CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 10: Integration Testing and Testing of OO Systems

Gregory Gay
DIT635 - February 21, 2020

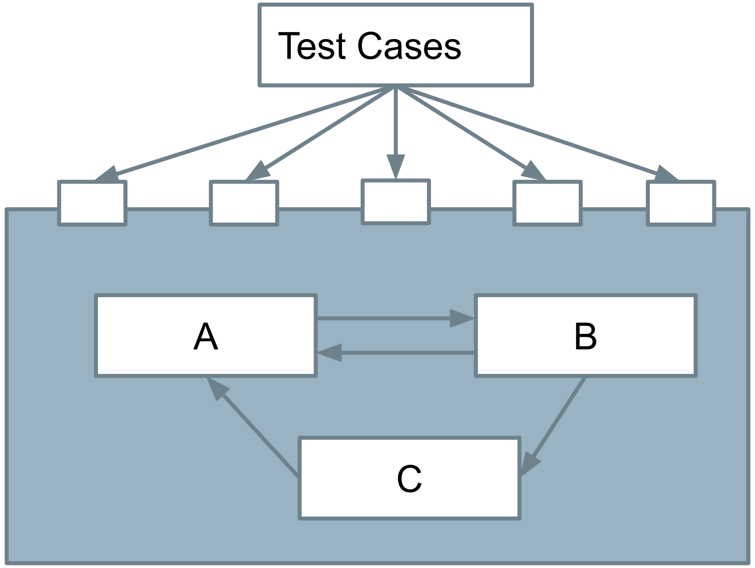
Integration Testing

- 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.
  - Integration testing focuses on showing that functionality accessed through this interface behaves according to the specifications.

Integration Testing

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

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

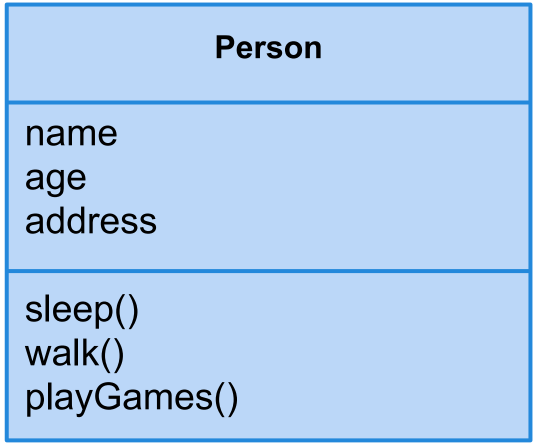


Object-Oriented Software

- 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...

Object-Oriented Software

- What defines an object:
  - Data representation
    - Characteristics that define an object (attributes).
  - Functionality
    - What the object can do (operations).



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

- Object behavior is **stateful**.
    - An object stores data and operates using that data.
    - The result of a method call depends on the state of the object - the values of its attributes.
- We cannot test a method in isolation.
    - Unit tests for classes in OO systems must put the object in the correct state by setting attributes and calling a *sequence* of methods.

State-Dependent Behavior

- The contents of the slots determine the legality of the model configuration.
- Are all components bound to compatible slots?
- Result of `checkConfiguration()` depends on the state.

```
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;
			}
		}
	}
}
```

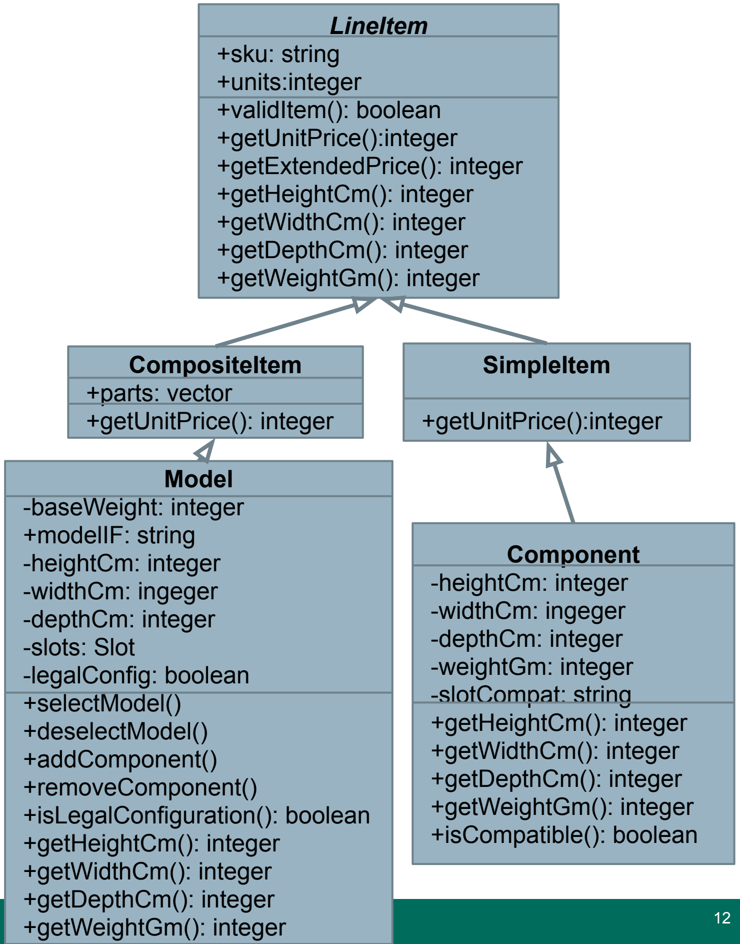
Encapsulation

- 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

- Child classes inherit attributes and operations from their parents.
  - Allows the creation of specialized versions of classes without reimplementing functionality.
  - All child objects are instances of that class and the parent class.

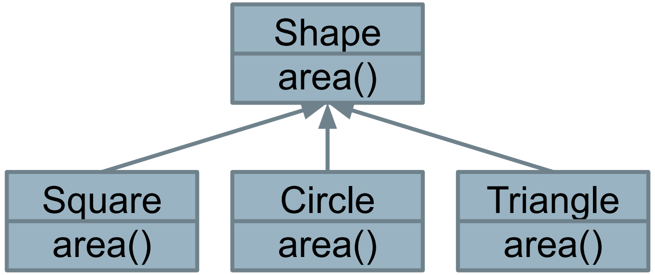


Inheritance

- 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.
  - 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.



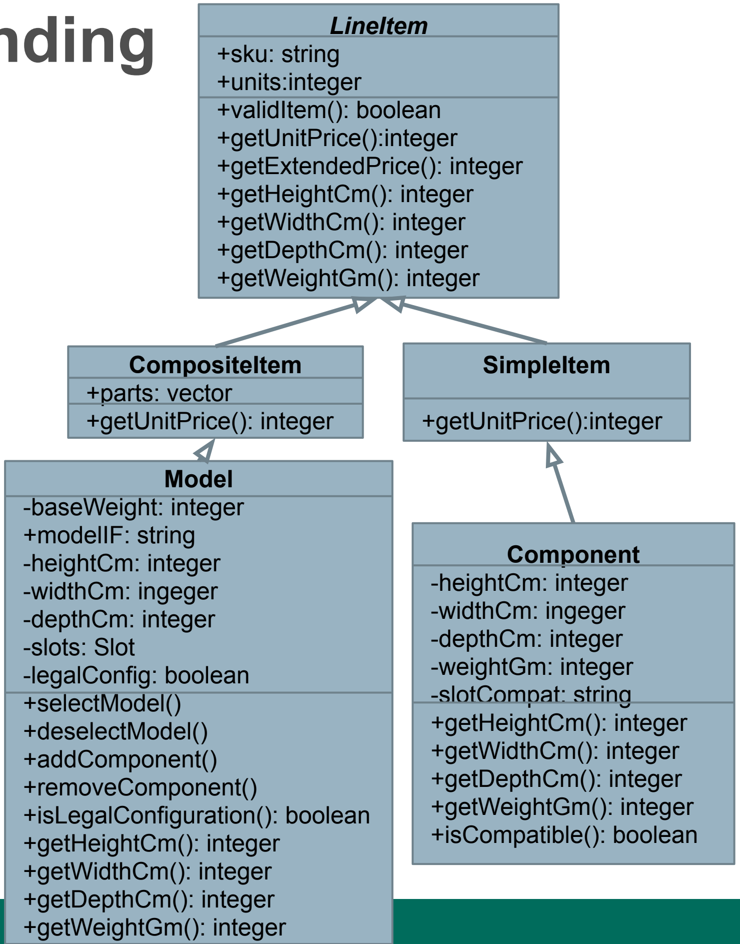
Because objects are instances of both their class and their parent class:

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

Gives the right answer if square, circle, triangle, etc are passed.

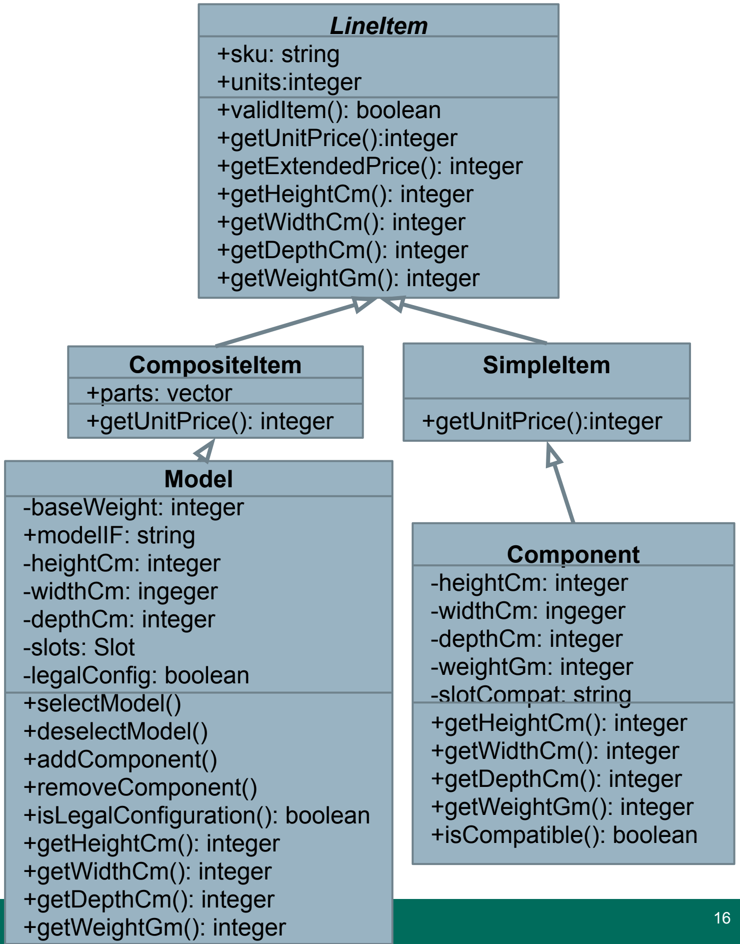
Polymorphism and Dynamic Binding

- 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 control-flow

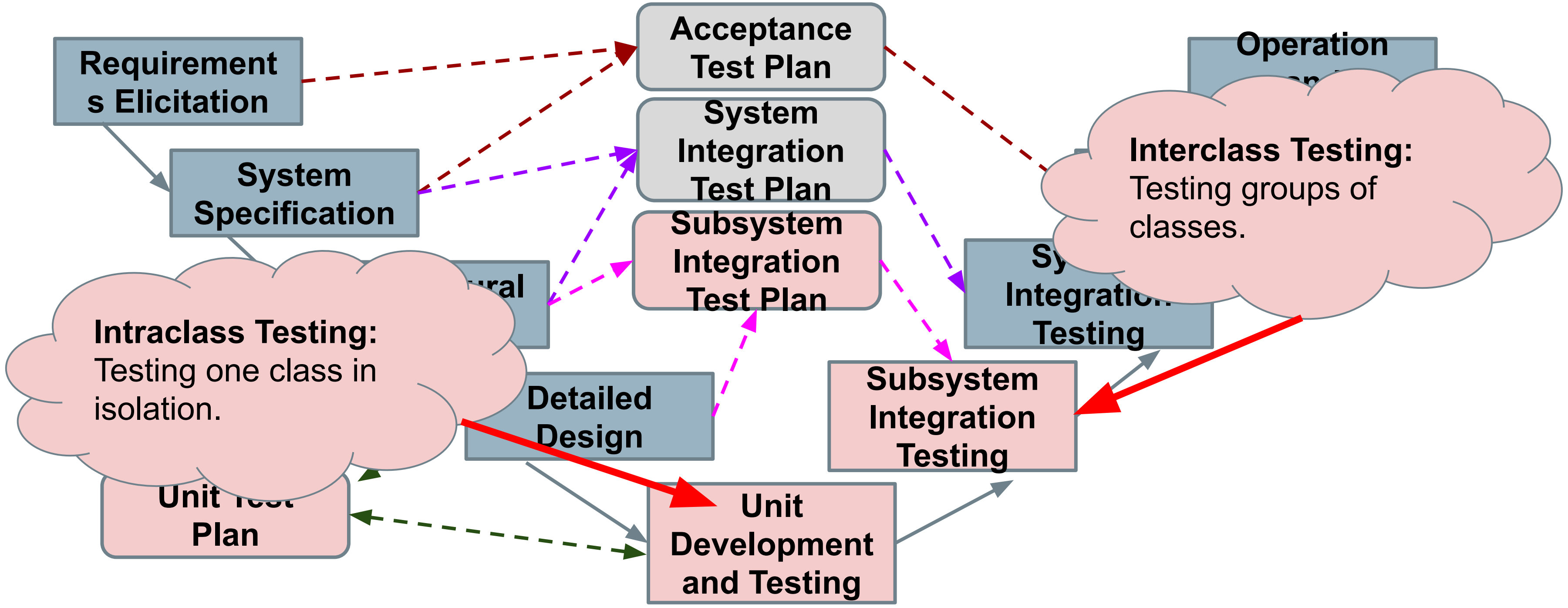
```
try{
    BufferedReader br = new
        BufferedReader(
        new File(“input.txt”));
    String line = br.readLine();
catch(IOException e){
    e.printStackTrace();
}


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


Requirements Elicitation
System Specification
ural
Detailed Design
Unit Test Plan
Acceptance Test Plan
System Integration Test Plan
Subsystem Integration Test Plan
Unit Development and Testing
Subsystem Integration Testing
System Integration Testing
Operation

Intraclass Testing: Testing one class in isolation.

Interclass Testing: Testing groups of classes.

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Intraclass Testing (single-class)

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

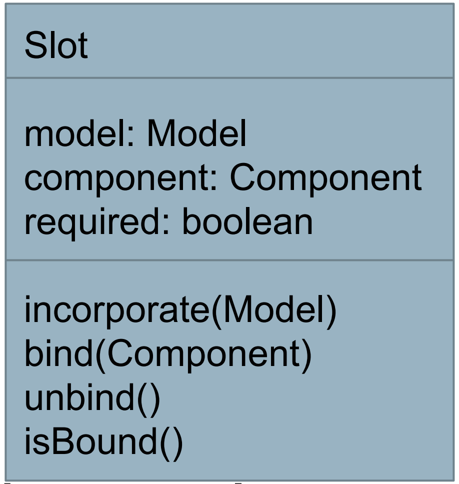
To test a class in isolation, we:

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.
    - Think of the class as a state machine model.

Intraclass Testing

4. Derive structural information from the source code 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.

Classes as State Machines

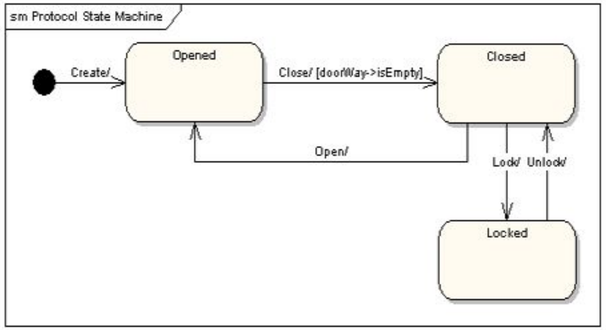


- The current value of the class-level variables defines the **state** of the class.
  - Combination of values of `model`, `component`, and `required`.
- The values of these variables influence the results of a method call.
  - `isBound()` returns true if `component` is not null.
- Values change as a result of method calls.
  - `bind(Component)` changes the value of `component`.

Classes as State Machines

- The state of an object implicitly impacts the result of a method call, and is changed by method calls.
  - Unit tests should attempt to cover the states of an object and transitions between those states.
  - Each unit test:
    - Consists of a series of method calls.
    - Should ensure that methods return the right result.
    - Should ensure that class-level attributes are set correctly (Is the class in the desired state?)

Finite State Machines



- A directed graph.
- Nodes represent states
  - An abstract description of the current value of an entity's attributes.
  - Should not represent **exact** values, but **types of values**.
    - (not 13, but "positive").
- Edges represent transitions between states.
  - Method calls cause the state to change.
  - Transitions represent method calls that change the state.

Using State Machine Models

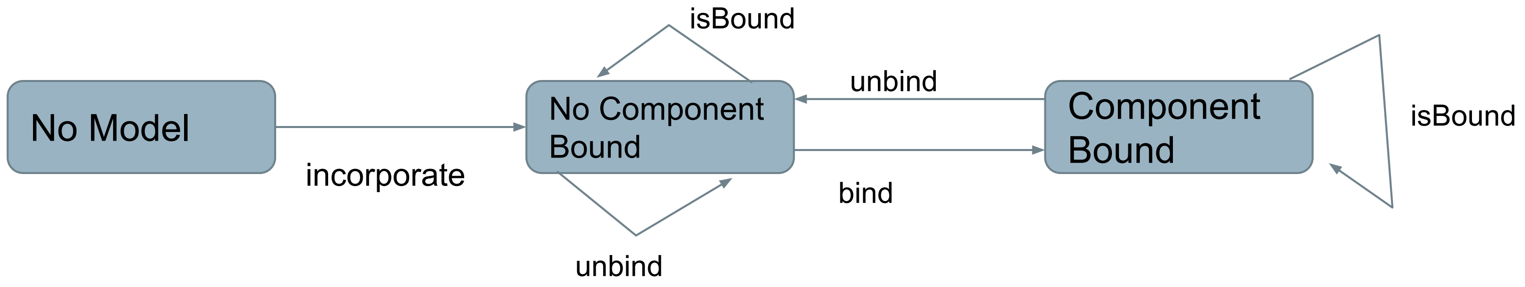
- We can identify method call sequences by covering a state machine model.
  - Map how method calls and attribute assignment can force the object into different states.
  - Sequence of transitions = sequence of method calls
  - Exercising that sequence should put the class into the the desired state.
    - (and cover different means of reaching those states)

Informal Specification

Slot represents a configuration choice in all instances of a particular model of computer. 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:

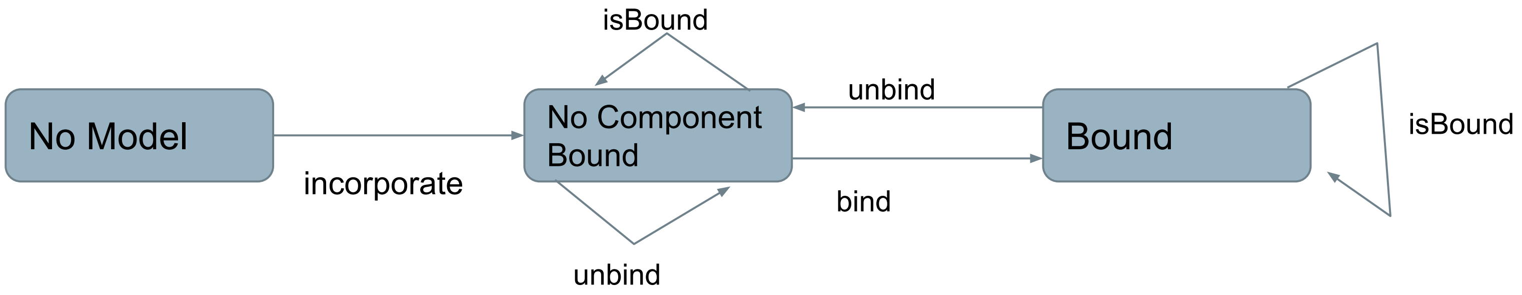
- **Incorporate:** Make a slot part of a model, and mark it as either required or optional. All instances of a model incorporate the same slots.
- **Bind:** Associate a compatible component with a slot.
- **Unbind:** The unbind operation breaks the binding of a component to a slot, reversing the effect 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.
  - Map variables to abstract values, not a state for each possible combination of values.
- Model how a method affects a class.
  - States only need to capture interactions between methods and the class state.

Test Coverage



- Tests should cover all states and transitions.
  - Do not do this in one test.
  - Split into smaller, targeted paths.
    - TC1: incorporate, isBound, bind, isBound
    - TC2: incorporate, unBind, bind, unBind, isBound

Example - Model

Model represents the current configuration of a model of computer.

- A given model may have zero or more slots, each of which is marked as required or optional.
- Each slot may contain a single component.
- To be a legal model, the model ID must exist in the ModelDB, each slot marked as required must be filled, the configuration must match that of the ModelDB entry for the model ID, and the optional components must match those allowed for that model in the ModelDB.

Example - Model

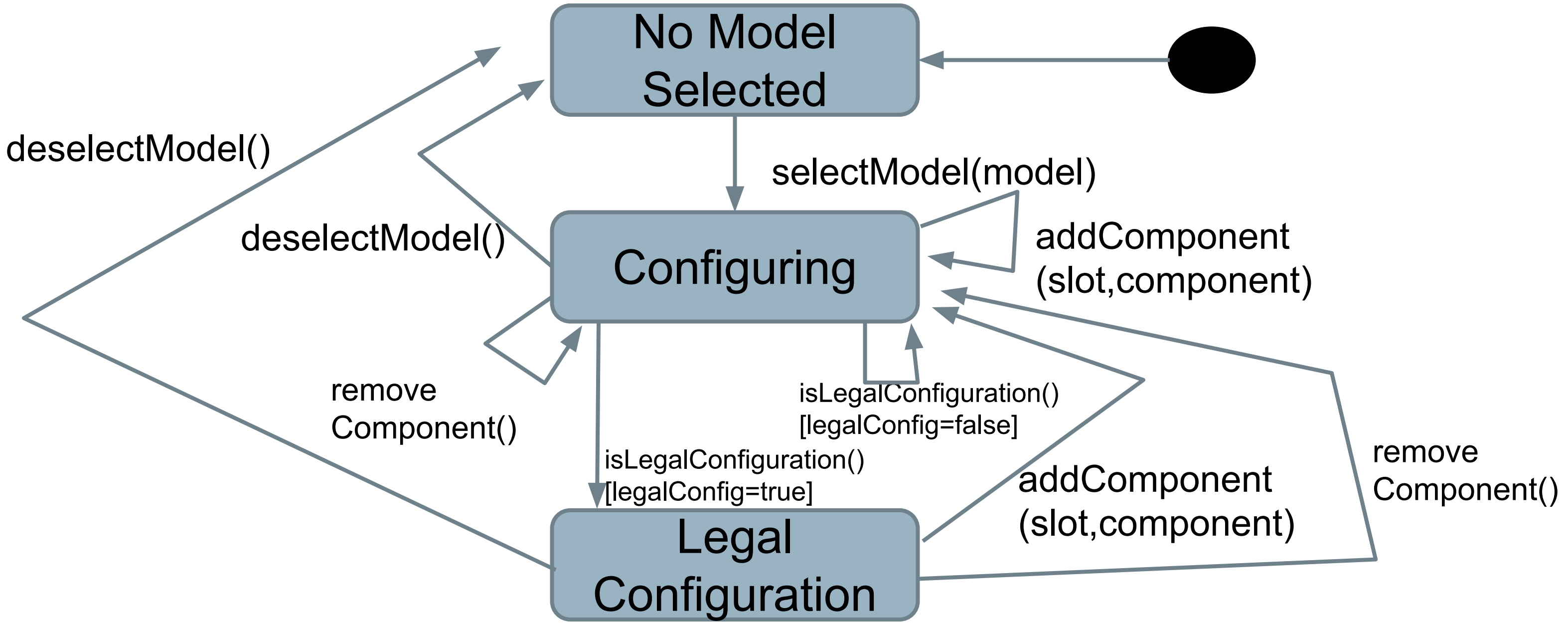
- **selectModel(modelId)**: Sets the model ID to the value passed in, as long as the model ID is set to “no model selected”. A model ID must be set before any other services are requested.
- **deselectModel():** Sets the model ID to “no model selected”. If the configuration was previously judged to be legal, it is no longer legal.
- **addComponent(slot, component):** Adds the selected component to the selected slot. If the configuration was previously judged to be legal, it is no longer legal.
- **removeComponent(slot)**: Removes the selected component to the selected slot. If the configuration was previously judged to be legal, it is no longer legal.
- **isLegalConfiguration():** Compares the current configuration to the entry in ModelDB. If the configuration is valid, the Model’s isLegal field is set to “true”.

Choosing States

No Model Selected | Configuring | Legal Configuration

- What does the class represent?
  - In this case: a computer model.
- What causes method results to differ?
  - Whether the model is legal or illegal.
- Can the class be in any other states?
  - We may not have set the model yet. We could still be making decisions and have not determined legality.

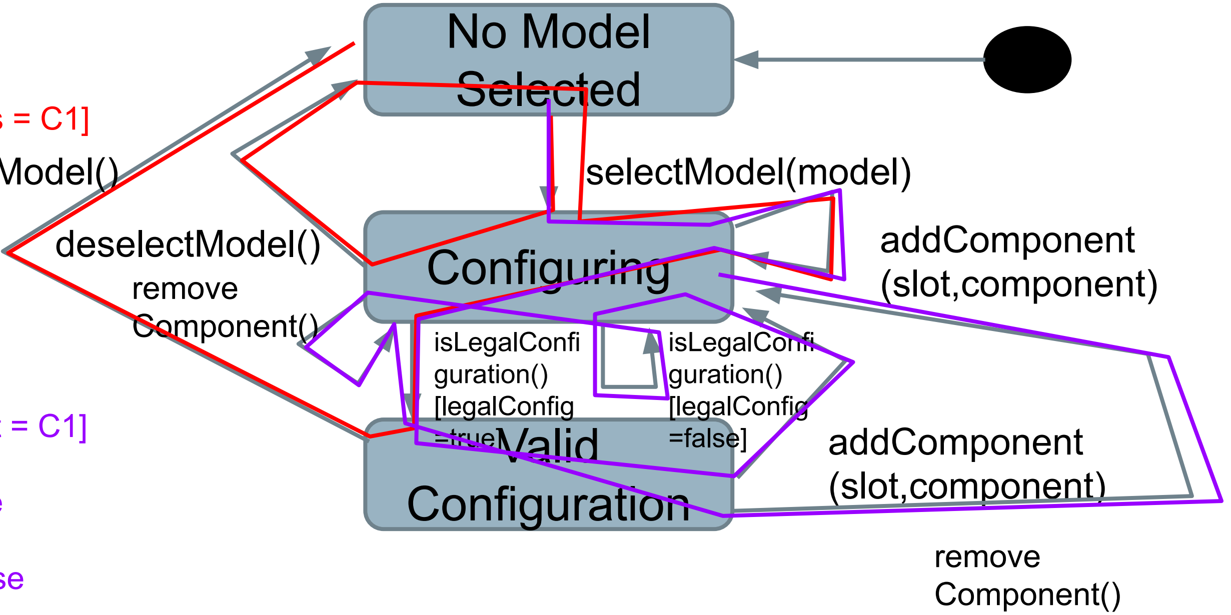
Choosing Transitions and Initial State


No Model Selected
Configuring
Legal Configuration
deselectModel()
deselectModel()
selectModel(model)
addComponent (slot,component)
remove Component()
isLegalConfiguration() [legalConfig=false]
isLegalConfiguration() [legalConfig=true]
addComponent (slot,component)
remove Component()

Choosing Test Cases

TC1:
selectModel(M1) [M1, 1 slots = C1]
deselectModel()
selectModel(M1)
addComponent(S1,C1)
isLegalConfiguration() //true
deselectModel()

TC2:
selectModel(M1) [M1, 1 slot = C1]
addComponent(S1,C1)
isLegalConfiguration() //true
addComponent(S2,C2)
isLegalConfiguration() // false
removeComponent(S2)
isLegalConfiguration() // true
removeComponent(S1)


No Model Selected
deselectModel()
selectModel(model)
deselectModel()
remove Component()
Configuring
addComponent (slot,component)
isLegalConfiguration() [legalConfig =true]
isLegalConfiguration() [legalConfig =false]
Valid Configuration
addComponent (slot,component)
remove Component()

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.

Let’s Take a Break

Interclass Testing (multi-class)

Interclass Testing

- Most software works by combining multiple, interacting components.
  - In addition to testing components independently, we must test their *integration*.
- Integration testing focuses on testing the compatibility of the interfaces of multiple classes.
  - Can they provide services to each other in a form that allows the calling class to complete its job.

Integration Faults

- Inconsistent interpretation of parameters or values
  - Developer implementing a class must interpret how to call methods from another class or subsystem.
  - Interpretation may be reasonable, but wrong.
  - Ex: Mix of metric and imperial units.
- Violation of value domains, capacity, or size limit
  - Implicit assumptions on ranges of values or sizes.
  - Buffer overflow when one class assumes a different memory capacity for a variable than another.

Integration Faults

- Side-effects on parameters or resources
  - Classes often make use of resources not mentioned by their interface. If two classes attempt to use the same resource, problems may arise.
  - Ex: Both classes create a temporary file called “tmp”
- Missing or misunderstood functionality
  - Underspecification of functionality may lead to incorrect assumptions about expected results.
  - Ex: multiple ways to count hits to a webpage. Client could get unexpected results if unknown how counting is done.

Integration Faults

- Nonfunctional problems
  - Performance expectations only explicitly stated if expected to be a problem.
  - Module interactions can lower performance, availability, security below acceptable thresholds.
- Dynamic mismatches
  - Polymorphic calls may be dynamically bound to incompatible methods.

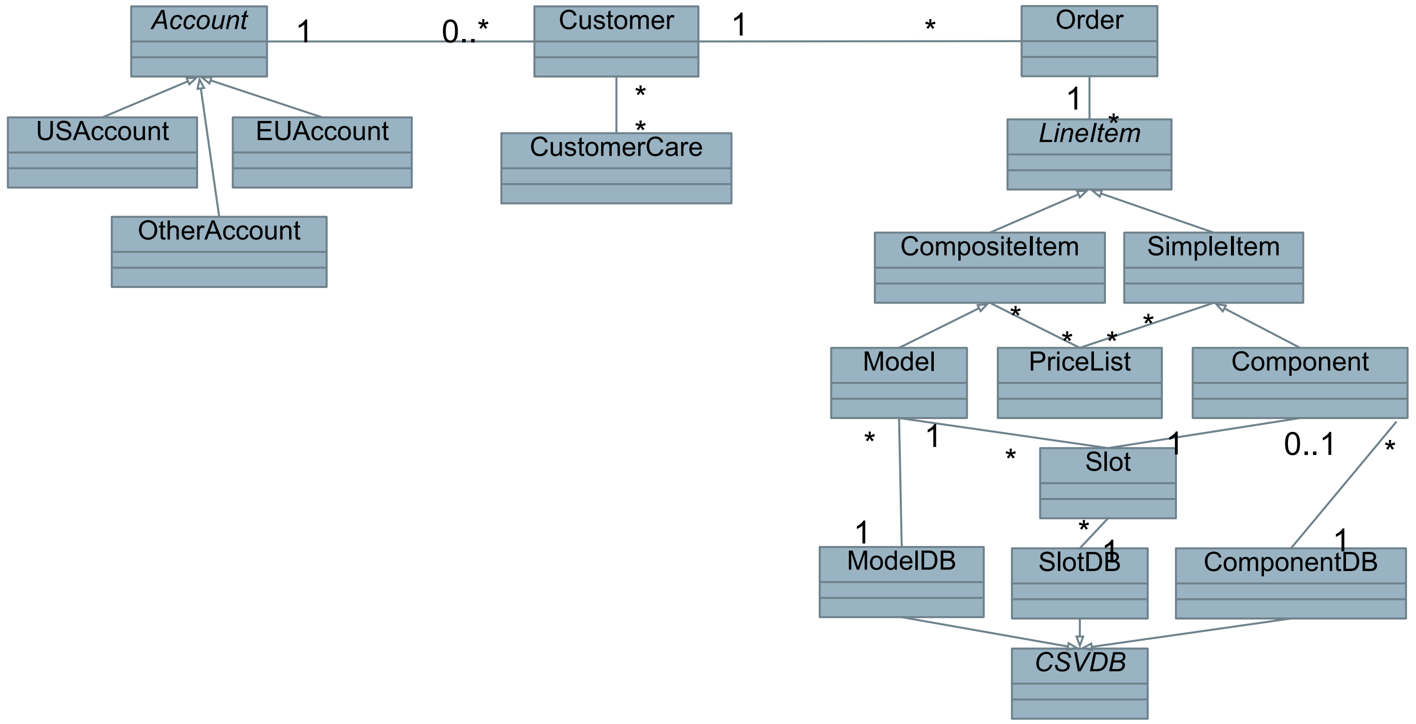
Interclass Testing

- When should we test a particular class that depends on other classes?
    - Identify a hierarchy of classes based on dependencies.
    - Use this hierarchy to decide how and when to integrate classes and test them.
        - Start from the bottom-up, or mock classes and work from the top-down.

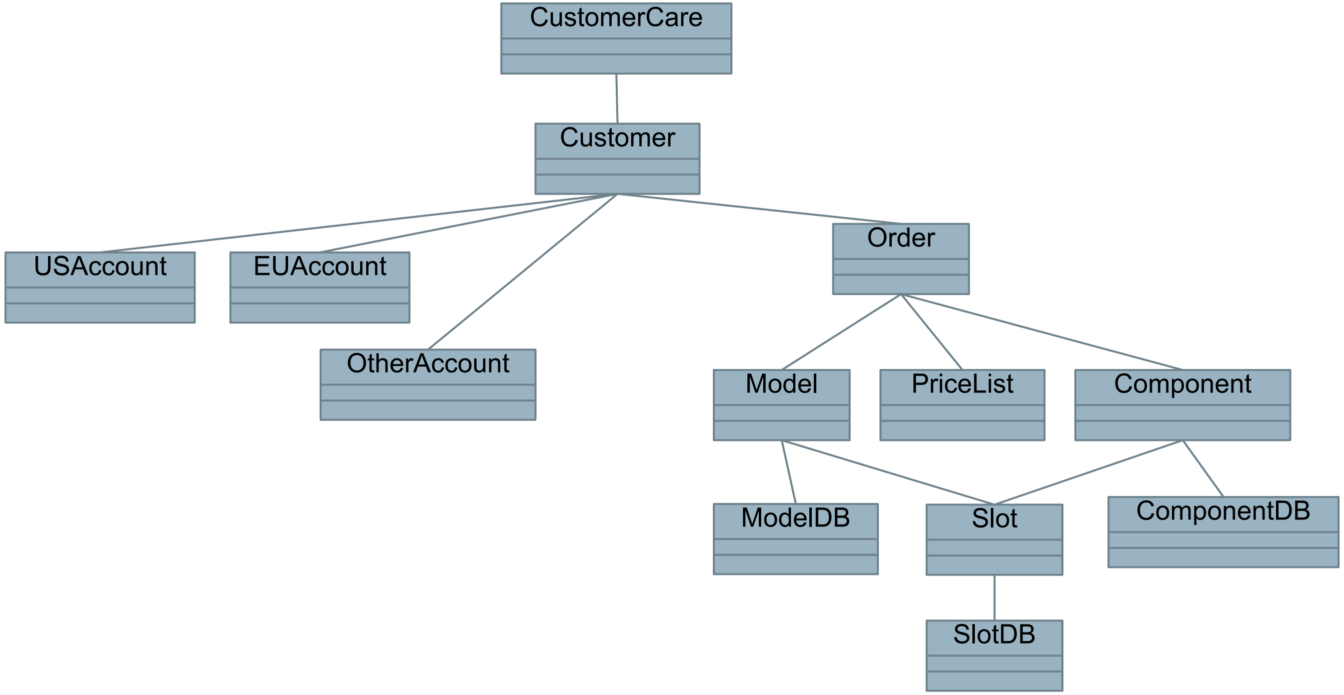
Dependency

- 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.

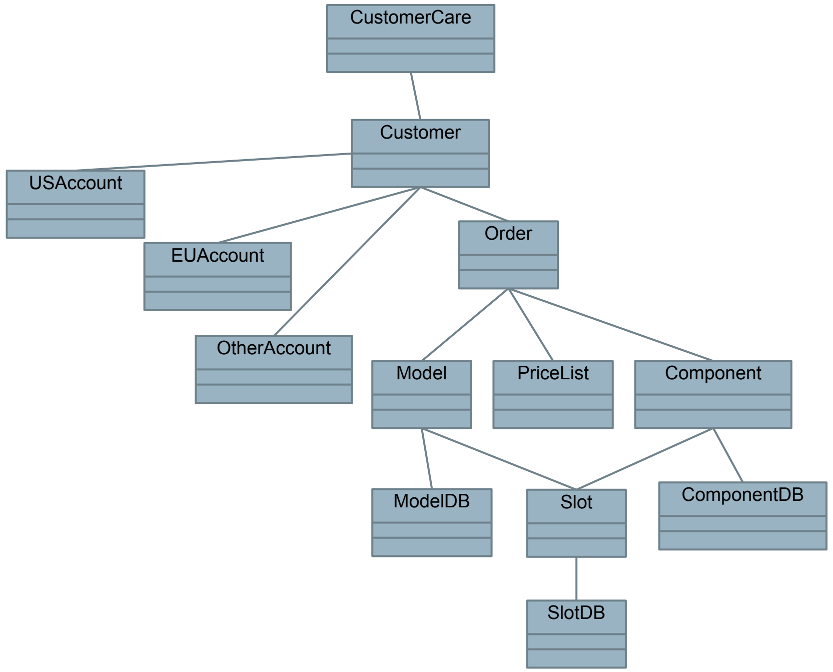
Deriving the Use/Include Hierarchy



Deriving the Use/Include Hierarchy



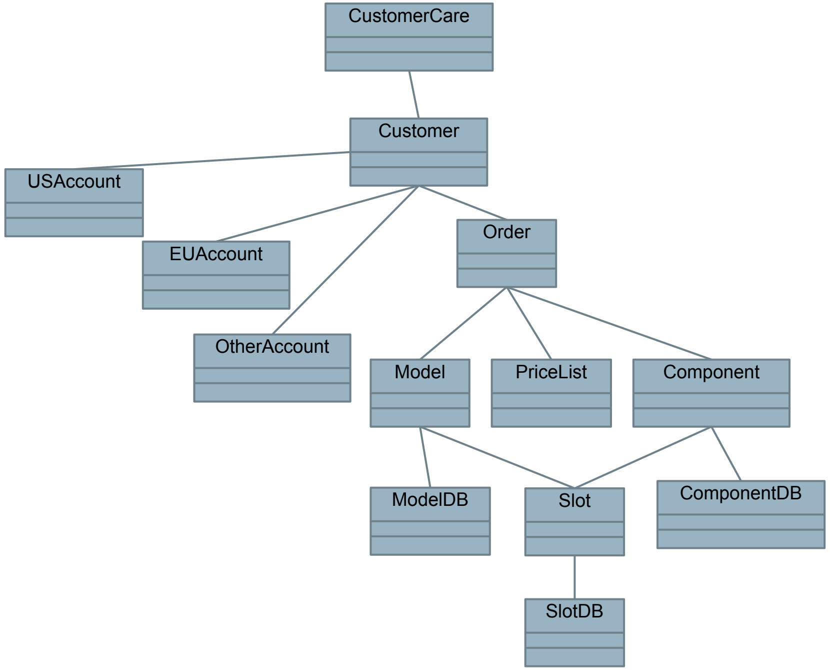
Incremental Testing



- Test pieces of the system as they are completed.
  - Use scaffolding (stubs, drivers) to test classes in isolation, then swap out for real components to test integration.
- Complex interactions can hide the source of failures, so test a small collection of classes before adding more.

Bottom-Up Testing

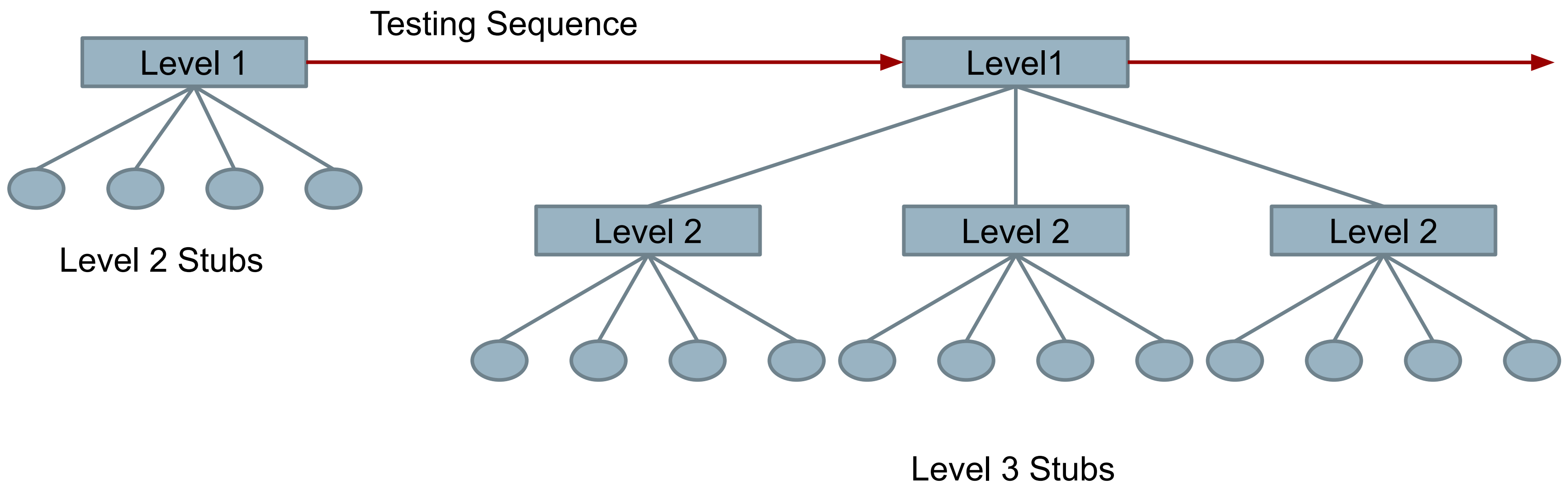
- Start testing from the bottom-up.
  - Start from classes with no dependency, then move up in the hierarchy.
  - Integrate SlotDB with Slot, Component with ComponentDB.
  - Then ModelDB with Model and Slot.
  - … up to Order with all below.



Bottom-Up Testing

- Start with the lower levels of a system and work your way upwards.
- Necessary for testing critical infrastructure.
- Very good at testing individual components.
  - But, may not find major architectural problems.
  - Top-Down Testing aids in finding issues related to how classes are integrated together.

Top-Down Testing



Top-Down Testing

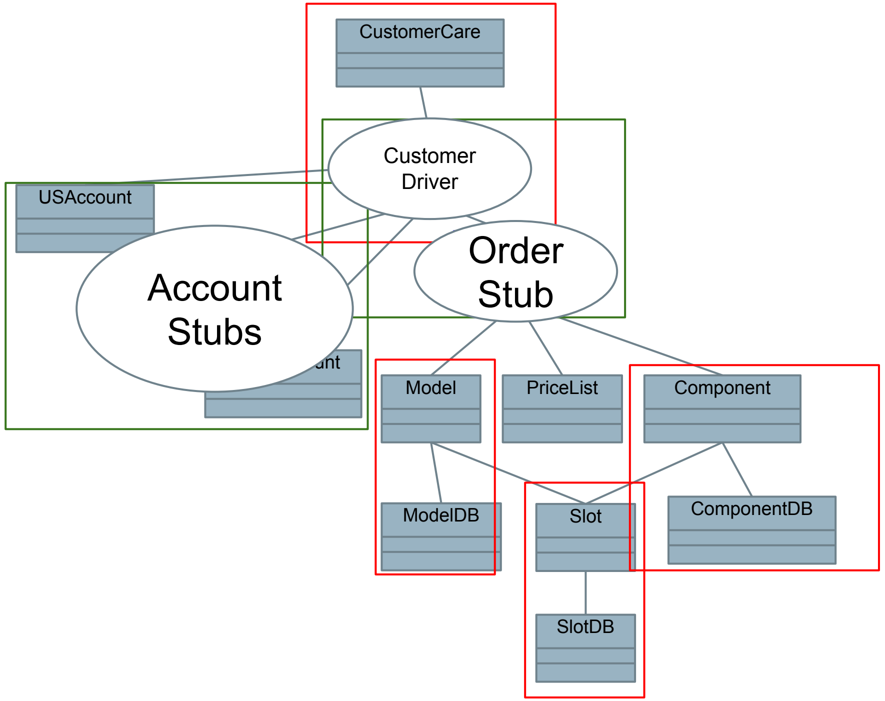
- Start with the high levels of system hierarchy and work your way downwards.
  - Lower levels are replaced with mock objects.
- Very good for finding architectural or integration errors.
- May need system infrastructure in place before testing is possible.
- Requires large effort in developing stubs.

Top-Down and Bottom-Up

- Both approaches can be applied if classes are delivered according to the hierarchy.
- Both approaches are effective at exposing integration issues.
- Real systems are rarely developed unidirectionally.
  - Driven by reuse of components or integration of libraries.
  - Driven by need to develop early prototypes.
  - Can combine elements of each approach.

Sandwich/Backbone Testing

- Starts top-down
  - Develop early prototype for feedback.
- Integrate modules bottom-up as built.
- Adds flexibility, but hard to plan and monitor.



CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

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 tests 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.

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 sequence diagrams.
    - Describe object interactions in service of a goal.
  - Vary these scenarios to capture additional illegal interaction sequences.

Addressing OO Testing Issues

Classes are Stateful

- A class has state, and state is impacted by the class' methods.
  - The state of the class is the result of a sequence of methods called.
  - Functional testing of classes focused on identifying the sequence of method calls that would cover different states.
- Structural techniques must also extend control and data flow across sequences of method calls.

Direct and Indirect Coverage

- Expressions in a method can be covered either through a **direct** call from a test case or a call from one method to another.
    - The test calls Method A, then Method A calls Method B. Code in Method B is covered **indirectly**.
- How coverage is attained may impact the likelihood of fault detection.
    - Coverage is attained contextually. Direct and indirect method calls cover same elements with different input.

Direct and Indirect Coverage

- Branch Coverage of `faultyAdd` can be attained through direct method calls from a test case, or by calling add (which calls `faultyAdd`).
- Indirect coverage cannot reveal the fault.
- Must consider the context in which coverage is attained.

```
public int[] add(int[] values, int valueToAdd){
    for(int i = 0; i < values.size(); i++){
        if(valueToAdd >= 0){ values[i] =
            faultyAdd(values[i], valueToAdd);
         }
    }
    return values;
}

public int faultyAdd(int value, int valueToAdd){
    if (valueToAdd <= 0){
        // FAULT, should be ==
        return value;
    }
    return value + valueToAdd;
}
```

Structural Testing of Classes

- Private methods can **only** be covered indirectly.
  - Coverage may be difficult to achieve.
- Context is important.
  - 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;
	}
}
```

Writing Oracles for Classes

- Correctness of a method is judged on the output of the method **and** the state of the object.
  - `deselectModel()` should clear array slots on the object.
- Oracles must check validity of 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 object inspection at runtime.
- Can be used to identify classes, 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.

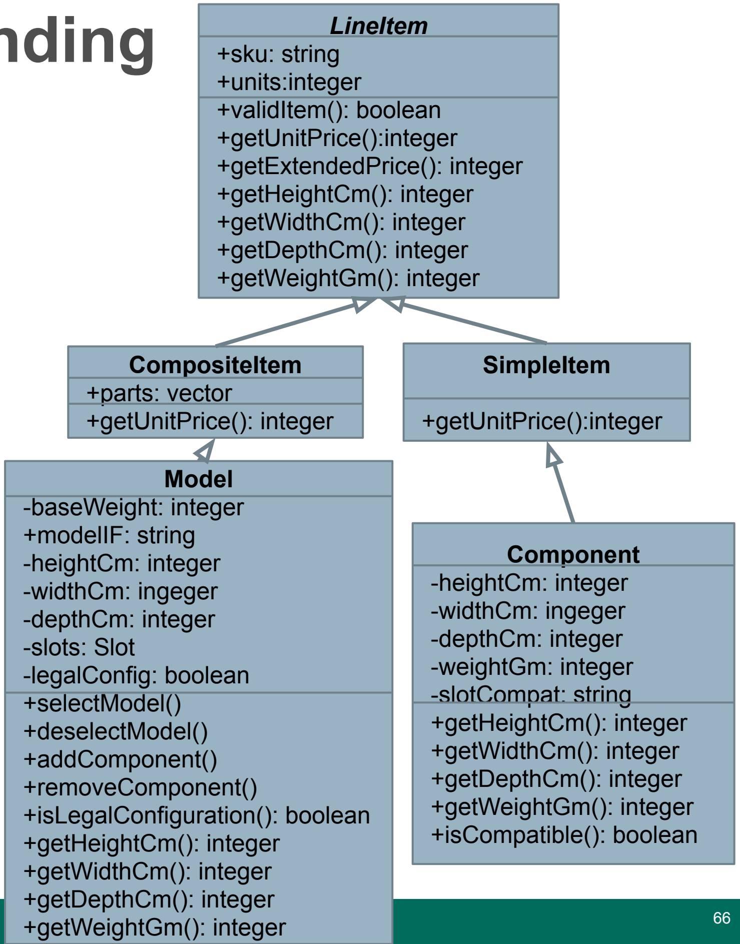
Option 2: Java Reflection

- 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("privateString");
privateStringField.setAccessible(true);

String fieldValue = (String)
privateStringField.get(privateObject);
System.out.println("fieldValue = " + fieldValue);
```

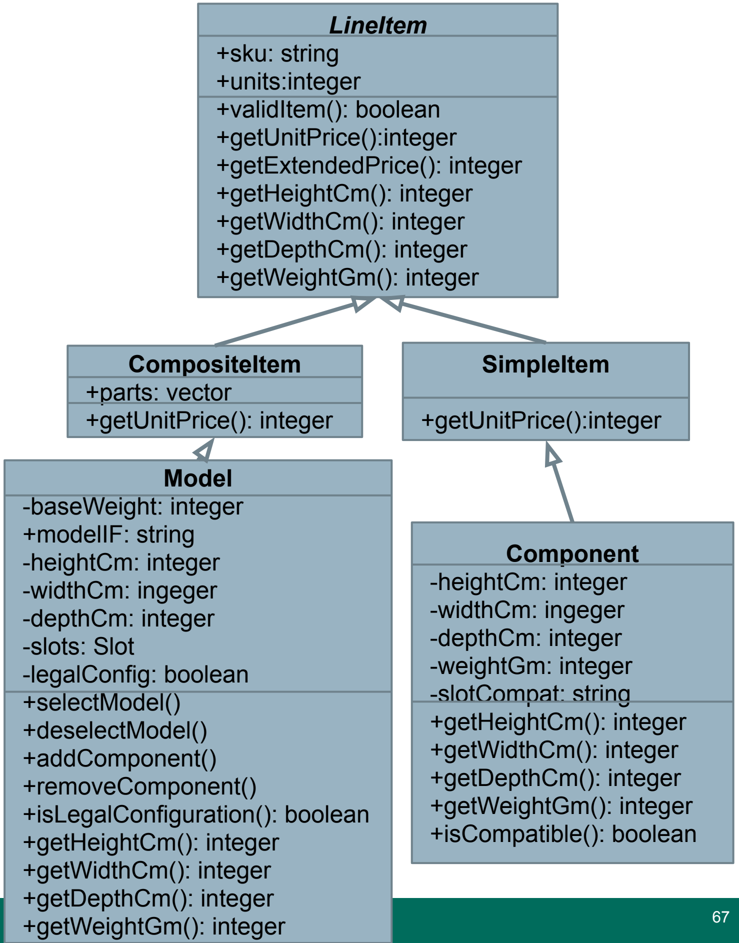
Polymorphism and Dynamic Binding

- 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 - try multiple bindings.



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

- A method can be abstract in parent, child, or both.
- We can categorize abstract methods as:
  - **Abstract New:** New and abstract in the child.
  - **Abstract Recursive:** Inherited when *the ancestor's version was abstract*.
    - Still abstract in child.
  - **Abstract Redefined:** Redefined when *the ancestor's version was abstract*.
    - Child version is not abstract.

Inheritance and Test Reuse

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

Genericity

- 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

- 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 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.
  - Still 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 indicating 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 indicating 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.

We Have Learned

- Testing of OO systems is impacted by
  - State Dependent Behavior
  - Encapsulation
  - Inheritance
  - Polymorphism and Dynamic Binding
  - Abstract Classes
  - Exception Handling
  - Concurrency

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 and top-down.

Next Class

- Exercise Session - Structural Testing
- Next class - Testing Near and Post-Release
  - Reading - Optional Reading - Pezze and Young, Chapter 22
- Assignment 2
  - Due March 1, questions?

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY