EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7a: OOP Testing Introduction

Creative
thinkers
made here.

Lecture Summary

The learning goals for this week are:

- To describe challenges associated with OO testing
- To describe concepts and techniques associated with OO testing

Module Expectations

How to draw a CFG for a class

- This one is easy

How OO characteristics can impact testing and what can we do

It’s a large topic for one week

- Take away the basic idea of how OO impacts testing

Learning objectives

Understand how object orientation impacts software testing

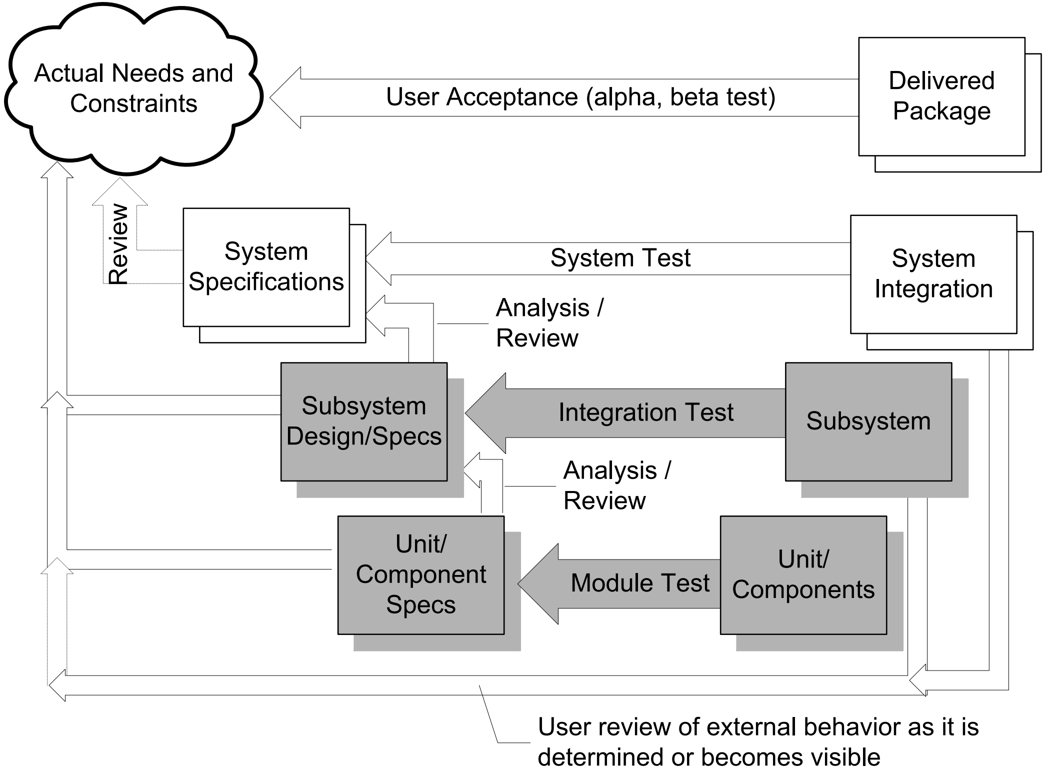
- What characteristics matter? Why?
- What adaptations are needed?
  - Understand basic techniques to cope with each key characteristic

Understand staging of unit and integration testing for OO software (intra-class and inter-class testing)

Similarities

- We begin with functional tests based on specification of intended behavior,
- add selected structural test cases based on the software structure,
- and work from unit testing and small-scale integration testing
- larger integration,
- and then system testing

Quality activities and OO SW



EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7b: OOP Terms

Creative
thinkers
made here.

OO definitions of unit and integration testing

Procedural software

- unit = single program, function, or procedure
  more often: a unit of work that may correspond to one or more intertwined functions or programs

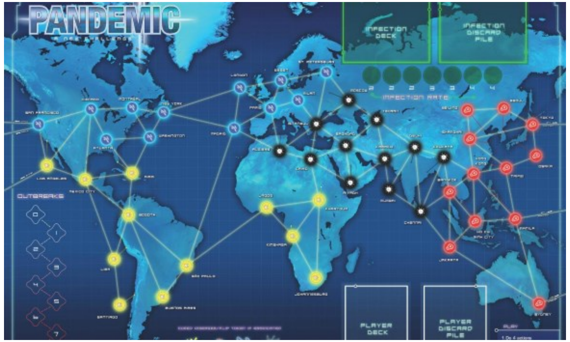
Object oriented software

- unit = class or (small) cluster of strongly related classes
  (e.g., sets of Java classes that correspond to exceptions)
- unit testing = **intra-class testing**
- integration testing = **inter-class testing** (cluster of classes)

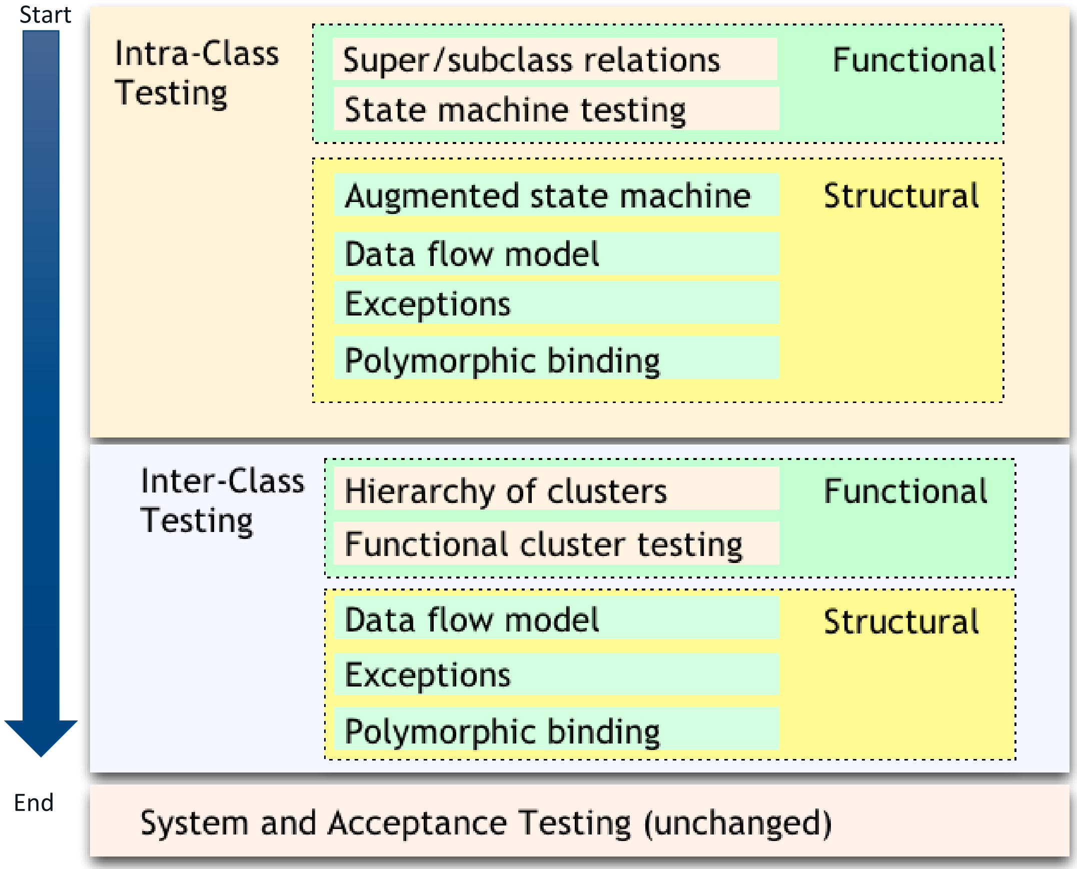
- dealing with single methods separately is usually too expensive (complex scaffolding), so methods are usually tested in the context of the class they belong to

OO definitions of unit and integration testing

- unit testing = **intra-class testing**
  - **INSIDE THE CLASS**
- integration testing = **inter-class testing** (cluster of classes)
  - Interaction of many classes
  - Think of traveling interstate or international!



Orthogonal approach: Stages



Start

Intra-Class Testing

Super/subclass relations

State machine testing

Functional

Augmented state machine

Data flow model

Exceptions

Polymorphic binding

Structural

Inter-Class Testing

Hierarchy of clusters

Functional cluster testing

Functional

Data flow model

Exceptions

Polymorphic binding

Structural

End

System and Acceptance Testing (unchanged)

EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7c: OOP Considerations State Based Behaviour

Creative
thinkers
made here.

Intraclass State Machine 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*.

Identifying states and transitions

From the informal specification we can identify three states:

- Not_installed
- Unbound
- Bound

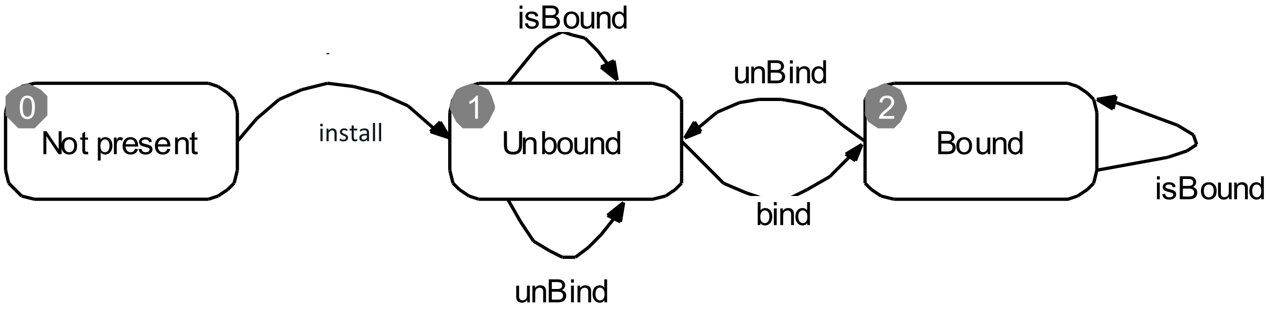
and four transitions

- 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: install, isBound, bind, isBound

TC-2: install, unBind, bind, unBind, isBound



Testing with State Diagrams

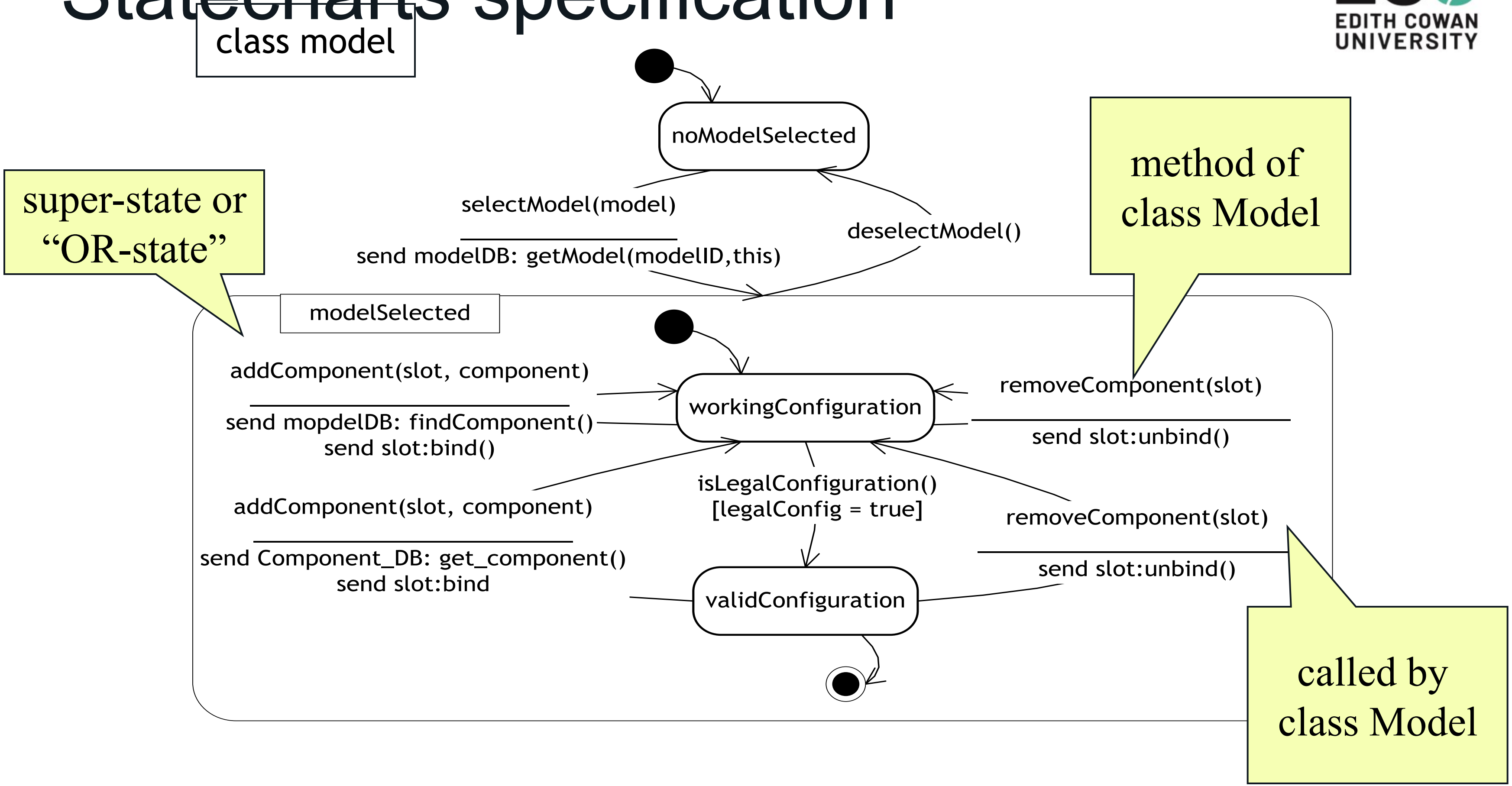
A statechart (called a “state diagram” in UML) may be produced as part of a specification or design

  - May also be implied by a set of message sequence charts (interaction diagrams), or other modeling formalisms

Two options:

- Convert (“flatten”) into standard finite-state machine, then derive test cases
- Use state diagram model directly

Statecharts specification

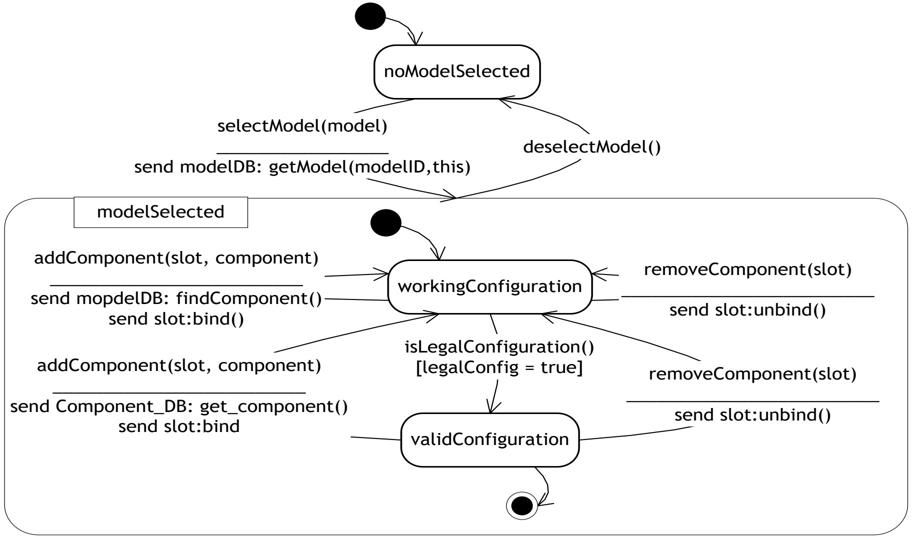


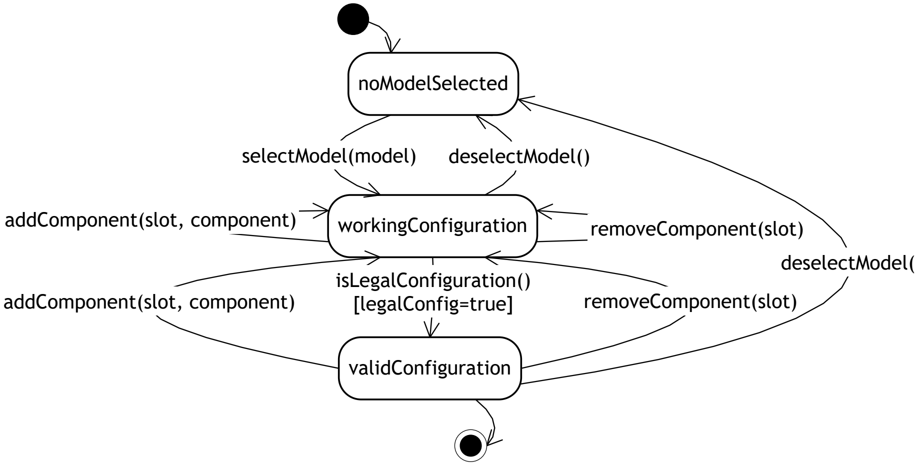
super-state or “OR-state”

method of class Model

called by class Model

From Statecharts to FSMs





From Statecharts to FSMs

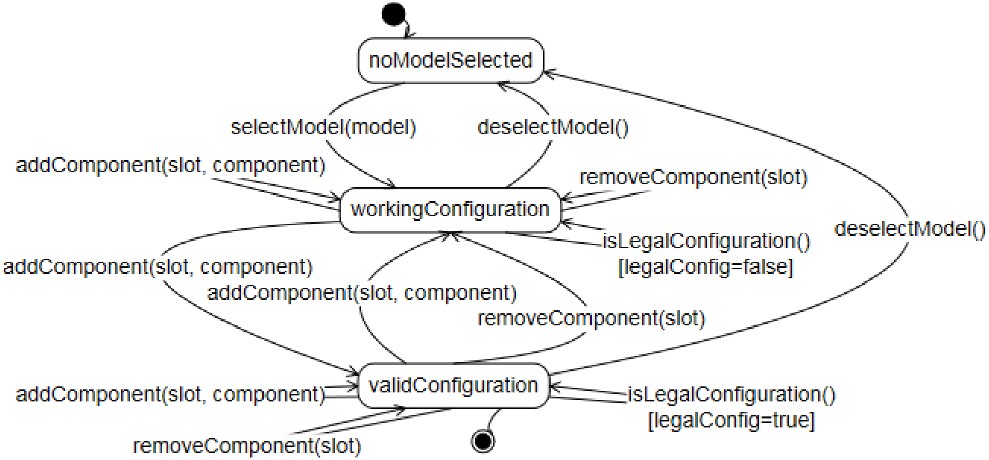
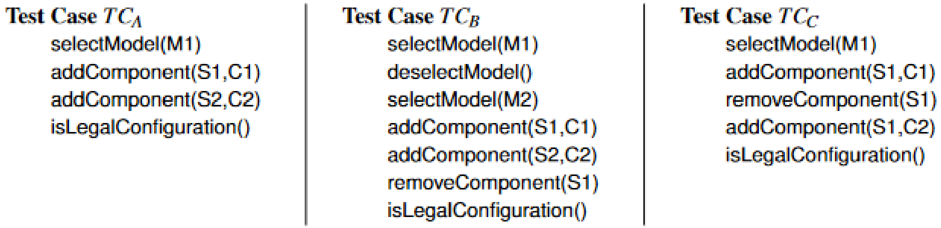


Figure 15.7: Finite state machine corresponding to the statechart of Figure 15.6.

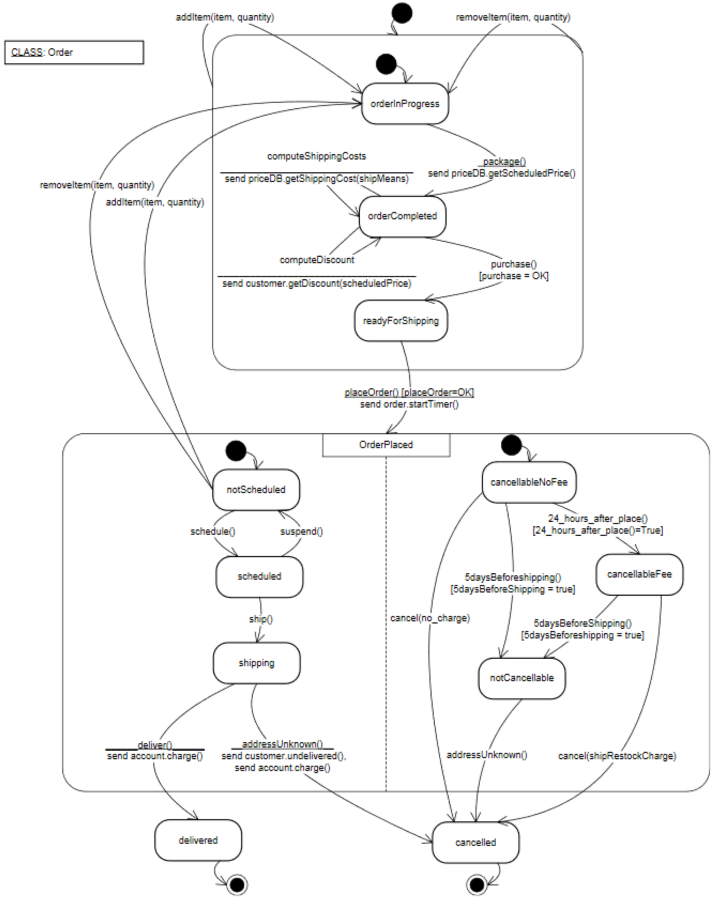


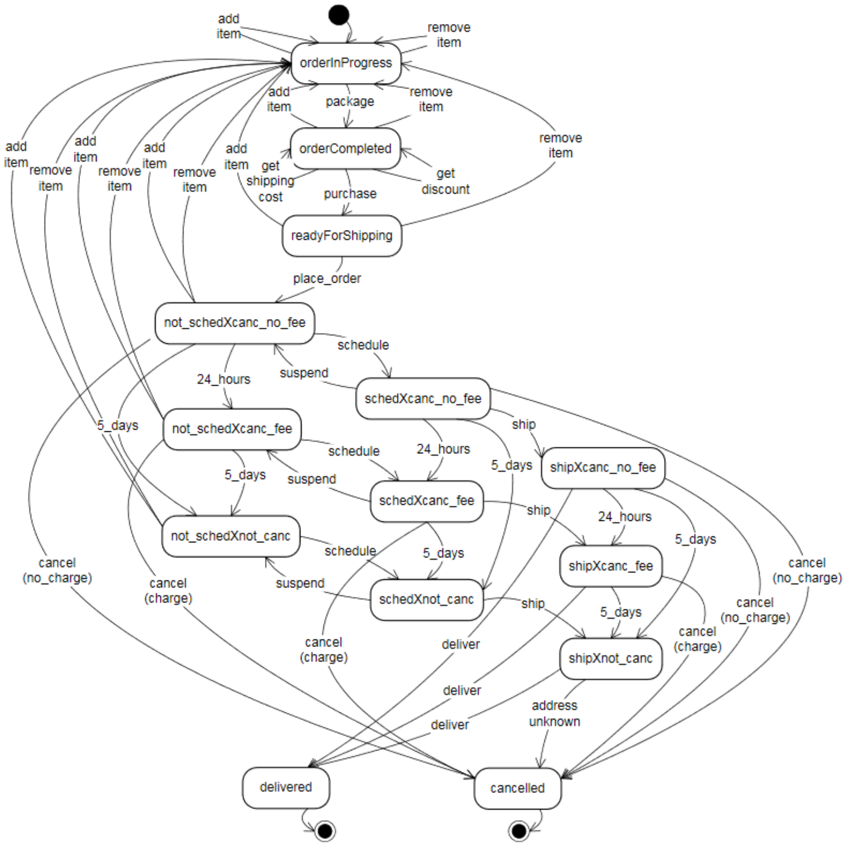
Test Case TC_A
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Test Case TC_B
selectModel(M1)
deselectModel()
selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
removeComponent(S1)
isLegalConfiguration()

Test Case TC_C
selectModel(M1)
addComponent(S1,C1)
removeComponent(S1)
addComponent(S1,C2)
isLegalConfiguration()

From Statecharts to FSMs





Statechart based criteria

In some cases, “flattening” a Statechart to a finite-state machine may cause “state explosion”

- Particularly for super-states with “history”

Alternative: Use the statechart directly

Simple transition coverage:
execute all transitions of the original Statechart

- incomplete transition coverage of corresponding FSM
- useful for complex statecharts and strong time constraints (combinatorial number of transitions)

EDITH COWAN
UNIVERSITY

CSI3105: Software Testing

Module 7d: OOP Considerations Encapsulation

Creative
thinkers
made here.

Characteristics of OO Software

Encapsulation

Encapsulating or hiding variables with the private keyword, adds in some complexity

We need to consider how to handle hidden or encapsulated fields.

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*

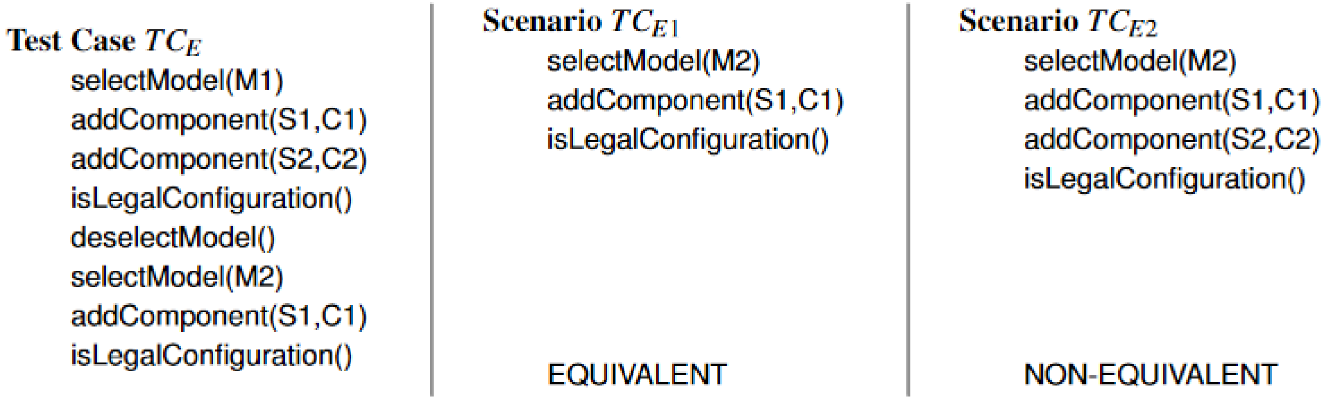
Equivalent scenarios approach:

- generate equivalent and non-equivalent sequences of method invocations
- compare the final state of the object after equivalent and non-equivalent sequences

Equivalent Scenarios Approach

Encapsulation….The state of model is hidden..

What do we do?



Test Case TC_E	Scenario TC_{E1}	Scenario TC_{E2}
selectModel(M1)	selectModel(M2)	selectModel(M2)
addComponent(S1,C1)	addComponent(S1,C1)	addComponent(S1,C1)
addComponent(S2,C2)	isLegalConfiguration()	addComponent(S2,C2)
isLegalConfiguration()		isLegalConfiguration()
deselectModel()		
selectModel(M2)		
addComponent(S1,C1)		
isLegalConfiguration()		
	EQUIVALENT	NON-EQUIVALENT

Verify equivalence

In principle: Two states are equivalent if all possible sequences of methods starting from those states produce the same results

Practically:

add inspectors that disclose hidden state and compare the results

- break encapsulation

examine the results obtained by applying a set of methods

- approximate results

add a method "compare" that specializes the default *equal* method

- design for testability

EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7e: OOP Considerations Inheritance

Creative thinkers made here.

Characteristics of OO Software

Inheritance

The effects of new and overridden methods on the behavior of inherited methods

We need to evaluate if the inherited methods from their ancestors need new test cases

- can use the same test cases as its ancestor or if this method doesn’t need to be retested.

Inheritance

When testing a subclass ...

- We would like to re-test only what has not been thoroughly tested in the parent class
- But we should test any method whose behavior may have changed

Reusing Tests with the Testing History Approach

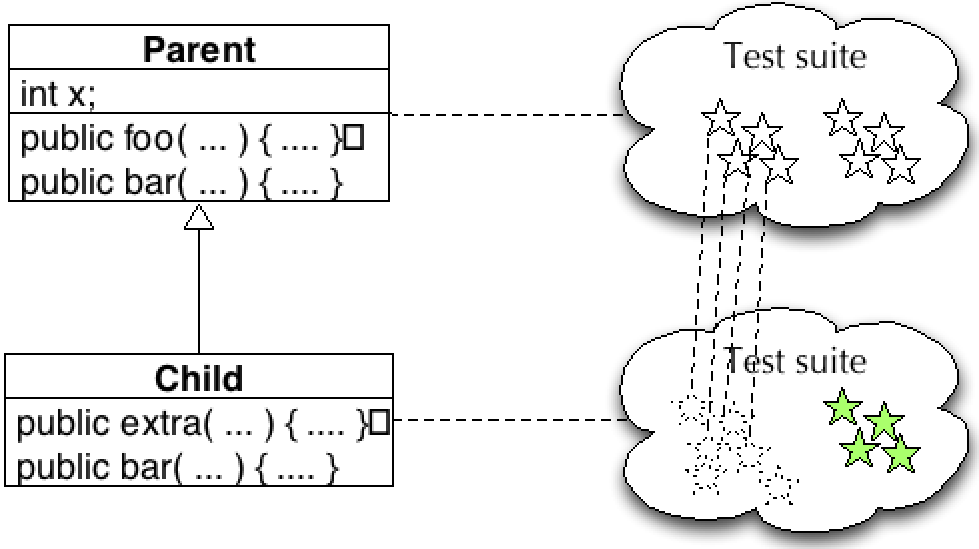
Track test suites and test executions

- determine which new tests are needed
- determine which old tests must be re-executed

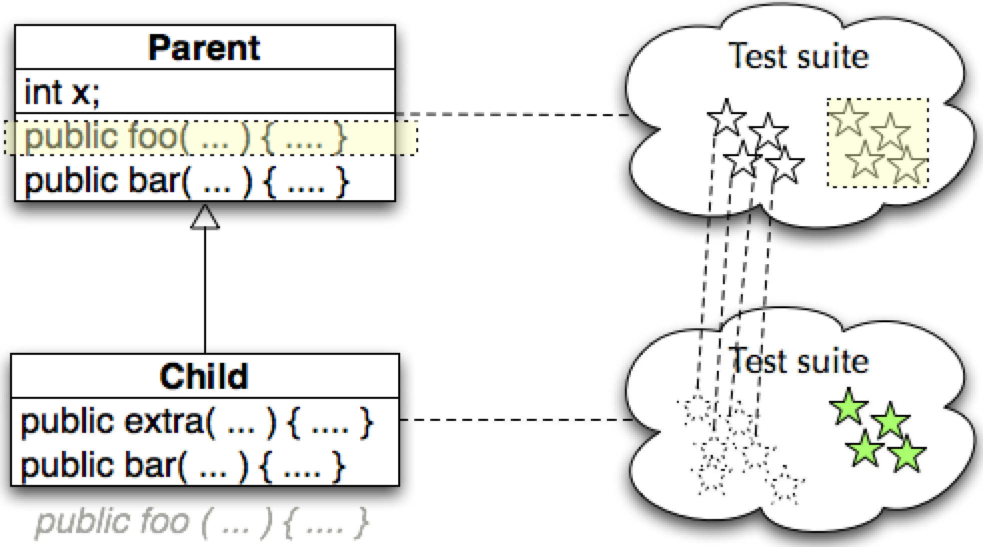
New and changed behavior ...

- 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

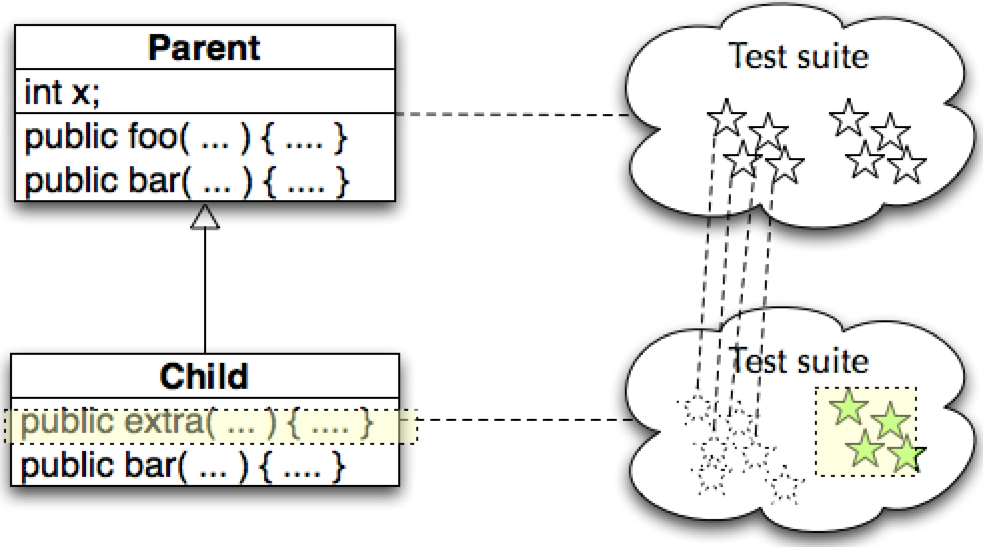

Parent
int x;
public foo(...) { .... }□
public bar(...) { .... }
Test suite
Child
public extra(...) { .... }□
public bar(...) { .... }
Test suite

Inherited, unchanged


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
public foo (...) { .... }
Test suite
Test suite

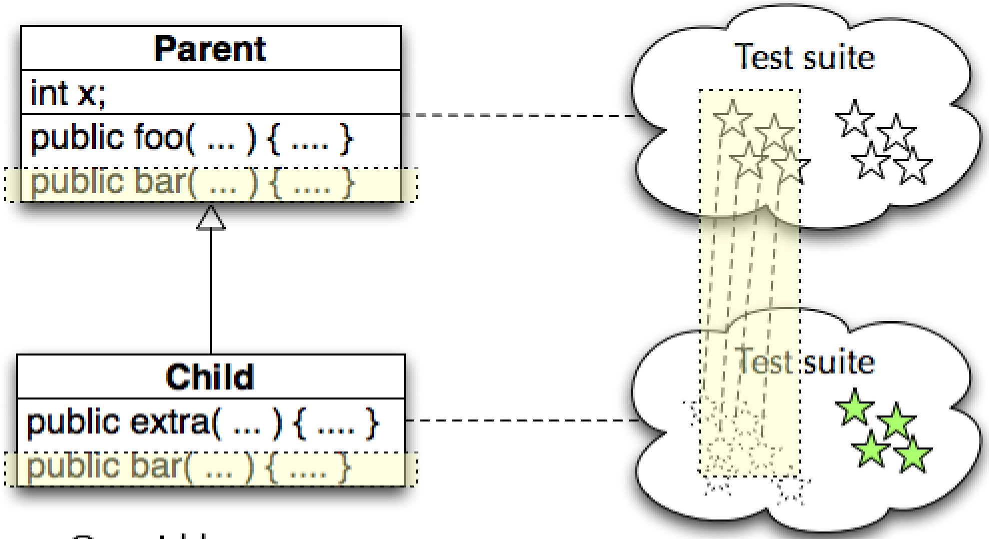
Inherited, unchanged ("recursive"):
No need to re-test

Newly introduced methods


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

New:
Design and execute new test cases

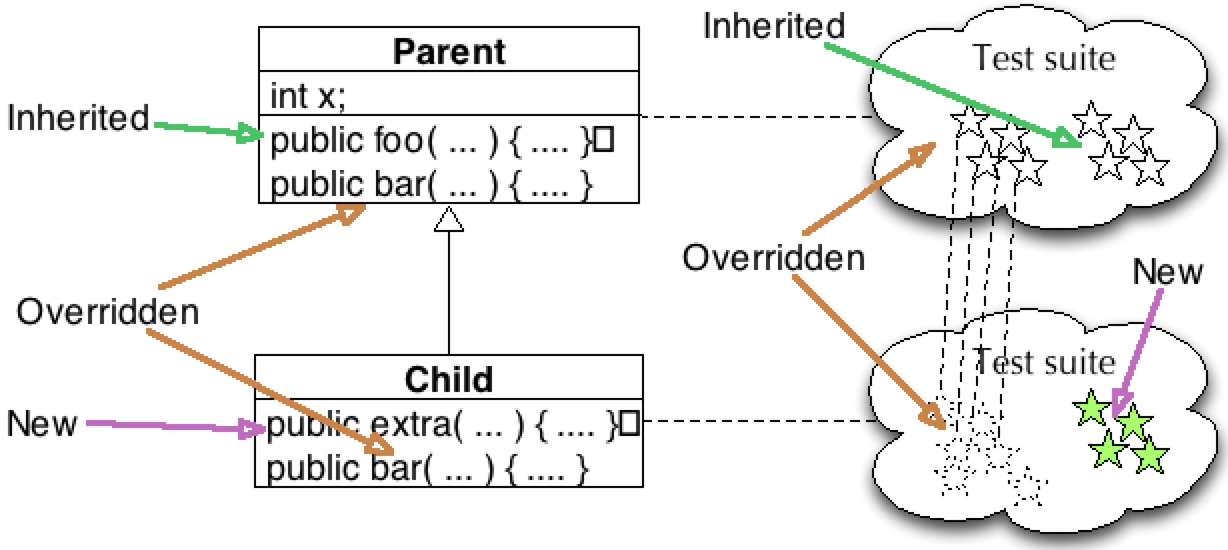
Overridden methods


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

Overridden:

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

Testing History - Summary


Inherited
Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Overridden
New
Child
public extra(...) { .... }
public bar(...) { .... }
Inherited
Test suite
Overridden
New
Test suite

EDITH COWAN
UNIVERSITY

CSI3105:
Software Testing

Module 7f: OOP Considerations Polymorphism

Creative
thinkers
made here.

Characteristics of OO Software

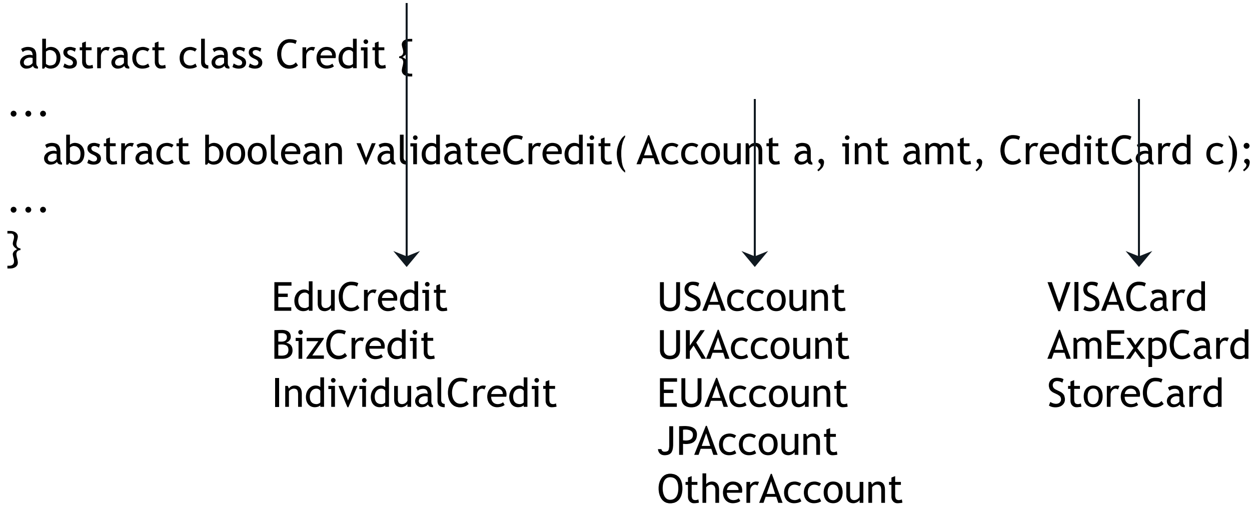
Polymorphism and dynamic binding

A single method call may be dynamically bound to different methods depending on the state of the computation.

Overridden method that acts different based on the parameters it receives.

Tests must exercise different bindings to reveal failures that depend on a particular binding or on interactions between bindings for different calls.

“Isolated” calls: the combinatorial explosion problem



```
 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!)

The combinatorial approach

Identify a set of combinations that cover all pairwise combinations of dynamic bindings

Same motivation as pairwise specification-based testing (module 8, next week)

Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit	VISACard
JPAccount	EduCredit	VISACard
JPAccount	BizCredit	ChipmunkCard
JPAccount	individualCredit	AmExpCard
OtherAccount	EduCredit	ChipmunkCard
OtherAccount	BizCredit	VISACard
OtherAccount	individualCredit	AmExpCard

EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7g: OOP Considerations Abstraction

Creative thinkers made here.

Characteristics of OO Software

Abstract and generic classes

Some classes in an object-oriented program are intentionally left incomplete and abstract classes cannot be directly instantiated

Abstract classes must be extended through subclasses

Testing generic classes

a generic class

```
class PriorityQueue<Elem Implements Comparable> {...}
```

is designed to be instantiated with many different parameter types

```
PriorityQueue<Customers>
PriorityQueue<Tasks>
```

A generic class is typically designed to behave consistently some set of permitted parameter types.

Testing can be broken into two parts

- Showing that some instantiation is correct
- showing that all permitted instantiations behave consistently

EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7h: OOP Considerations Exceptions

Creative
thinkers
made here.

Characteristics of OO Software

Exception handling

- Attempt to handle run time errors
- separate handling of error cases from the primary program logic

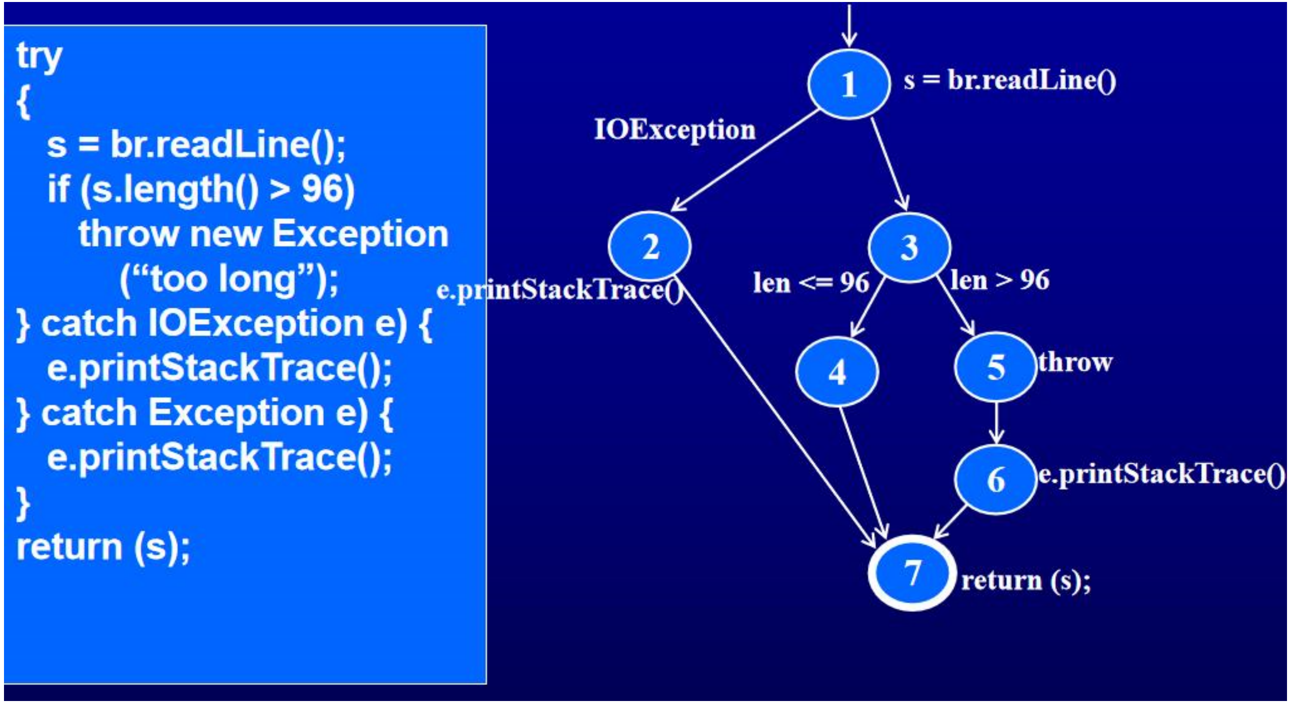
Try catch block etc..

- Alters the flow of code

We need to..

- Test where exceptions are thrown
- Test where exceptions are handled
- Test for unhandled exceptions

Characteristics of OO Software



```
try
{
   s = br.readLine();
   if (s.length() > 96)
      throw new Exception
         ("too long");
} catch IOException e) {
   e.printStackTrace();
} catch Exception e) {
   e.printStackTrace();
}
return (s);
```

Characteristics of OO Software

Since testing all chains through which exceptions can propagate is impractical, it is best to make it unnecessary..

Design test cases to exercise each point at which an exception is explicitly thrown by application code, and each handler in application code, but not necessarily all their combinations.

Testing exception handling

What we do test: Each exception handler, and each explicit throw or re-throw of an exception

EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7i: OOP Drawing CFGs

Creative thinkers made here.

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

+

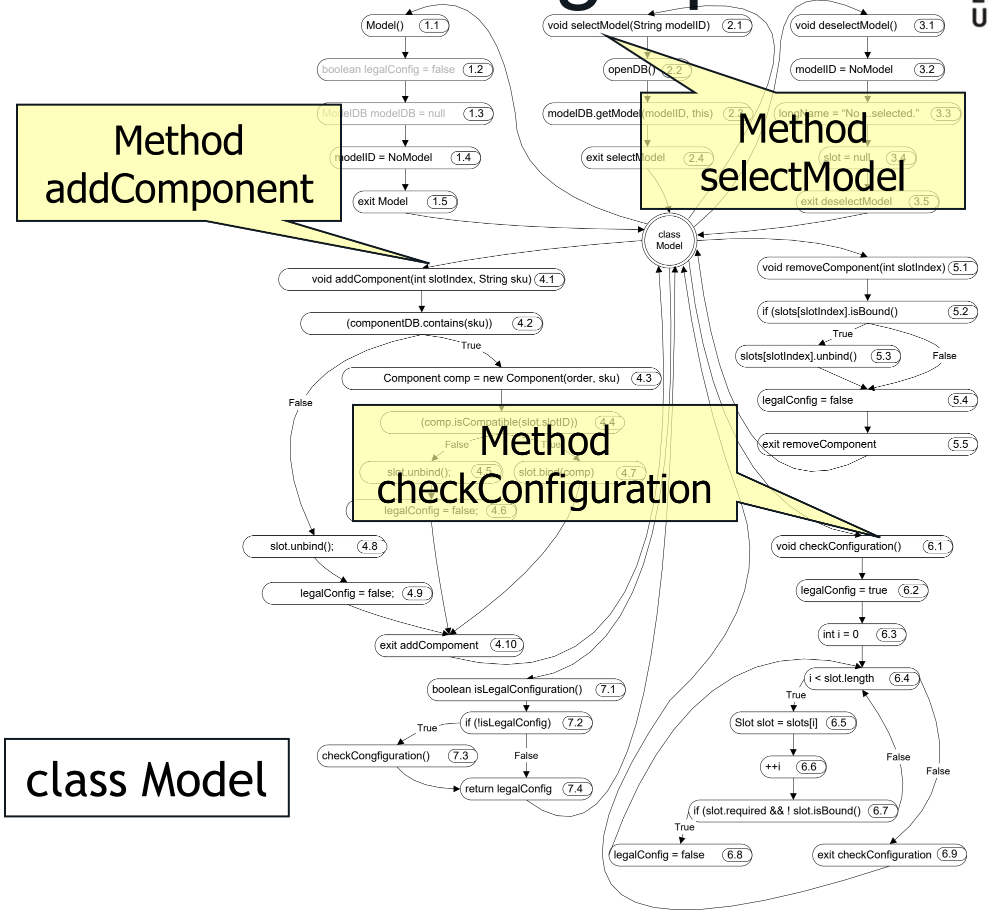
node for class

+

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



Model() 1.1
boolean legalConfig = false 1.2
ModelDB modelDB = null 1.3
modelID = NoModel 1.4
exit Model 1.5
void selectModel(String modelID) 2.1
openDB() 2.2
modelDB.getModel(modelID, this) 2.3
exit selectModel 2.4
void deselectModel() 3.1
modelID = NoModel 3.2
longName = "No ...selected." 3.3
slot = null 3.4
exit deselectModel 3.5
class Model
void addComponent(int slotIndex, String sku) 4.1
(componentDB.contains(sku)) 4.2
True
False
Component comp = new Component(order, sku) 4.3
(comp.isCompatible(slot.slotID)) 4.4
False
True
slot.unbind(); 4.5
slot.bind(comp) 4.7
legalConfig = false; 4.6
slot.unbind(); 4.8
legalConfig = false; 4.9
exit addCompoment 4.10
void removeComponent(int slotIndex) 5.1
if (slots[slotIndex].isBound() 5.2
True
slots[slotIndex].unbind() 5.3
False
legalConfig = false 5.4
exit removeComponent 5.5
void checkConfiguration() 6.1
legalConfig = true 6.2
int i = 0 6.3
i < slot.length 6.4
True
Slot slot = slots[i] 6.5
++i 6.6
False
False
if (slot.required && ! slot.isBound() 6.7
True
legalConfig = false 6.8
exit checkConfiguration 6.9
boolean isLegalConfiguration() 7.1
if (!isLegalConfig) 7.2
True
checkCongfiguration() 7.3
False
return legalConfig 7.4

Method addComponent

Method selectModel

Method checkConfiguration

class Model

Definition-Use (DU) pairs

instance variable **legalConfig**

<model (1.2), isLegalConfiguration (7.2)>
<addComponent (4.6), isLegalConfiguration (7.2)>
<removeComponent (5.4), isLegalConfiguration (7.2)>
<checkConfiguration (6.2), isLegalConfiguration (7.2)>
<checkConfiguration (6.3), isLegalConfiguration (7.2)>
<addComponent (4.9), isLegalConfiguration (7.2)>

Each pair corresponds to a test case
note that

- some pairs may be infeasible
- to cover pairs we may need to find complex sequences

EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 7j: OOP Considerations Scaffolding

Creative
thinkers
made here.

Nearly There..

Other OOP Considerations

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*

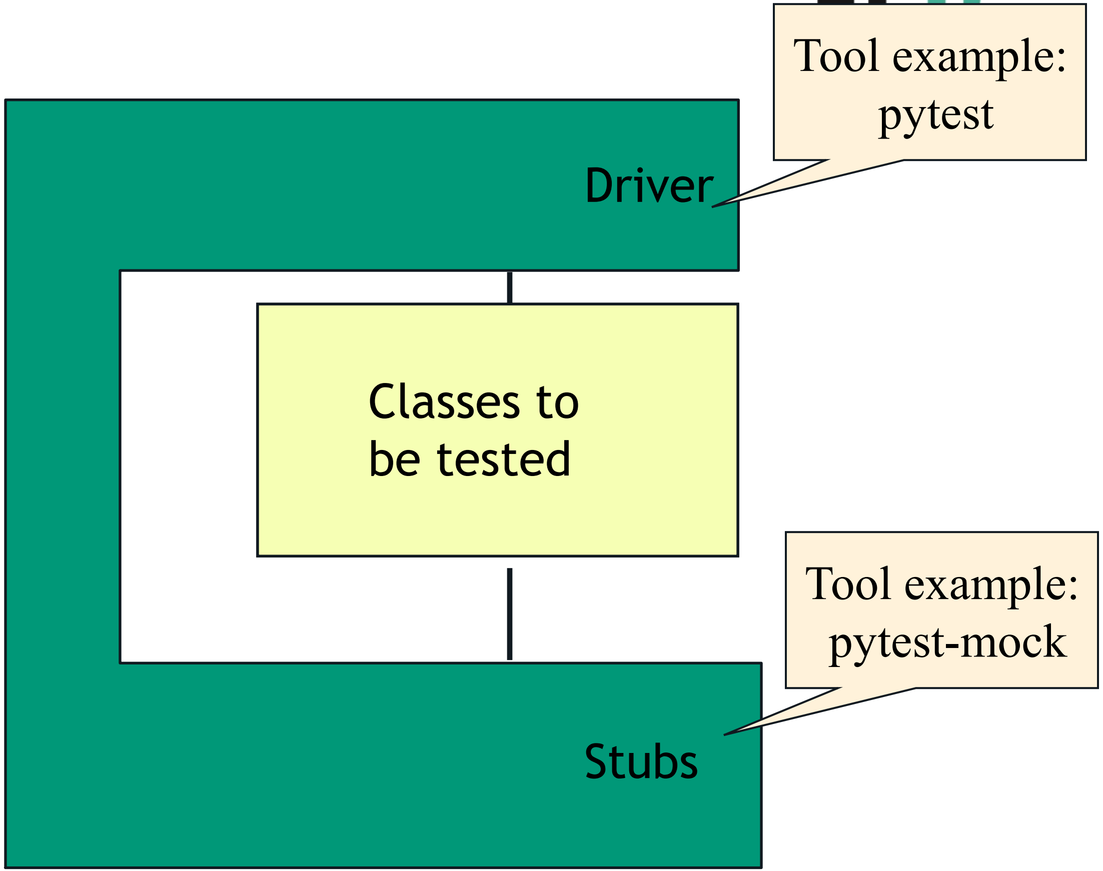
Scaffolding

Used to create test dependencies like drivers and stubs.

•**pytest-mock** – Provides integration with unittest.mock to replace real functions or objects.

•**Monkeypatching (monkeypatch.setattr)** – Temporarily replaces attributes or methods for testing.

This setup ensures **unit isolation** and allows us to **test components independently** of external dependencies.



Approaches

Requirements on scaffolding approach: Controllability and Observability

General/reusable scaffolding

- Across projects; build or buy tools

Project-specific scaffolding

- Design for test
- Ad hoc, per-class or even per-test-case

Usually a combination

Summary

Object-oriented testing can be broken into three phases, progressing from individual classes toward consideration of integration and interactions.

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