Testing Object Oriented Software

Dr. Paul West

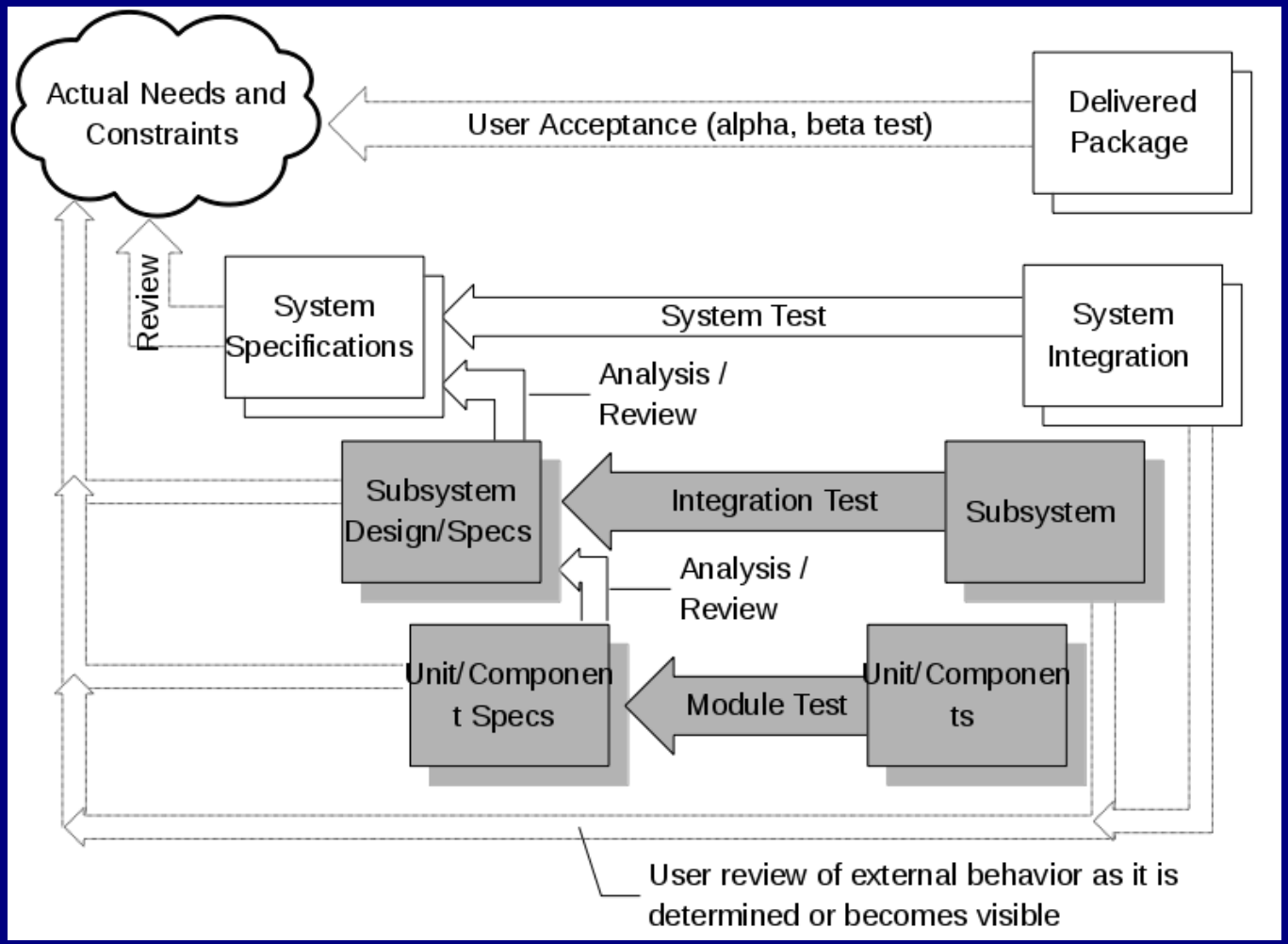
Department of Computer Science
College of Charleston

March 20, 2014

Typical OO software characteristics that impact testing

- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling

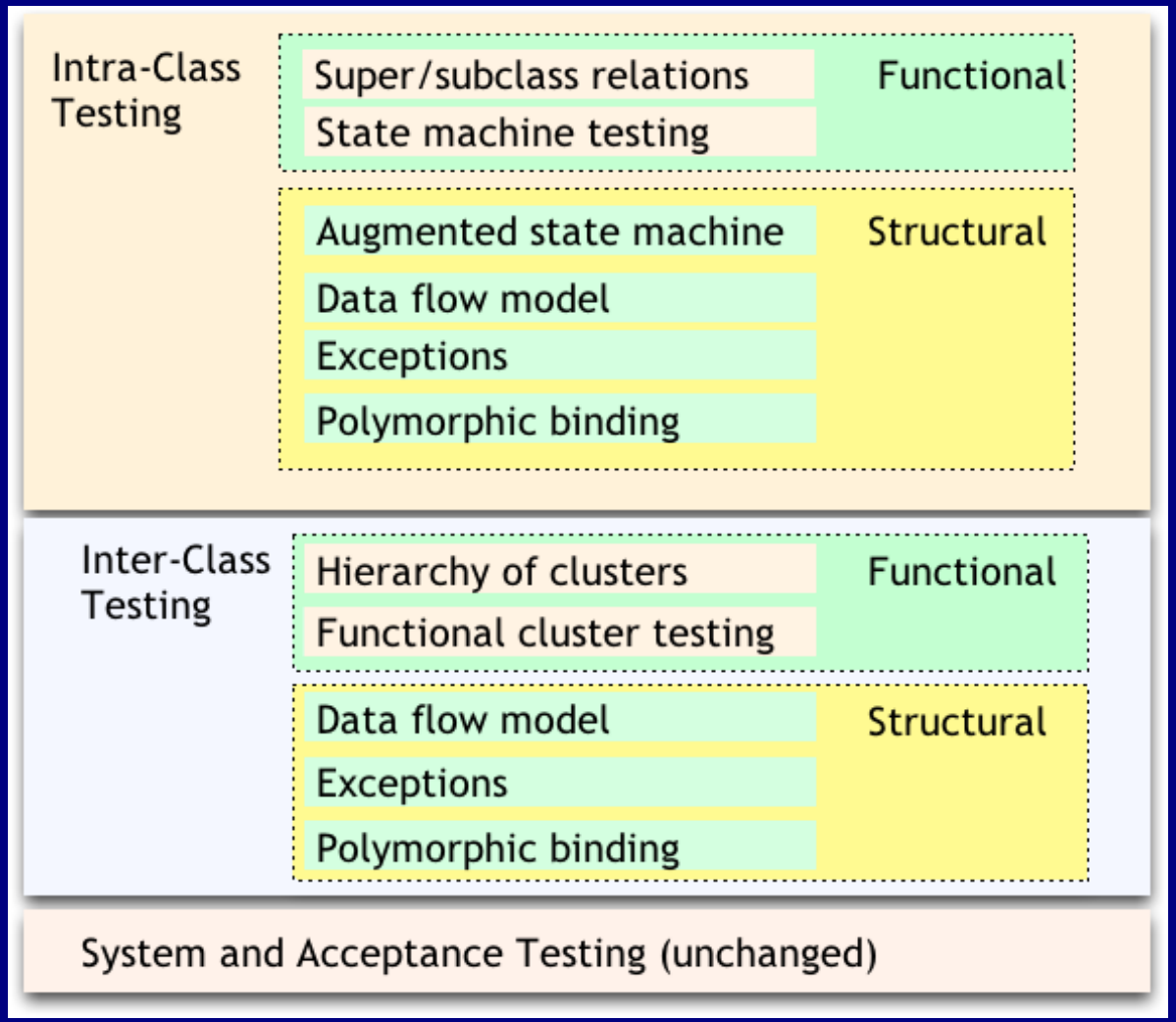
Quality activities and OO SW



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

Orthogonal approach: Stages



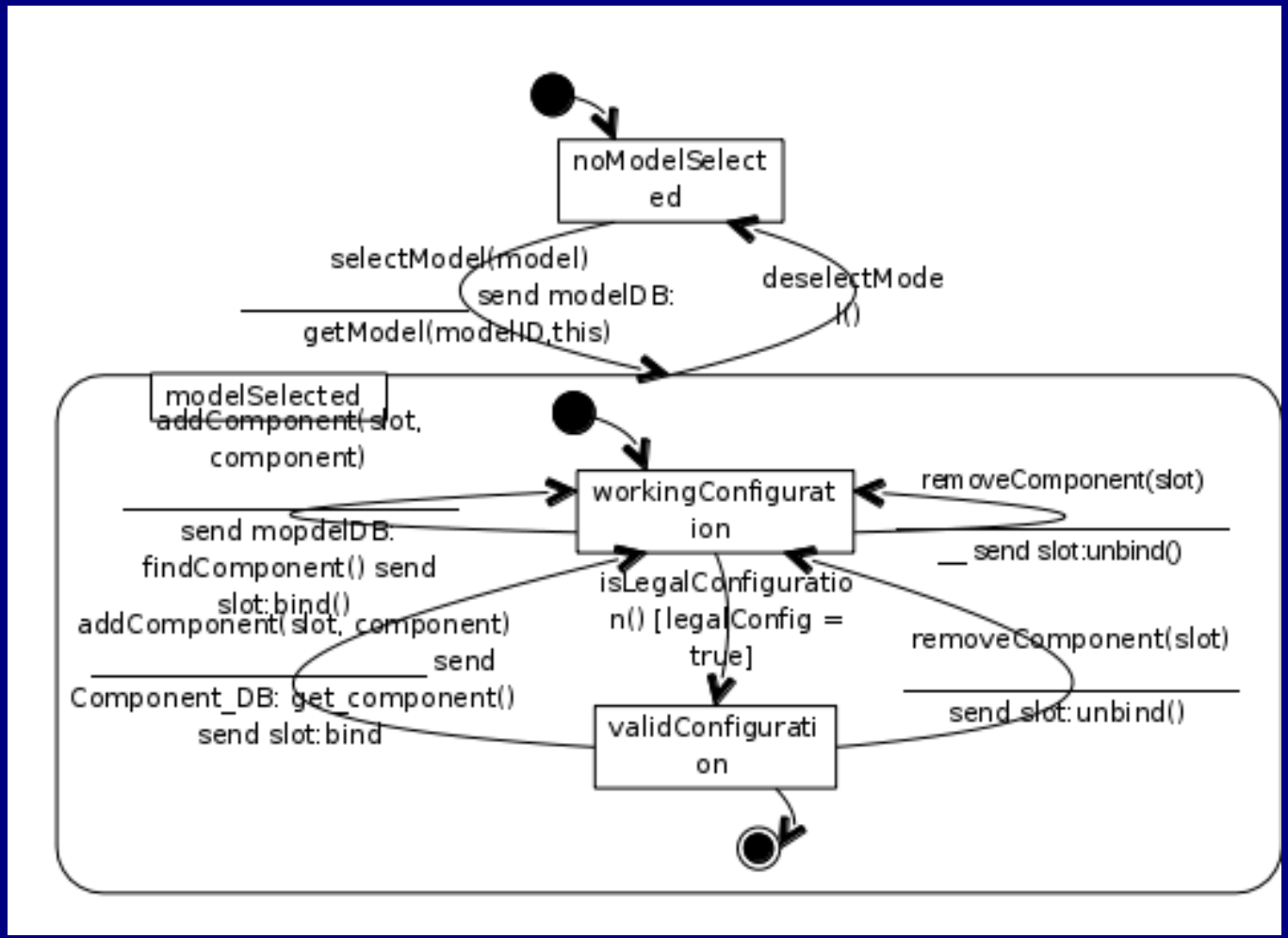
Intraclass State Machine Testing

- 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

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



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)

Interclass Testing

- The first level of integration testing for object-oriented software
  - Focus on interactions between classes
- Bottom-up integration according to “depends” relation
  - A depends on B: Build and test B, then A
- Start from use/include hierarchy
  - Implementation-level parallel to logical “depends” relation
  - Class A makes method calls on class B
  - Class A objects include references to class B methods
    - but only if reference means “is part of”

Interactions in Interclass Tests

- Proceed bottom-up
- Consider all combinations of interactions
  - example: a test case for class Order includes a call to a method of class Model, and the called method calls a method of class Slot, exercise all possible relevant states of the different classes
  - problem: combinatorial explosion of cases
  - so select a subset of interactions:
    - arbitrary or random selection
    - plus all significant interaction scenarios that have been previously identified in design and analysis: sequence + collaboration diagrams

Using Structural Information

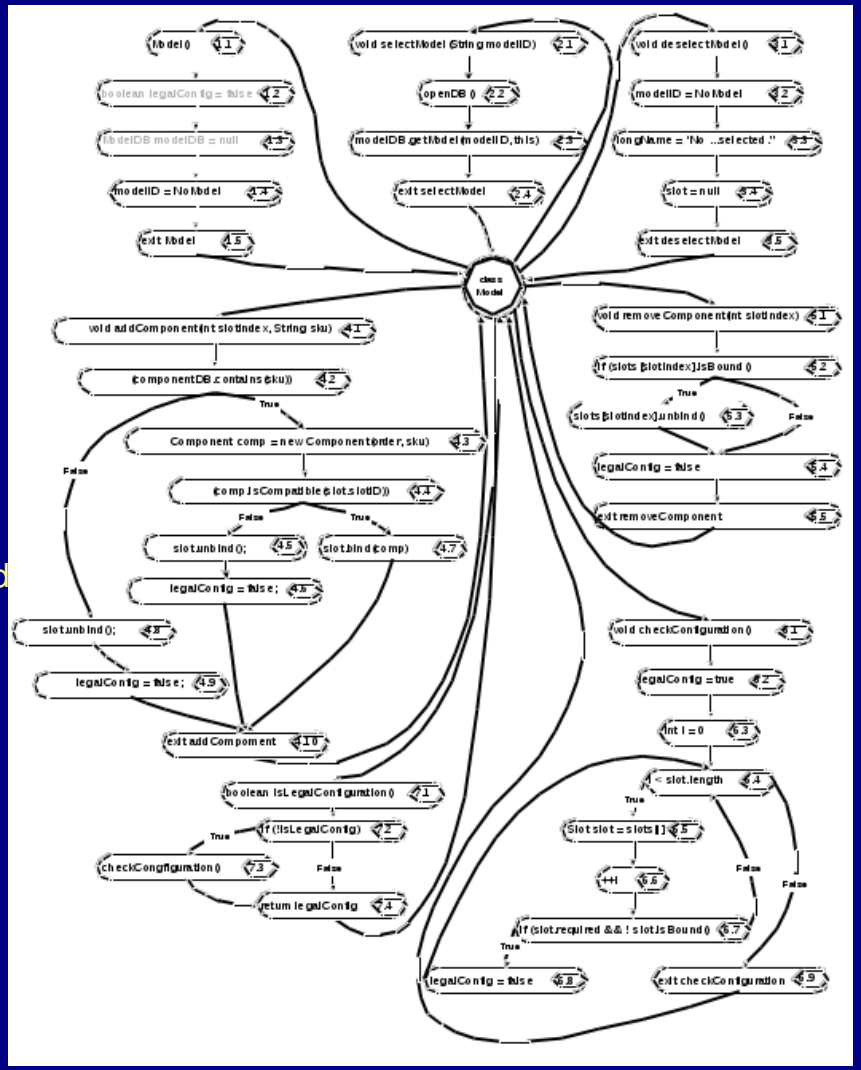
- Start with functional testing
  - As for procedural software, the specification (formal or informal) is the first source of information for testing object-oriented software
    - “Specification” widely construed: Anything from a requirements document to a design model or detailed interface description
- Then add information from the code (structural testing)
  - Design and implementation details not available from other sources

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 ...

Control flow **for** each method
+
node **for** class
+
edges
from node class to the start nodes of the method
from the end nodes of the methods to node class

=> control flow through sequences of method call



Interclass structural testing

- Working “bottom up” in dependence hierarchy
  - Dependence is not the same as class hierarchy; not always the same as call or inclusion relation.
  - -> May match bottom-up build order
- Starting from leaf classes, then classes that use leaf classes, ...
  - Summarize effect of each method: Changing or using object state, or both
  - Treating a whole object as a variable (not just primitive types)

Inspectors and modifiers

- Classify methods (execution paths) as
  - inspectors: use, but do not modify, instance variables
  - modifiers: modify, but not use instance variables
  - inspector/modifiers: use and modify instance variables
- Example - class slot:
  - Slot() modifier
  - bind() modifier
  - unbind() modifier
  - isbound() inspector

Definition-Use (DU) pairs

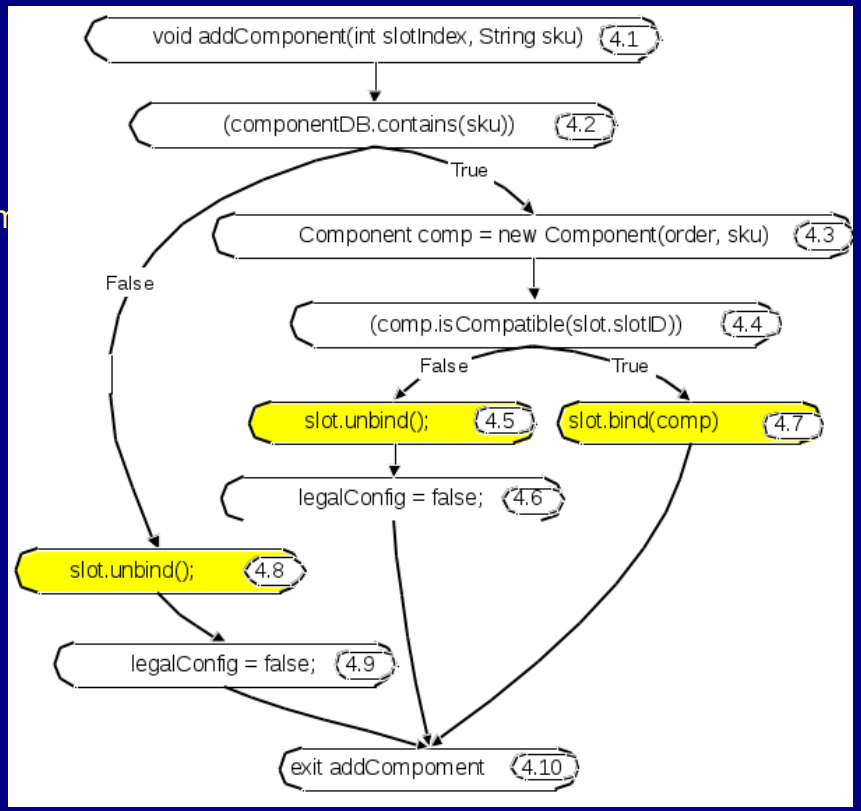
instance variable legalConfig

<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

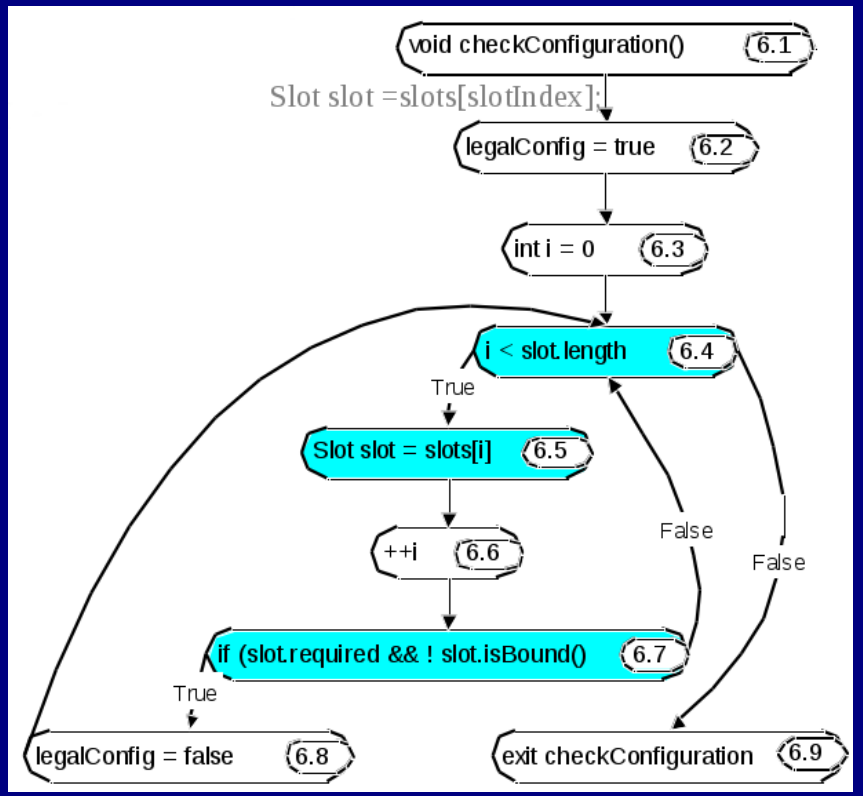
Definitions of instance variable slot in class m
addComponent (4.5)
addComponent (4.7)
addComponent (4.8)
selectModel (2.3)
removeComponent (5.3)

Slot() modifier
bind() modifier
unbind() modifier
isbound() inspector



Uses of instance variables slot in class model
removeComponent (5.2)
checkConfiguration (6.4)
checkConfiguration (6.5)
checkConfiguration (6.7)

Slot() modifier
bind() modifier
unbind() modifier
isbound() inspector



Stubs, Drivers, and Oracles for Classes

- 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 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

Oracles

- Test oracles must be able to check the correctness of the behavior of the object when executed with a given input
- Behavior produces outputs and brings an object into a new state
  - We can use traditional approaches to check for the correctness of the output
  - To check the correctness of the final state we need to access the state

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
  - Whitebox!
- 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

```
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()
deselectModel()
selectModel(M2)
addComponent(S1,C1)
isLegalConfiguration()
```

EQUIVALENT

```
selectModel(M2)
addComponent(S1,C1)
isLegalConfiguration()
```

NON EQUIVALENT

```
selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()
```

Generating Equivalent Sequences

- remove unnecessary (“circular”) methods
  selectModel(M1)
  addComponent(S1,C1)
  addComponent(S2,C2)
  isLegalConfiguration()
  deselectModel()
  selectModel(M2)
  addComponent(S1,C1)
  isLegalConfiguration()

Generating non-equivalent scenarios

- Remove and/or shuffle essential actions
- Try generating sequences that resemble real faults

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