Testing Object-Oriented Systems

CSCE 747 - Lecture 12 - 02/18/2016

Object-Oriented Software

- Most software is designed as a collection of interacting objects that model concepts in the problem domain.
  - Concrete concepts in the real world
    - A driver's license, an aircraft, a document…
  - Logical concepts
    - A scheduling policy, conflict resolution rules...
- What defines an object:
  - Data representation
    - Characteristics that define an object (attributes).
  - Functionality
    - What the object can do (operations).

Classes

- A class describes a **type** of object where each instance has the same attributes and behaviors, the same relationships to other classes, and common meaning.
- **Objects are instances of classes**, where each object has the same structure and behavior.

- Person instances:
  - Greg Gay, Jason Biatek
- Credit Card instances:
  - Greg's credit card, Jason's credit card

Testing Object-Oriented Software

- Most of the techniques we have covered have been introduced using non-OO examples (a single procedure, multiple procedures within one class).
- These techniques work on OO systems…
  - But, there are a few complications.
  - Today - we will discuss these complications and factors that must be considered in testing OO code.

Issues With Testing OO Systems

OO Testing Issues

- State Dependent Behavior
- Encapsulation
- Inheritance
- Polymorphism and Dynamic Binding
- Abstract Classes
- Exception Handling
- Concurrency

State-Dependent Behavior

- The behavior of a method depends on the current state of the object.
- Two objects might return different results if their state differs.
- Here - the contents of slots determines the legality of the model configuration.

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

Encapsulation

- Classes may have public and private members.
- Other objects must work with public methods and variables.
- To run a test, we may not be able to put an object in particular states.
- To check test results, we may need access to private information.

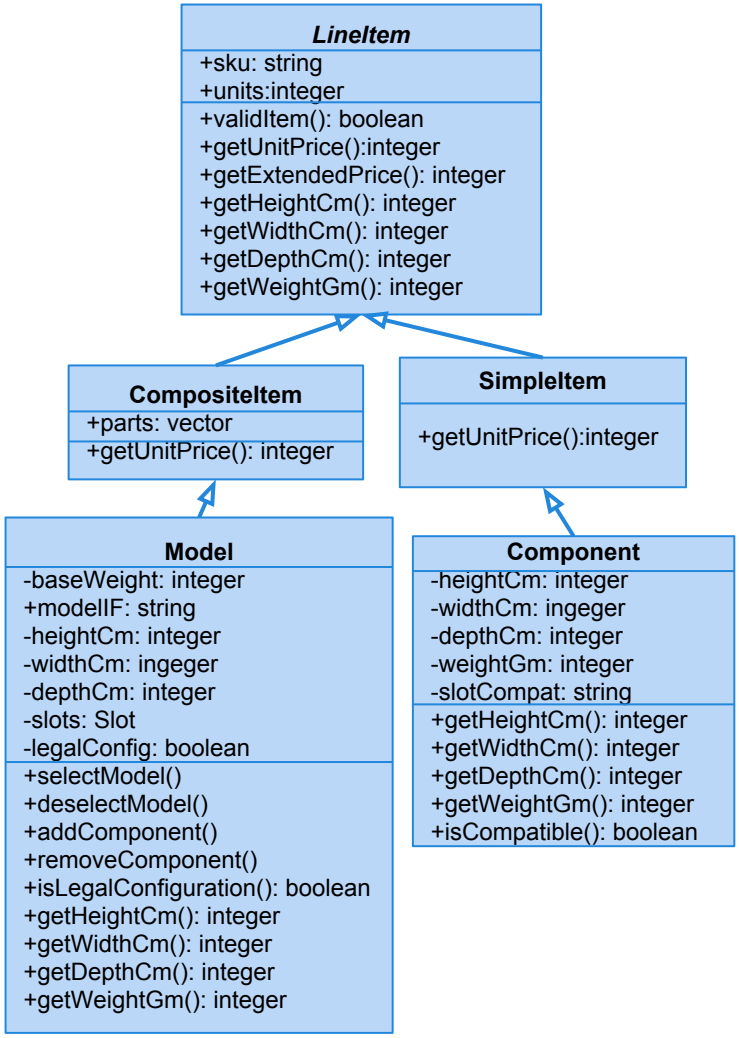
```
public class Model extends Orders.CompositeItem
{
      public String modelID;
      private int baseWeight;
      private int heightCm, widthCM, depthCM;
      private Slot[] slots;
      private boolean legalConfig = false;
      private static final String NoModel = "NO
MODEL SELECTED";

      private void checkConfiguration(){
            ...
      }

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

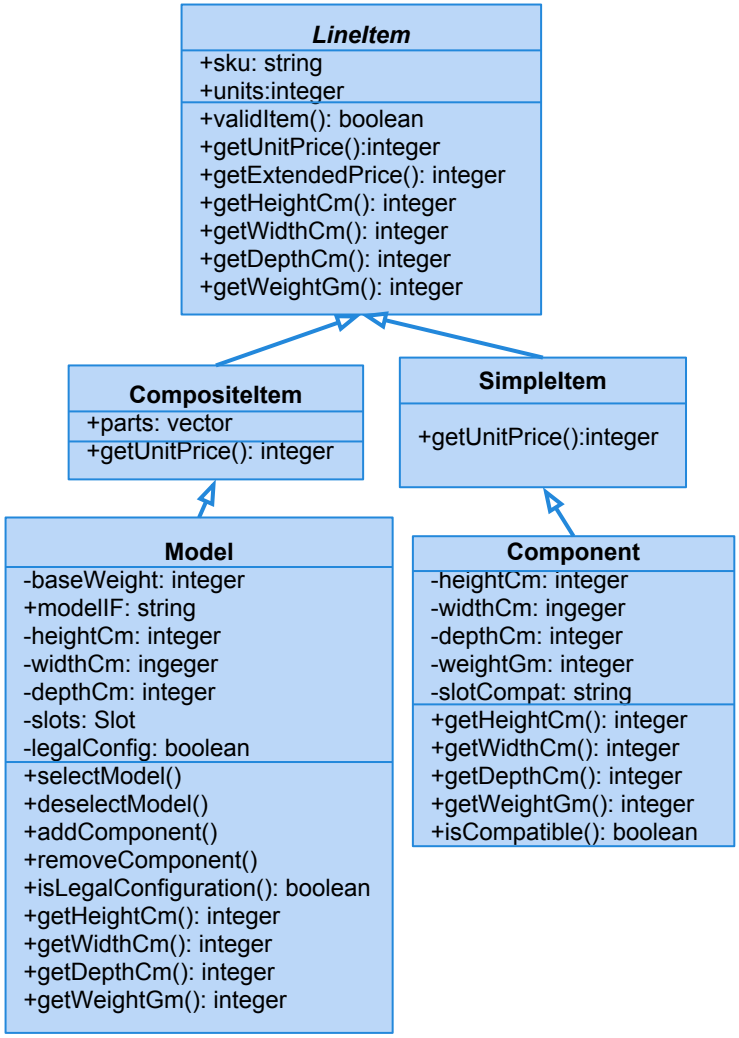
Inheritance

- We can define child classes that share attributes and operations based on a hierarchical relationship.
  - Allows the creation of specialized versions of classes without reimplementing functionality or including attributes and operations where they aren't needed.
  - All objects of a child are both instances of that class and of the parent class.



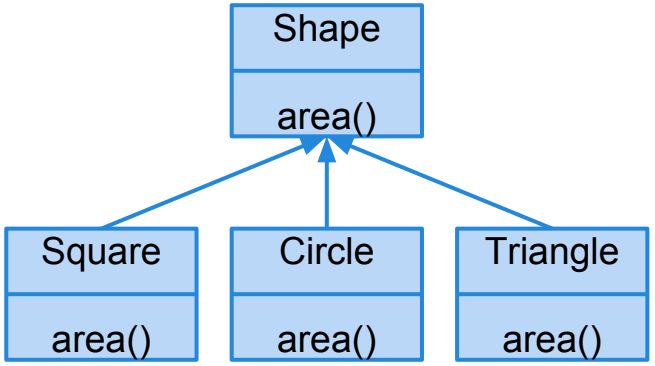
Inheritance

- Inherited methods may not exhibit the same behavior in children as they do in parent:
  - Child may *override* the method with its own implementation.
  - A method may depend on other parts of the class that have changed.
  - Can often establish that the method is truly unchanged and does not need to be retested.
  - If is has changed, it must be retested in the right context.



Polymorphism and Dynamic Binding

- The same operation may behave differently when used on different classes.
  - Specifically, we can *redefine operations* in each related class.
- Because Shape defines an area() method, we know all children offer that method.
  - But, we can redefine that method in each child to offer the right answer.



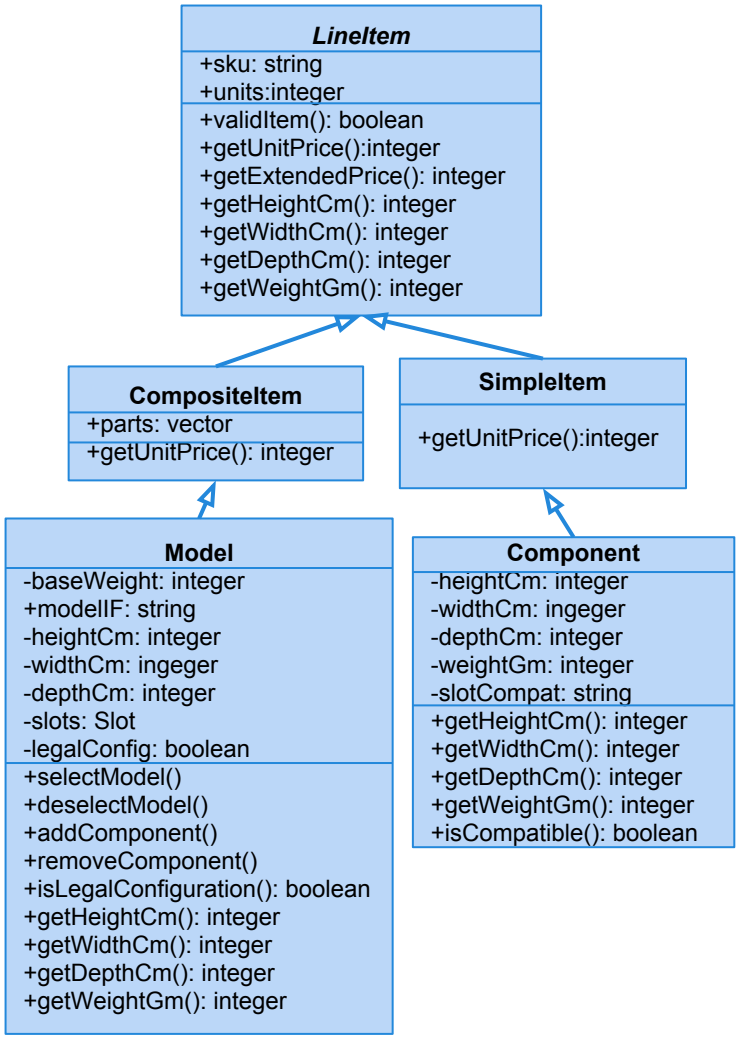
Because objects are instances of both their class and their parent class:

```
void getArea(Shape s){
        System.out.println(s.area());
}
```

Gives the right answer if a square, circle, triangle, etc is passed in.

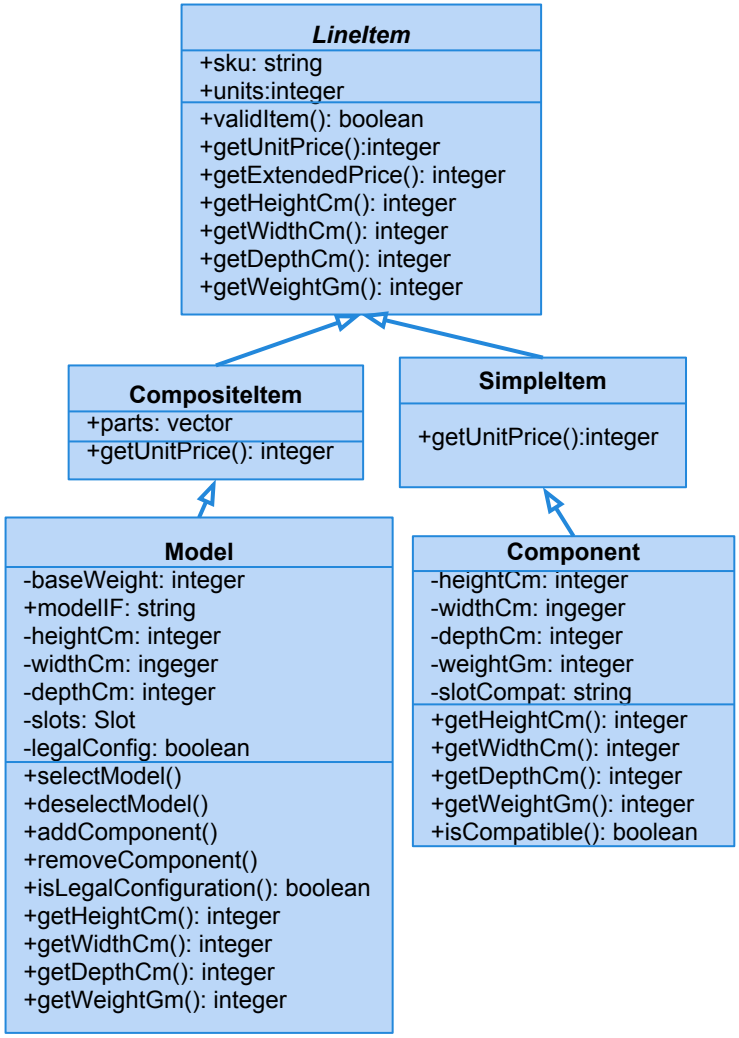
Polymorphism and Dynamic Binding

- 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.



Abstract Classes

- Classes that are incomplete and cannot be instantiated.
  - LineItem
- Define templates for other classes to follow.
- These still must be tested in some form.
  - Can test all of the child classes.
  - Techniques for testing what is declared in the abstract class.



Exceptions

- Used to handle erroneous execution conditions.
- Either handled directly in code, or declared in method header.
- Where an exception is caught and where it is handled differ.
  - Impacts the control-flow of the code.

```
try{
    BufferedReader br = new
        BufferedReader(
        new File("input.txt"));
    String line = br.readLine();
catch(IOException e){
    e.printStackTrace();
}
```

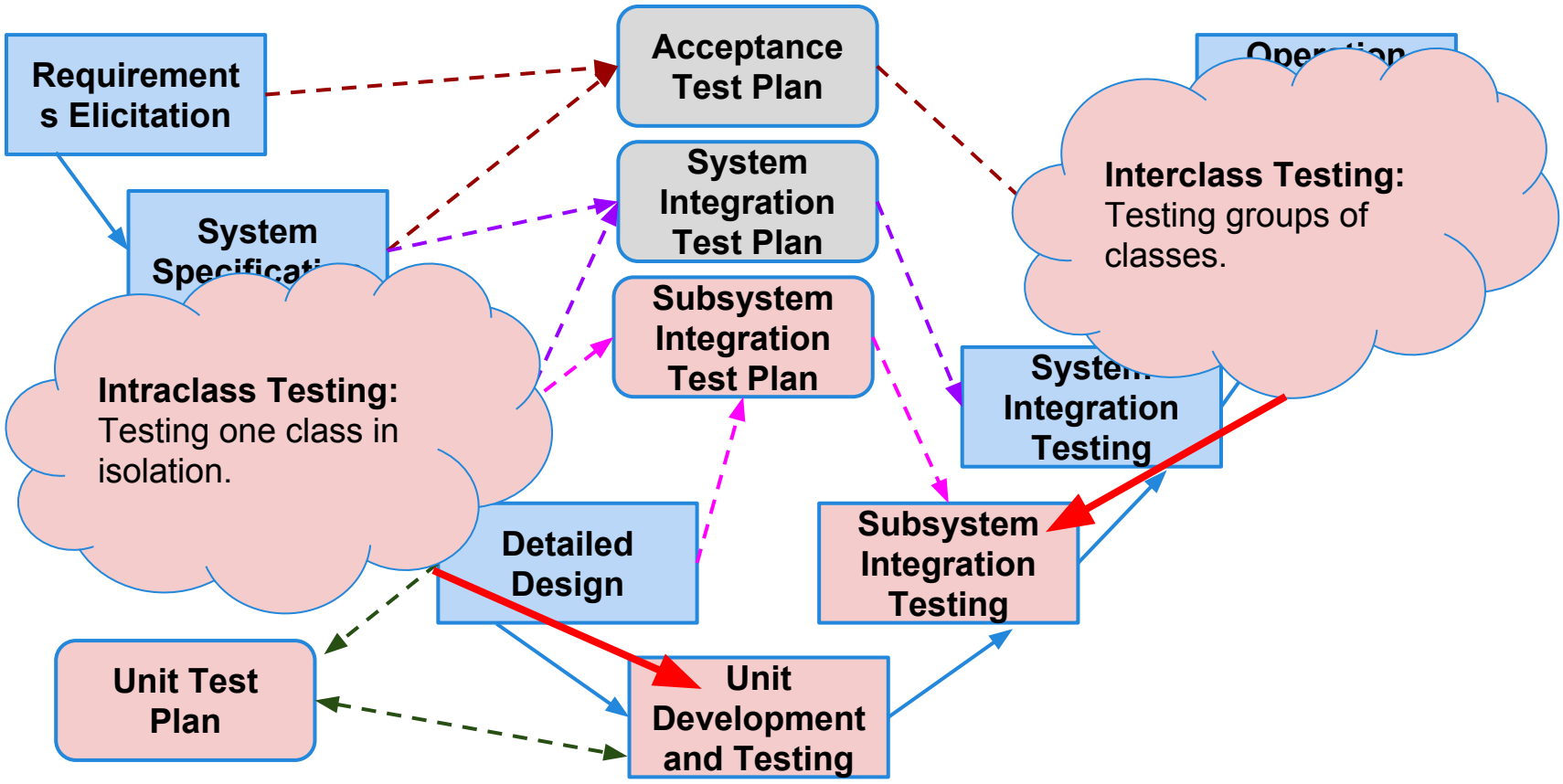
```
public int tryThis()
    throws NullPointerException{
    ...
}
```

Concurrency

- A program can be designed to execute over multiple, concurrently-executing processes.
- Introduces new sources of failure:
  - Deadlock, race conditions, timing of data synchronization.
- System is dependent on scheduler decisions that a tester cannot control.

Approaches to Testing OO Systems

The V-Model of Development



Unit Testing

- Unit testing is the process of testing the smallest isolated “unit” that can be tested.
  - Allows testing to begin as code is written.
  - Allows testing of system components in isolation from other components.
- Before the system is built, each component should work in isolation.
- Usually in OO, a unit is a class.
  - Individual methods depend on and modify object state and are dependent on other methods.

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.

Using State Machine Models

- The state of an object implicitly impacts the result of a method call.
  - Tests should examine the possible states of an object and transitions between states.
- Create tests by covering a state machine model.
  - Sequence of transitions ~ sequence of method calls
  - Exercising that sequence will put the class into the different possible states (and cover different means of reaching those states).

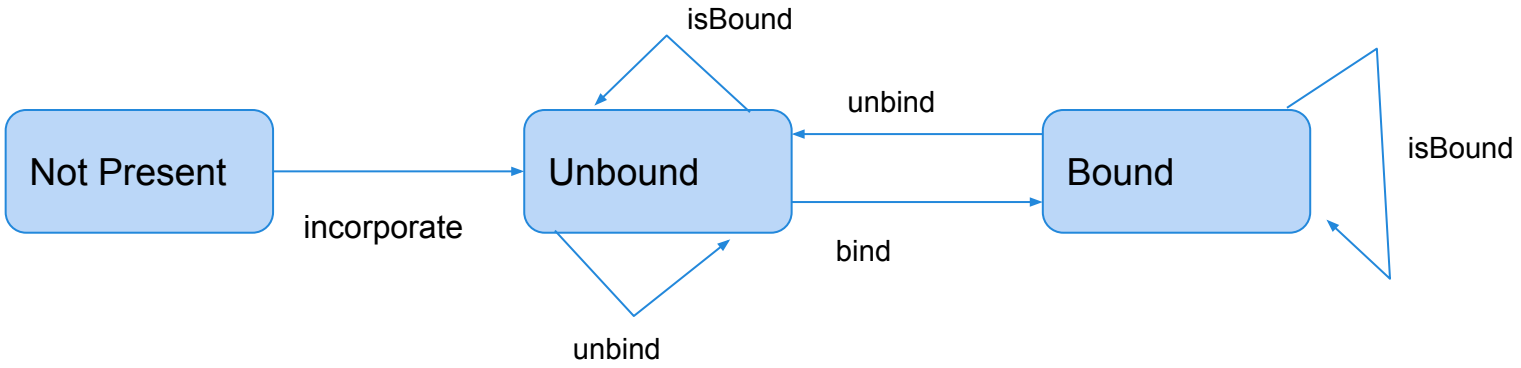
Informal Specification

Slot represents a configuration choice in all instances of a particular model of computer. It may or may not be implemented as a physical slot on a bus. A given model may have zero or more slots, each of which is marked as required or optional. If a slot is marked as required, it must be bound to a suitable component in all legal configurations.

Slot offers the following services:

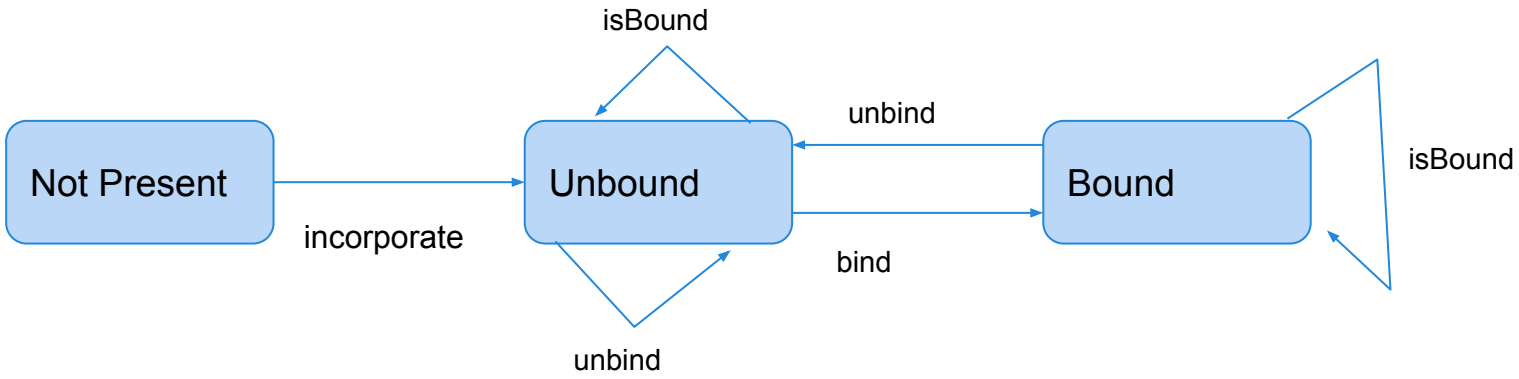
- **Incorporate:** Make a slot part of a model, and mark it as either required or optional. All instances of a model incorporate the same slots.
- **Bind:** Associate a compatible component with a slot.
- **Unbind:** The unbind operation breaks the binding of a component to a slot, reversing the effect of a previous bind operation.
- **IsBound**: Returns true if a component is currently bound to a slot, or false if the slot is currently empty.

… To State Machine



- Do not derive too many states.
  - Integer mapped to “zero” and “nonzero”, not a state for each possible value.
- Model how a method affects a class. States only need to capture interactions between methods and the class state.

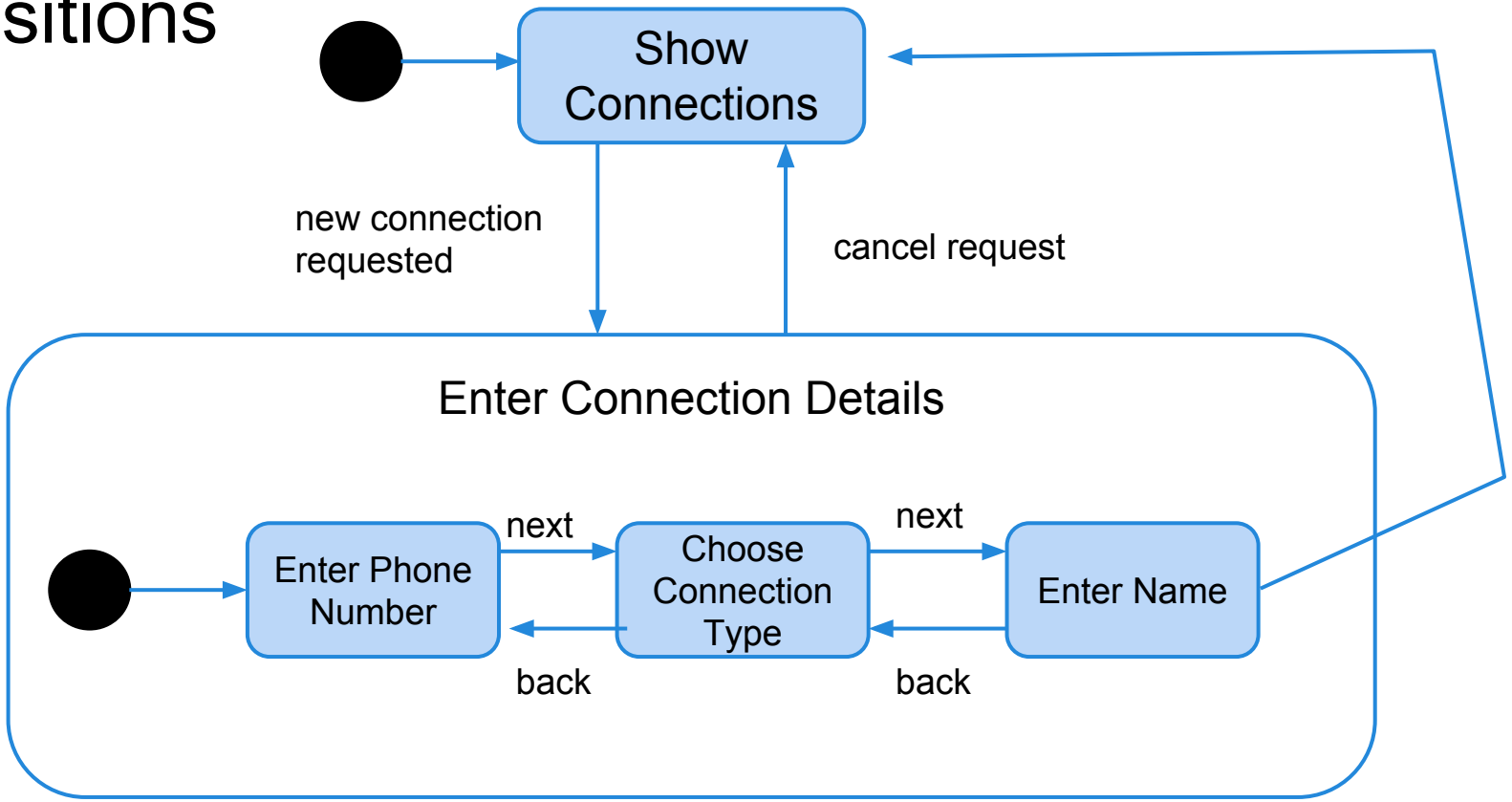
Test Coverage



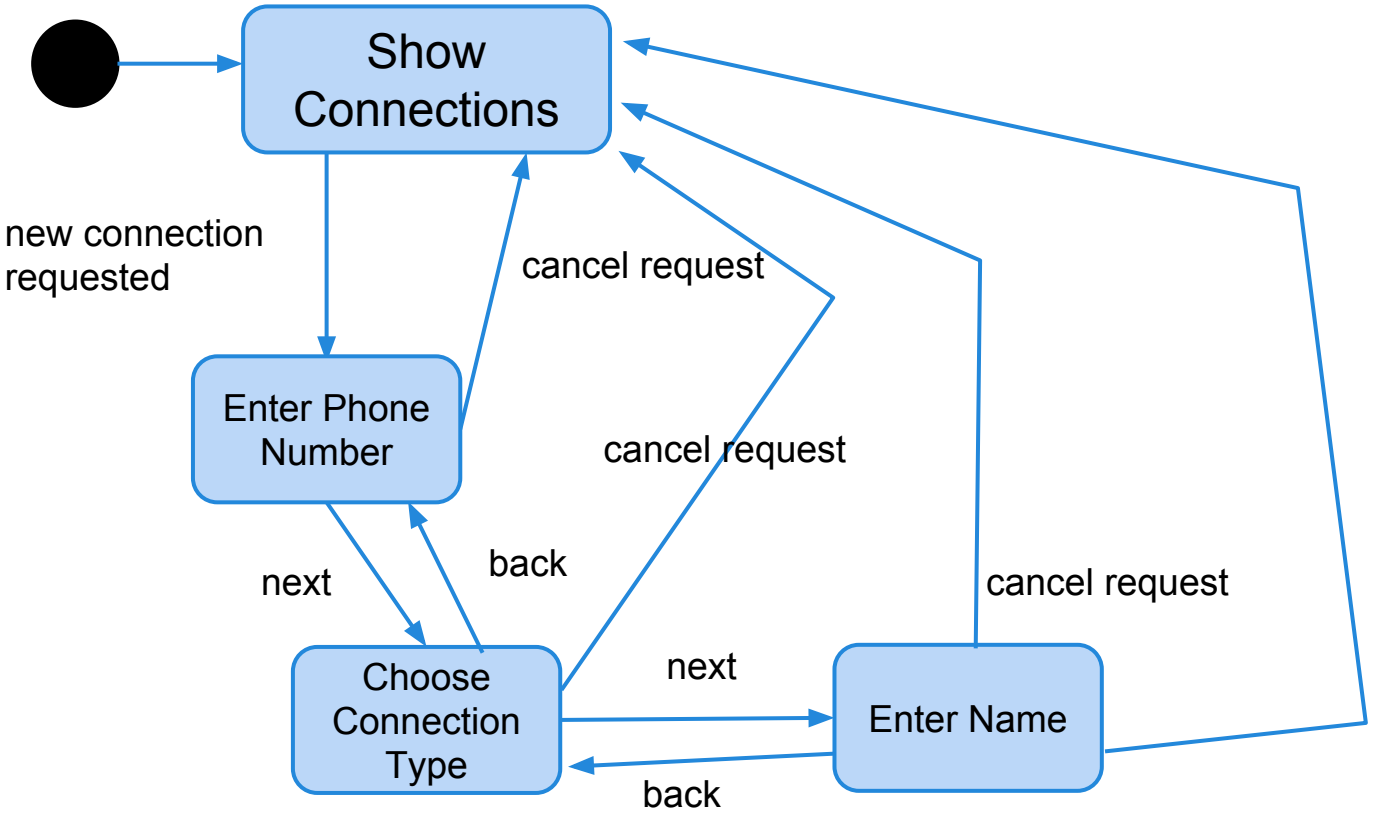
- To test: apply model coverage criteria.
  - Transition Coverage.
    - TC1: incorporate, isBound, bind, isBound
    - TC2: incorporate, unBind, bind, unBind, isBound

Superstates

Use superstates to encapsulate common transitions

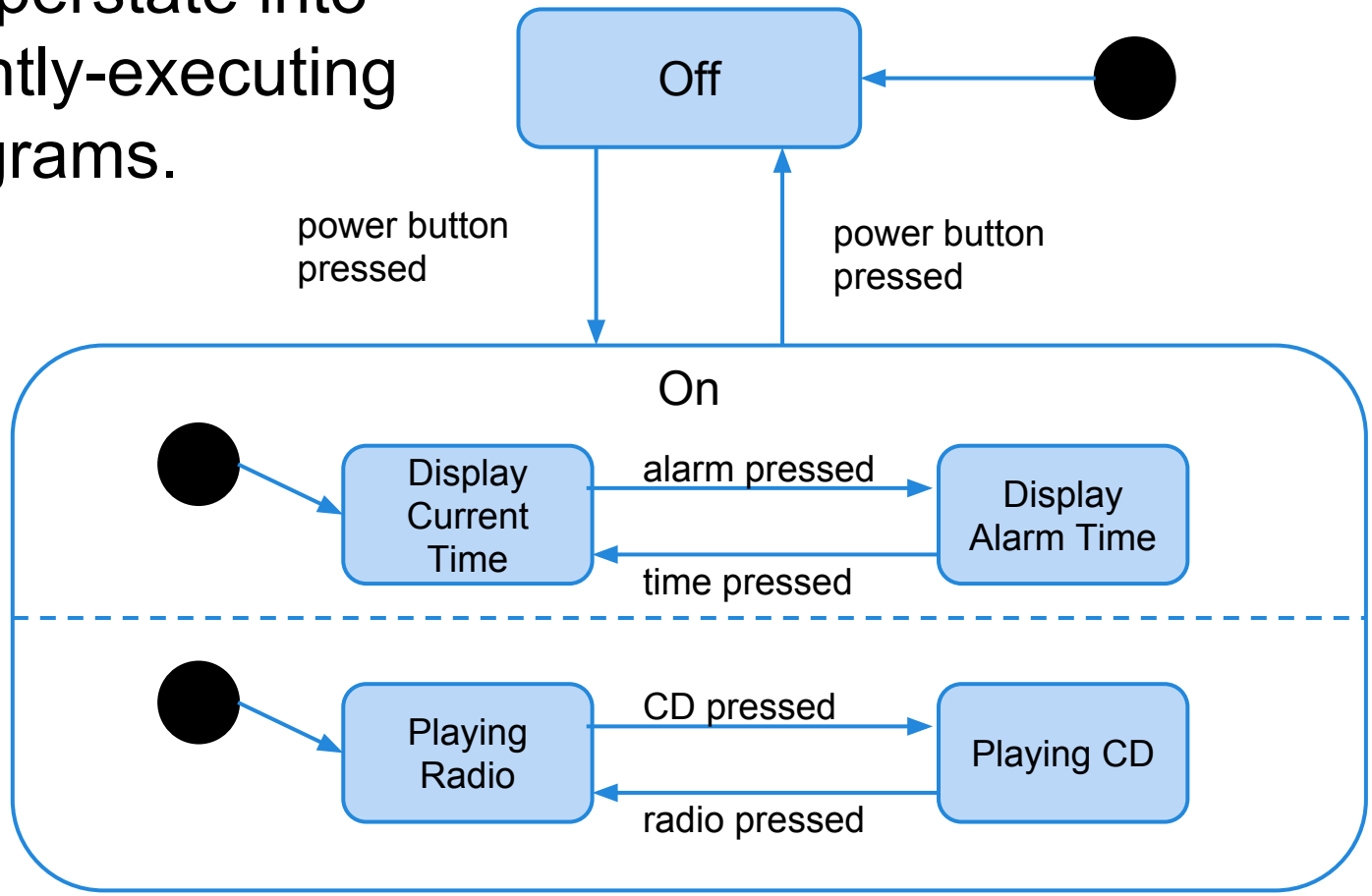


Superstates - Flattened

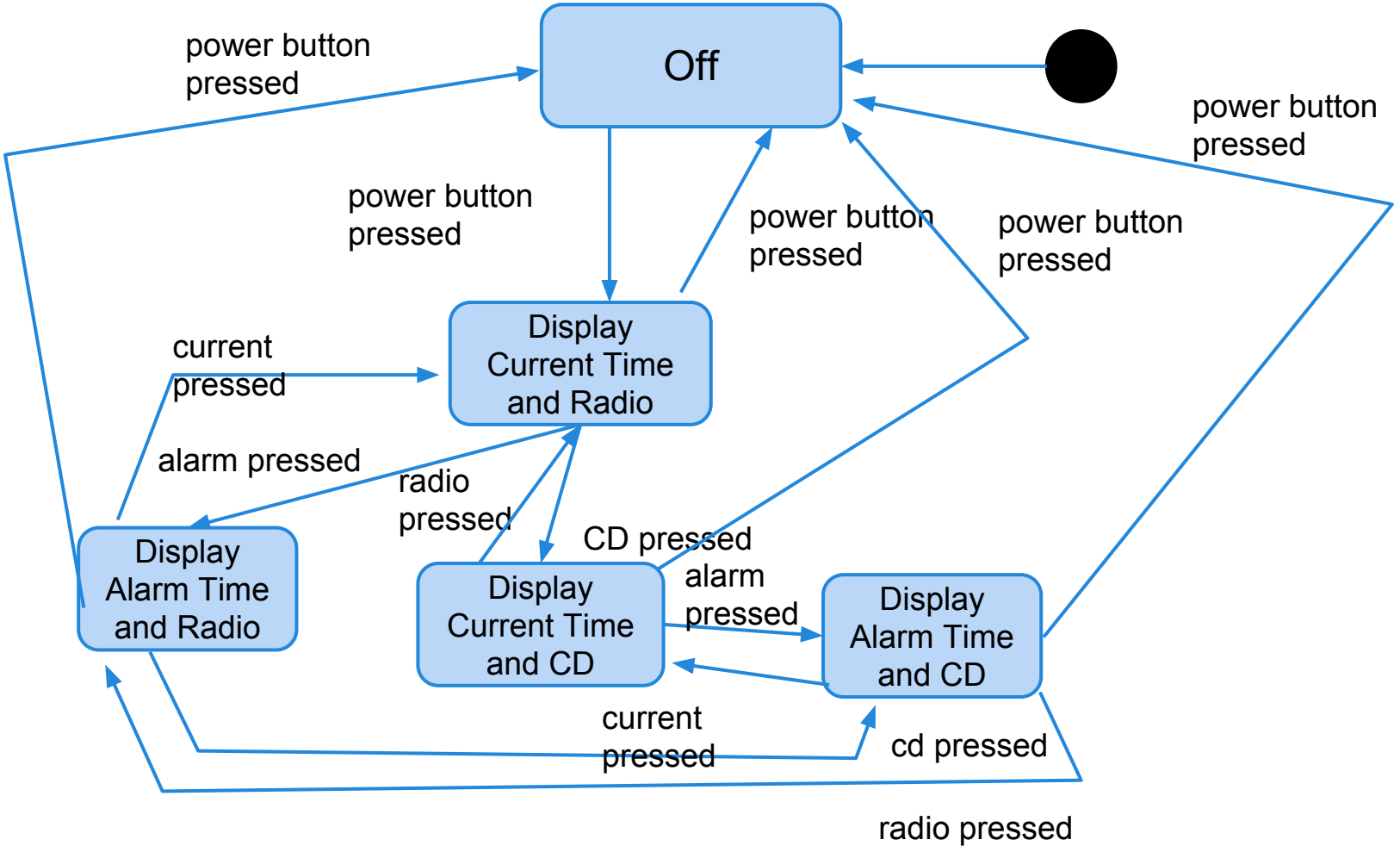


Concurrent States

Divide superstate into concurrently-executing state diagrams.



Concurrent States - Flattened



Simple Transition Coverage

- Do not flatten the state machine. Instead, cover all visible transitions.
  - Rather than trying all combinations of the concurrent threads, cover their transitions in isolation.
  - Rather than exercising all possible entry/exit transitions from substates, try any of them.
- Weaker than full transition coverage, but requires far fewer test cases.

An Important Reminder

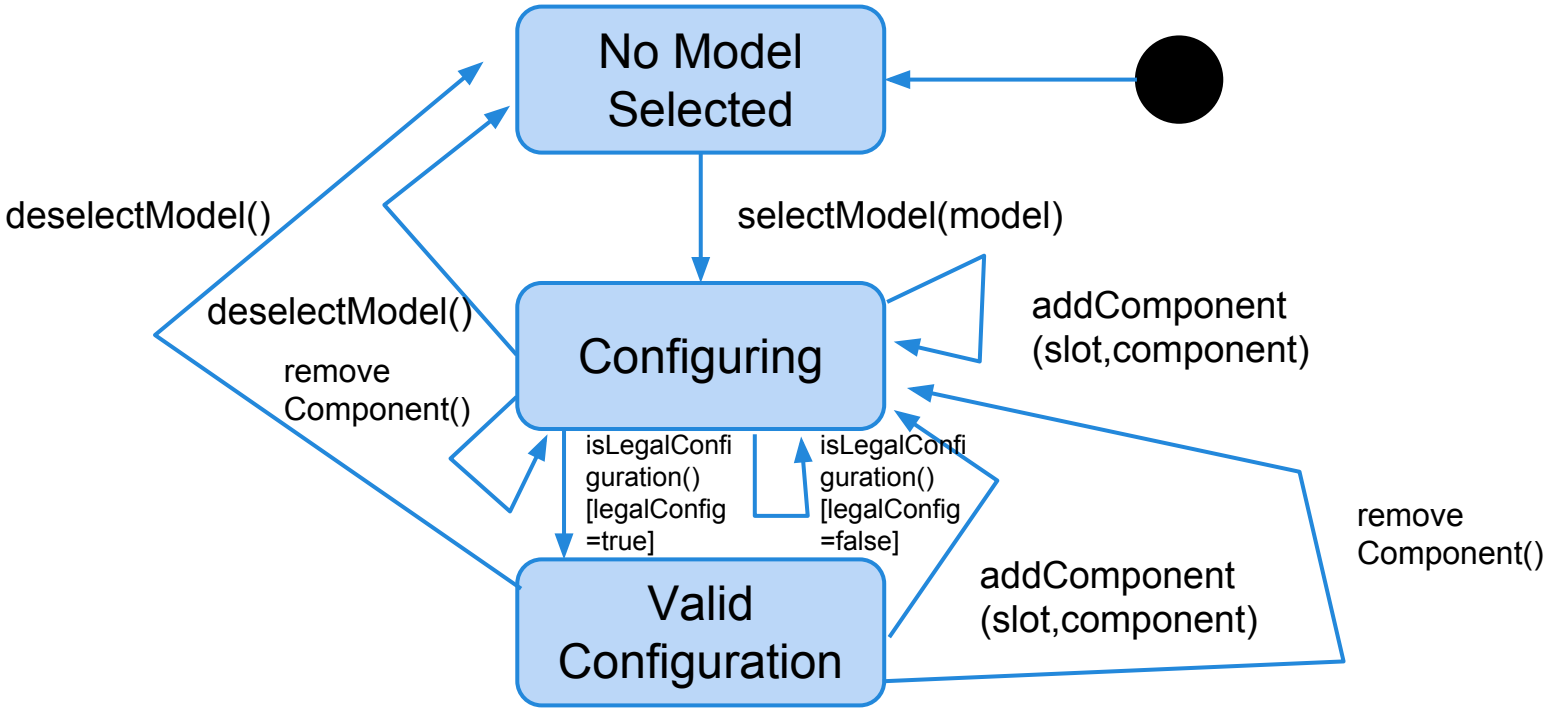
- Do not do this for all classes in your system.
  - State does not always have a significant impact.
  - Some classes are simple enough to cover through basic functional testing
  - Building state machines requires a lot of work.
  - Many real world systems have too many classes.
    - Facebook's iOS app - 18000 classes.
- Look for classes where state clearly matters. Model and cover those classes.

Activity

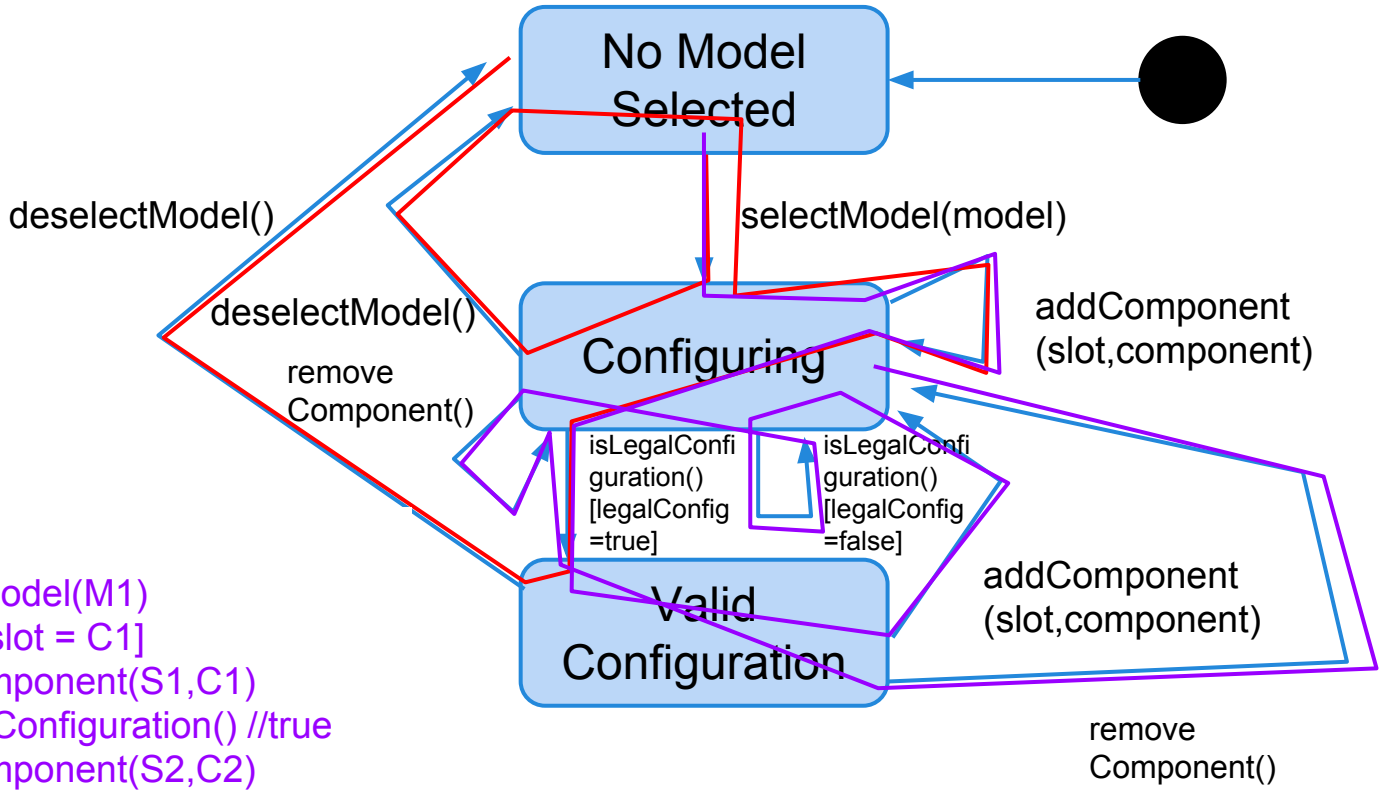
Informal specification for class Model.

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)

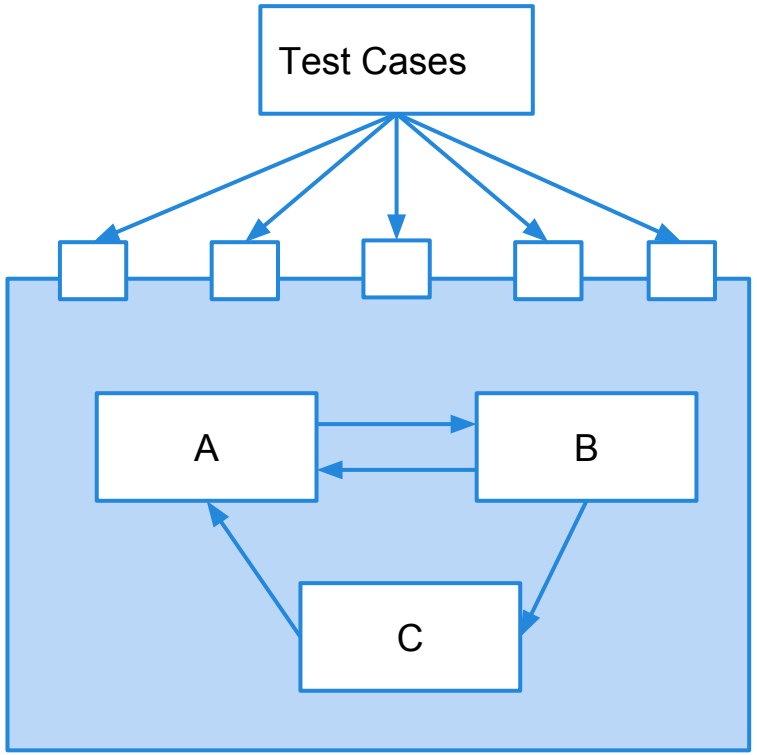
Interclass Testing

- Most software works by combining multiple, interacting components.
  - In addition to testing components independently, we must test their *integration*.
- Functionality performed across components is accessed through a defined interface.
  - Therefore, integration testing focuses on showing that functionality accessed through this interface behaves according to the specifications.

Interclass Testing

We have a subsystem made up of classes A, B, and C. We have performed unit testing...

- However, they work together to perform functions.
- Therefore, we apply test cases not to the classes, but to the interface of the subsystem they form.
- Errors in their combined behavior result are not caught by unit testing.



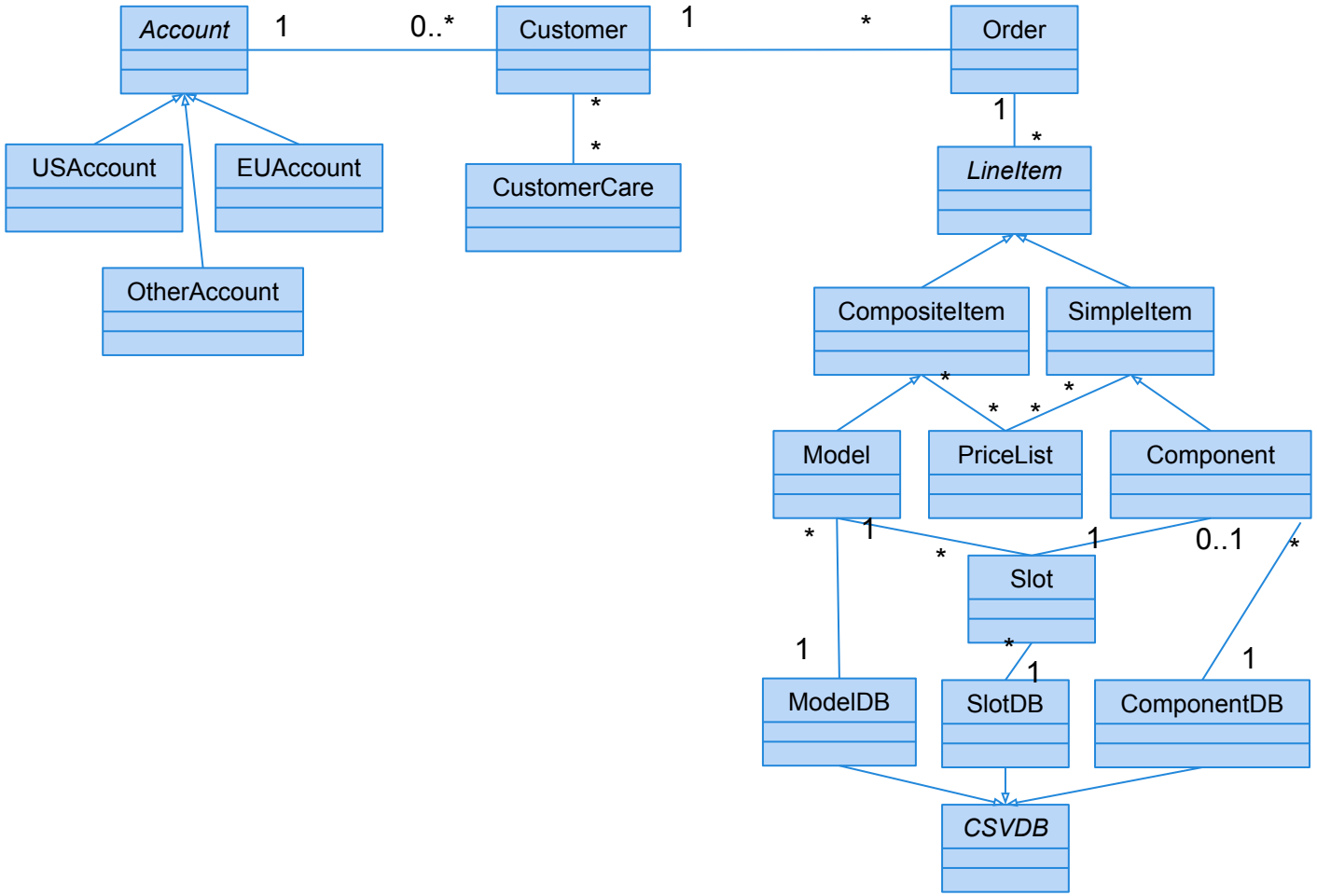
Interclass Testing

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.

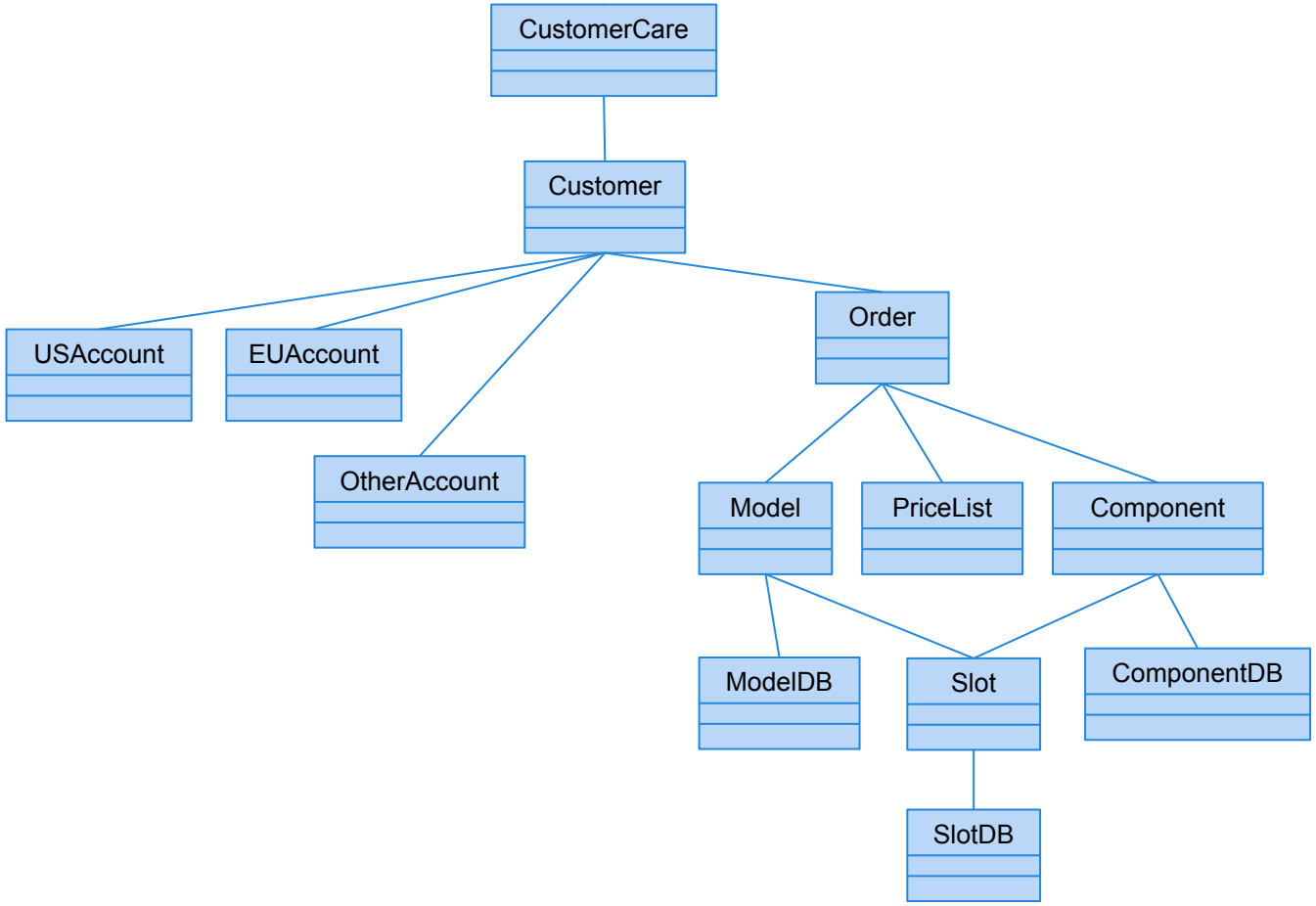
Interclass Testing

- As the point of interclass testing is to verify interactions, we need to understand how classes make use of each other.
- Class A *depends* on B if the functionality of B must be present for the functionality of A to be provided.
  - Model the use/include relation between classes.
  - If objects of class A contain references to objects of class B, A and B have a use/include relation.
  - Ignores inheritance and abstract classes.

Deriving the Use/Include Hierarchy

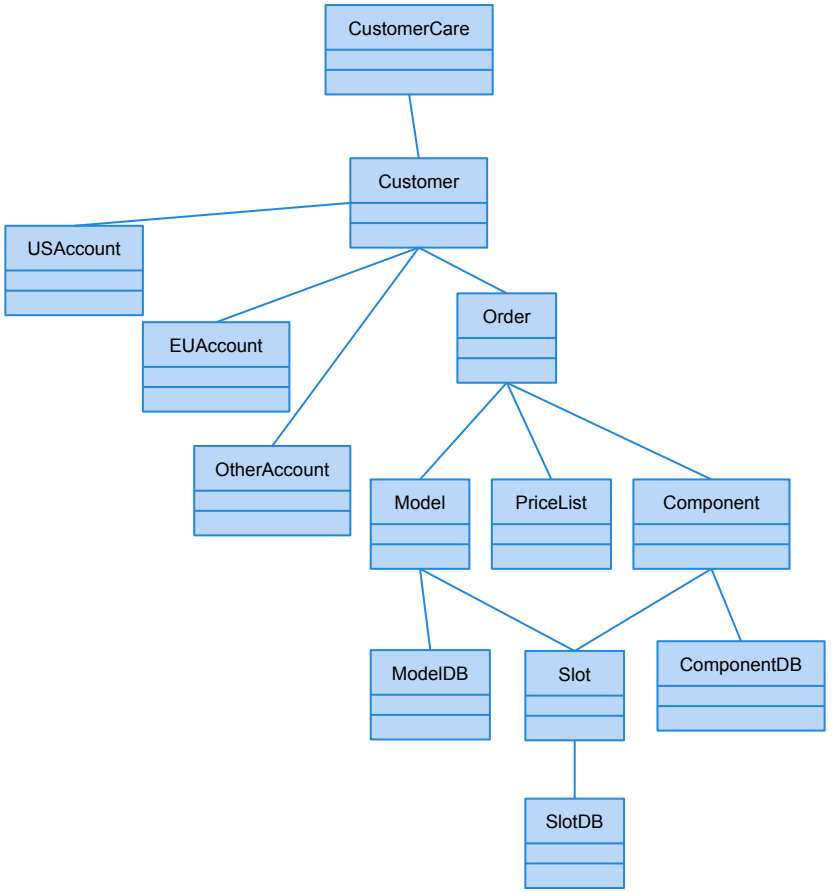


Deriving the Use/Include Hierarchy



Interclass Testing

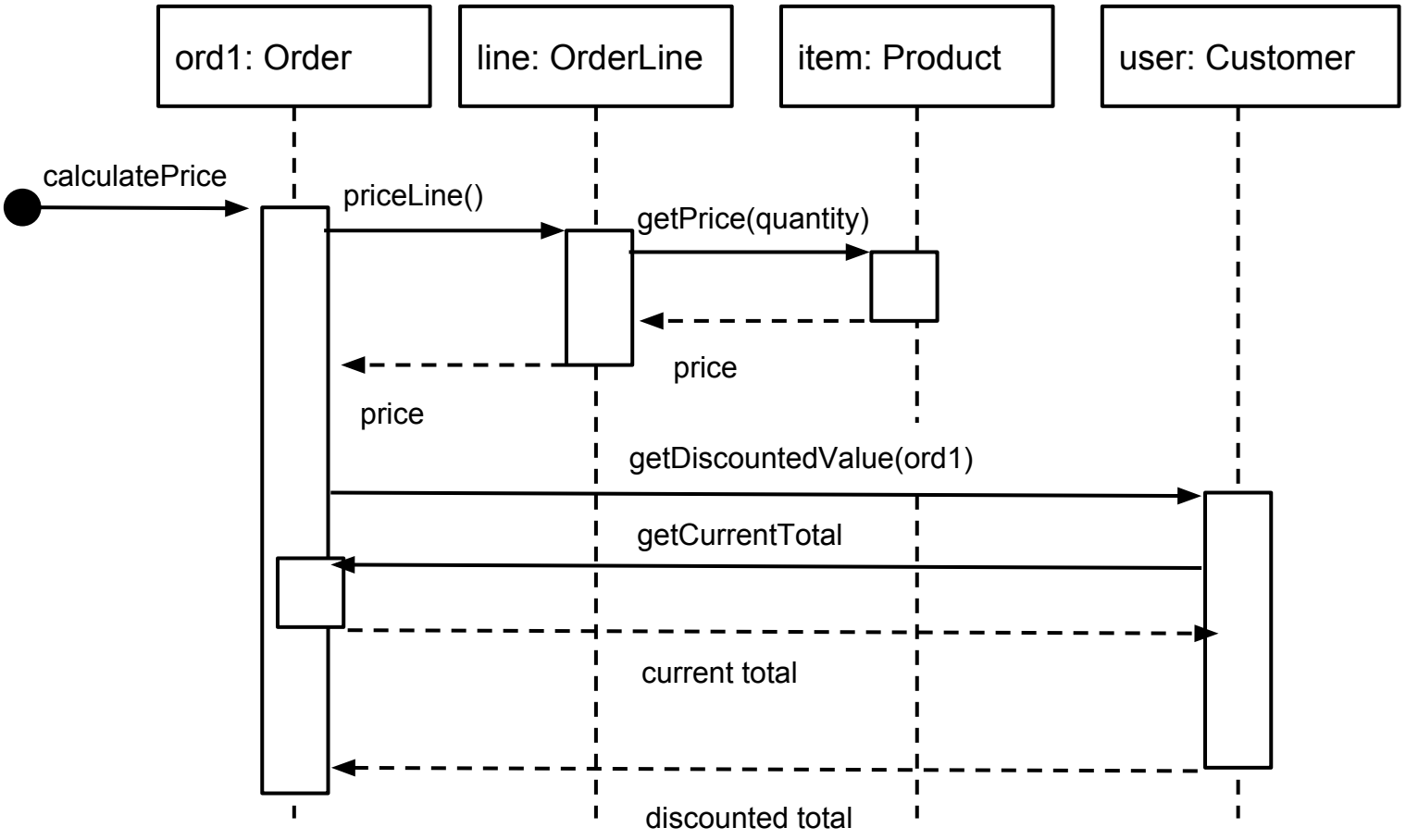
- Start testing from the bottom-up.
  - Start from classes with no dependency, then move up in the hierarchy.
  - Integrate SlotDB with Slot, Component with ComponentDB.
  - Then ModelDB with Model and Slot.
  - … up to Order with all below.



Choosing Interactions

- We would like to cover all possible interactions between classes.
  - All possible states of each and all ways they can interact.
  - This is clearly not possible.
- Need to choose significant scenarios.
  - May be captured already in UML sequence diagrams.
    - Describe object interactions in service of a goal.
  - Vary these scenarios to capture additional illegal interaction sequences.

Sequence Diagram



We Have Learned

- 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.

We Have Learned

- As classes are impacted by state, we can test them effectively by building state machines and deriving transition-covering tests.
  - A path is a set of method calls on that class.
- Groups of classes should be arranged by their dependence relationships, then tested from the bottom-up.

Next Time

- More OO Testing
  - Structural Testing
  - Exceptions
  - Polymorphism
  - Oracles and Encapsulation

- Homework:
  - Assignment 2 - due next Tuesday.