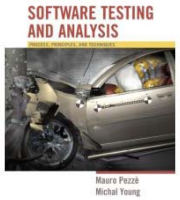
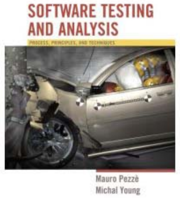
Testing Object Oriented Software

Chapter 15



Learning objectives

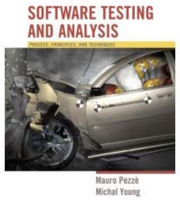
- Understand how object orientation impacts software testing
  - What characteristics matter? Why?
  - What adaptations are needed?
    - Understand basic techniques to cope with each key characteristic
- Understand staging of unit and integration testing for OO software (intra-class and inter-class testing)



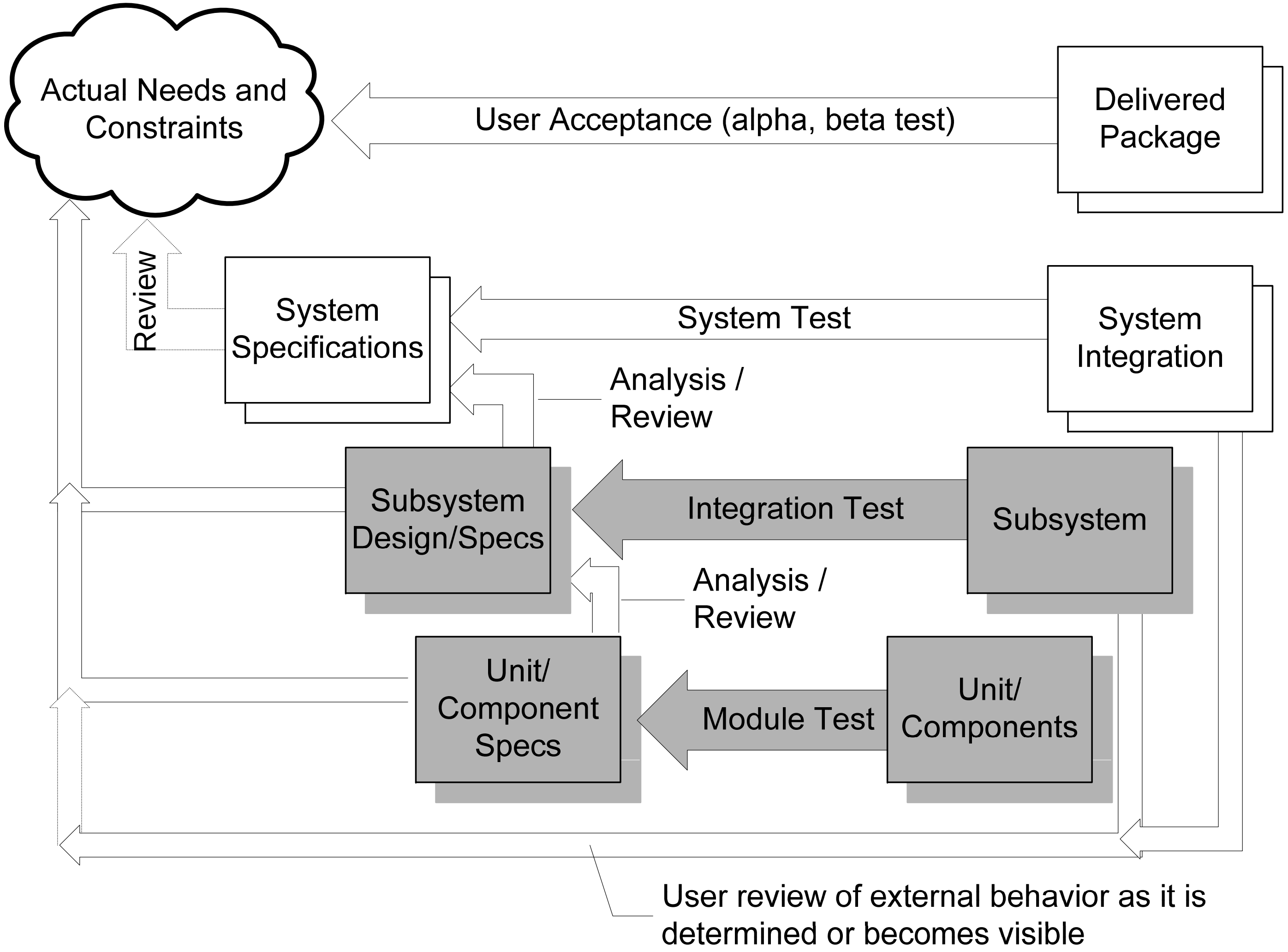
Characteristics of OO Software

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



Actual Needs and Constraints

User Acceptance (alpha, beta test)

Delivered Package

Review

System Specifications

System Test

System Integration

Analysis / Review

Subsystem Design/Specs

Integration Test

Subsystem

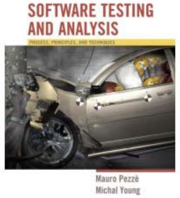
Analysis / Review

Unit/ Component Specs

Module Test

Unit/ Components

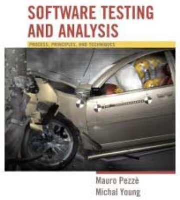
User review of external behavior as it is determined or becomes visible



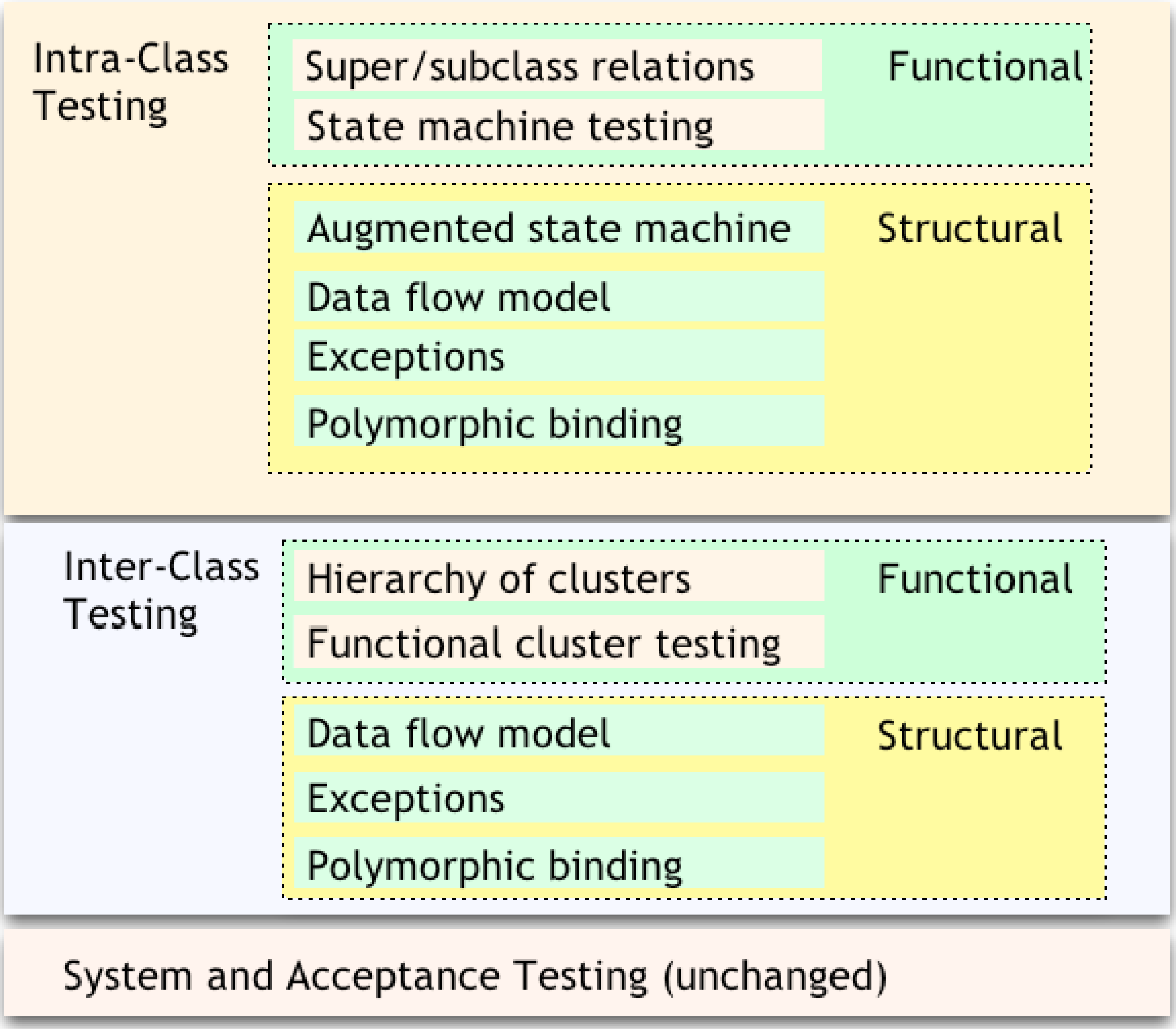
OO definitions of unit and integration testing

- 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



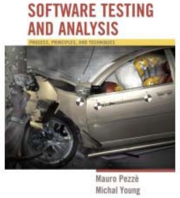
Intra-Class Testing

- Functional
  - Super/subclass relations
  - State machine testing
- Structural
  - Augmented state machine
  - Data flow model
  - Exceptions
  - Polymorphic binding

Inter-Class Testing

- Functional
  - Hierarchy of clusters
  - Functional cluster testing
- Structural
  - Data flow model
  - Exceptions
  - Polymorphic binding

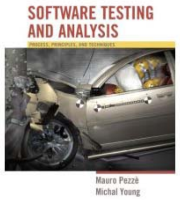
System and Acceptance Testing (unchanged)



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

[Later: Inheritance and dynamic binding]



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:

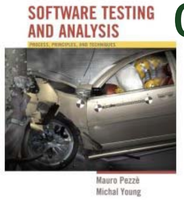
- **Install**: slots can be installed on a model as *required* or *optional*.

  …

- **Bind**: slots can be bound to a compatible component.

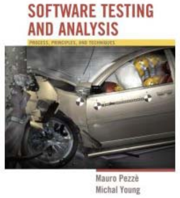
  …

- **Unbind**: bound slots can be unbound by removing the bound component.
- **IsBound**: returns the current binding, if bound; otherwise returns the special value *empty*.

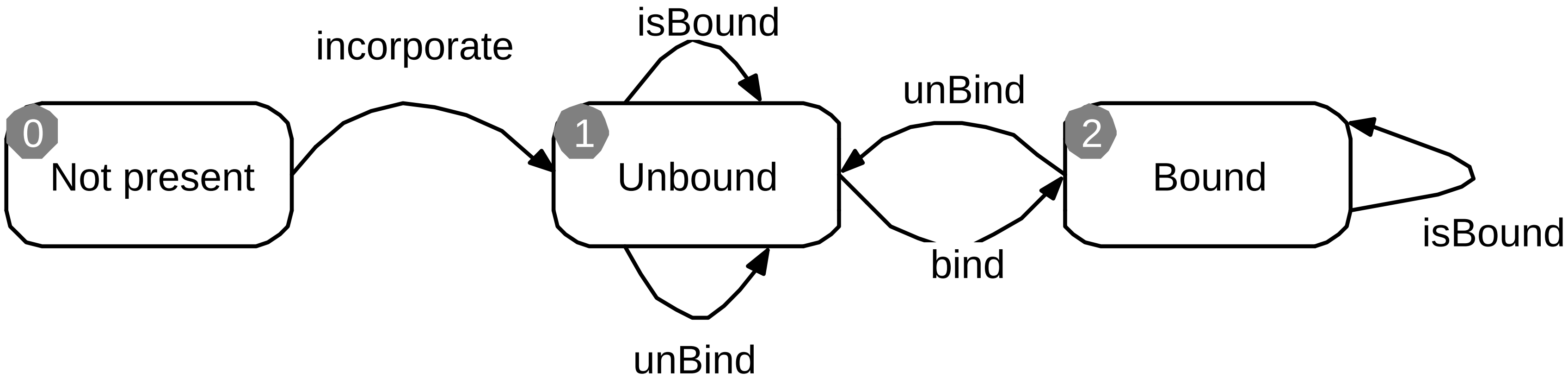


Identifying states and transitions

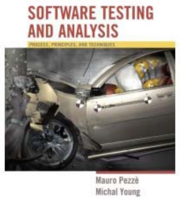
- From the informal specification we can identify three states:
  - Not_installed
  - Unbound
  - Bound
- and four transitions
  - install: from Not_installed to Unbound
  - bind: from Unbound to Bound
  - unbind: …to Unbound
  - isBound: does not change state



Deriving an FSM and test cases

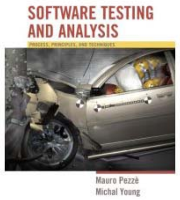


- TC-1: incorporate, isBound, bind, isBound
- TC-2: incorporate, unBind, bind, unBind, isBound

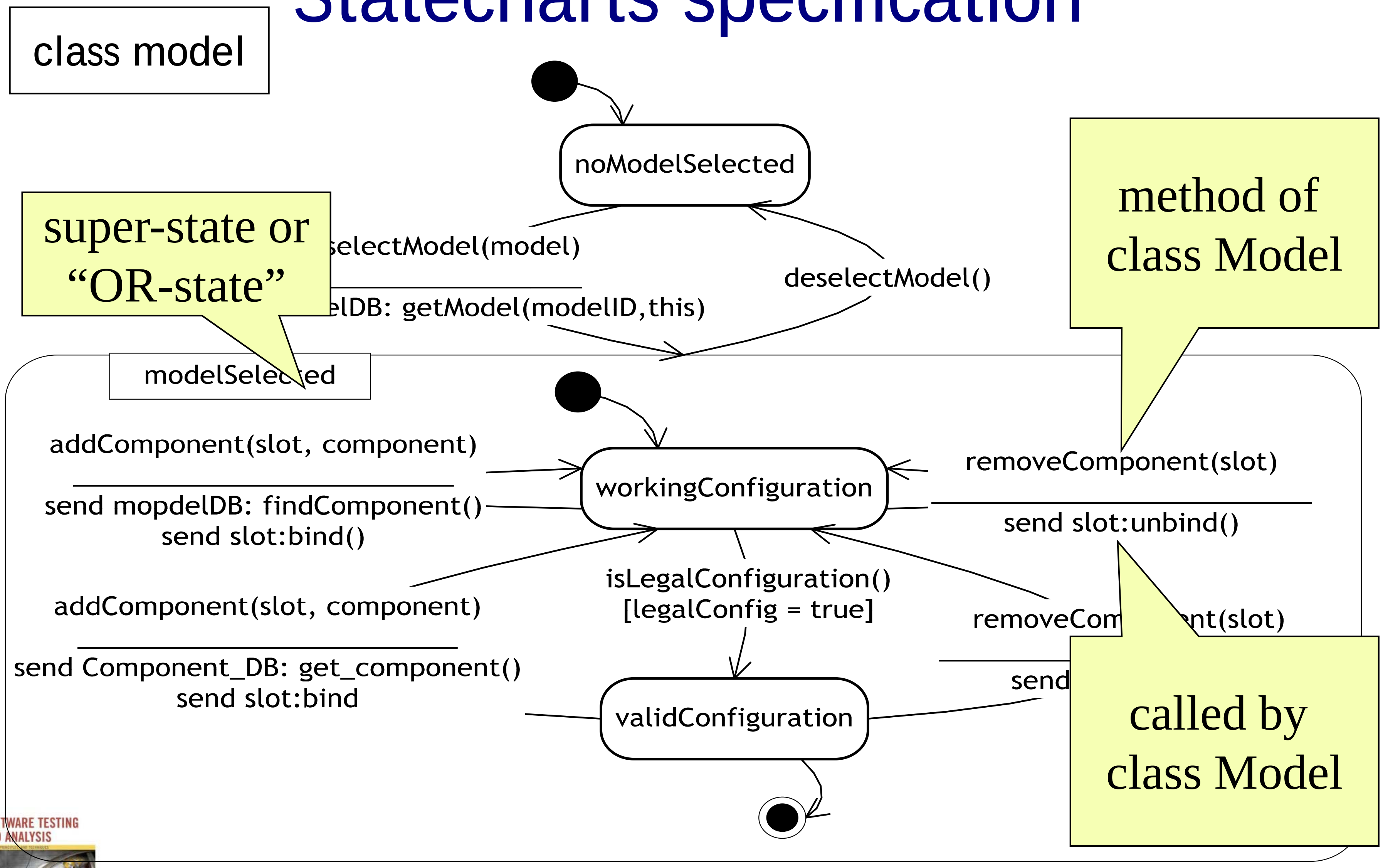


Testing with State Diagrams

- 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



class model

noModelSelected

selectModel(model)

elDB: getModel(modelID,this)

deselectModel()

modelSelected

workingConfiguration

addComponent(slot, component)

send mopdelDB: findComponent()
send slot:bind()

removeComponent(slot)

send slot:unbind()

isLegalConfiguration()
[legalConfig = true]

addComponent(slot, component)

send Component_DB: get_component()
send slot:bind

removeCom nt(slot)

send

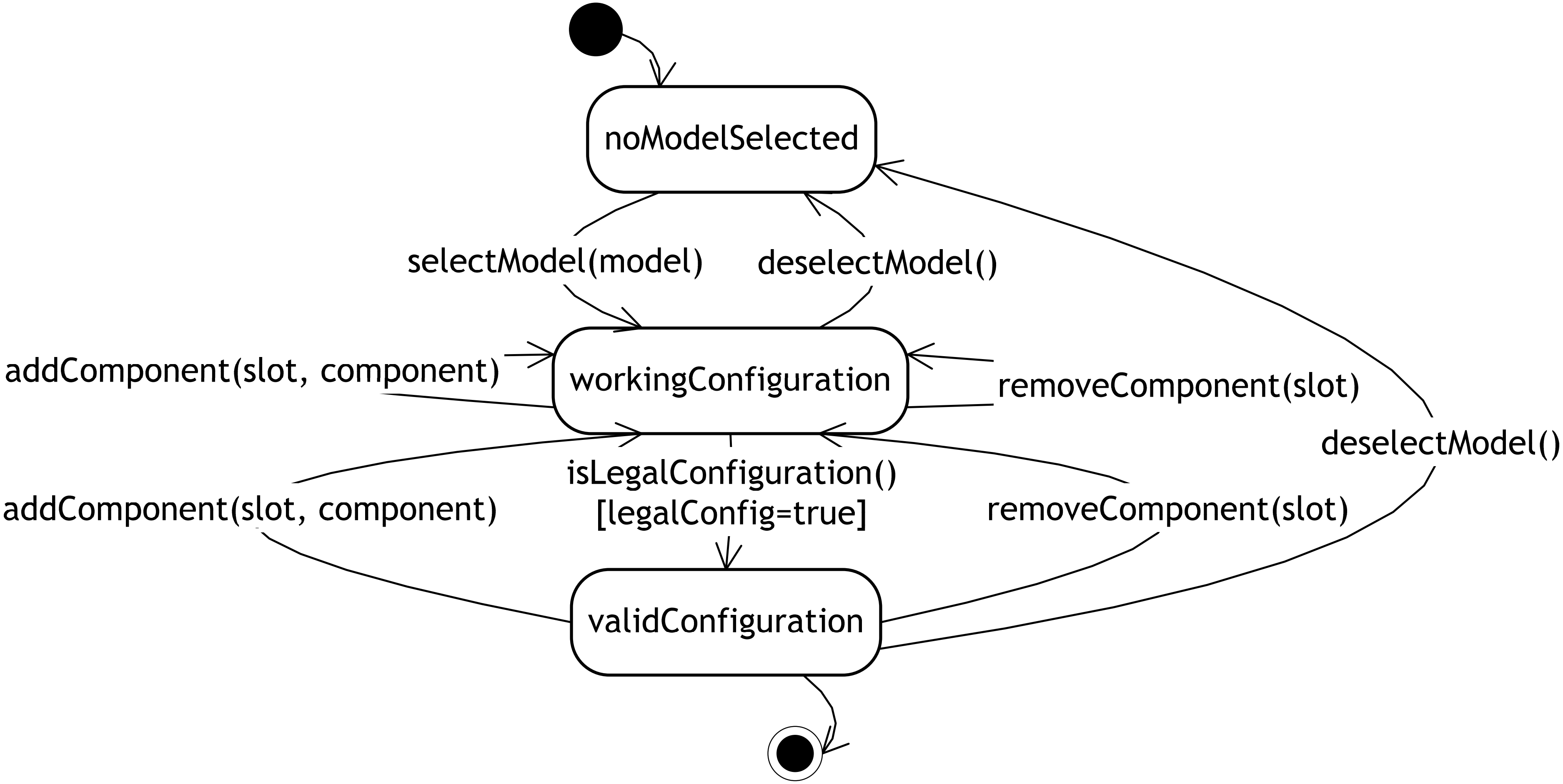
validConfiguration

super-state or "OR-state"

method of class Model

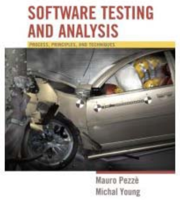
called by class Model

From Statecharts to FSMs



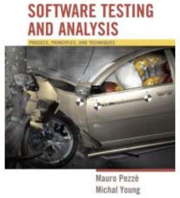
Statechart based criteria

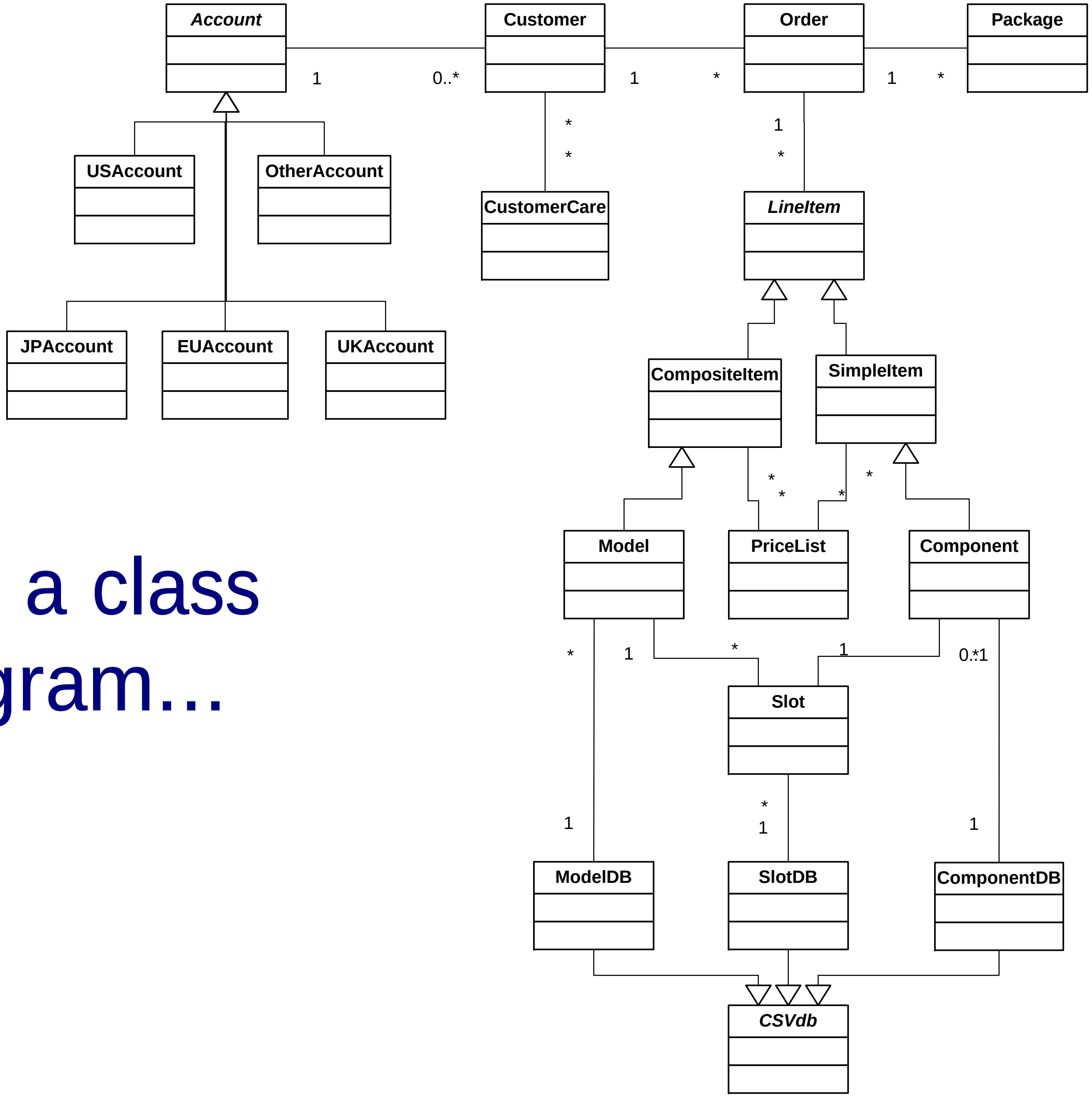
- In some cases, "flattening" a Statechart to a finite-state machine may cause "state explosion"
    - Particularly for super-states with "history"
- Alternative: Use the statechart directly
- Simple transition coverage:
  execute all transitions of the original Statechart
    - incomplete transition coverage of corresponding FSM
    - useful for complex statecharts and strong time constraints (combinatorial number of transitions)



Interclass Testing

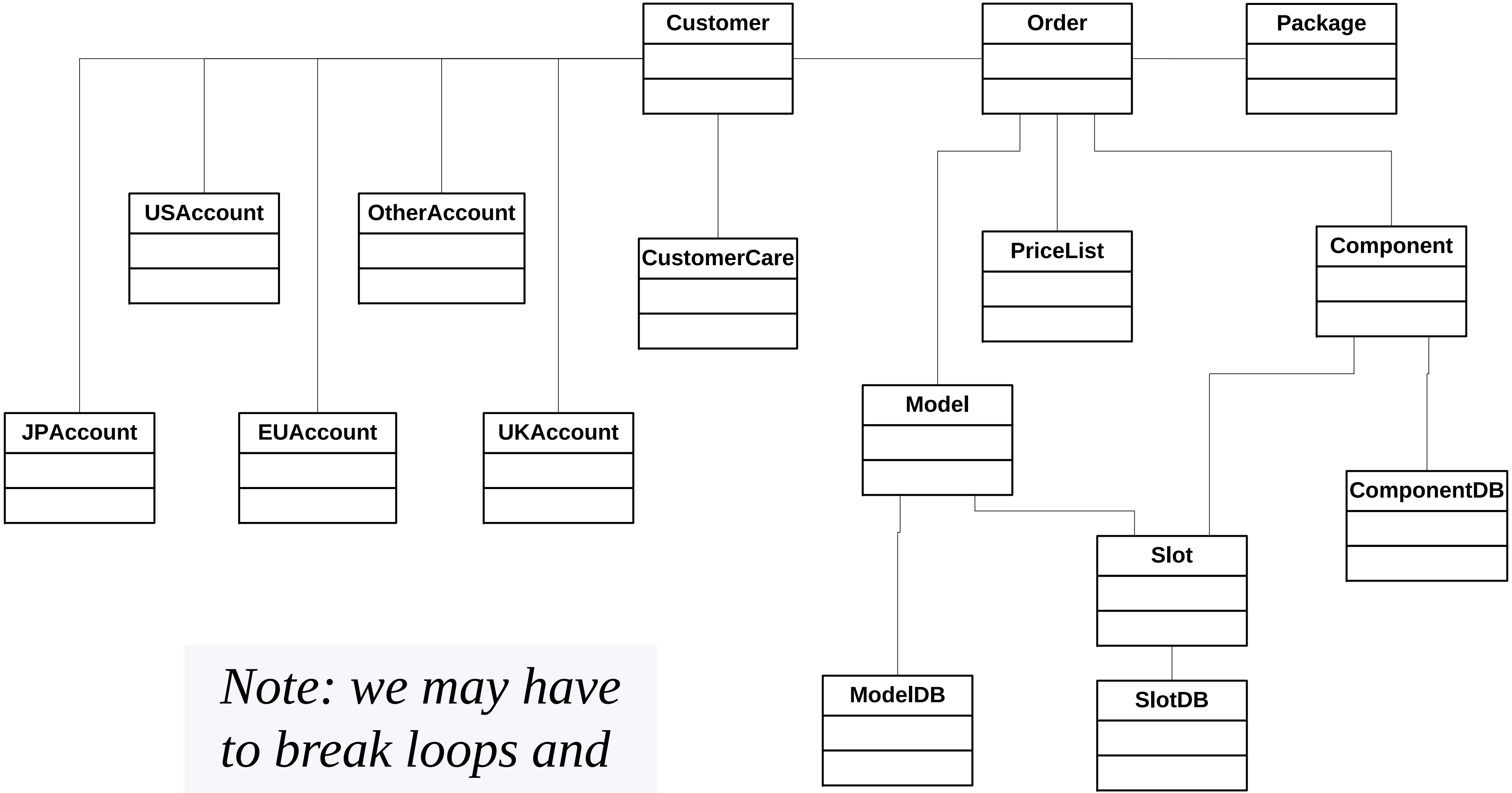
- The first level of *integration testing* for object-oriented software
  - Focus on interactions between classes
- Bottom-up integration according to “depends” relation
  - A depends on B: Build and test B, then A
- Start from use/include hierarchy
      - Implementation-level parallel to logical “depends” 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”





from a class diagram...

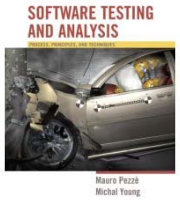
....to a hierarchy



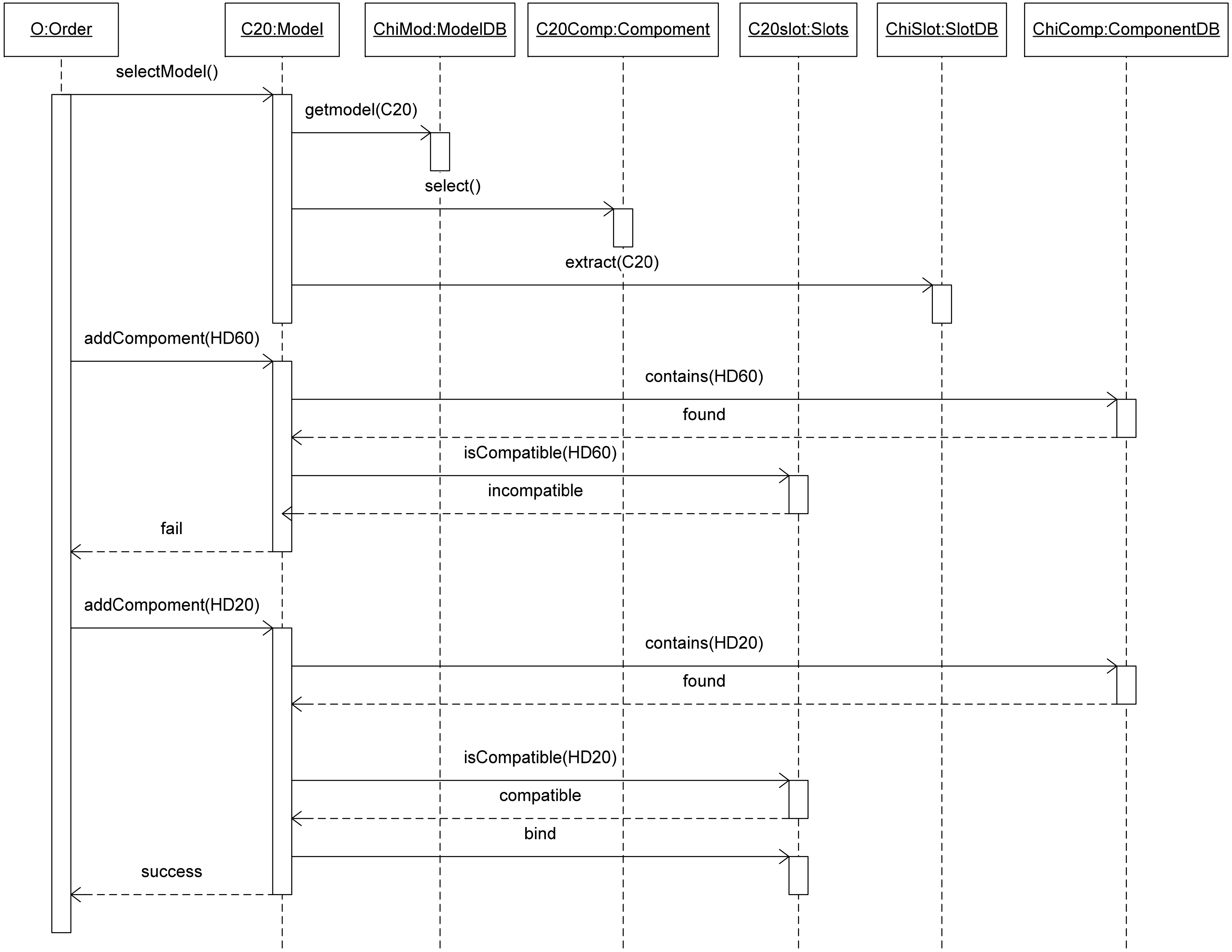
Note: we may have to break loops and generate stubs

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

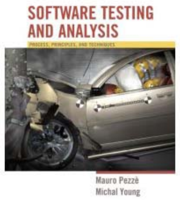


sequence diagram



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

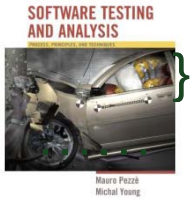


From the implementation ...

```
public class Model  extends Orders.CompositeItem {
....
   private boolean legalConfig = false;  // memoized
....
   public boolean isLegalConfiguration() {
    if (! legalConfig) {
       checkConfiguration();
    }
    return legalConfig;
   }
.....
   private void checkConfiguration() {
    legalConfig = true;
    for (int i=0; i < slots.length; ++i) {
       Slot slot = slots[i];
       if (slot.required && ! slot.isBound()) {
          legalConfig = false;
       } ...}  ...  }
```

private instance variable

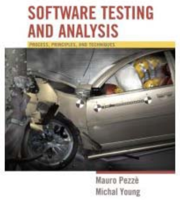
private method



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

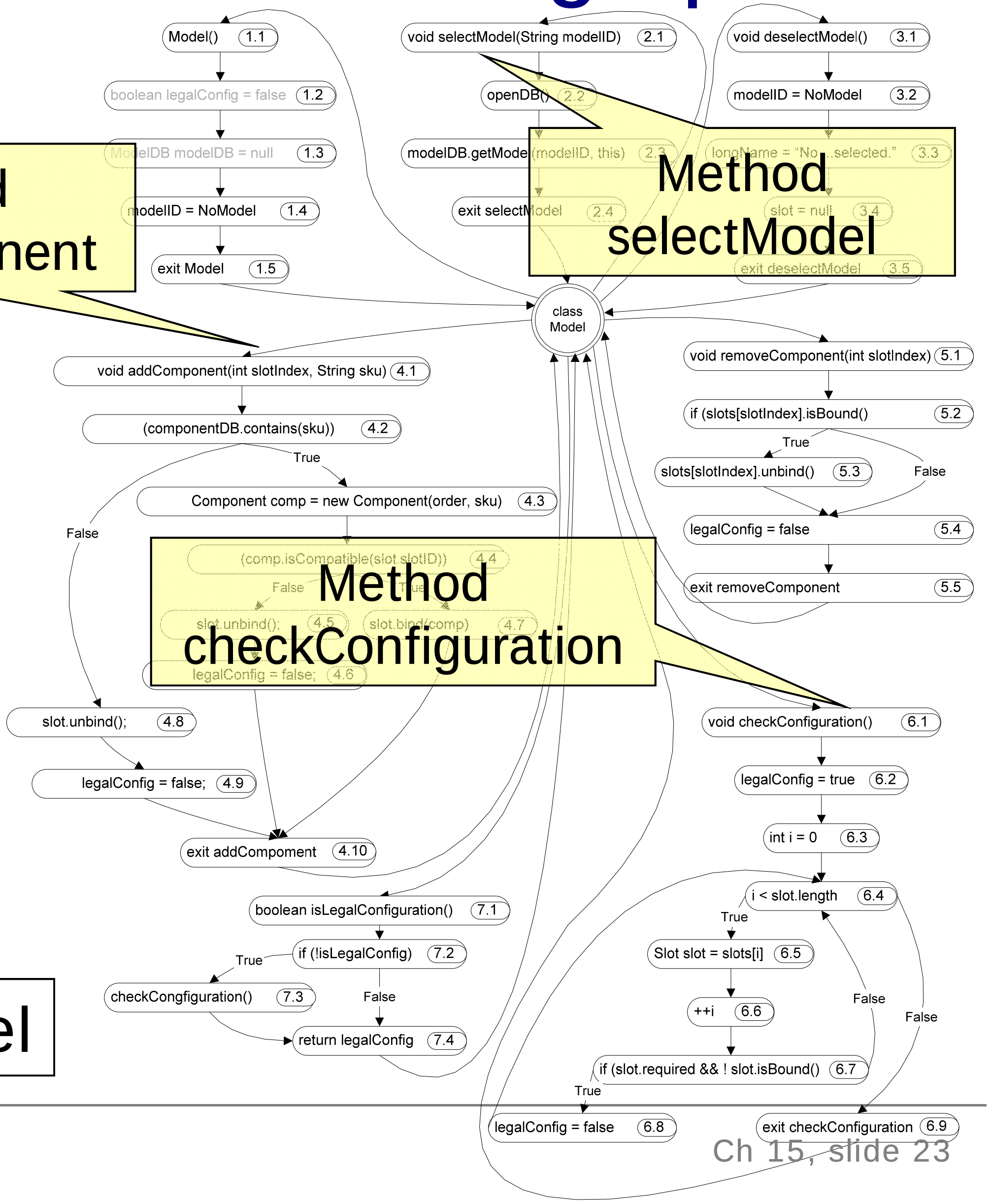
+

edges

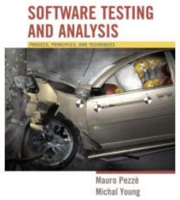
from node *class* to the start nodes of the methods

from the end nodes of the methods to node *class*

=> control flow through *sequences* of method calls



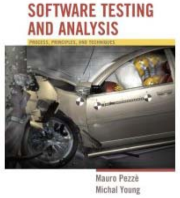
class Model



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)

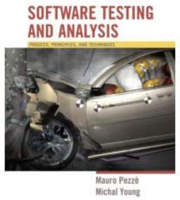


Inspectors and modifiers

- 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*



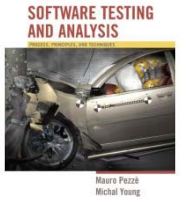
Definition-Use (DU) pairs

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

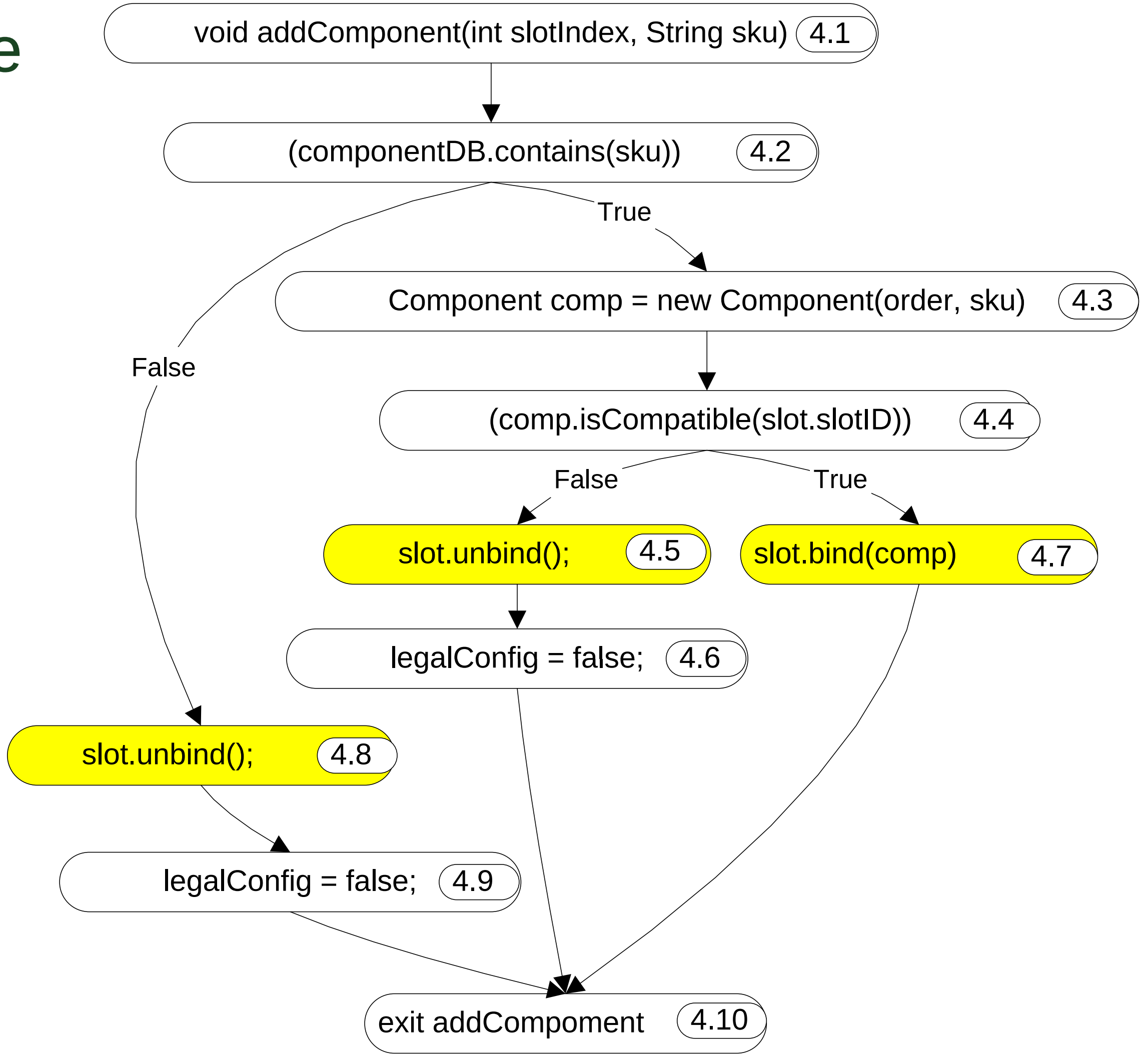


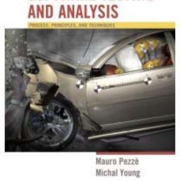
Definitions from modifiers

Definitions of instance variable *slot* in class *model*

addComponent (4.5)
addComponent (4.7)
addComponent (4.8)
selectModel (2.3)
removeComponent (5.3)

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



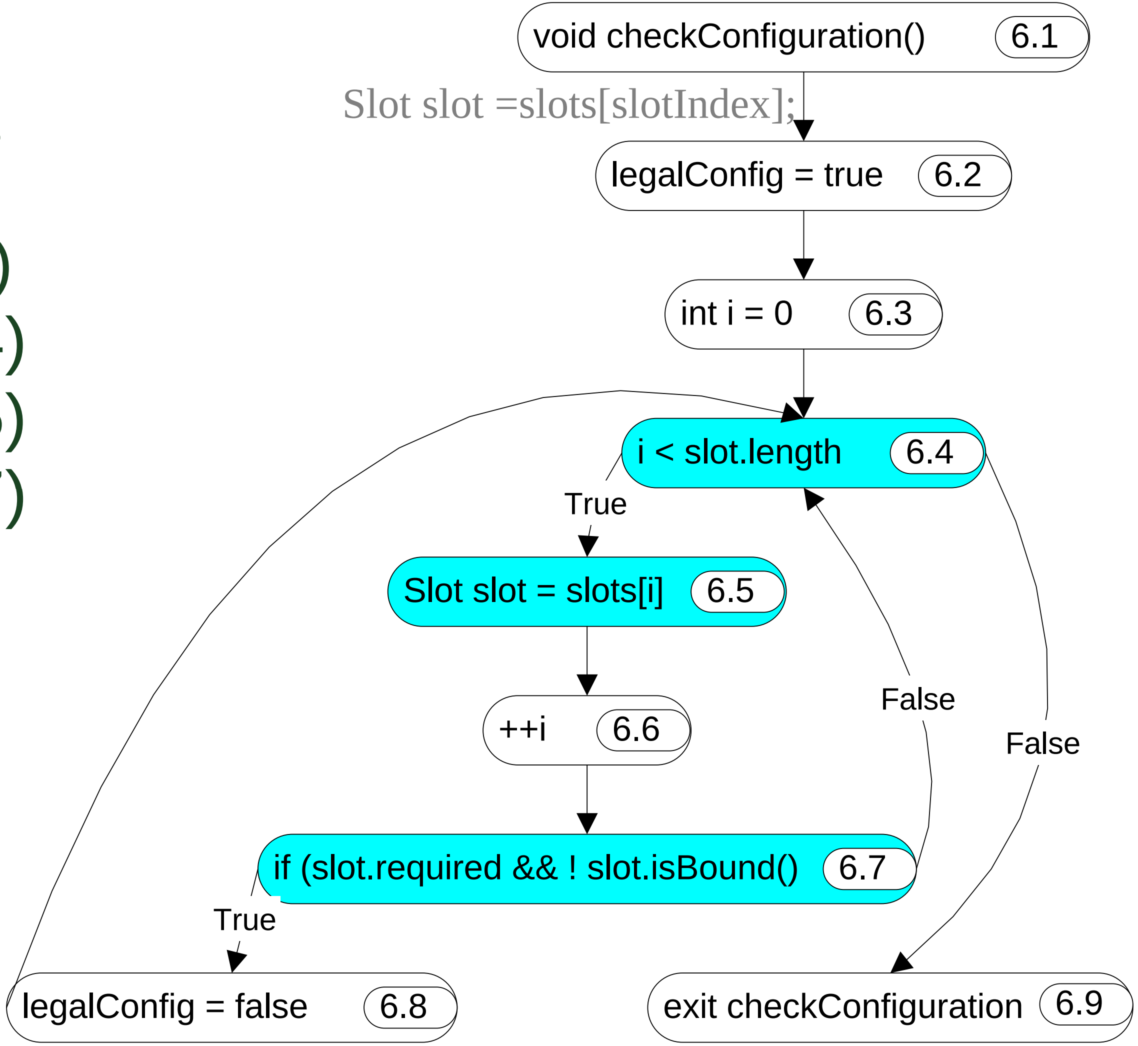


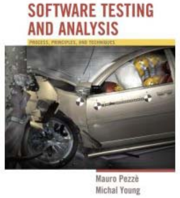
Uses from inspectors

Uses of instance variables *slot* in class *model*

removeComponent (5.2)
checkConfiguration (6.4)
checkConfiguration (6.5)
checkConfiguration (6.7)

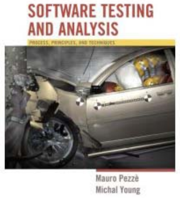
Slot()	modifier
bind()	modifier
unbind()	modifier
isbound()	inspector



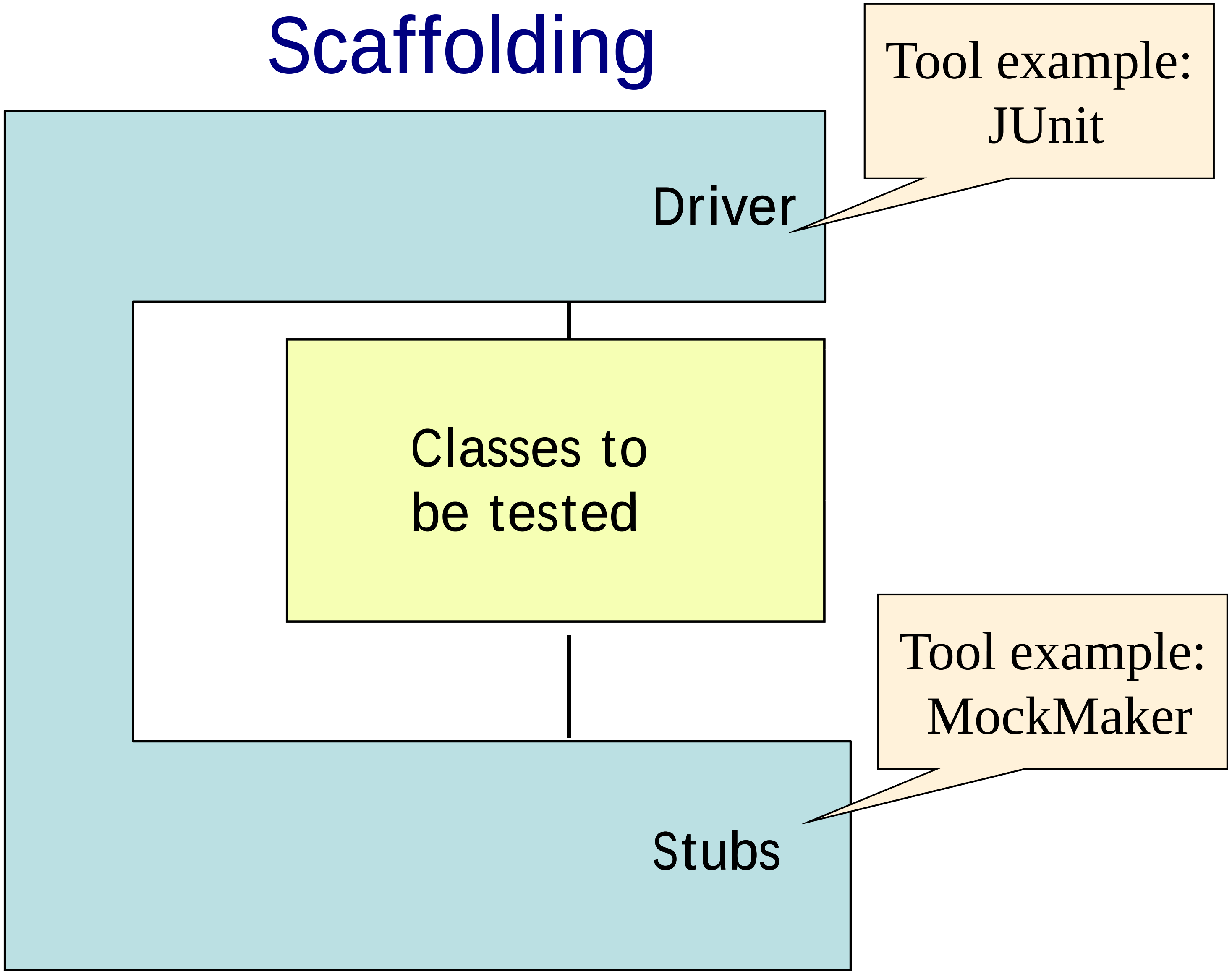


Stubs, Drivers, and Oracles for Classes

- 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



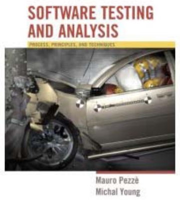
Tool example: JUnit

Driver

Classes to be tested

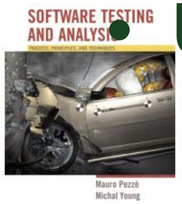
Tool example: MockMaker

Stubs



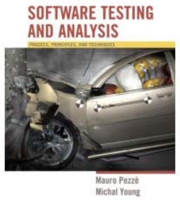
Approaches

- 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



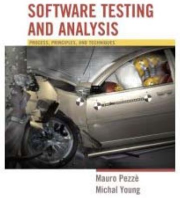
Oracles

- 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*
- 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

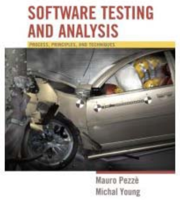


Equivalent Scenarios Approach

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)

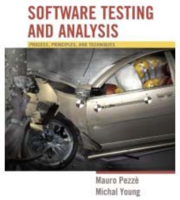
isLegalConfiguration()

deselectModel()

selectModel(M2)

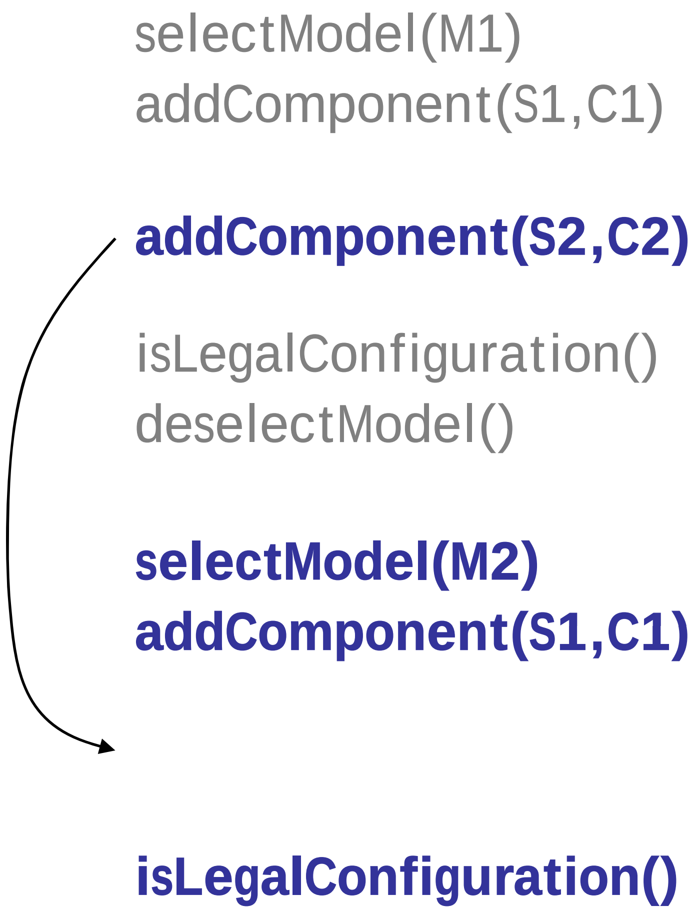
addComponent(S1,C1)

isLegalConfiguration()



Generating non-equivalent scenarios

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



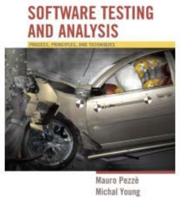
selectModel(M1)
addComponent(S1,C1)

addComponent(S2,C2)

isLegalConfiguration()
deselectModel()

selectModel(M2)
addComponent(S1,C1)

isLegalConfiguration()

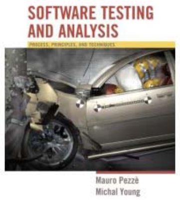


Verify equivalence

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

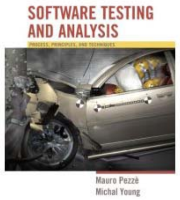
Practically:

- 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

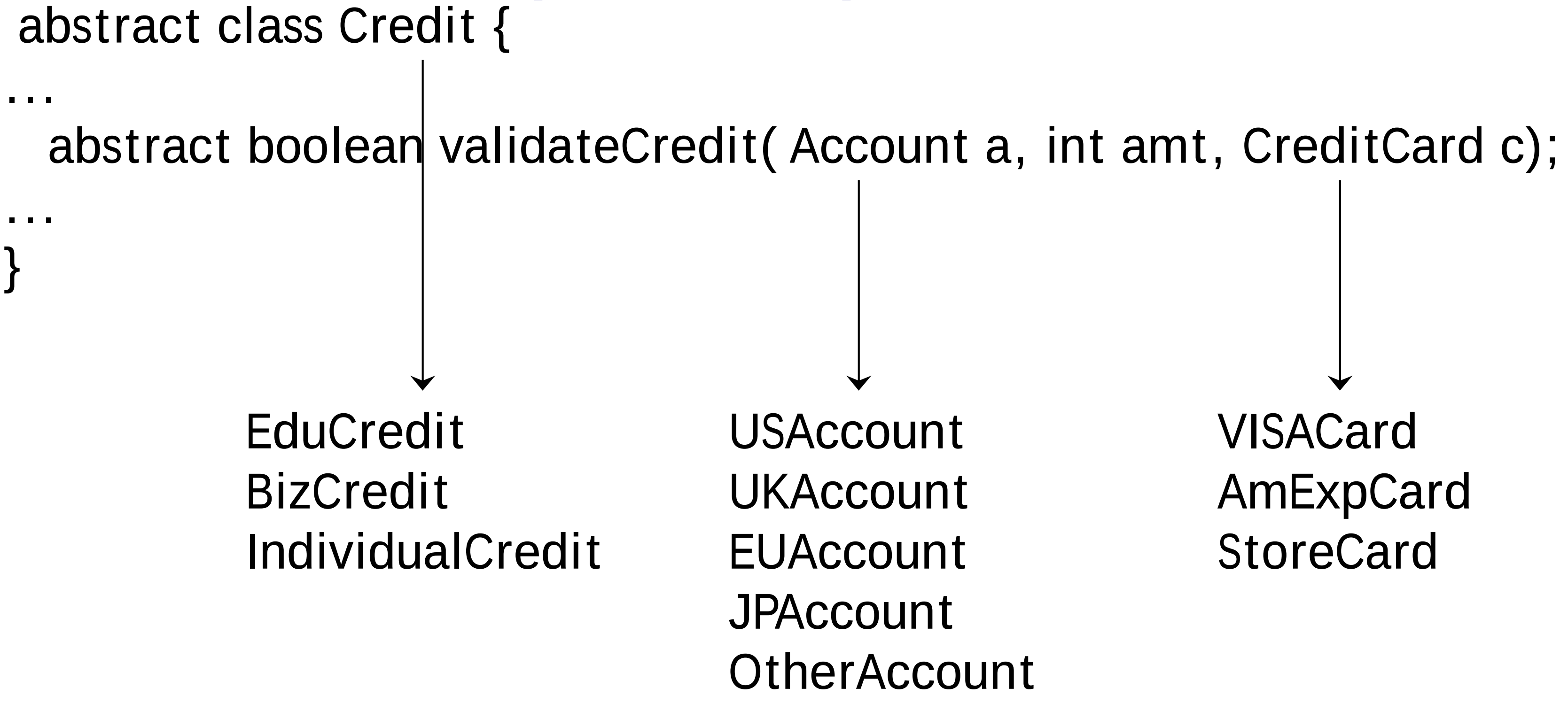


Polymorphism and dynamic binding

One variable potentially bound to methods of different (sub-)classes



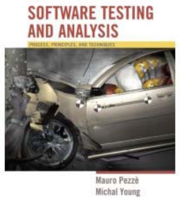
“Isolated” calls: the combinatorial explosion problem



```
 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

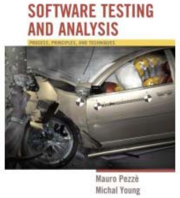
Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit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

Same motivation as pairwise specification-based testing

Combined calls: undesired effects

```
public abstract class Account { …
    public int getYTDPurchased() {
     if (ytdPurchasedValid) { return ytdPurchased; }
     int totalPurchased = 0;
     for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements(); )
        {   Account subsidiary = (Account) e.nextElement();
            totalPurchased += subsidiary.getYTDPurchased();
        }
     for (Enumeration e = customers.elements(); e.hasMoreElements(); )
        {   Customer aCust = (Customer) e.nextElement();
            totalPurchased += aCust.getYearlyPurchase();
        }
     ytdPurchased = totalPurchased;
     ytdPurchasedValid = true;
     return totalPurchased;
    }  …  }
```

Problem:
different implementations of methods getYDTPurchased refer to different currencies.



A data flow approach

```
public abstract class Account {
...
   public int getYTDPurchased() {
                    if (ytdPurchasedValid) {                rchased; }
                    int totalPurchased =
                    for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements
                         {
                                   Account subsidiary = (Account) e.nextElement();
                                   totalPurchased += subsidiary.getYTDPurchased();
                         }
                    for (Enumeration e = customers.elements(); e.hasMoreElement
                         {
                                   Customer aCust = (Customer) e.nextElement();
                                   totalPurchased += aCust.getYearlyPurchase();
                         }
                    ytdPurchased = totalPurchased;
                    ytdPurchasedValid = true;
                    return totalPurchased;
   }
…
}
```

step 1: identify polymorphic calls, binding sets, defs and uses

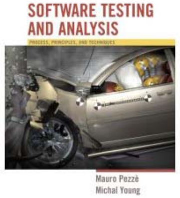
totalPurchased defined

totalPurchased used and defined

totalPurchased used and defined

totalPurchased used

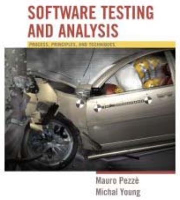
totalPurchased used



Def-Use (dataflow) testing of polymorphic calls

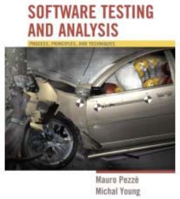
- Derive a test case for each possible polymorphic <def,use> pair
  - Each binding must be considered individually
  - Pairwise combinatorial selection may help in reducing the set of test cases

- *Example*: Dynamic binding of currency
  - We need test cases that bind the different calls to different methods *in the same run*
  - We can reveal faults due to the use of different currencies in different methods



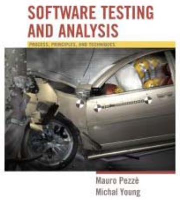
Inheritance

- When testing a subclass …
  - We would like to re-test only what has not been thoroughly tested in the parent class
    - for example, no need to test hashCode and getClass methods inherited from class Object in Java
  - But we should test any method whose behavior may have changed
    - even accidentally!

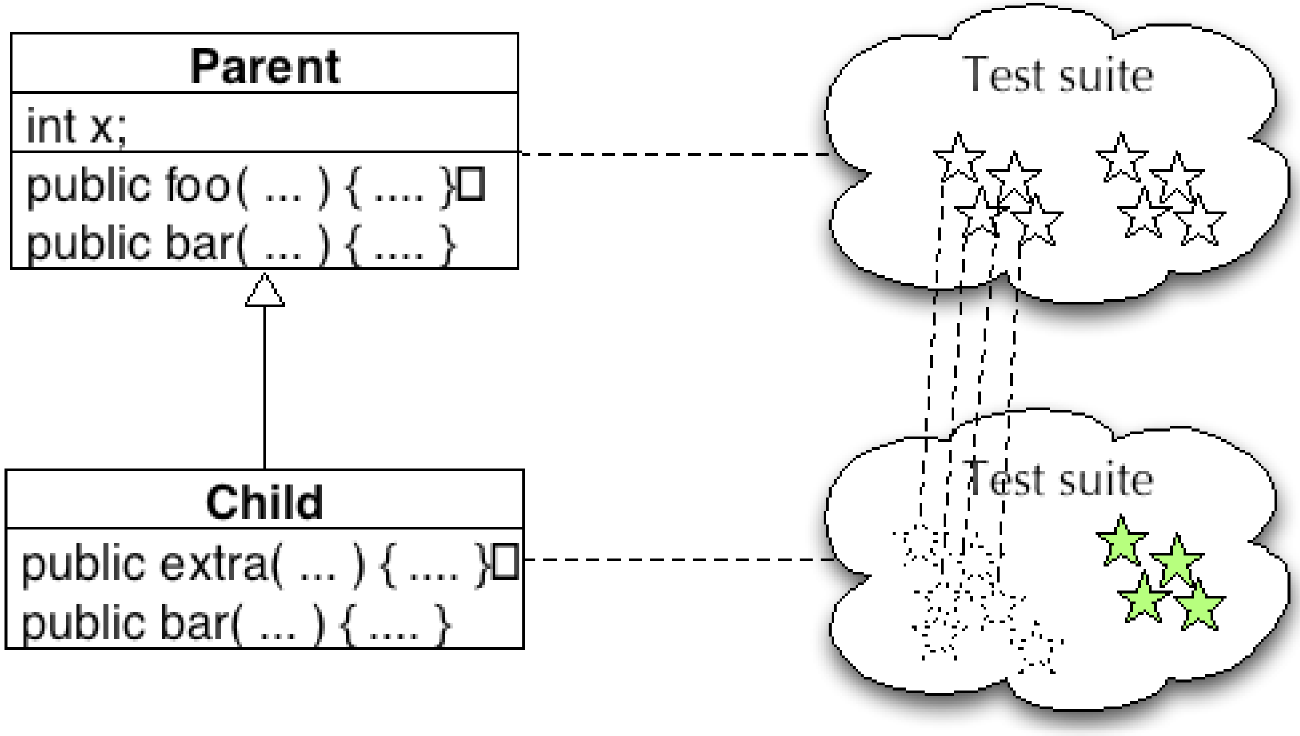


Reusing Tests with the Testing History Approach

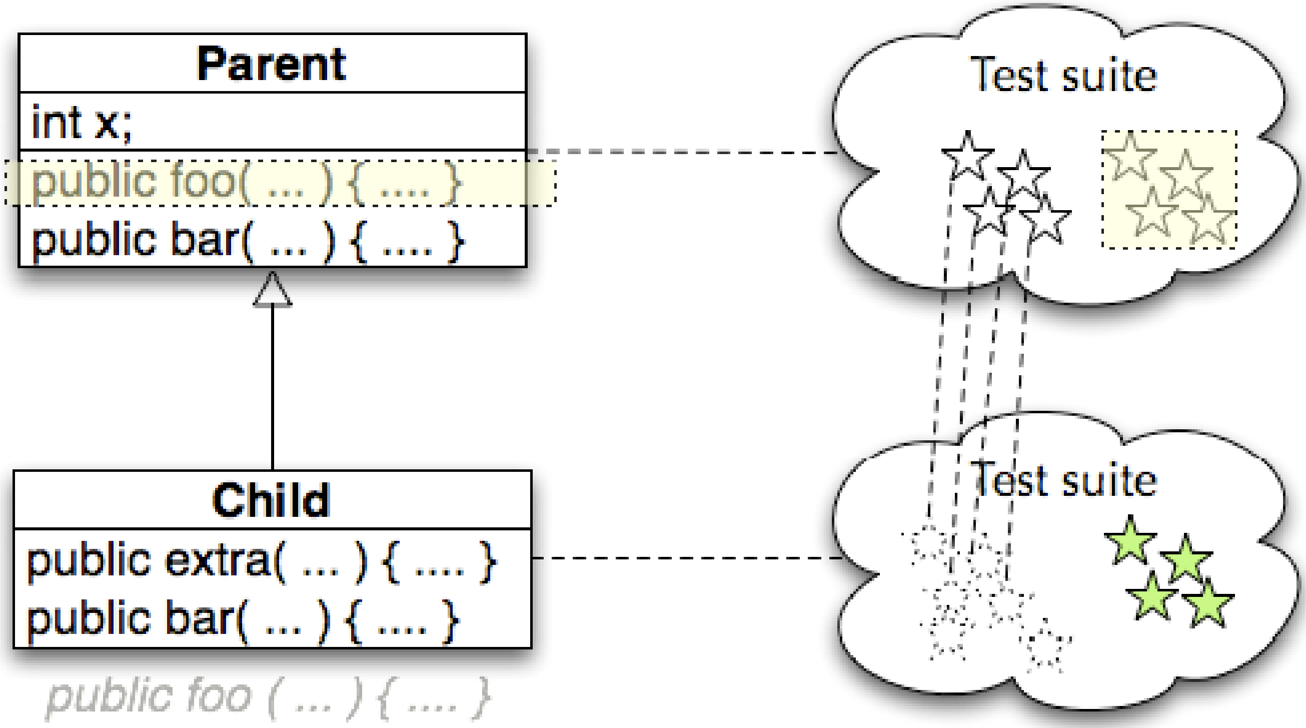
- Track test suites and test executions
  - determine which new tests are needed
  - determine which old tests must be re-executed
- New and changed behavior …
  - new methods must be tested
  - redefined methods must be tested, but we can partially reuse test suites defined for the ancestor
  - other inherited methods do not have to be retested



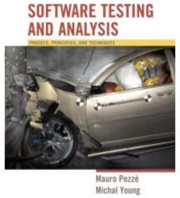
Testing history



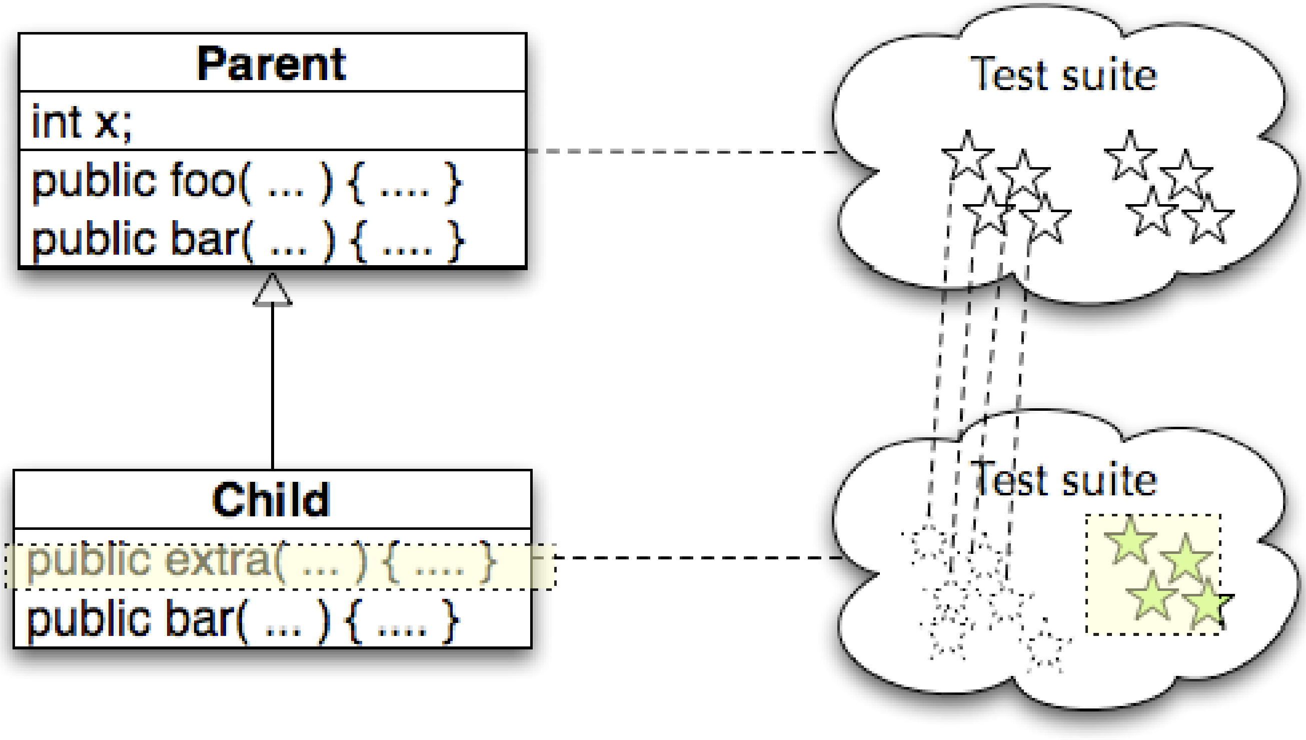
Inherited, unchanged



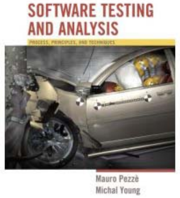
Inherited, unchanged ("recursive"):
No need to re-test



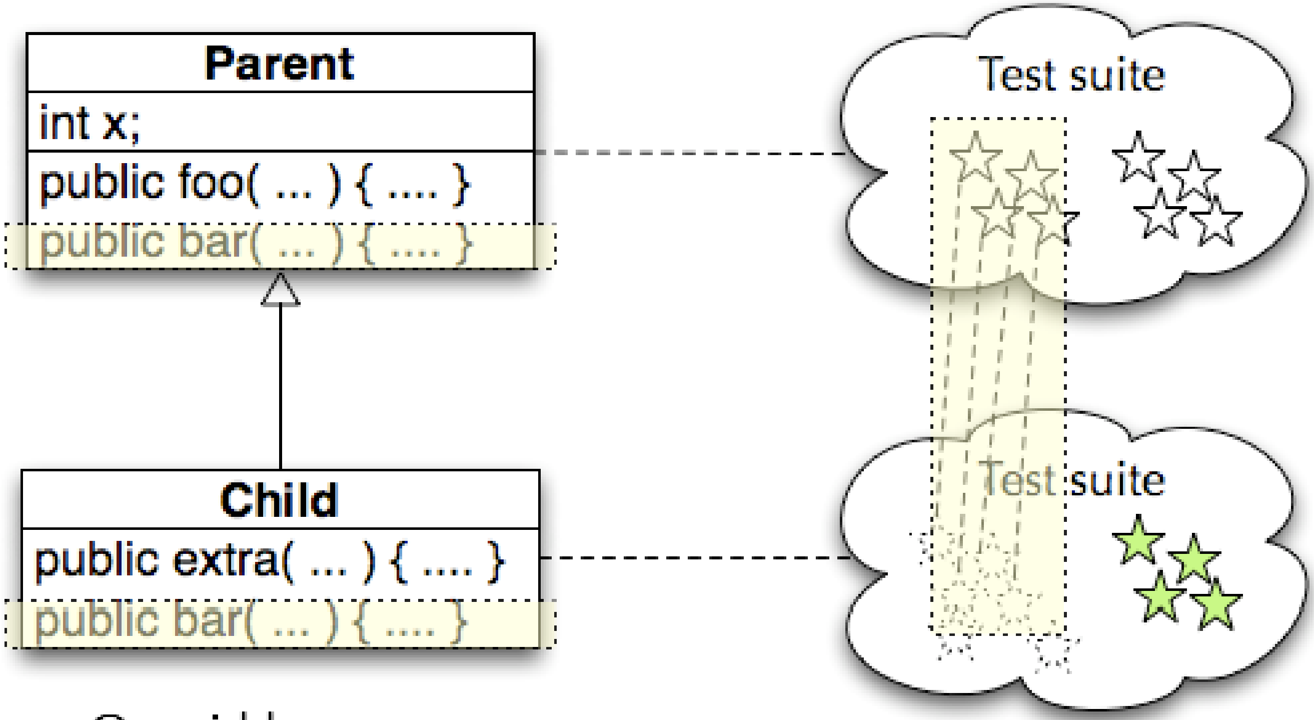
Newly introduced methods



New:
Design and execute new test cases

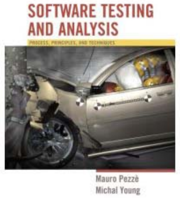


Overridden methods



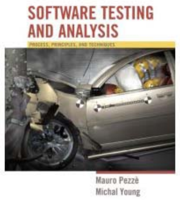
Overridden:

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

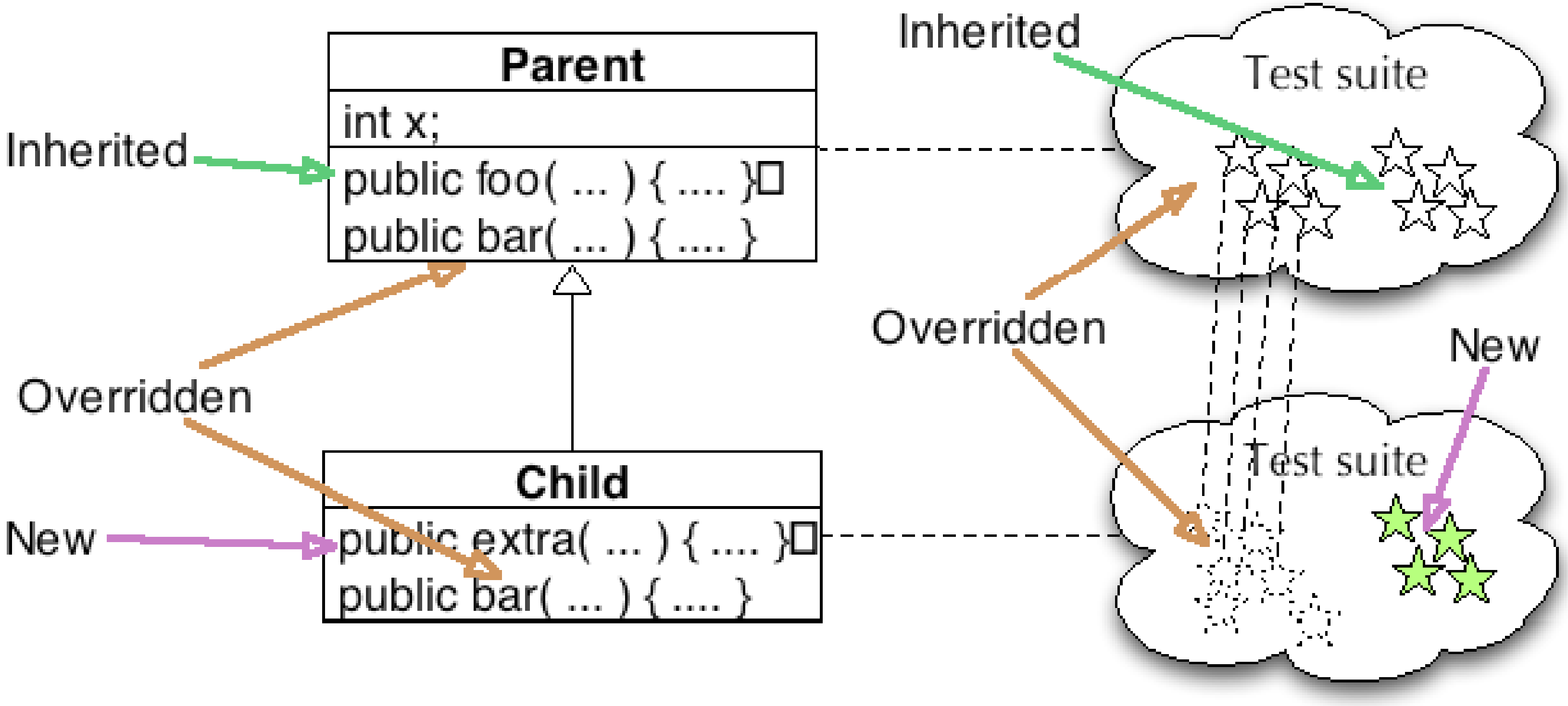


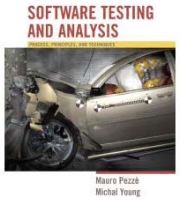
Testing History – some details

- Abstract methods (and classes)
  - Design test cases when abstract method is introduced (even if it can't be executed yet)
- Behavior changes
  - Should we consider a method "redefined" if another new or redefined method changes its behavior?
    - The standard "testing history" approach does not do this
    - It might be reasonable combination of data flow (structural) OO testing with the (functional) testing history approach



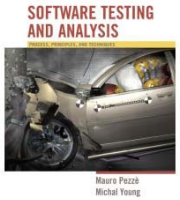
Testing History - Summary





Does testing history help?

- Executing test cases should (usually) be cheap
  - It may be simpler to re-execute the full test suite of the parent class
  - … but still add to it for the same reasons
- But sometimes execution is not cheap …
  - Example: Control of physical devices
  - Or very large test suites
    - Ex: Some Microsoft product test suites require more than one night (so daily build cannot be fully tested)
  - Then some use of testing history is profitable



Testing generic classes

a generic class

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

is designed to be instantiated with many different parameter types

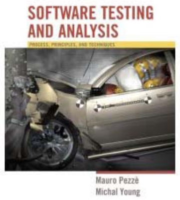
`PriorityQueue<Customers>`

`PriorityQueue<Tasks>`

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

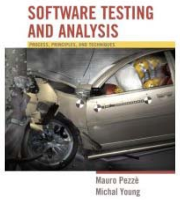
Testing can be broken into two parts

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



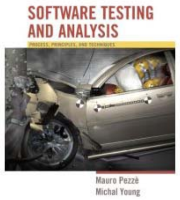
Show that some instantiation is correct

- Design tests as if the parameter were copied textually into the body of the generic class.
  - We need source code for both the generic class and the parameter class



Identify (possible) interactions

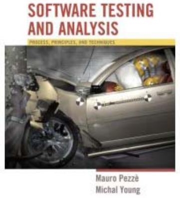
- Identify potential interactions between generic and its parameters
  - Identify potential interactions by inspection or analysis, not testing
  - Look for: method calls on parameter object, access to parameter fields, possible indirect dependence
  - Easy case is no interactions at all (e.g., a simple container class)
- Where interactions are possible, they will need to be tested



Example interaction

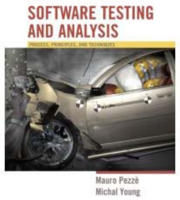
```
class PriorityQueue
  <Elem implements Comparable> {...}
```

- Priority queue uses the “Comparable” interface of Elem to make method calls on the generic parameter
- We need to establish that it does so consistently
  - So that if priority queue works for one kind of Comparable element, we can have some confidence it does so for others



Testing variation in instantiation

- We can’t test every possible instantiation
  - Just as we can’t test every possible program input
- … but there is a contract (a specification) between the generic class and its parameters
  - Example: “implements Comparable” is a specification of possible instantiations
  - Other contracts may be written only as comments
- Functional (specification-based) testing techniques are appropriate
  - Identify and then systematically test properties implied by the specification



Example: Testing instantiation variation

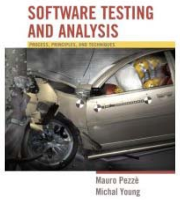
Most but not all classes that implement Comparable also satisfy the rule

`(x.compareTo(y) == 0) == (x.equals(y))`

(from java.lang.Comparable)

So test cases for PriorityQueue should include

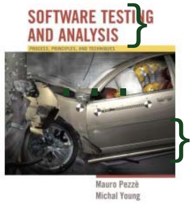
- instantiations with classes that do obey this rule:
  `class String`
- instantiations that violate the rule:
  `class BigDecimal` with values `4.0` and `4.00`



Exception handling

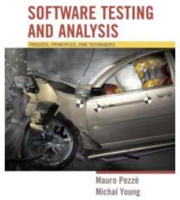
```
void addCustomer(Customer theCust) {
  customers.add(theCust);
    }
    public static Account
  newAccount(...)
  throws InvalidRegionException
    {
  Account thisAccount = null;
  String regionAbbrev = Regions.regionOfCountry(
                    mailAddress.getCountry());
  if (regionAbbrev == Regions.US) {
      thisAccount = new USAccount();
  } else if (regionAbbrev == Regions.UK) {
      ....
  } else if (regionAbbrev == Regions.Invalid) {
      throw new
  InvalidRegionException(mailAddress.getCountry());
  }
    }
```

exceptions create implicit control flows and may be handled by different handlers



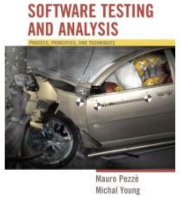
Testing exception handling

- Impractical to treat exceptions like normal flow
  - too many flows: every array subscript reference, every memory allocation, every cast, …
  - multiplied by matching them to every handler that could appear immediately above them on the call stack.
  - many actually impossible
- So we separate testing exceptions
  - and ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: Each exception handler, and each explicit throw or re-throw of an exception



Testing program exception handlers

- Local exception handlers
  - test the exception handler (consider a subset of points bound to the handler)
- Non-local exception handlers
  - Difficult to determine all pairings of <points, handlers>
  - So enforce (and test for) a design rule:
    if a method propagates an exception, the method call should have *no other effect*



Summary

- 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