Object Oriented Testing

Validating Object Oriented Systems

- Do OO systems make validation harder or easier?
- Does code reuse lead to validation reuse?
- Do we need to change existing techniques?
  - If so, how?
- Do we need to develop new techniques?

What is an Object Oriented Programming Language?

- Supports abstract data types (ADTs)
  - Information hiding
  - Encapsulation
- Supports inheritance
  - Change to a parent type is reflected in the children
  - Supports reuse
  - Subtype or Subclass
    - Subclass- reuse implementation information
    - Subtype-child type must be a legal member of the parent type
- Supports dynamic binding/dispatch or polymorphism

may have additional features, but at least should have these

Some terminology

- A class is a type
  - Access methods
  - Instance variables (attributes)
    - Any access method may access the instance variables
  - An object is an instance of a class
    - May have multiple instances of a class, each with their own instance variables
- Methods are invoked via messages
  - Not referring to concurrency but to dynamic binding
  - Actual method that is invoked may need to be determined at runtime

Example: inheritance

```
class Table
        create( );
        insert (int entry);
        delete (int entry);
        isEmpty() returns boolean;
        isEntered(int entry) returns boolean;
endclass;
```

```
class UniqueTable extends Table
        insert(int entry);
endclass;
```

Is UniqueTable a subtype or subclass of Table?

$T \in UniqueTable \Rightarrow T \in Table$

Example: dynamic binding

```
t.insert(entry);
```

=>Which insert method gets called depends on the type of t

Example: instance variables

```
class Table
        int numberElements;
        create( );
        insert (int entry);
        delete (int entry);
        isEmpty() returns boolean;
        isEntered(int entry) returns boolean;
endclass;
```

Example: generic (parameterized class)

```
class Table (elemType)
        int numberElements;
        create( );
        insert (elemType entry);
        delete (elemType entry);
        isempty() returns boolean;
        isentered(elemType entry) returns boolean;
endclass;
```

Some more terminology

- Single inheritance
  - A class may inherit from only one parent
- Multiple inheritance
  - A class may inherit from one or more parents
  - Need to define what happens if there are conflicts
    - E.g., each parent has an insert method
- Parent class is also called supertype/superclass
- Child class is also called a subtype/subclass

Validating Object Oriented Systems

- How are dynamic analysis approaches affected?
  - E.g., coverage criteria
- How are testing processes affected?
  - Unit testing
  - Integration testing
  - Regression testing
- How are static analysis approaches affected?
  - Dependency analysis

Issues in O-O testing

- basic unit for unit testing
- implications of encapsulation
- implications of inheritance
- implications of genericity
- implications of polymorphism/dynamic binding
- implications for testing processes

Unit Testing Object-Oriented Systems

- procedural programming
  - basic component: subroutine
  - results: output data and out parameters
- object-oriented programming
  - basic component:
    class = owned data structures + set of operations
  - objects are instances of classes
  - Results: output data, out parameters and state
    - data structures define the state of the object
  - state is not directly accessible, but can only be accessed using the access methods (encapsulation)

Basic Unit for Testing

- the class is the natural unit for unit test case design
- methods are meaningless apart from their class
- testing a class instance (an object) can validate a class in isolation
- when individually validated classes are used to create more complex classes in an application system, the entire subsystem must be tested as a whole before it can be considered to be validated (integration testing)

Issues in O-O testing

- basic unit for unit testing
- implications of encapsulation
- implications of inheritance
- implications of genericity
- implications of polymorphism/dynamic binding
- implications for testing processes

Encapsulation

- not a source of errors but may be an obstacle to testing
- how to get at the concrete state of an object?
  - break the encapsulation
    - using features of the languages
      - C++ friend
      - Ada95 Child Unit
    - use low level probes or debugging tools to manually inspect

How to get at the concrete state of an object?

- Use the abstraction
  - State is inspected via access methods
  - Scenarios--examine sequences of events
    - t. create ( ); t. push (item); t. pop( ) = t. create ( )
    - Need to be able to define what equivalent sequences are and need to determine equal states

- Use or provide hidden functions to examine the state
  - Useful for debugging throughout the life of the system
    - But, modified code may alter the behavior
    - Especially true for languages that do not support strong typing

Example: local state of an object

```
class Table
        private int numberelements;
        create( );
        insert (int entry);
        delete (int entry);
        isEmpty() returns boolean;
        isEntered(int entry) returns boolean;
endclass;

class UniqueTable extends Table
        insert(entry) returns table;
endclass;
```

ASTOOT

- Proposed by Phyllis Frankl and R.K. Doong
- Requires each class to provide its own simplified "oracle"
  - Determines if two instances of a class are equivalent
- Uses a class' algebraic specification to derive alternative equivalent test cases
  - A form of specification-based testing
- Uses an oracle to determine if the implementation of the class satisfies the specification of the class for the test cases

Algebraic Specification

- Specifies signatures of all the methods
- Specifies axioms that the class is supposed to maintain
  - expected results from combinations of method invocations
  - Usually need to consider all type compatible combinations of the methods

Algebraic Specification :Stack Example

Class Stack

Signatures:

create: -> stack;

pop: stack -> stack;

push: stack x value -> stack;

top: stack -> value;

isEmpty: stack -> Boolean;

Algebraic Specification :Stack Example

Variables:

s: stack; val: value;

Axioms:

s.push(val).isEmpty = false;

s.push(val).pop = s;

create.isEmpty = true;

create.pop = error;

create.top = error;

s.push(val).top = val;

ASTOOT creates pairs of equivalent test cases

- Uses algebraic specifications to define test cases
  - Create test cases that are syntactically correct sequences of access methods
  - Can be either user defined or automatically generated from the algebraic specification
  - Using algebraic specifications, simplify or extend sequences to create "equivalent" test cases

Example equivalent test cases

create(s);push(s,5) =

create(s);push(s,5);top(s) =

create(s);push(s,5);top(s);push(s,10);pop(s)

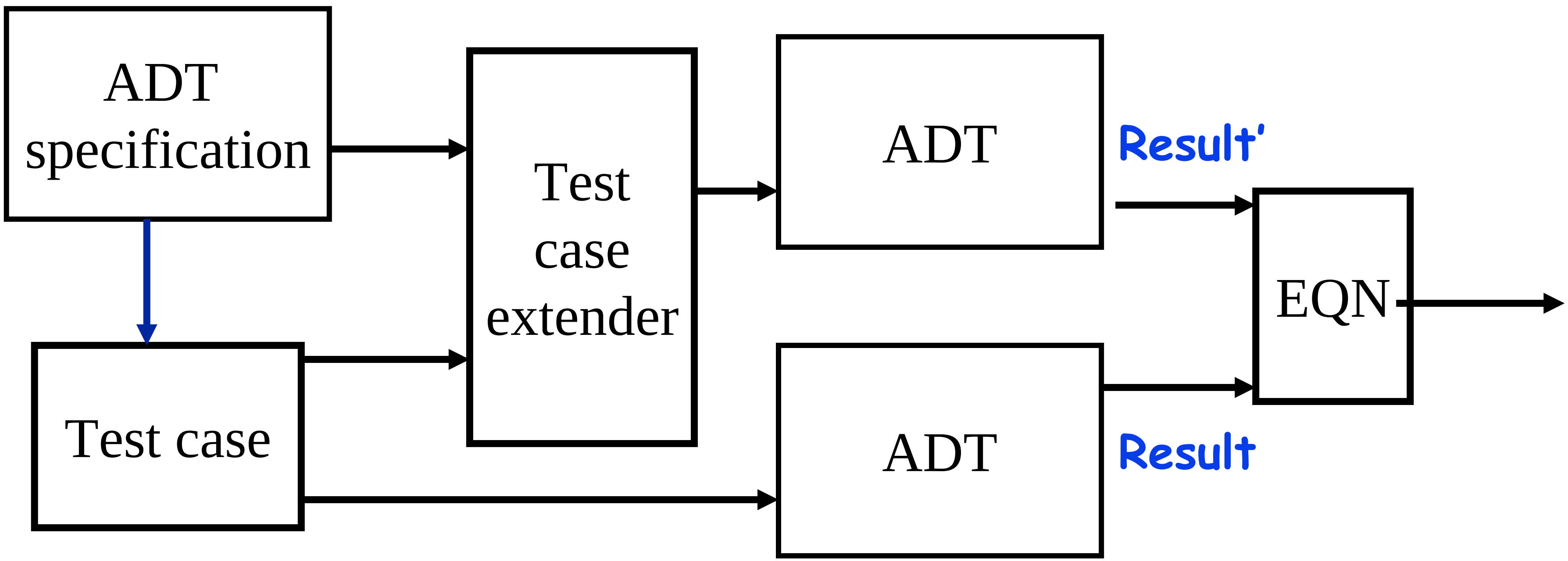
Kinds of Methods/Transformations

- Constructors (creators)-return initial objects
  - Not all methods can be applied to an initial object
  - Create(s); pop(s)
- Observers-return state information but do not change the state
  - A no op in terms of impact on state
    - Identity function f(s) = s
  - create(s);push(s,5);top(s);push(s,10);pop(s)
- Transformers-changes the value of at least one element of the state
  - Inverse functions $s = f(s);\ f^{-1}(s)$
  - create(s);push(s,5);top(s);push(s,10);pop(s)

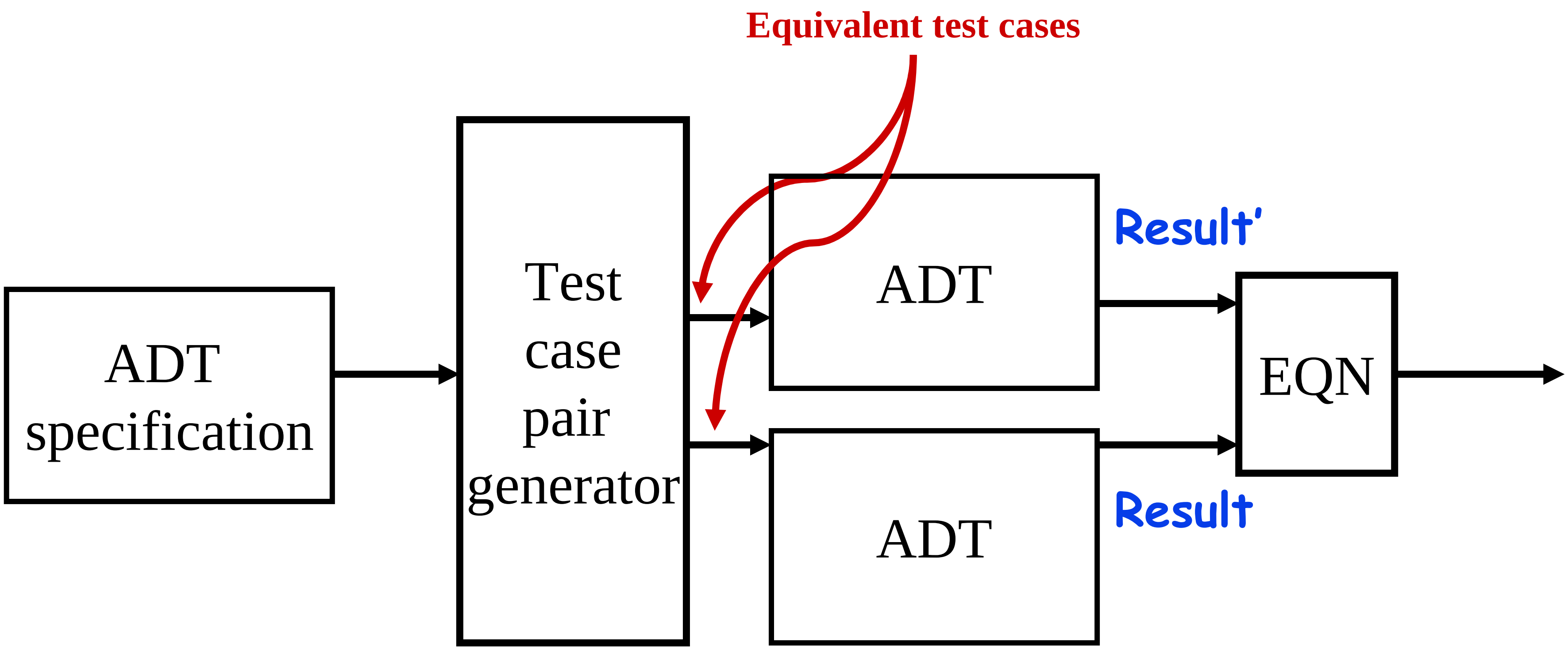
Using the EQN test oracle

- Using EQN function, determine if the class returns the same results for both test cases
  - Tests whether the specification is defined correctly
  - Tests whether the implementation meets the specifications

ASTOOT usage model



ASTOOT alternative usage model



EQN: Simplified oracle

- Requires that each class have an equivalence function, EQN, that determines if two instances of the same class are "equivalent"
  - E.g. EQN(create;push(5);push(6);pop,
    create;push(5))
    would return true
- Can define EQN recursively using the access methods
- Can define EQN using the underlying implementation

Example: recursive definition of EQN

```
if IsEmpty(s1) and IsEmpty (s2) then true
     elseif IsEmpty(s1) then false
     elseif IsEmpty(s2) then false
     elseif Top(s1)≠Top(s2) then false
     else
              EQN (Pop(s1),Pop(s2))
endif
```

Example:implementation based definition of EQN

```
EQN(s1, s2) returns flag
s1,s2: stack;
flag := true;
If size(s1) ≠ size(s2) then flag := false;
i := firstIndex(s1);
while i≤size(s1) and flag =true do
   if s1(i) ≠ s2(i) then flag := false
   i := i+1;
endwhile;
return flag;
```

size, firstIndex, and s1(I), s2(1) are all hidden operations

Identical versus Observational Equivalence of Instances

- Two objects are observationally equivalent, if they "look" the same according to any sequence of access methods
- Example:
  - Specification based definition of EQN only uses access methods
    - evaluates if the two instances are observationally equivalence
  - Implementation based definition of EQN
    - evaluates if the two objects are identical in structure

How do we select the equivalent pairs?

- **Basically an infinite number of equivalent pairs**
- **Is there a subset of equivalent pairs that is sufficient?**

In general, can not determine observational equivalence with a subset of the state, must consider white box information

Example

```
ParentExample{
        if (val < 0) message(“Less”)
        else if(val==0) message(“Equal”)
        else message(“More”)}
```

```
ChildExample extendsParentExample{
        if (val < 0) message(“Less”)
        else if(val==0) message(“Zero Equal”)
        else
        {    message(“More”)
             if(val==42) message(“Jackpot”)
        }  }
```

Must Also Consider Non-Equivalent Pairs

- Equivalent pairs could be correct, but non-equivalent relationships could produce erroneous results
  - May want to assure other types of relationships
    - E.g., Bigger > Smaller
  - Certain instances may not have multiple creation paths
    - One of a kind

Some observations about ASTOOT

- Exploiting abstract data type representations
  - Assumes it is easy to create an algebraic specification
  - Basis for EQN recursive definition
  - Basis for test data generation
- Provides considerable automated support
  - Test cases generation
  - Result comparison
- Interesting way to use specifications to help derive test cases
- Interesting way to define a test oracle in terms of EQN (or other predefined relationships)
- Predecessor to JUnit approach

Issues in O-O testing

- basic unit for unit testing
- implications of encapsulation
- implications of inheritance
- implications of genericity
- implications of polymorphism/dynamic binding
- implications for testing processes

Implications of Inheritance

- inherited features often require re-testing
  - because a new context of usage results when features are inherited
- multiple inheritance increases the number of contexts to test

Which functions must be tested in a subclass?

```
class parent {
  void foo(int x);
  int range(); // returns between 1-10
}
class child extends parent {
  int range(); // returns between 1-20
  // inherits foo
}
```

- **When testing child, we need to retest range**
- **Do we need to retest foo?**

Suppose foo contained the line:

```
x = x / (20-range( ) );
```

Retesting is necessary, but maybe we don't have to retest everything

Can tests for a parent class be reused for a child class?

- parent.range() and child.range() are two different functions with different specifications and implementations
  - tests are derived from the different specifications and implementations
  - but the functions are likely to be similar, so the cleaner the OO design, the greater the overlap
- new tests are needed for child.range() requirements that are not satisfied by the parent.range test cases
  - the simpler a test, the more likely it is to be reusable in subclasses

Incremental testing of OO class structures

- **Mary Jean Harrold and John D. McGregor**
- **Exploits the inheritance hierarchy to minimize the amount of testing that must be done**

Incremental Inheritance based testing

- First test each base class (no parents)
  - Test each method
  - Test the interactions among methods
- Then consider all classes that use only previously tested classes
- Child inherents its parent's test suite
  - Used as the basis for test planning
  - Only need to develop new test cases for those entities that are directly or indirectly changed

Incremental Inheritance based testing

- Saves time
  - Reduces number of new test cases
  - Reduces execution time since there are fewer test cases
  - Reduces number of test results that need to be evaluated
- May increase the cost of selecting new test cases
  - Easily offset by reduction in human labor
- Actually a form of regression testing
  - Minimizes the number of test cases needed to exercise a modified class

Approaches to Inheritance Testing

- flattening inheritance
  - each subclass is tested as if all inherited features were newly defined
  - tests used in the super-classes can be reused
  - many tests are redundant
- incremental testing
  - limit tests only to new/modified features
  - determining what needs to be tested requires automated support

Issues in O-O testing

- basic unit for unit testing
- implications of encapsulation
- implications of inheritance
- implications of genericity
- implications of polymorphism/dynamic binding
- implications for testing processes

Example: generic (parameterized class)

```
class Table (elemType)
        int numberElements;
        create( );
        insert (elemType entry);
        delete (elemType entry);
        isempty() returns boolean;
        isentered(elemType entry) returns boolean;
endclass;
```

Testing generics

- Basically a change in the underlying structure
- Need to apply white box testing techniques that exercise this change
  - Parameterization may or may not affect the functionality of the access methods
  - In Tableclass, elemType may have little impact on the implementations of the access methods of Table
  - But, UniqueTable class would need to evaluate the equivalence of elements and this could vary depending on the representation of elemType

Example: generic (parameterized class)

```
class Table (elemType)
        int numberElements;
        create( );
        insert (elemType entry);
        delete (elemType entry);
        isempty() returns boolean;
        isentered(elemType entry) returns boolean;
endclass;
```

```
class UniqueTable extends Table
        insert(elemType entry);
endclass;
```

Issues in O-O testing

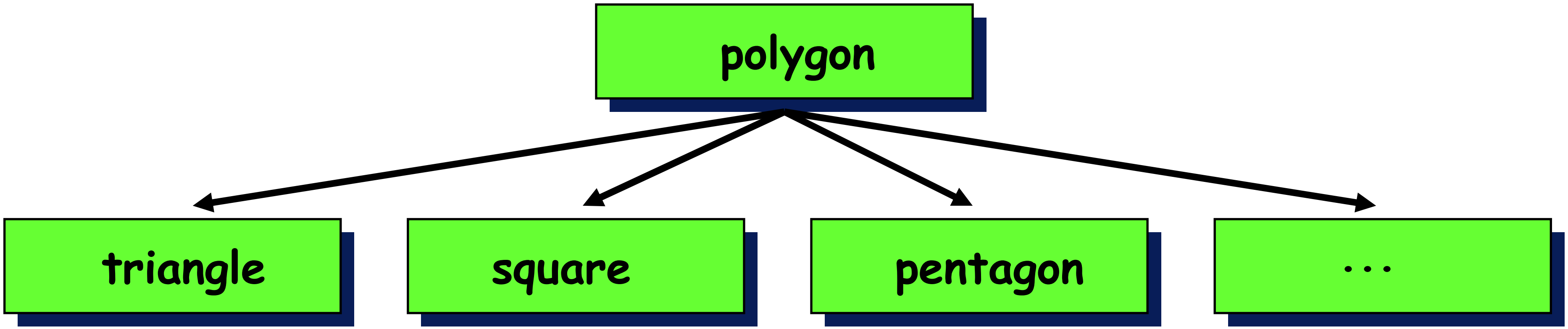
- basic unit for unit testing
- implications of encapsulation
- implications of inheritance
- implications of genericity
- implications of polymorphism/dynamic binding
- implications for testing processes

Polymorphism

- in procedural programming, procedure calls are statically bound
- each possible binding of a polymorphic component requires a separate set of test cases
  - many server classes may need to be integrated before a client class can be tested
    - E.g., t.insert would need to be tested for Table and UniqueTable

- may be hard to determine all such bindings
- complicates integration planning and testing

Example




```
void resize( )
{
...
data = polygon.area;
...
}
```

- Which implementation of area is actually called?
- Need to test all bindings

Approaches to the Dynamic Binding Problem

- Try to reduce combinatorial explosion in the number of possible combinations of polymorphic calls
  - Use static analysis (data flow analysis) to determine possible bindings
    - At most call sites, the average number of “possible” bindings is 2

Issues in O-O testing

- basic unit for unit testing
- implications of encapsulation
- implications of inheritance
- implications of genericity
- implications of polymorphism/dynamic binding
- implications for testing processes
  - Need to re-examine all testing techniques and processes

White-box vs. Black-box Testing of O-O

- In OO systems, inheritance can change both the implementation and specification
- UniqueTable example
  - Black box testing should focus on how the spec has changed
  - White box testing should focus on how the insert implementation has changed
- Jackpot in previous example shows same concerns

White box O-O Testing

- these techniques can be adapted to method testing, but are not sufficient for class testing
- conventional flow-graph approaches
  - What about flow between methods?
  - Do methods in a class have a special relationship that deserves special consideration or are standard interprocedural techniques adequate?
    - Must deal with instance variables

Black-box O-O Testing

- conventional black-box methods are useful for object-oriented systems
- Additional techniques
  - Utilize assertions specifications integrated with the implementation
    - C++ and Java assertions, Eiffel pre/post-conditions offer self-checking
  - Utilize method (event) sequence information
    - Usually don't have history of method invocations so can't do this with assertions

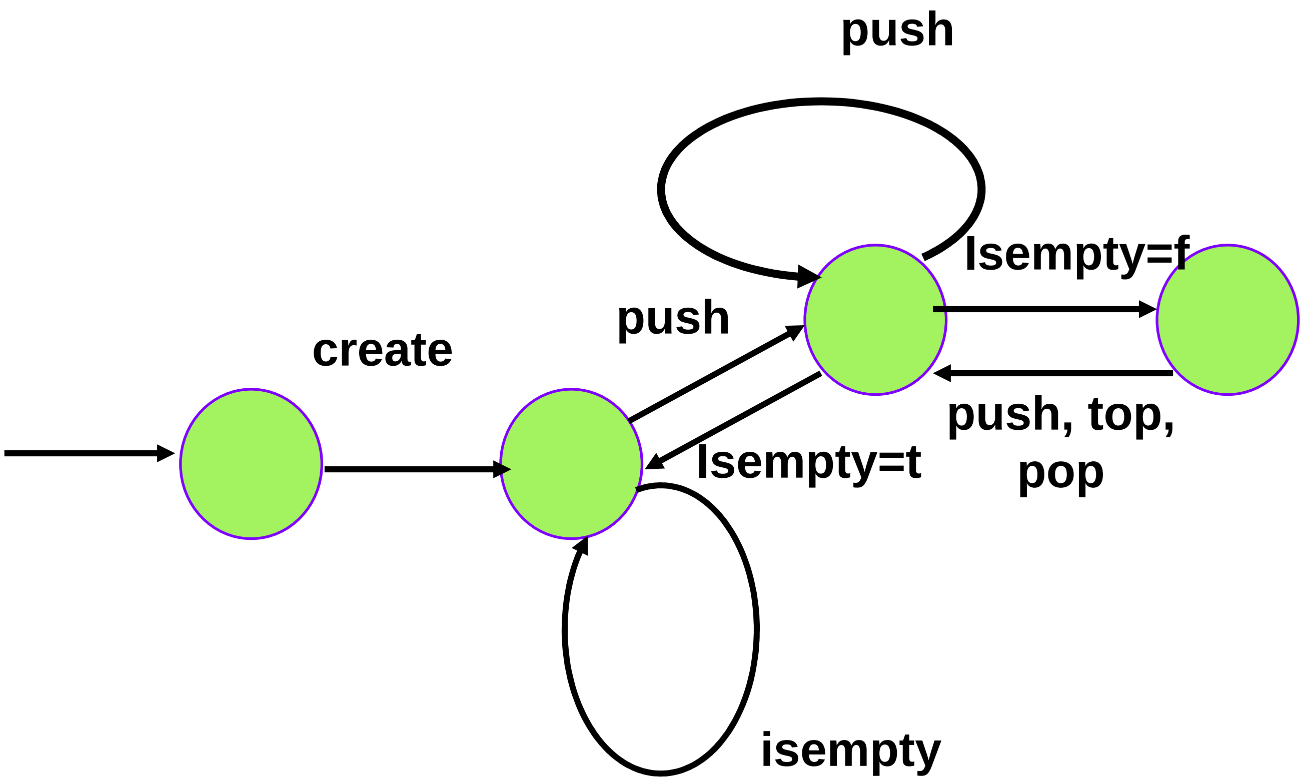
Method Invocation Model for Testing

- Consider the "implied" contract about how methods can be invoked
  - Applies to a class in isolaton
  - Applies to a cluster of classes
- Use state transition diagrams to represent the contract
  - Called a
    - State model
    - Event model

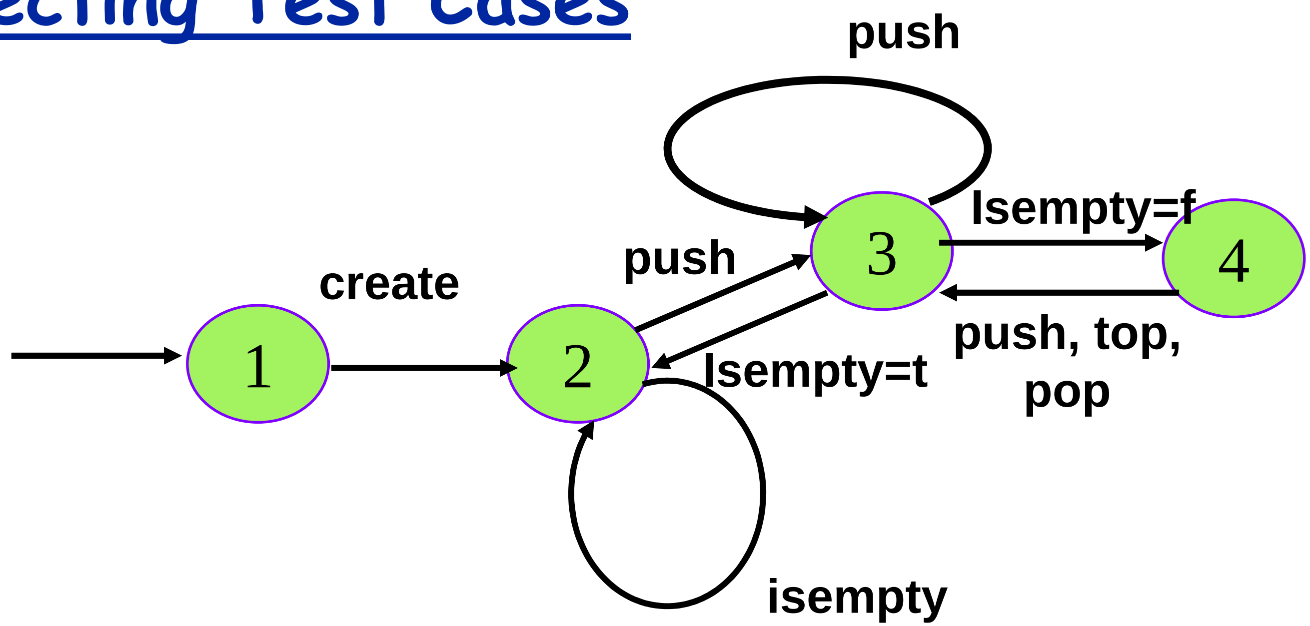
Method Invocation Model Testing

- derives test cases by modeling a class as a state machine
- methods result in state transitions
- state model defines allowable transition sequences
  - e.g., an instance must be created before it can be updated or deleted
- test cases are devised to
  - Exercise each transition
  - Exercise paths through the graph
    - Usually a small number of acyclic or simple cycle paths through the model
  - Exercise different call stacks

Example: model of a stack



Selecting Test Cases



- Each transition/method
- Each simple path
- Each unique call stack
  - Unique sequences of method calls
  - Up to a certain length
    - From the start state
    - Any subsequence

- push, top, pop
- push, pop, top
- top, pop, push
- top, push, pop
- …

Problems with Method Invocation Model Testing

- may take a lengthy sequence of operations to get an object in a desired state
- may not be productive if a class is designed to accept any possible sequence of method activation
- control may be distributed over an entire application or cluster
- system-wide control makes it difficult to verify a class in isolation
  - a global state model is needed to show how classes interact

Footprint of a “modern” OO system is very different

- More reuse
  - More contexts to test each entity
  - More unused code in a system
- More dynamism
  - Data structures
  - Dynamic binding
  - Introspection
- More method calls, exceptions, concurrency

Summary: Impact of OO on testing processes

- Affects unit testing
  - Changes what we mean by unit
  - Changes concerns
    - State of instance/class variables
    - Sequences of methods calls
      - Based on equivalence, ASTOOT
        - Applies to a single class
      - Based on a method invocation Contract
        - Applies to a single or multiple classes
    - Must test a class and its specializations
      - E.g., Harrold and McGregor

Summary: Impact of OO on the testing process (continued)

- Affects integration testing
  - Need to test component interaction
  - Need to test specific context
    - Specialized classes via inheritance and generics
- Affects regression testing
  - Changes may have greater impact because of inheritance, dynamic binding
- May not affect system testing
  - Requirements are not usually impacted

Summary: OO testing

- ADT's
  - well-defined interfaces and centralized focus help with testing
    - E.g. ASTOOT, algebraic specification based
- Inheritance and Generics
  - Increases reuse and thus reuse of test results
    - But, impact of change must be carefully assessed and taken into account
- Dynamic binding
  - Simplifies code but testing must consider all possible bindings

Summary: OO testing

- Overall, OO simplifies design and coding
  - Increases reuse
  - Reduces faults (?)
- Various OO interactions must be validated
  - Need automated support to determine these interactions
  - Need testing/analysis strategies that take these interactions into account