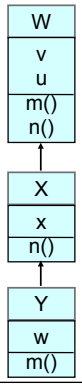
DU Pairs and Anomalies



State Definition Anomaly

X extends *W*, and *X* overrides some methods

Overriding methods in *X* fail to define some variables that the overridden methods in *W* defined



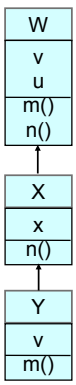
- ***W::m()*** defines ***v*** and ***W::n()*** uses ***v***
- ***X::n()*** uses ***v***
- ***Y::m()*** does not define ***v***

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

State Definition Inconsistency

Hiding 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***

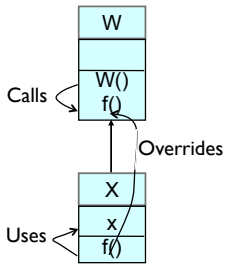
For an object of type ***Y***, a data flow inconsistency may exist and result in a fault if ***m()*** is called, then ***n()***

Anomalous Construction Behavior

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()***.

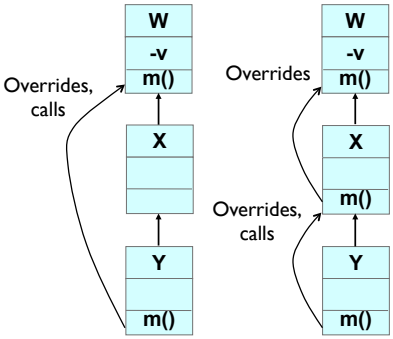
When ***W()*** calls ***X::f()***, ***x*** is used, but has <u>not</u> yet been given a value!

State Visibility Anomaly

A private variable ***v*** is declared 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()***!

OO Mutation

AMC - Access Modifier Change
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion
ATC - Actual Type Change
DTC - Declared Type Change
RTC - Reference Type Change
PTC - Parameter Type Change
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

OO Mutation

Information Hiding
AMC - Access Modifier Change

Inheritance
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion

Polymorphism
ATC - Actual Type Change
DTC - Declared Type Change

RTC - Reference Type Change
PTC - Parameter Type Change

Overloading
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change

Java Specific
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

OO Mutation

Information Hiding
AMC - Access Modifier Change

Inheritance
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion

Polymorphism
ATC - Actual Type Change
DTC - Declared Type Change

RTC - Reference Type Change
PTC - Parameter Type Change

Overloading
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change

Java Specific
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

AMC - Access Modifier Change

```
public Stack s;
```

```
private Stack s;
```

```
protected Stack s;
```

```
Stack s;
```

This operator does not find any functional faults, but shows where encapsulation is not done properly. Because it leads to many uncompilable or equivalent mutants it is not used in practice.

OO Mutation

Information Hiding
AMC - Access Modifier Change

Inheritance
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion

Polymorphism
ATC - Actual Type Change
DTC - Declared Type Change

RTC - Reference Type Change
PTC - Parameter Type Change

Overloading
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change

Java Specific
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

HVD - Hiding variable deletion

```
class List {
  int size;
  ... ...
}
class Stack extends List {
  int size;
  ... ...
}
```

```
class List {
  int size;
  ... ...
}
class Stack extends List {
Δ //  int size;
  ... ...
}
```

This mutant can only be killed by a test case that is able to show that the reference to the parent variable is incorrect.

HVI – Hiding variable insertion

```
class List {
  int size;
  ... ...
}
class Stack extends List {
  ... ...
}
```

```
class List {
  int size;
  ... ...
}
class Stack extends List {
Δ int size;
  ... ...
}
```

This mutant can only be killed by a test case that is able to show that the reference to the overridden variable is incorrect.

OMD – Overriding method deletion

```
class List {
  void clear() { ...}
}
class Stack extends List {
  ... ...
  void clear() {...}
}
```

```
class List {
  void clear() { ...}
}
class Stack extends List {
  ... ...
Δ // void clear() {...}
}
```

This mutant is killed by a test case that is able to show that the behavior of the parent's method is incorrect.

OMM – Overridden method calling position change

```
class List {
  ... ...
  void SetEnv() {
    size = 5; ...
  }
}
class Stack extends List {
  int size;
  ... ...
  void SetEnv() {
    super.SetEnv();
    size = 10;
  }
}
```

```
class List {
  ... ...
  void SetEnv() {
    size = 5; ...
  }
}
class Stack extends List {
  int size;
  ... ...
  void SetEnv() {
Δ   size = 10;
Δ   super.SetEnv();
  }
}
```

OMR – Overridden method rename

```
class List {
  void m() {
    ...
    f();
    ...
  }
  void f() { ... }
}
class Stack extends List {
  ... ...
  void f() { ... }
}
```

```
class List {
  void m() {
    ...
    f'();
    ...
  }
  void f'() { ... }
}
class Stack extends List {
  ... ...
  void f() { ... }
}
```

These mutants can only be killed by a test case that causes different behavior when the overriding (child's) version is not called.

SKR – super keyword deletion

```
class Stack extends List {
  ... ...
  int MyPop( ) {
    ... ...
    return val*super.num;
  }
}
```

```
class Stack extends List {
  ... ...
  int MyPop( ) {
    ... ...
Δ    return val*num;
  }
}
```

Ensures that hiding/hidden variables and overriding/overridden methods are used appropriately.

PCD – Explicit call of a parent's constructor deletion

```
class Stack extends List {
  ... ...
  Stack (int a) {
    super (a);
    ... ...
  }
}
```

```
class Stack extends List {
  ... ...
  Stack (int a) {
Δ   // super (a);
    ... ...
  }
}
```

Causes calling of default constructor. Can only be killed by a test case for which the parent's default constructor creates an initial state that is incorrect.

OO Mutation

Information Hiding
AMC - Access Modifier Change

Inheritance
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion

Polymorphism
ATC - Actual Type Change
DTC - Declared Type Change

RTC - Reference Type Change
PTC - Parameter Type Change

Overloading
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change

Java Specific
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

Polymorphic Mutations

```
A a;
a = new A();
```

ATC
```
A a;
Δ a = new B();
```

DTC
```
Δ B a;
a = new A();
```

```
boolean equals (B o)
{ ... }
```

PTC
```
Δ boolean equals (A o)
{ ... }
```

```
Object obj;
String s = “Hello”;
Integer i = new Integer(4);
obj=s;
```

RTC
```
Object obj;
String s = “Hello”;
Integer i = new Integer(4);
Δ obj=i;
```

Causes object reference to refer to an object of a type that is different from the declared type.

OO Mutation

Information Hiding
AMC - Access Modifier Change

Inheritance
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion

Polymorphism
ATC - Actual Type Change
DTC - Declared Type Change

RTC - Reference Type Change
PTC - Parameter Type Change

Overloading
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change

Java Specific
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

OMR – Overloading method contents change

```
class List {
  ... ...
  void Add (int e) {
    ... ...
  }
  void Add (int e, int n) {
    ... ...
  }
}
```

```
class List {
  ... ...
  void Add (int e) {
    ... ...
  }
  void Add (int e, int n) {
Δ    this.Add(e);
  }
}
```

Checks that overloaded methods are invoked correctly.

OMD – Overloading method deletion

```
class Stack extends List
{ ... ...
  void Push (int i)
   { ... }
  void Push (float f)
   { ... }
}
```

```
class Stack extends List
{ ... ...
Δ // void Push (int i)
Δ //  { ... }
 void Push (float f)
  { ... }
}
```

Ensures coverage of overloaded methods.

Changing Arguments

```
s.Push (0.5, 2);
```

AOC

```
Δ s.Push (2, 0.5);
```

ANC

```
Δ s.Push (2);
Δ s.Push (0.5);
Δ s.Push ();
```

Only if there is an overloaded method that accepts the parameters.

OO Mutation

Information Hiding
AMC - Access Modifier Change

Inheritance
HVD - Hiding Variable Deletion
HVI - Hiding Variable Insertion
OMD - Overriding Method Deletion
OMM - Overridden Method Moving
OMR - Overridden Method Rename
SKR - Super Keyword Deletion
PCD - Parent Constructor Deletion

Polymorphism
ATC - Actual Type Change
DTC - Declared Type Change

RTC - Reference Type Change
PTC - Parameter Type Change

Overloading
OMC - Overloading Method Change
OMD - Overloading Method Deletion
AOC - Argument Order Change
ANC - Argument Number Change

Java Specific
TKD - this Keyword Deletion
SMV - Static Modifier Change
VID - Variable Initialization Deletion
JDC - Java Supported Default Constructor

TKD – this keyword deletion

```
class Stack {
  int size;
  ... ...
  void setSize(int size) {
    this.size = size;
  }
}
```

```
class Stack {
  int size;
  ... ...
  void setSize(int size) {
Δ    size = size;
  }
}
```

Similar to hiding variables.

SMV – static modifier change

```
public static int s = 100;
private String name;
```

```
public int s = 100;
public static String name;
```

Change class to instance variables.

VID – Member variable initialization deletion

```
class Stack {
  int size = 100;
  ... ...
}
```

```
  class Stack {
Δ   int size;
    ... ...
  }
```

Ensures correct initialization of instance variables.

Others

EOA – Reference assignment and content assignment replacement

```
List list1, list2;
list1 = new List();
list2 = list1;
```

```
List list1, list2;
list1 = new List();
Δ list2 = list1.clone();
```

EOC – Reference comparison and content comparison replacement

```
Fract f1 = new Fract (1, 2);
Fract f2 = new Fract (1, 2);
boolean b = (f1==f2);
```

```
Fract f1 = new Fract (1, 2);
Fract f2 = new Fract (1, 2);
Δ boolean b = f1.equals(f2);
```

EAM – Accessor method change

```
point.getX();
```

```
Δ point.getY();
```

EAM – Modifier method change

```
point.setX(2);
```

```
Δ point.setY(2);
```

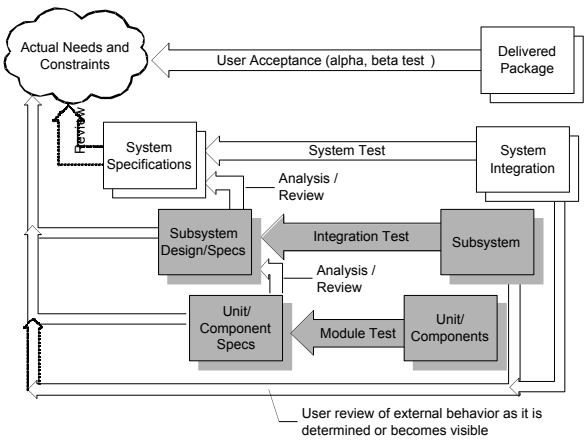
OO Mutation

- Creates far less mutants than traditional mutation
- Large percentage of equivalent mutants

OO Testing

Using Structural Information

- Start with functional testing
  - As for procedural software, the specification (formal or informal) is the first source of information for testing object-oriented software
    - "Specification" widely construed: Anything from a requirements document to a design model or detailed interface description
- Then add information from the code (structural testing)
  - Design and implementation details not available from other sources



OO Testing Levels

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

Intra-Method Testing

- Procedural testing also applies to methods
  Coverage of control flow, data-flow, ...
- Setup-code necessary to get into right state
- Need to treat exceptions

Exception handling

```
void addCustomer(Customer theCust) {
  customers.add(theCust);
}
public static Account newAccount(...)
   throws InvalidRegionException
{
  Account thisAccount = null;
  String regionAbbrev = Regions.regionOfCountry(
                mailAddress.getCountry());
  if (regionAbbrev == Regions.US) {
      thisAccount = new USAccount();
  } else if (regionAbbrev == Regions.UK) {
    //...
  } else if (regionAbbrev == Regions.Invalid) {
    throw new InvalidRegionException(mailAddress.getCountry());
  }
  //...
}
```

exceptions create implicit control flows and may be handled by different handlers

Testing exception handling

- Impractical to treat exceptions like normal flow
  - too many flows: every array subscript reference, every memory allocation, every cast, ...
  - multiplied by matching them to every handler that could appear immediately above them on the call stack.
  - many actually impossible
- So we separate testing exceptions
  - and ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: Each exception handler, and each explicit throw or re-throw of an exception

Testing exception handlers

- Local exception handlers
  - test the exception handler (consider a subset of points bound to the handler)
- Non-local exception handlers
  - Difficult to determine all pairings of <points, handlers>
  - So enforce (and test for) a design rule:
    if a method propagates an exception, the method call should have *no other effect*

Inheritance

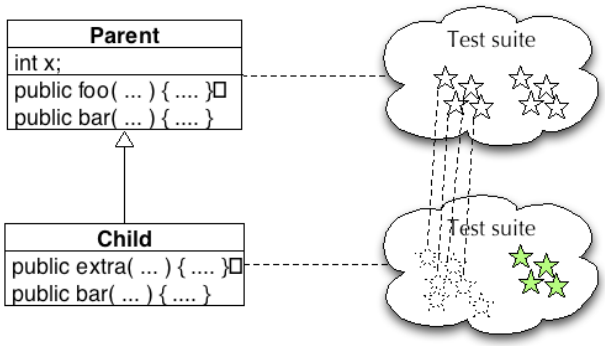
When testing a subclass ...

- We would like to re-test only what has not been thoroughly tested in the parent class
  - for example, no need to test hashCode and getClass methods inherited from class Object in Java
- But we should test any method whose behavior may have changed
  - even accidentally!

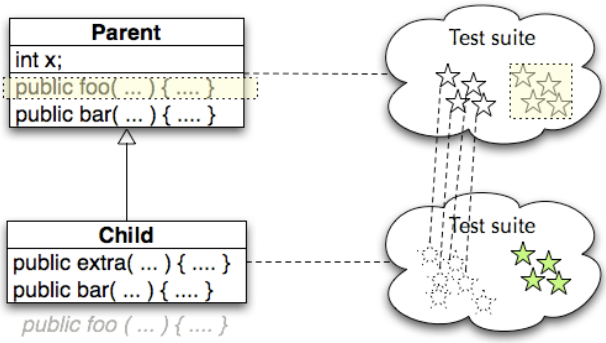
Reusing Tests with the Testing History Approach

- Track test suites and test executions
  - determine which new tests are needed
  - determine which old tests must be re-executed
- New and changed behavior ...
  - new methods must be tested
  - redefined methods must be tested, but we can partially reuse test suites defined for the ancestor
  - other inherited methods do not have to be retested

Testing history

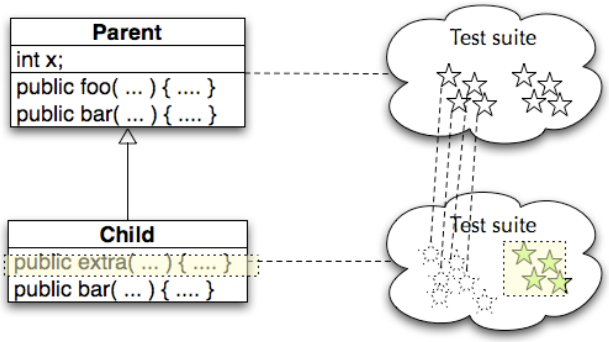


Inherited, unchanged



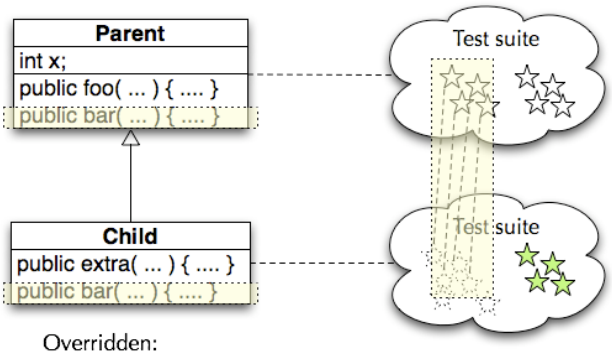
Inherited, unchanged ("recursive"):
No need to re-test

Newly introduced methods



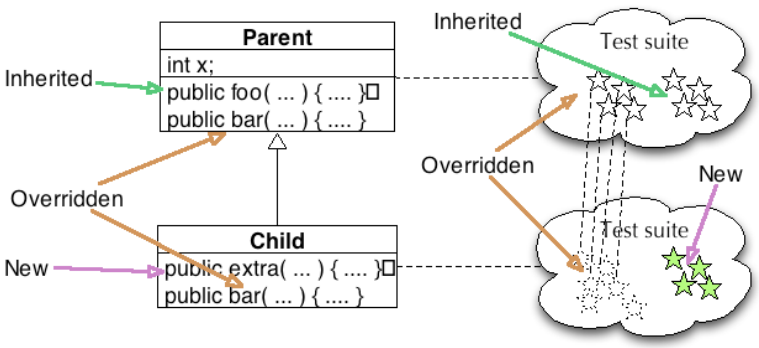
New:
Design and execute new test cases

Overridden methods



Overridden:
Re-execute test cases from parent,
add new test cases as needed

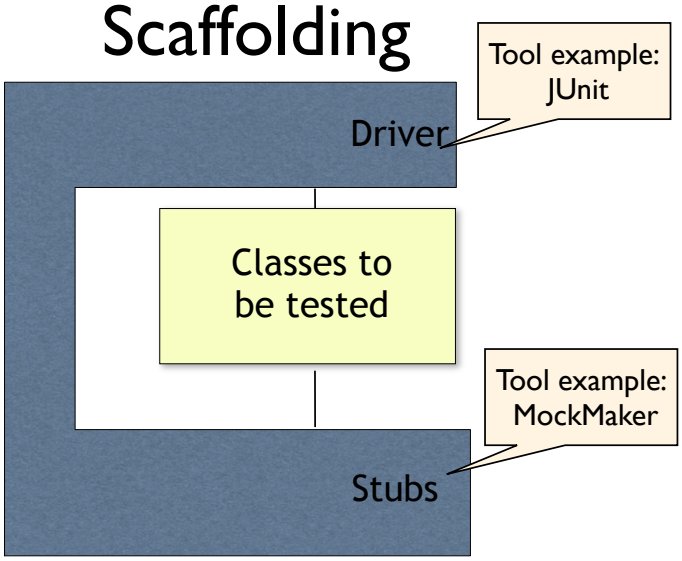
Testing History - Summary



Stubs, Drivers, and Oracles for Classes

- Problem: State is encapsulated
  - How can we tell whether a method had the correct effect?
- Problem: Most classes are not complete programs
  - Additional code must be added to execute them
- We typically solve both problems together, with *scaffolding*

Scaffolding



A test scaffolding is composed of:

one or more drivers, that provide a prototype activation environment for the unit under test . For procedural programs, drivers initialize non-local variables and parameters, and call the unit.

one or more stubs, that provides a prototype of the units used by the program to be tested.

one or more oracles, i.e., acceptors, that identify the tests that cause failures.

Oracles

- Test oracles must be able to check the correctness of the behavior of the object when executed with a given input
- Behavior produces *outputs* and brings an object into a *new state*
  - We can use traditional approaches to check for the correctness of the output
  - To check the correctness of the final state we need to access the state

Accessing the state

- Intrusive approaches
  - use language constructs (C++ friend classes)
  - add inspector methods
  - *in both cases we break encapsulation and we may produce undesired results*
- Equivalent scenarios approach:
  - generate equivalent and non-equivalent sequences of method invocations
  - compare the final state of the object after equivalent and non-equivalent sequences

Equivalent Scenarios

selectModel(M1)

addComponent(S1,C1)

addComponent(S2,C2)

isLegalConfiguration()

deselectModel()

selectModel(M2)

addComponent(S1,C1)

isLegalConfiguration()

EQUIVALENT

selectModel(M2)
addComponent(S1,C1)
isLegalConfiguration()

NON EQUIVALENT

selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

OO Testing Levels

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

Inter-Method Testing

- Complexity due to connections and interactions
- Simple structural criteria cannot capture this complexity
  - Dataflow within one method
  - Dataflow within one class
  - Inter-class data flow
- Data flow couplings among units and classes are more complicated than control flow couplings

Data flow couplings among units and classes are more complicated than control flow couplings
When values are passed, they "change names"
Many different ways to share data
Finding defs and uses can be difficult – finding which uses a def can reach is very difficult
When software gets complicated … testers should get interested
That's where the faults are!

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

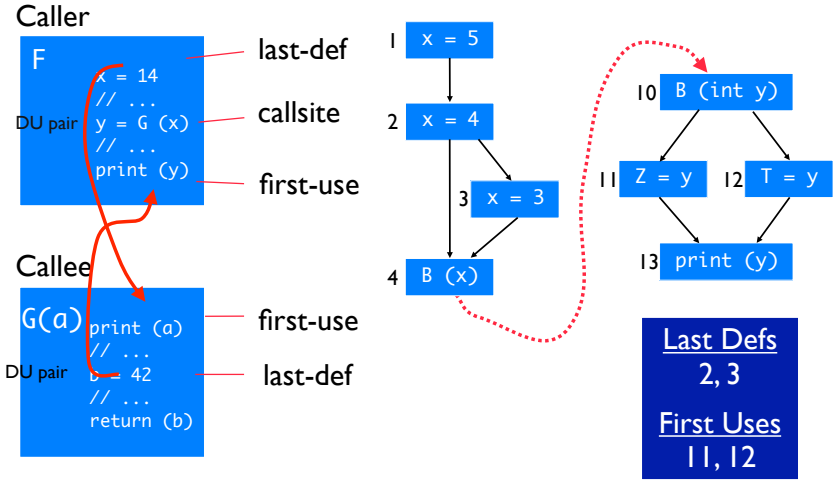
If we focus on the interface, then we just need to consider the last definitions of variables before calls and returns and first uses inside units and after calls

Last-def : The set of nodes that define a variable x and has a def-clear path from the node through a callsite to a use in the other unit

Can be from caller to callee (parameter or shared variable) or from callee to caller as a 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 callsite to the nodes

Example Inter-procedural DU Pairs



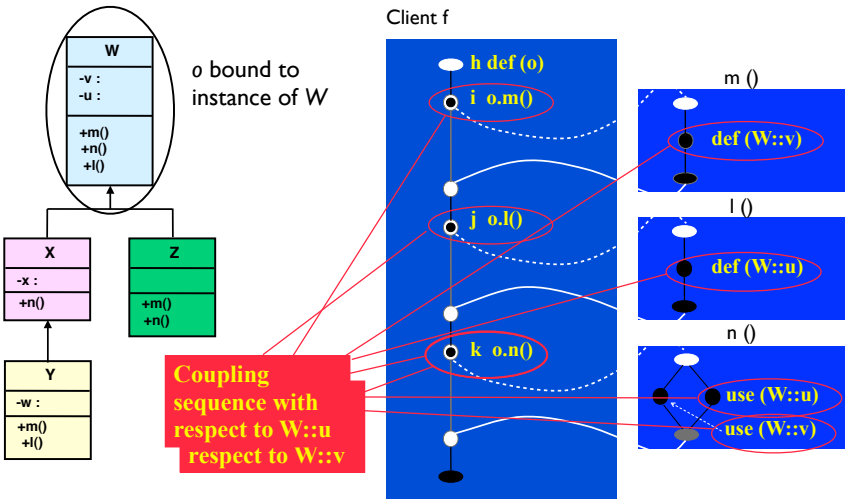
Polymorphic Call Set

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

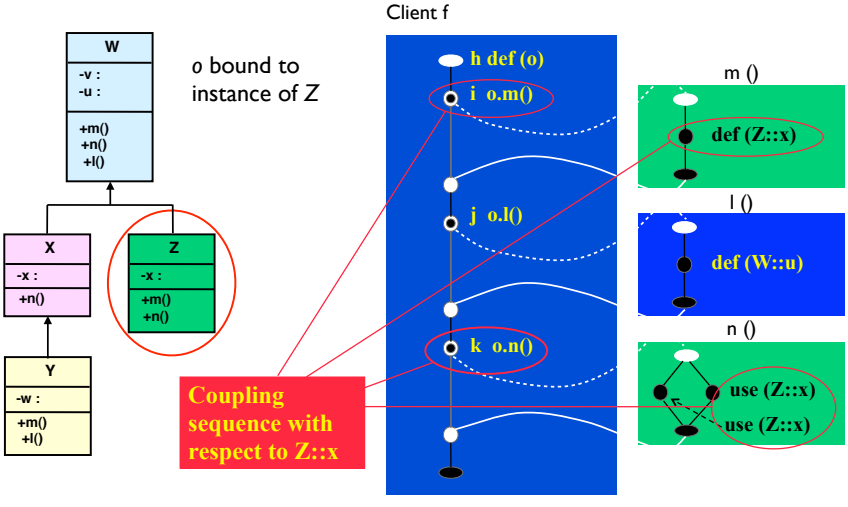
Set of methods that can potentially execute as result of a method call through a particular instance context

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

Example Coupling Sequence



Example Coupling Sequence (2)



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
- 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 last one test case ***t***

- Every last definition of a coupling variable reaches every first use
- Does not consider inheritance and polymorphism

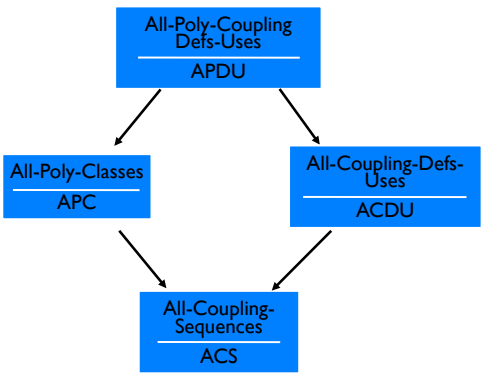
All-Poly-Coupling-Defs-and-Uses

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 ***Sj,k***, for every coupling variable ***v*** of Sj,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 ***Sj,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 Criteria Subsumption



OO Testing Levels

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

Intra-Class Testing

- Basic idea:
- The state of an object is modified by operations
- Methods can be modeled as state transitions
- Test cases are sequences of method calls that traverse the state machine model
- State machine model can be derived from specification (functional testing), code (structural testing), or both

Informal state-full specifications

Slot: represents a slot of a computer model.

.... slots can be bound or unbound. Bound slots are assigned a compatible component, unbound slots are empty. Class slot offers the following services:

- **Install**: slots can be installed on a model as *required* or *optional*.
  ...
- **Bind**: slots can be bound to a compatible component.
  ...
- **Unbind**: bound slots can be unbound by removing the bound component.
- **IsBound**: returns the current binding, if bound; otherwise returns the special value *empty*.

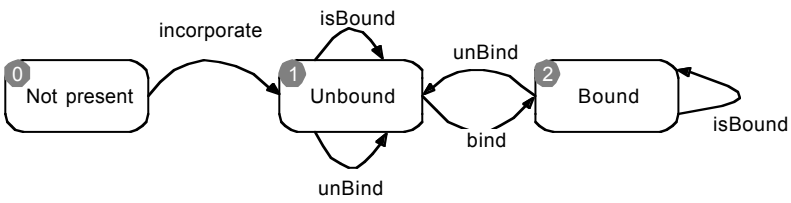
Part I: generating test cases from informal specifications:

Let us start with an informal description of a simple class *slot* of our running example: the web presence of the Chipmunk Coputer Company.

Identifying states and transitions

- From the informal specification we can identify three states:
  - Not_installed
  - Unbound
  - Bound
- and four transitions
  - install: from Not_installed to Unbound
  - bind: from Unbound to Bound
  - unbind: ...to Unbound
  - isBound: does not change state

Deriving an FSM and test cases



- TC-1: incorporate, isBound, bind, isBound
- TC-2: incorporate, unBind, bind, unBind, isBound

A simple analysis of the informal specification of class Slots allows to identify states and transitions.

The analysis reveals some ambiguities that have been solved in the finite state machine model.

Intraclass data flow testing

- Exercise sequences of methods
  - From setting or modifying a field value
  - To using that field value
- We need a control flow graph that encompasses more than a single method ...

The intraclass control flow graph

Control flow for each method

+

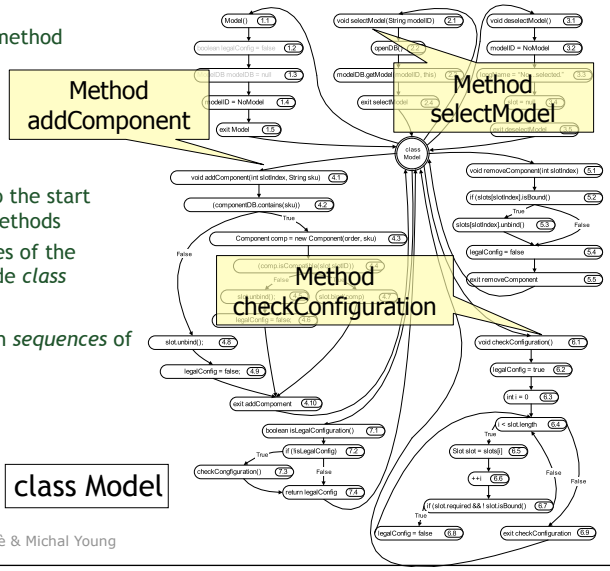
node for class

+

edges

- from node *class* to the start nodes of the methods
- from the end nodes of the methods to node *class*

=> control flow through *sequences* of method calls



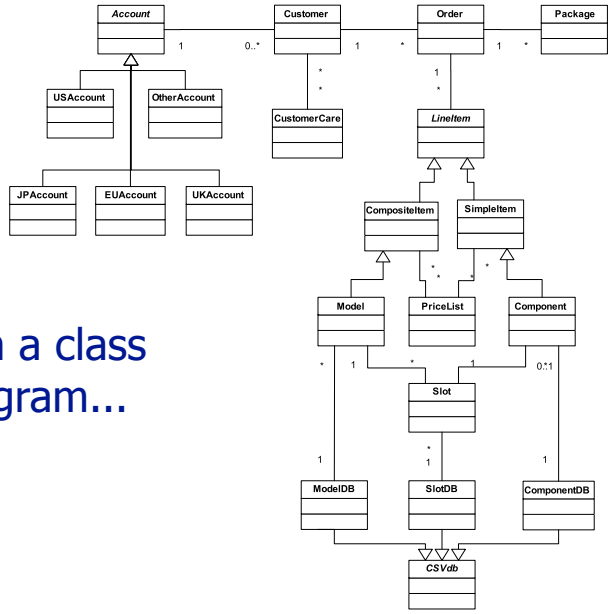
OO Testing Levels

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

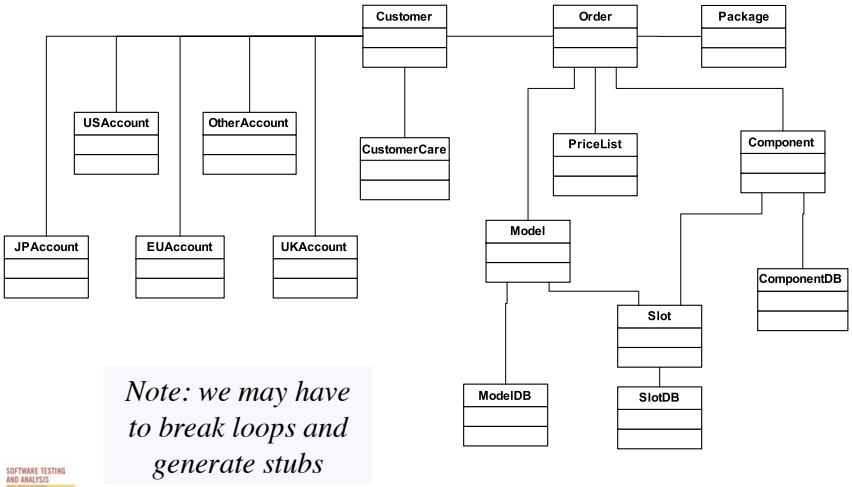
Inter-Class Testing

- The first level of *integration testing* for object-oriented software
  - Focus on interactions between classes
- Bottom-up integration according to "depends" relation
  - A depends on B: Build and test B, then A
    - Class A makes method calls on class B
    - Class A objects include references to class B methods
      - but only if reference means "is part of"

from a class diagram...



....to a hierarchy



Note: we may have to break loops and generate stubs

Summary

- Several features of object-oriented languages and programs impact testing
  - from encapsulation and state-dependent structure to generics and exceptions
  - but only at unit and subsystem levels
  - and fundamental principles are still applicable
- Basic approach is orthogonal
  - Techniques for each major issue (e.g., exception handling, generics, inheritance, ...) can be applied incrementally and independently

Random Testing Revisited: Method Sequences

Based on slides by Carlos Pacheco

OO Testing

- Testing object oriented systems is complicated
- A test case is a sequence of method calls
- Testing object oriented systems is an area of active research
- What can we do automatically?

Deriving Method Sequences

- Symbolic execution
- Evolutionary search
- Random search

Test Cluster

- Java Reflection
- What's the class under test?
- What are the dependencies (parameters)
- All methods, constructors, fields form the test cluster

Feedback-directed random test generation

- Build test inputs incrementally
  - New test inputs extend previous ones
  - In our context, a test input is a method sequence
- As soon as a test is created, execute it
- Use execution results to guide the search
  - away from redundant or illegal method sequences
  - towards sequences that create new object states

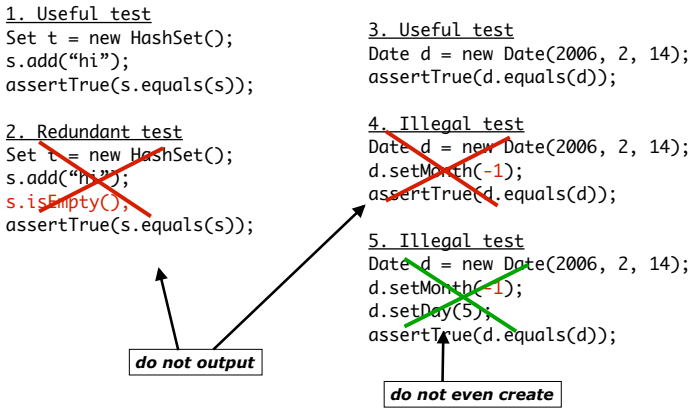
1. Useful test
```
Set t = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

2. Redundant test
```
Set t = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

3. Useful test
```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

4. Illegal test
```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

5. Illegal test
```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```



Technique input/output

- Input:
  - classes under test
  - time limit
  - set of contracts
    - Method contracts (e.g. "o.hashCode() throws no exception")
    - Object invariants (e.g. "o.equals(o) == true")
- Output: contract-violating test cases

Technique input/output

no contracts violated up to last method call

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

fails when executed

Creating a sequence

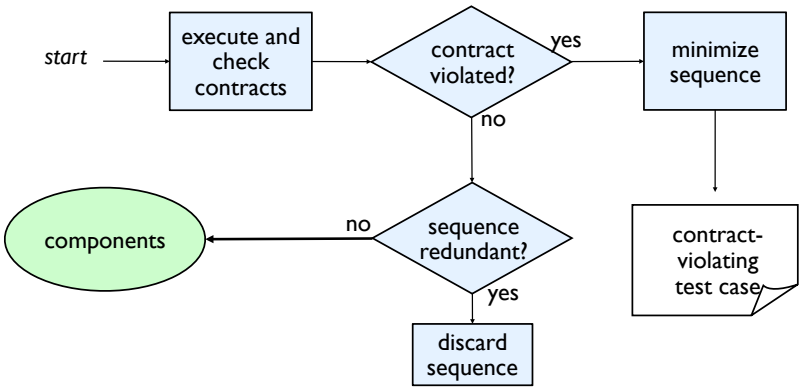
- Each sequence "constructs" several objects, available after the last method call is executed:
  - < result, receiver, $param_1$, ..., $param_n$ > of last method call
- Example: sequence constructs two objects:

  < LinkedList, HashSet >

```
LinkedList l = new LinkedList();
Set h = new HashSet();
l.addFirst(h);
```

- To create a new sequence:
  i. Randomly pick a method call $m(T_1...T_k)/T_{ret}$ from a tested class
  ii. For each input parameter of type T_i, randomly pick a sequence S_i that constructs an object of type T_i
  iii. Create new sequence $S_{new} = S_1 \bullet ... \bullet S_k \bullet T_{ret}\ v_{new} = m(v_1...v_k)$;
  iv. if S_{new} was previously created (lexically), go to i

Classifying a sequence



Redundant Sequences

- During generation, maintain a set of all objects created.
- A sequence is redundant if all the objects created during its execution are members of the above set (using **`equals`** to compare)
  - We incorporate redundancy checking as part of the generation process.

Randoop

- Implements feedback-directed random test generation
- Input:
  - An assembly (for .NET) or a list of classes (for Java)
  - Generation time limit
  - Optional: a set of contracts to augment default contracts
- Output: a test suite (Junit or Nunit) containing
  - Contract-violating test cases
  - Normal-behavior test cases

Randoop outputs oracles

- Oracle for contract-violating test case:

```
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

- Oracle for normal-behavior test case:

```
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
l.add(o);
assertEquals(2, l.size());
assertEquals(false, l.isEmpty());
```

Randoop uses ***observer methods*** *to capture object state*

Errors found: examples

- JDK Collections classes have 4 methods that create objects violating o.equals(o) contract
- Javax.xml creates objects that cause hashCode and toString to crash, even though objects are well-formed XML constructs
- Apache libraries have constructors that leave fields unset, leading to NPE on calls of equals, hashCode and toString (this only counts as one bug)
- Many Apache classes require a call of an init() method before object is legal—led to many false positives
- .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, of illegal state exception)
- .Net framework has 8 methods that violate o.equals(o)
- .Net framework loops forever on a legal but unexpected input