

Testing Object-Oriented Systems (Part 2)

CSCE 747 - Lecture 20 - 03/23/2017

Testing Object-Oriented Software

- Testing of OO systems is impacted by
 - State Dependent Behavior
 - Encapsulation
 - Inheritance
 - Polymorphism and Dynamic Binding
 - Abstract Classes
 - Exception Handling
 - Concurrency
- To test such systems, we must test both individual classes and groups of related classes.

Intraclass Testing

To test a class in isolation, we:

1. If the class is abstract, derive a set of instantiations to cover significant cases.
2. Design test cases to check correct invocation of inherited and overridden methods.
3. Design a set of test cases based on the states that the class can be put into.
 - a. Build a state machine model based on the class.

Intraclass Testing

4. Derive structural information from the source code (control and data-flow) and cover the code structure of the class.
5. Design test cases for exception handling.
 - a. Exercising exceptions that should be thrown by methods in the class and exceptions that should be caught and handled by them.
6. Design test cases for polymorphic calls.
 - a. Calls to superclass or interface methods that can be bound to different subclass objects.

Interclass Testing

1. Identify a hierarchy of classes to be tested incrementally.
2. Design a set of interclass test cases for the cluster-under test.
3. Add test cases to cover data flow between method calls.
4. Integrate the intraclass exception-handling tests with interclass exception-handling tests.
5. Integrate polymorphism test suite with tests that check for interclass interactions.

Structural Testing for Classes

Structural Testing of Classes

- Main difference for functional testing - central role of object state.
 - Sequences of method calls are needed to manipulate object state.
- Structural techniques must extend control and data flow across sequences of method calls.

```
public class Model extends Orders.CompositeItem{
    public String modelID;
    private int baseWeight;
    private int heightCm, widthCM, depthCM;
    private Slot[] slots;
    private boolean legalConfig = false;
    private static final String NoModel = "NO
MODEL SELECTED";

    private void checkConfiguration(){
        ...
    }

    public boolean isLegalConfiguration(){
        if(!legalConfig){
            this.checkConfiguration();
        }
        return legalConfig;
    }
}
```

Structural Testing of Classes

- One approach - add variables in the code, but not the specification to the state machine.
 - Adding a boolean variable = 2x states, 4x the transitions.
- What is important is not the *values* of the variables, but tracking *how values are changed*

```
public class Model extends Orders.CompositeItem{
    public String modelID;
    private int baseWeight;
    private int heightCm, widthCM, depthCM;
    private Slot[] slots;
    private boolean legalConfig = false;
    private static final String NoModel = "NO
MODEL SELECTED";

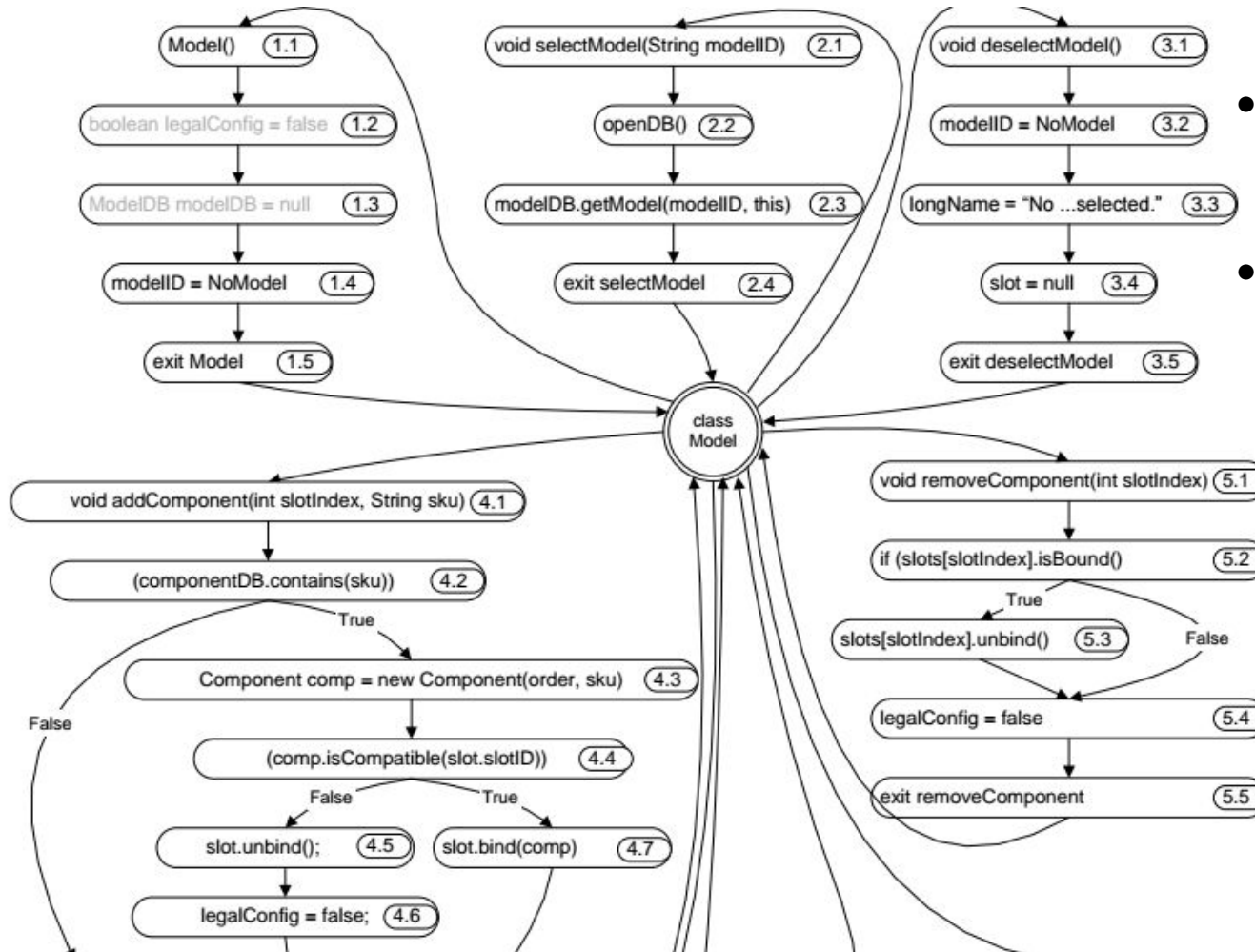
    private void checkConfiguration(){
        ...
    }

    public boolean isLegalConfiguration(){
        if(!legalConfig){
            this.checkConfiguration();
        }
        return legalConfig;
    }
}
```


Intraclass Structural Testing

- Need to track definitions and uses across multiple methods instead of just one.
- Derive this information from the source code.
- Create a control-flow graph across the whole class.
 - Create control-flow graphs for each method.
 - Add a central node representing the class itself.
 - Edges from node Class to the entry of each method
 - Edge from exit of each method to node Class.
 - Add global declarations to the constructor's CFG.

Intraclass CFG



- For now - treat calls to methods of other classes as simple statements.
- Need a strategy for arrays and objects, like with one-method data-flow.

Structural Testing of Classes

- A test case to exercise a DU pair is a sequence of method calls that starts with the constructor and includes a definition-clear path.
- Can use All DU Pair, All DU Path, All Definitions coverage metrics.
- Covers variables not mentioned in the specification (or modeled in state machine)

```
public class Model extends Orders.CompositeItem{
    private boolean legalConfig = false;

    private void checkConfiguration() {}
    // 2 definitions

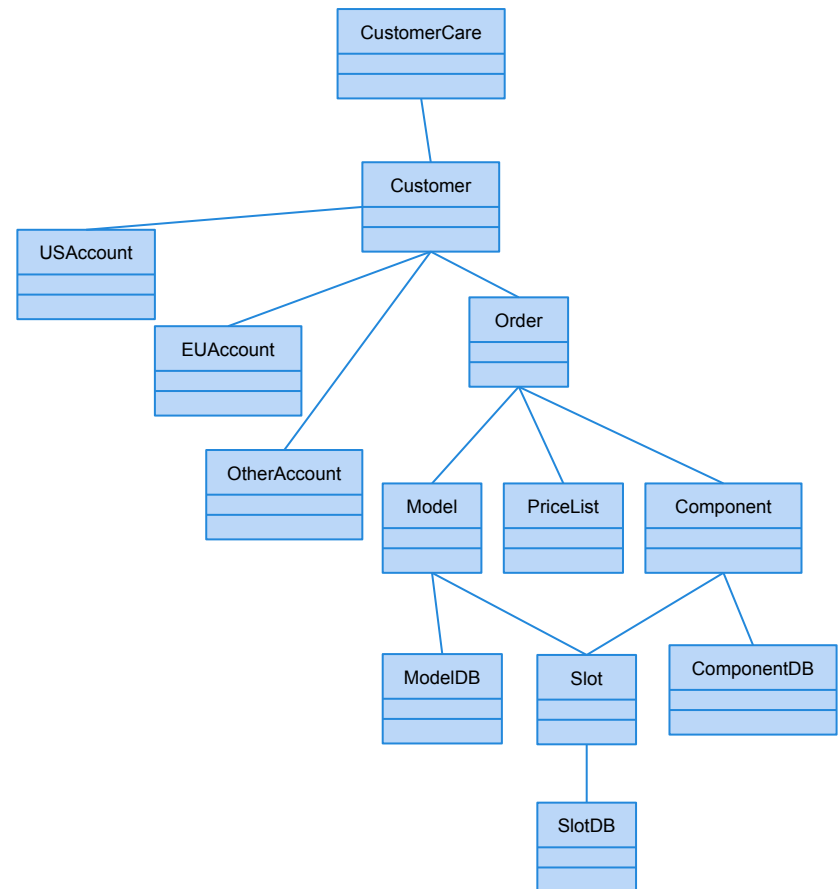
    public boolean isLegalConfiguration(){
        if(!legalConfig){
            this.checkConfiguration();
        }
        return legalConfig;
    }

    public void addComponent(int slotIndex,
String sku){}    // 2 definitions

    public void removeComponent(int
slotIndex){}    // 1 definition
}
```

Interclass Structural Testing

- Follow the dependence relation.
- Test the leaf nodes in isolation, using interclass methods.
- Examine how each class can define and use variables from its leaves.
 - Classify inspectors, modifiers, inspector/modifier methods.



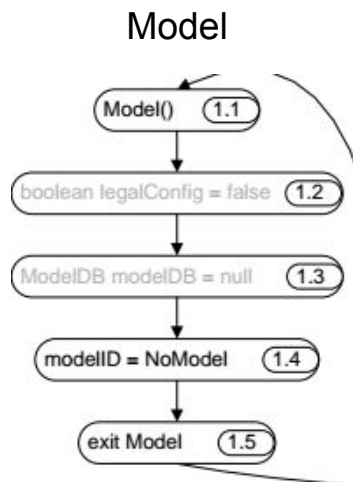
Interclass Structural Testing

- **Inspectors**
 - Methods that access, but do not modify the state.
 - Uses, no definitions.
- **Modifiers**
 - Methods that modify, but do not access the state.
 - Definitions, no uses.
- **Inspector/Modifiers**
 - Methods that both define and use variables.
- **Other methods do not need to be considered in interclass structural testing.**

Interclass Structural Testing

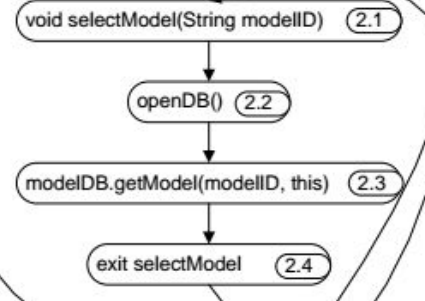
- Consider the whole object state when classifying.
 - Method is an inspector/modifier, even if it inspects one variable and modifies another.
 - Important for improving scalability.
- If a method has multiple execution paths
 - Can classify whole method.
 - Or split into separate paths if they would have different classifications.

Example - Model



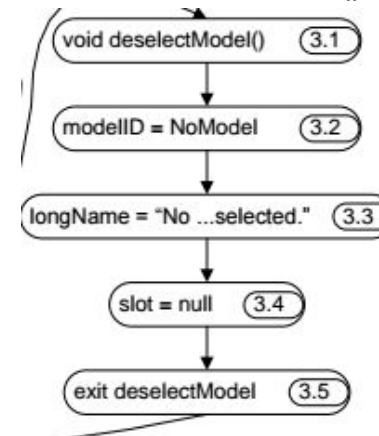
modifier

void selectModel(String modelID)



modifier

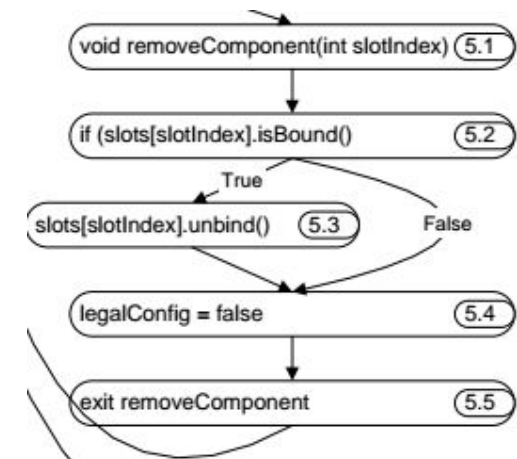
void deselectModel()



modifier

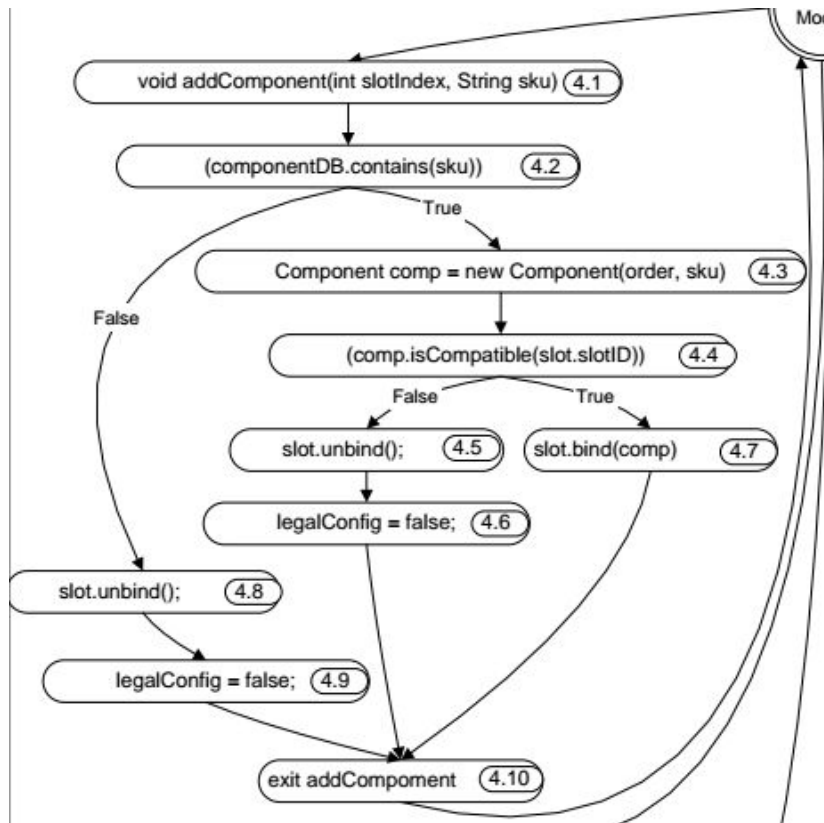
Path 1: 1-2-3-4-5: inspector/modifier
Path 2: 1-2-4-5: inspector/modifier

void removeComponent(int slotIndex)



Example - Model

```
void addComponent(int slotIndex, String sku)
```



Paths:

1-2-3-4-7-10

1-2-3-4-5-6-10

1-2-8-9-10

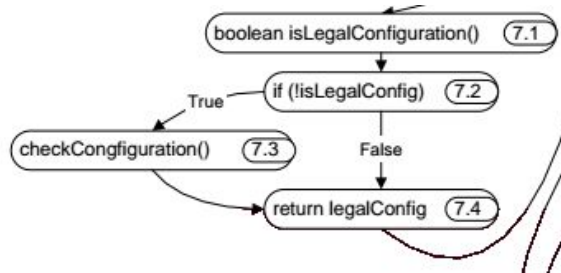
Path 1: inspector/modifier

Path 2: inspector/modifier

Path 3: inspector/modifier

Example - Model

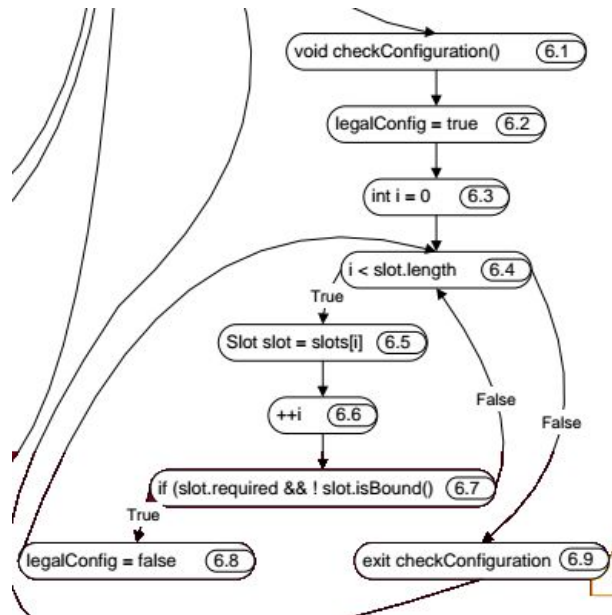
boolean isLegalConfiguration()



Paths:

- 1: 1-2-3-[1-2-3-4-5-6-7-8-4-9]-4
- 2: 1-2-3-[1-2-3-4-5-6-7-4-9]-4
- 3: 1-2-3-[1-2-3-4-9]-4
- 4: 1-2-4

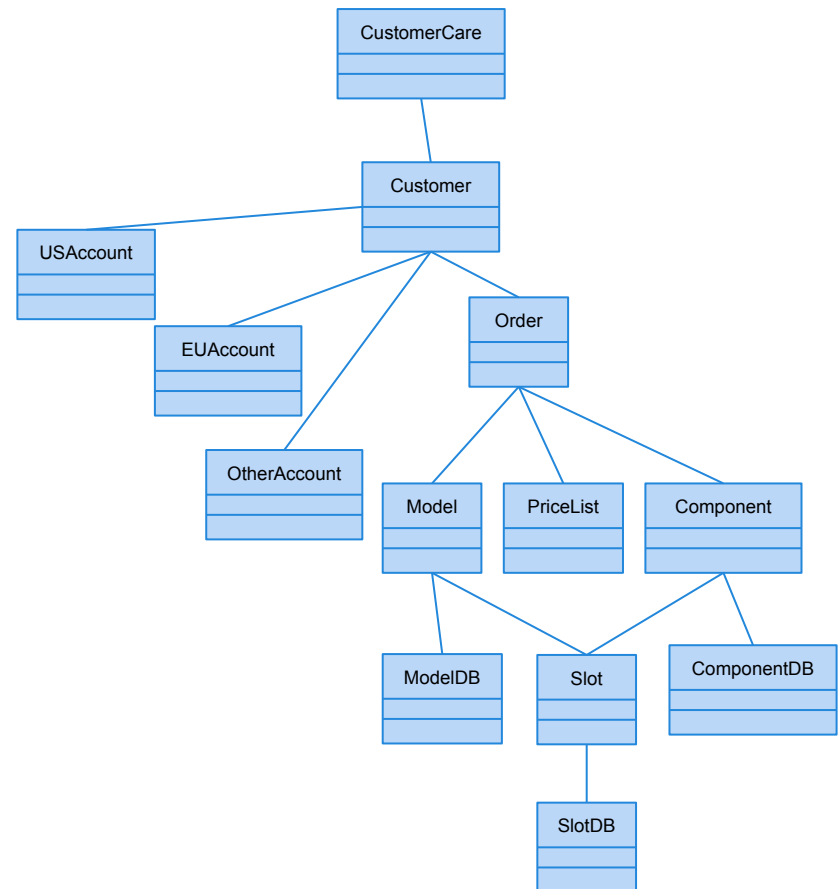
void checkConfiguration()



Path 1: inspector/modifier
Path 2: inspector/modifier
Path 3: inspector/modifier
Path 4: modifier

Interclass Structural Testing

- Test classes that use or contain leaf classes.
- Invocations of modifiers and inspector/modifiers are treated as definitions
- Invocations of inspectors and inspector/modifiers are treated as uses.
- Analyze classes that depend on classes already analyzed.



Addressing OO Testing Issues

Oracles for Classes

- The correctness of a method is not just judged on the output of the method, but on the state of the object.
 - `deselectModel()` should clear the array slots on the object.
- Oracles must check the validity of both output and state.
- State may not be directly accessible.
 - Private variables.

Option 1: Modify the Code

- Break encapsulation by making variables public while testing.
 - Risk - different behavior between testing and production code.
 - C++ has friend classes
- Add “getter” methods.
- Add a method that produces a representation of the entire state of the object.
 - `object.toString()` in Java.

Option 2: Java Reflection

- Reflection allows Java code to inspect objects at runtime.
- Can be used to identify their class, fields, and methods, and use them to perform tasks.

```
Method[] methods = MyObject.class.getMethods();  
  
for(Method method : methods){  
    System.out.println("method = " + method.getName());  
}
```

- This code gets the class and prints out the list of methods.

Option 2: Java Reflection

- Reflection can be used to access private fields and methods.
- Protects the real object from modification, but can be used to get information for testing.

```
public class PrivateObject {  
  
    private String privateString = null;  
  
    public PrivateObject(String privateString) {  
        this.privateString = privateString;  
    }  
}  
  
PrivateObject privateObject = new  
PrivateObject("The Private Value");  
  
Field privateStringField =  
PrivateObject.class.getDeclaredField("privateStrin  
g");  
privateStringField.setAccessible(true);  
  
String fieldValue = (String)  
privateStringField.get(privateObject);  
System.out.println("fieldValue = " + fieldValue);
```

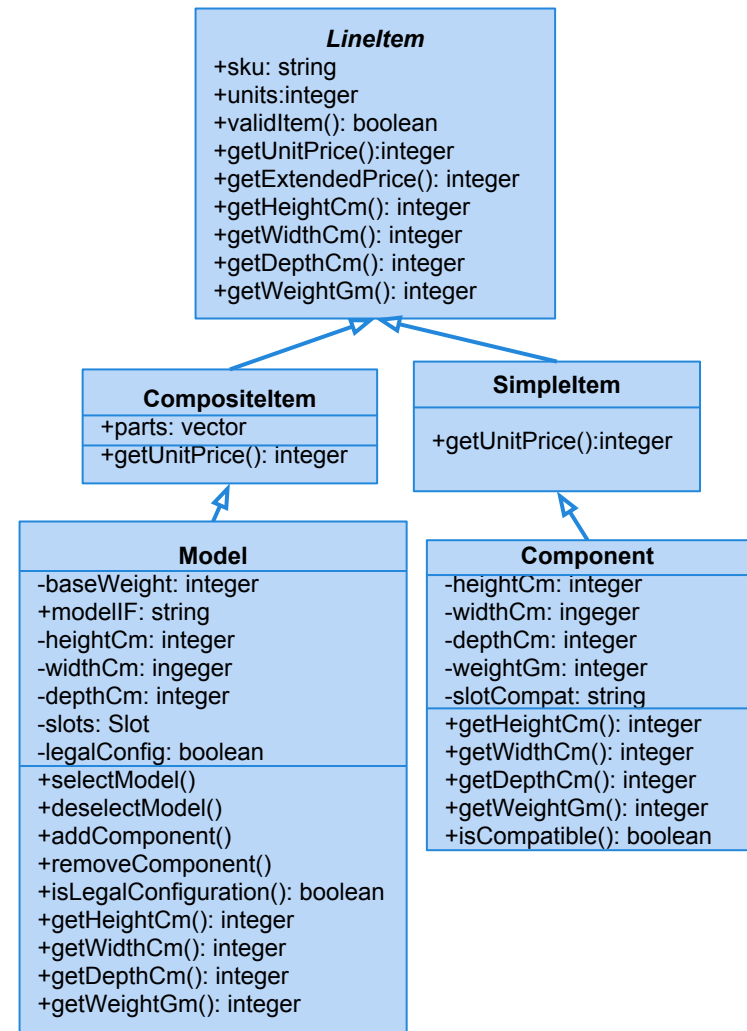
Option 3: Equivalent Scenarios

- Rather than exposing internal state, execute two *equivalent* scenarios and check whether the inspectable portions of the two produced objects match.
 - Then, run a third *non-equivalent* scenario and make sure it differs from the first two.

| | | |
|--|---|--|
| selectModel(M1) addComponent(S1,C1) addComponent(S2,C2) isLegalConfig() deselectModel() selectModel(M2) addComponent(S1,C1) isLegalConfig() | selectModel(M2) addComponent(S1,C1) isLegalConfig() | selectModel(M2) addComponent(S1,C1) addComponent(S2,C2) isLegalConfig() |
| | Equivalent | Non-Equivalent |

Polymorphism and Dynamic Binding

- Behavior depends on the object assigned at runtime.
 - If `LineItem.getUnitPrice()` is called, it may actually be `SimpleItem.getUnitPrice()`.
 - Wrong object might be bound to the variable.
 - May be difficult to tell which class has the fault.
 - Fault may be a result of a combination of bindings.
- Testing one possible binding is not enough - must try multiple bindings.



Polymorphism and Dynamic Binding

- Limited use of polymorphism: Unfold calls.
 - Try each possible binding.
- Challenge - layers of polymorphic calls.

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

```
Credit c;  
boolean canPurchase = c.validateCredit(a,amt,cc);
```

Credit can be one of:
EduCredit, BusinessCredit,
IndividualCredit

Account can be one of:
USAccount,
UKAccount,EUAccount,
JPAccount, or OtherAccount

CreditCard can be one of:
VISACard, AmexCard,
MasterCard

Apply Combinatorial Testing

- This is the same problem faced in functional testing, with parameter combinations.
- Use combinatorial interaction testing to test all n-way combinations.

| Account | Credit | creditCard |
|--------------|------------------|------------|
| USAccount | EDUCredit | VISACard |
| USAccount | BusinessCredit | AmExCard |
| USAccount | IndividualCredit | MasterCard |
| UKAccount | EDUCredit | AmExCard |
| UKAccount | BusinessCredit | VISACard |
| UKAccount | IndividualCredit | MasterCard |
| EUAccount | EDUCredit | MasterCard |
| EUAccount | BusinessCredit | AmExCard |
| EUAccount | IndividualCredit | VISACard |
| JPAccount | EDUCredit | VISACard |
| JPAccount | BusinessCredit | MasterCard |
| JPAccount | IndividualCredit | AmExCard |
| OtherAccount | EDUCredit | MasterCard |
| OtherAccount | BusinessCredit | VISACard |
| OtherAccount | IndividualCredit | AmExCard |

Polymorphism and Dynamic Binding

- Problem - bad polymorphic call infects a variable definition.

```
public abstract class Account{
    public int getYTDPurchased(){
        int totalPurchased = 0;

        for(Enumeration e = subsidiaries.elements(); e.hasMoreElements(); ){
            Account subsidiary = (Account) e.nextElement();
            totalPurchased += subsidiary.getYTDPurchased();
        }
        return totalPurchased;
    }
}
```

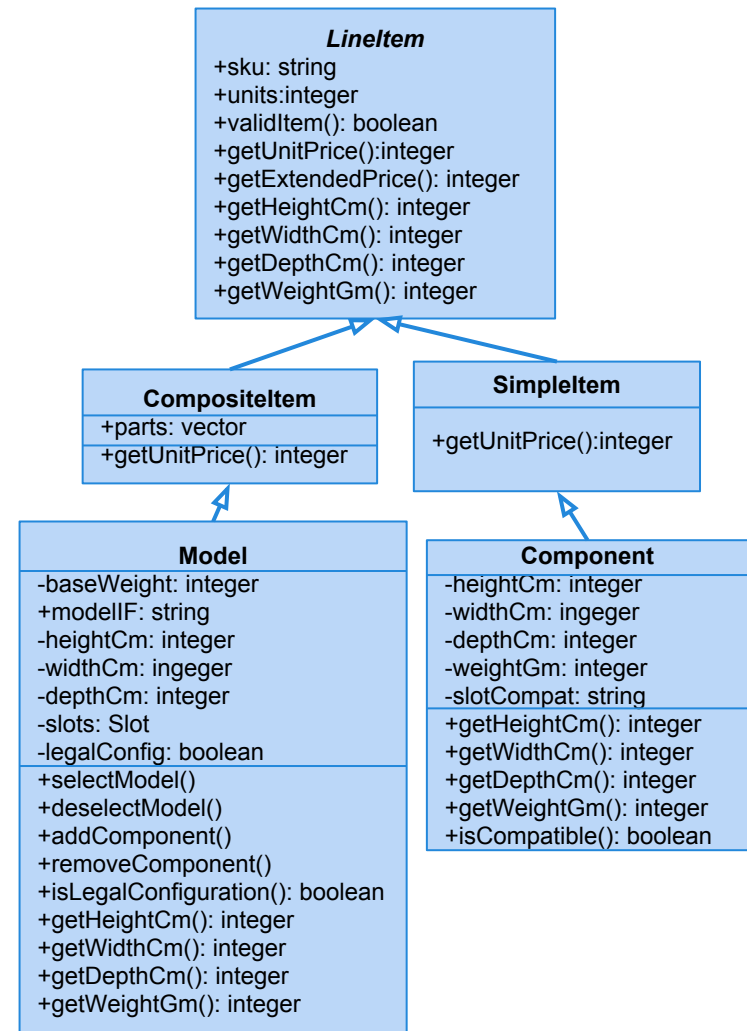
- Combine polymorphic variations with data-flow techniques.

Data-Flow x Polymorphic Calls

- Use data flow analysis to identify DU pairs.
 - At each definition and use bound to a polymorphic call, note the possible bindings.
- One DU pair becomes $N \times M$ pairs.
 - N ways the definition can be bound.
 - M ways the point of use can be bound.
- High number of tests, but only those polymorphic calls that can corrupt variables.
 - If too many, cover all N and M settings in any combination, rather than their product.

Inheritance

- We can define child classes that inherit attributes and operations.
- Most inheritance issues are really polymorphism issues.
- However, inheritance may allow us to **reduce** the number of test cases required.



Inheritance and Test Reuse

- Subclasses share methods with ancestors.
- We can categorize methods as:
 - **New:** If not inherited, we need to test them.
 - If the name is the same, but parameters have changed, it is new.
 - **Recursive:** Inherited from the ancestor without change. Code only appears in the ancestor.
 - **Redefined:** Overridden in the subclass.

Inheritance and Test Reuse

- We can categorize methods as:
 - **Abstract New:** New and abstract in the subclass.
 - **Abstract Recursive:** Inherited when *the ancestor's version was abstract*.
 - **Abstract Redefined:** Redefined when *the ancestor's version was abstract*.

Inheritance and Test Reuse

- In general, four sets of tests for a method:
 - Intraclass Functional, Intraclass Structural
 - Interclass Functional, Interclass Structural
- When we test a subclass, new methods need test cases.
- Recursive/Abstract Recursive methods do not need to be retested.
- Redefined/Abstract Redefined must be retested.

Genericity

- Generics
 - Generic class is instantiated with different types:
 - `LinkedList<String>`, `LinkedList<Integer>`
 - `HashMap<String,Integer>`,
`HashMap<ArrayList<Integer>,Boolean>`
- Important for building reusable components and libraries.
- Challenging to test:
 - Can only test instantiations, not the generic class.
 - May not know all ways it can be instantiated.

Testing Generics

- Designed to behave consistently.
- First, testing requires showing that any instantiation is correct.
 - In general, this is straightforward if we have code of the generic class and the parameterized version.
- Second, do all possible parameterizations behave identically to the tested one?

Testing Generics

- Second, do all possible parameterizations behave identically to the tested one?
 - Potential challenge - does the generic class interact with the parameterized version?
 - i.e., the generic makes use of a service the parameterized version might also make use of.
 - `class PriorityQueue<Elem implements Comparable> {...}`
 - Behavior of `PriorityQueue<E>` depends on `E`.
 - Acceptable as long as `E` behaves correctly when fulfilling requirements of `Comparable`.
 - Interfaces are a type of specification.

Exceptions

- Exceptions separate error handling from the primary program logic.
 - Common fault in C - not checking for error indications returned by a function.
 - In Java, a thrown exception interrupts control.
- Introduces implicit control flow
 - The point where an exception is caught and handled may not match where it is thrown.
 - Associations of exceptions with handlers is dynamic.
 - Exception propagates up stack of calling methods until it reaches a matching handler.

Exceptions

- Cannot be treated as normal control flow.
 - Would have to add branches for every possible exception (array index references, memory allocations, casts, etc.) and match to any handler.
- Separate exceptions from normal, explicit control flow.
 - Dismiss any exceptions triggered by program errors signaled by the system.
 - Subscript errors, bad casts.
 - Exercising these does not help prevent or find errors.

Exceptions

- Unless we have explicitly written code to handle those kind of exceptions.
 - Fault-tolerant systems.
 - Must test the error recovery code.
 - Still do not need to couple recovery code to every point where there might be an error.
- Must handle exceptions that indicate abnormal cases.
 - If exception handler is local, must test the handler.
 - Do not need to test each point the exception might be raised.

Exceptions

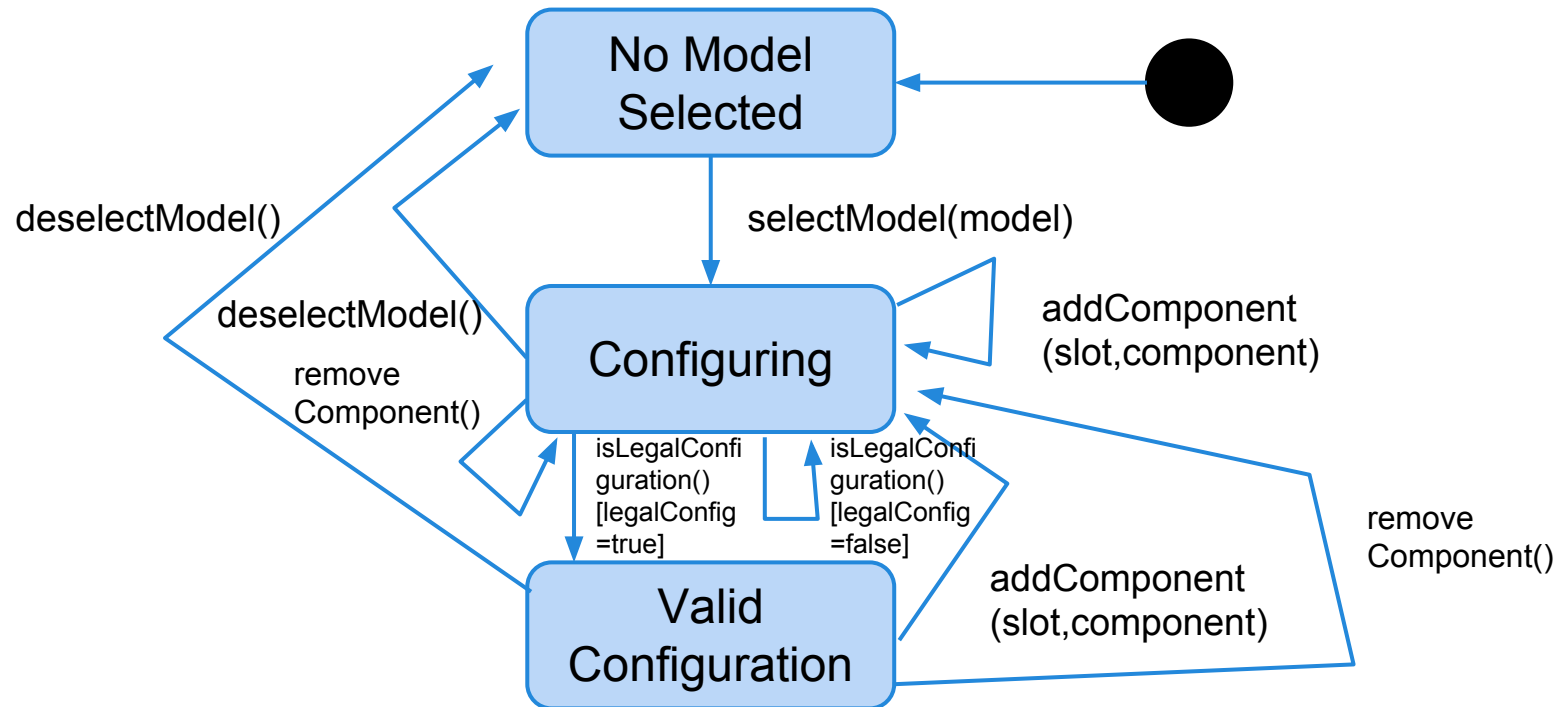
- Must handle exceptions that indicate abnormal cases.
 - If the handler is not local...
 - The exception will be passed up the stack until it is handled. There could be many potential handlers.
 - It is very hard to determine *where* it will be handled.
 - We can't test all possible chains.
 - Instead, enforce a design rule:
 - If a method can propagate an exception without catching it, that call should have no other effect.

Activity

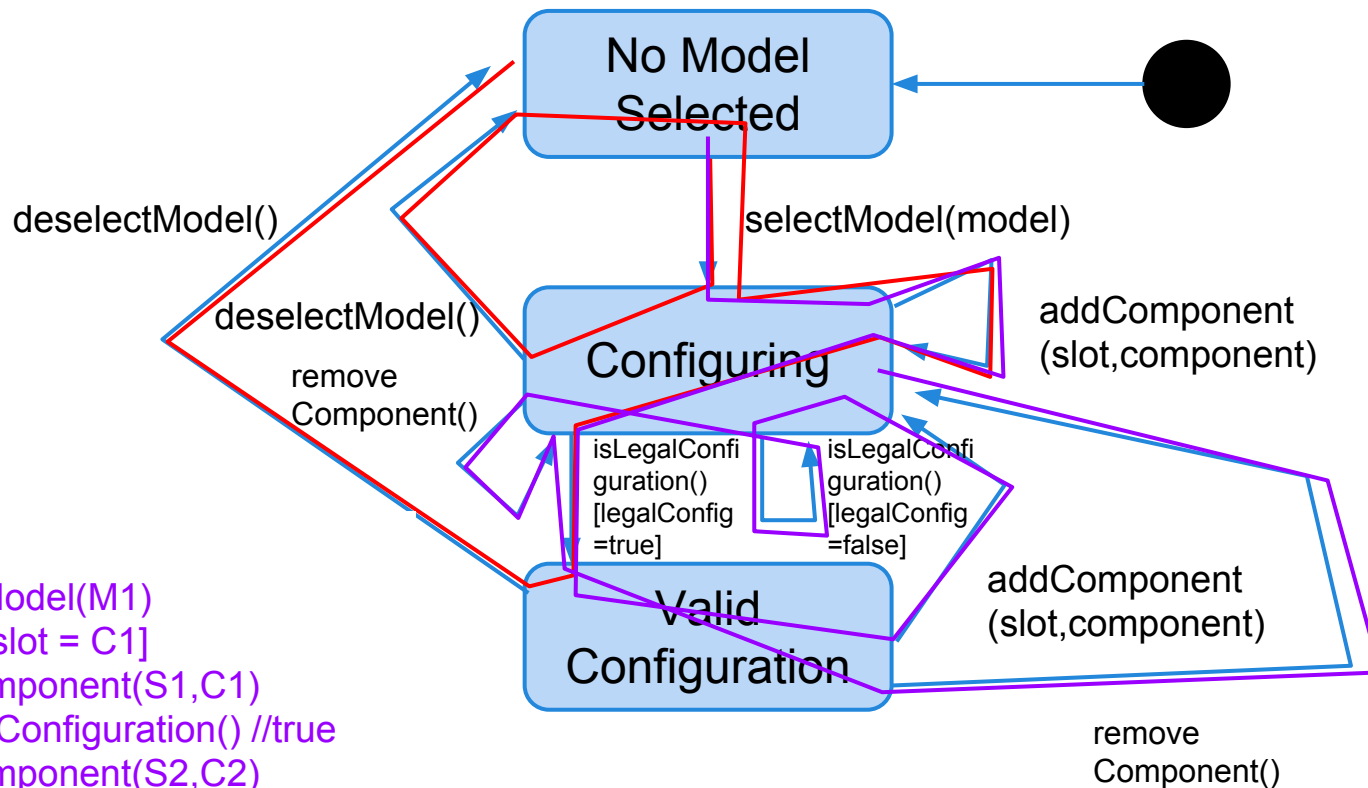
Informal specification for class Model.

1. Derive a state machine representation of the class from the specification.
2. Identify test cases (sequences of method calls) to achieve transition coverage over the model.

Activity - Sample Solution



Activity - Sample Test Cases



TC2:

```
selectModel(M1)
[M1, 1 slot = C1]
addComponent(S1,C1)
isLegalConfiguration() //true
addComponent(S2,C2)
isLegalConfiguration() // false
removeComponent(S2)
isLegalConfiguration() // true
removeComponent(S1)
```

We Have Learned

- Basic functional and structural testing techniques can be applied to OO systems, with a few adaptations.
 - When testing one class, build an intraclass CFG and cover control and data-flow.
 - When testing multiple classes, categorize methods as inspectors and modifiers and cover DU pairs between them.

We Have Learned

- Oracles can inspect hidden state through added code, code changes, or reflection.
- Polymorphic bindings can be covered through combinatorial testing and DU pairings.
- Inheritance can reduce testing effort.
- Exceptions require special handling.

Next Time

- Finite State Verification
 - Proving properties over models.
 - Reading: Chapter 8
- Homework:
 - Assignment 3 - due 03/28