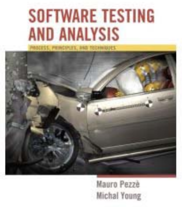
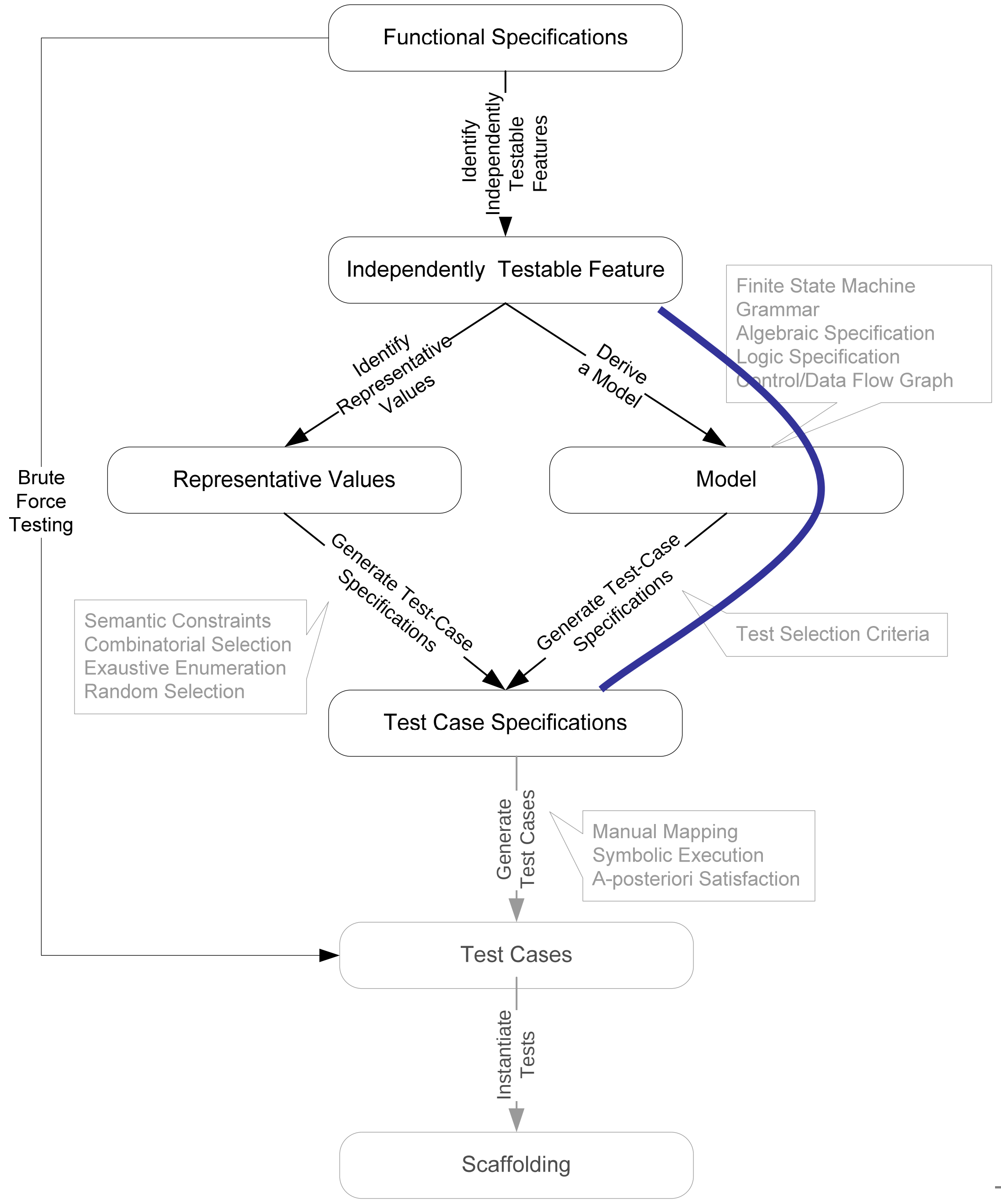
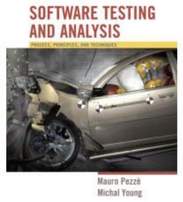
Model based testing

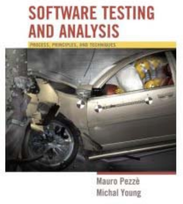



Functional Specifications
Identify Independently Testable Features
Independently Testable Feature
Identify Representative Values
Derive a Model
Finite State Machine
Grammar
Algebraic Specification
Logic Specification
Control/Data Flow Graph
Brute Force Testing
Representative Values
Model
Generate Test-Case Specifications
Generate Test-Case Specifications
Semantic Constraints
Combinatorial Selection
Exaustive Enumeration
Random Selection
Test Selection Criteria
Test Case Specifications
Generate Test Cases
Manual Mapping
Symbolic Execution
A-posteriori Satisfaction
Test Cases
Instantiate Tests
Scaffolding



Why model-based testing?

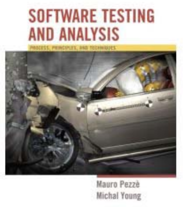
- Models used in specification or design have structure
  - Useful information for selecting representative classes of behavior; behaviors that are treated differently with respect to the model should be tried by a thorough test suite
  - In combinatorial testing, it is difficult to capture that structure clearly and correctly in constraints
- We can devise test cases to check actual behavior against behavior specified by the model
  - “Coverage” similar to structural testing, but applied to specification and design models



Deriving test cases from finite state machines

A common kind of model for describing behavior that depends on sequences of events or stimuli

Example: UML state diagrams



From an informal specification...

Maintenance: The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or mainte
requested either by calling the maintenance to · or
by bringing the item to a designated maintenan

Multiple choices in the first step ...

If the maintenance is requested by phone or we
resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided · lows
the procedure for items not covered by warran

... determine the possibilities for the next step ...

If the product is not covered by warranty or ma · e
requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate.

If the customer does not accept the estimate,

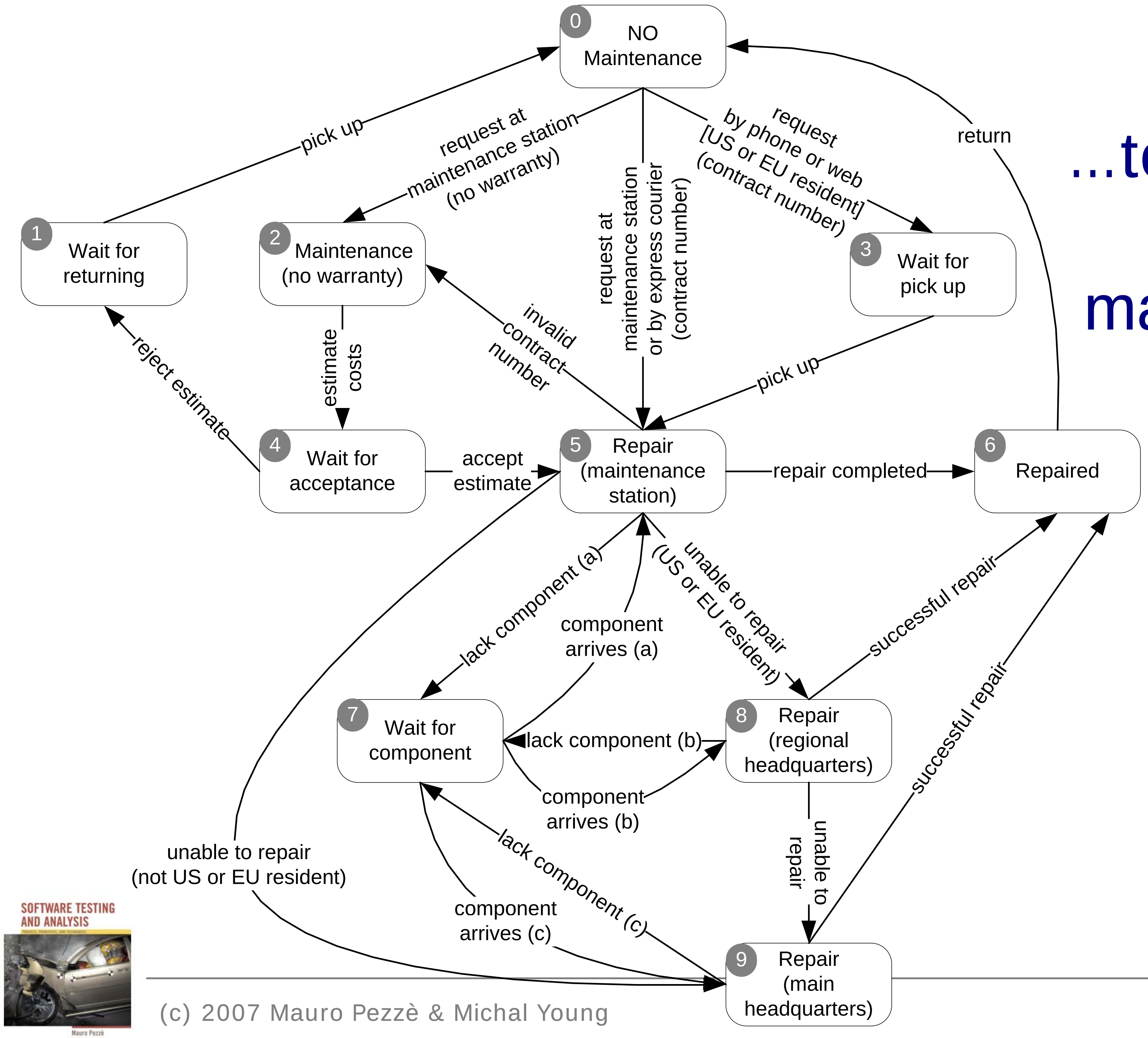
Small problems can be repaired directly at the · e
station cannot solve the problem, the product
headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

... and so on ...

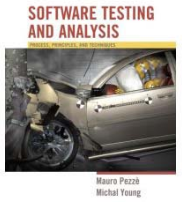
If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.



...to a finite state machine...

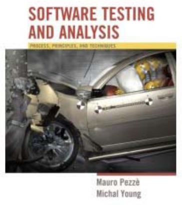


...to a test suite

TC1	0	2	4	1	0						
TC2	0	5	2	4	5	6	0				
TC3	0	3	5	9	6	0					
TC4	0	3	5	7	5	8	7	8	9	6	0

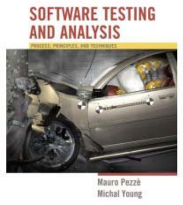
Meaning: From state 0 to state 2 to state 4 to state 1 to state 0

Is this a thorough test suite?
How can we judge?



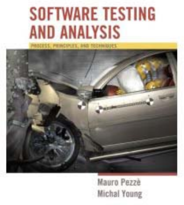
“Covering” finite state machines

- State coverage:
  - Every state in the model should be visited by at least one test case
- Transition coverage
  - Every transition between states should be traversed by at least one test case.
  - *This is the most commonly used criterion*
    - A transition can be thought of as a (precondition, postcondition) pair



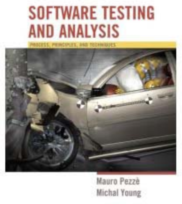
Path sensitive criteria?

- Basic assumption: States fully summarize history
    - No distinction based on how we reached a state; this should be true of well-designed state machine models
- If the assumption is violated, we may distinguish paths and devise criteria to cover them
    - Single state path coverage:
        - traverse each subpath that reaches each state at most once
    - Single transition path coverage:
        - “ ” “ ” each transition at most once
    - Boundary interior loop coverage:
        - each distinct loop of the state machine must be exercised the minimum, an intermediate, and the maximum or a large number of times
        - *Of the path sensitive criteria, only boundary-interior is common*



Testing decision structures

Some specifications are structured as decision tables, decision trees, or flow charts. We can exercise these as if they were program source code.



...to a decision table ...

	edu		individual					
EduAc	T	T	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-
YP > YT1	-	-	-	-	-	-	-	-
CP > CT2	-	-	-	-	F	F	T	T
YP > YT2	-	-	-	-	-	-	-	-
SP < Sc	F	T	F	T	-	-	-	-
SP < T1	-	-	-	-	F	T	-	-
SP < T2	-	-	-	-	-	-	F	T
out	Edu	SP	ND	SP	T1	SP	T2	SP

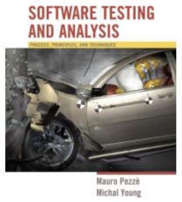
Example MC/DC

	C.1	C.1a	C.1b	C
EduAc	T	F	T	-
BusAc	-	-	-	T
CP > CT1	-	-	-	F
YP > YT1	-	-	-	F
CP > CT2	-	-	-	-
YP > YT2	-	-	-	-
SP > Sc	F	F	T	T
SP > T1	-	-	-	-
SP > T2	-	-	-	-
out	Edu	*	*	SP

Generate C.1a and C.1b by flipping one element of C.1

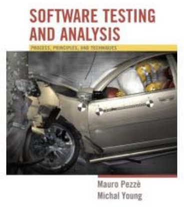
C.1b can be merged with an existing column (C.10) in the spec

Outcome of generated columns must differ from source column



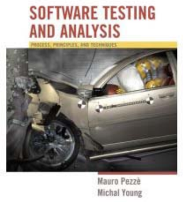
Summary: The big picture

- Models are useful abstractions
  - In specification and design, they help us think and communicate about complex artifacts by emphasizing key features and suppressing details
  - Models convey structure and help us focus on one thing at a time
- We can use them in systematic testing
  - If a model divides behavior into classes, we probably want to exercise each of those classes!
  - Common model-based testing techniques are based on state machines, decision structures, and grammars
    - but we can apply the same approach to other models



Testing Object Oriented Software

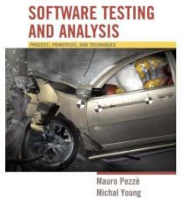
Chapter 15



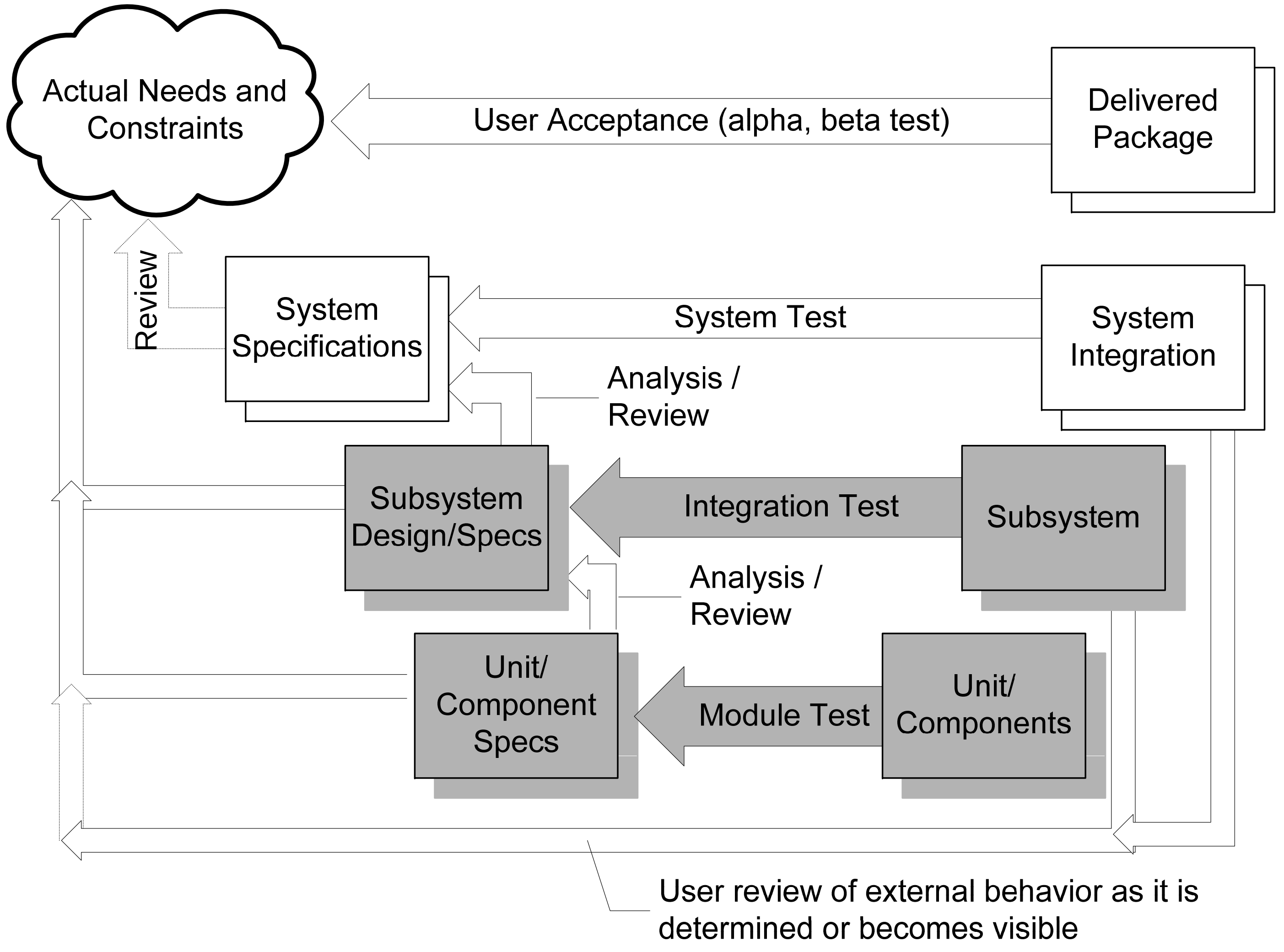
Characteristics of OO Software

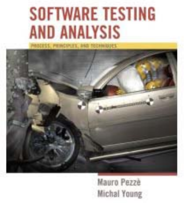
Typical OO software characteristics that impact testing

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



Quality activities and OO SW

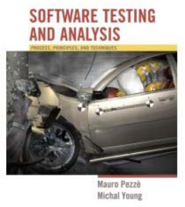




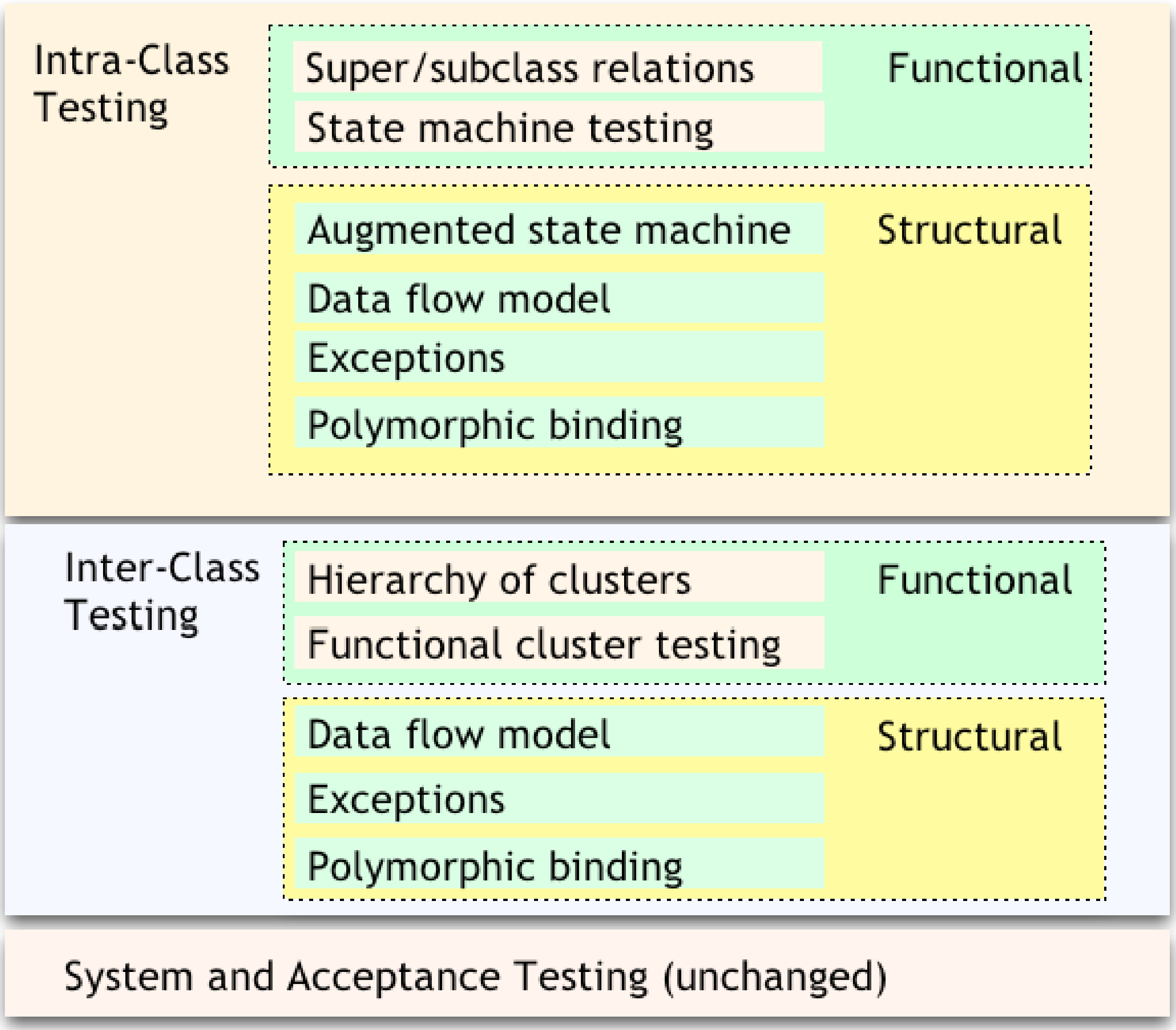
OO definitions of unit and integration testing

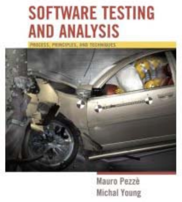
- Procedural software
  - unit = single program, function, or procedure
    more often: a unit of work that may correspond to one or more intertwined functions or programs
- Object oriented software
  - unit = class or (small) cluster of strongly related classes
    (e.g., sets of Java classes that correspond 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

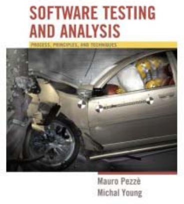




Intraclass State Machine Testing

- 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

[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:

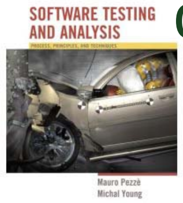
- **Install**: slots can be installed on a model as *required* or *optional*.

  …

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

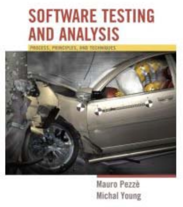
  …

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

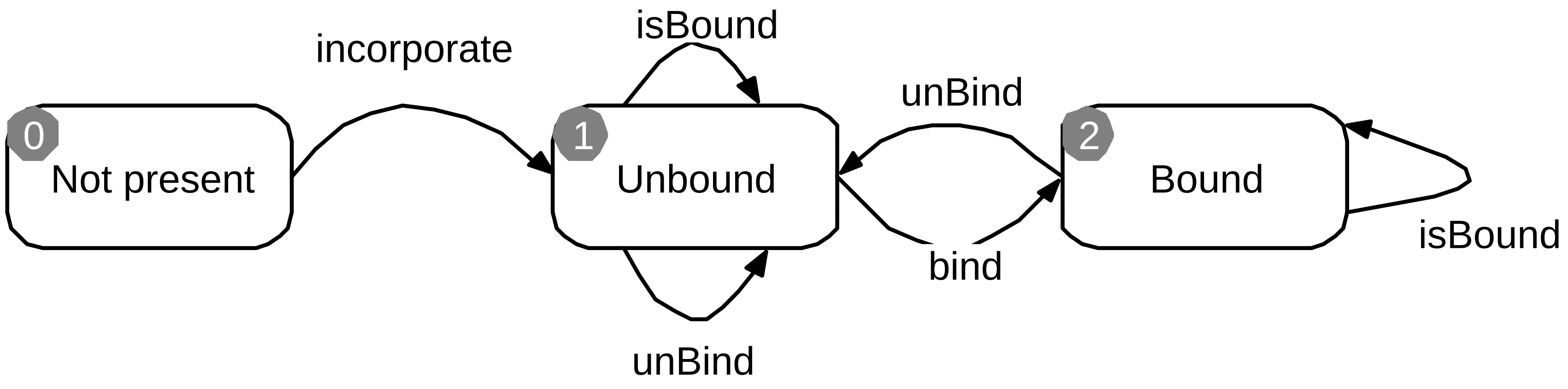


Identifying states and transitions

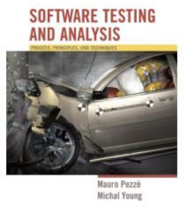
- From the informal specification we can identify three states:
  - Not_installed
  - Unbound
  - Bound
- and four transitions
  - install: from Not_installed to Unbound
  - bind: from Unbound to Bound
  - unbind: …to Unbound
  - isBound: does not change state



Deriving an FSM and test cases

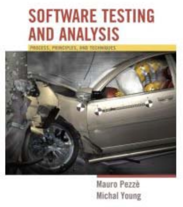


- TC-1: incorporate, isBound, bind, isBound
- TC-2: incorporate, unBind, bind, unBind, isBound



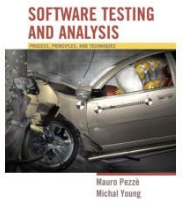
Testing with State Diagrams

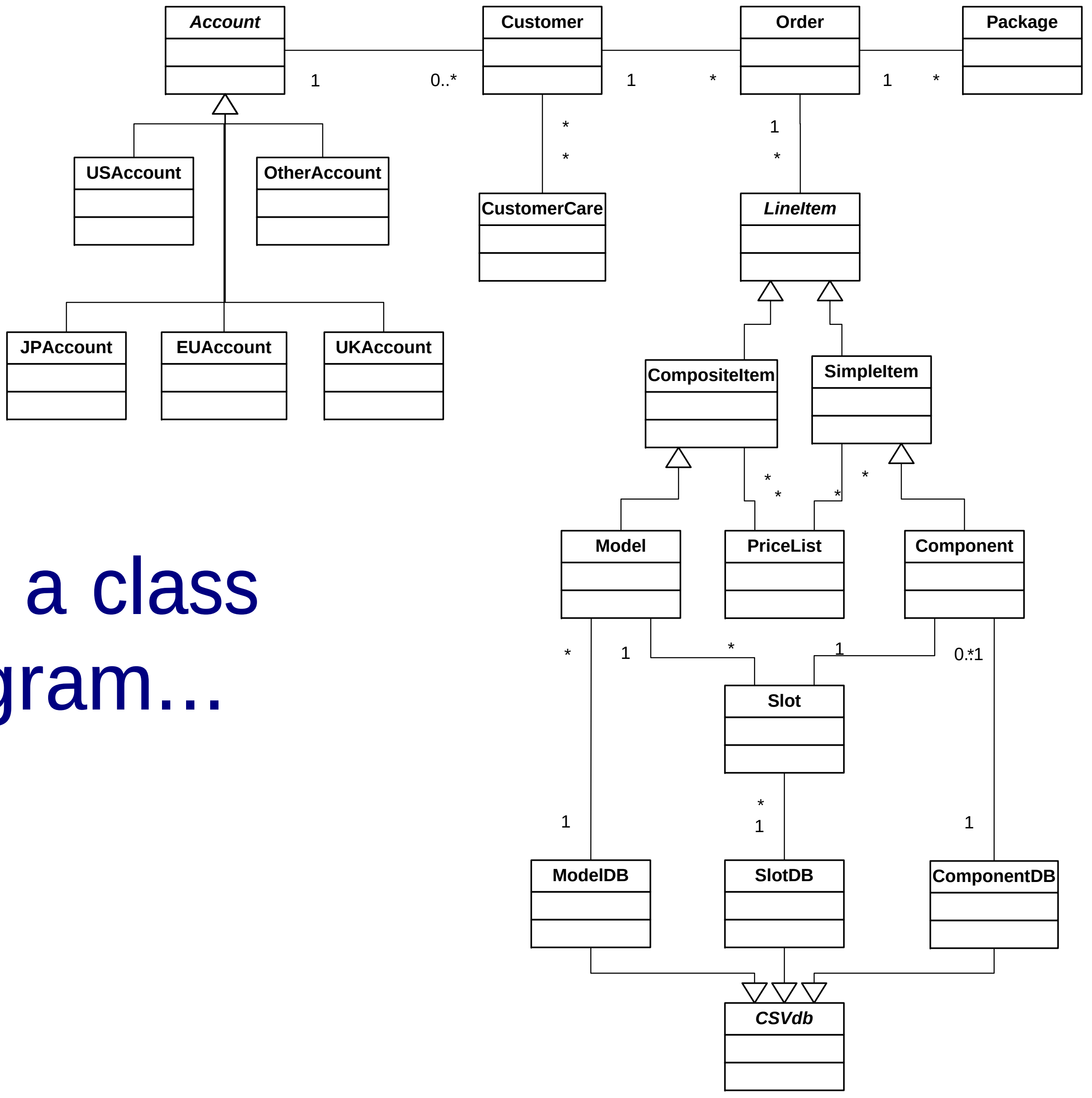
- A statechart (called a “state diagram” in UML) may be produced as part of a specification or design
    - May also be implied by a set of message sequence charts (interaction diagrams), or other modeling formalisms
- Two options:
    - Convert (“flatten”) into standard finite-state machine, then derive test cases
    - Use state diagram model directly



Interclass Testing

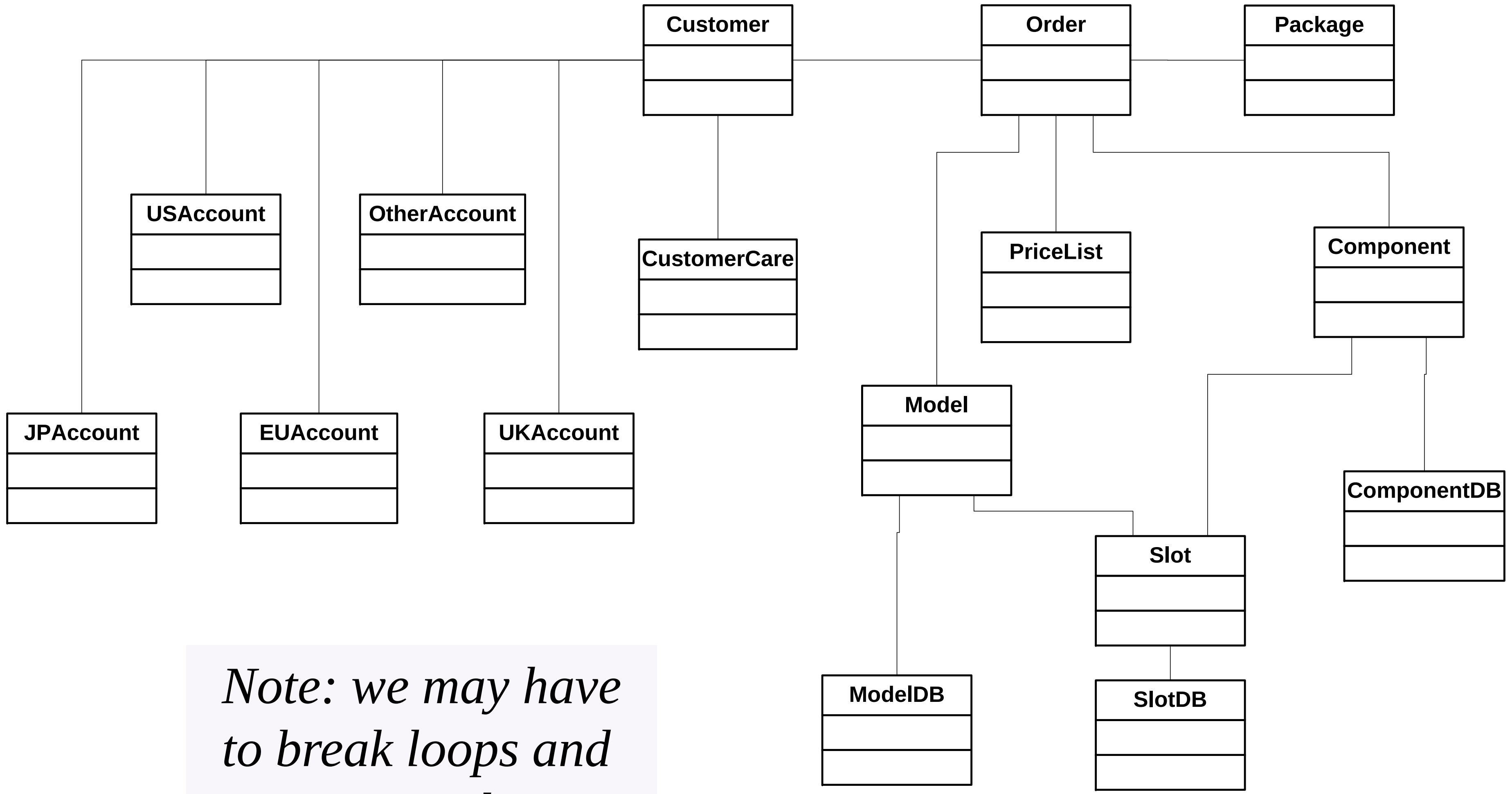
- The first level of *integration testing* for object-oriented software
  - Focus on interactions between classes
- Bottom-up integration according to “depends” relation
  - A depends on B: Build and test B, then A
- Start from use/include hierarchy
  - Implementation-level parallel to logical “depends”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



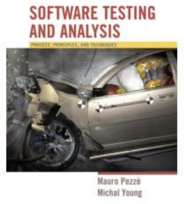


from a class diagram...

....to a hierarchy

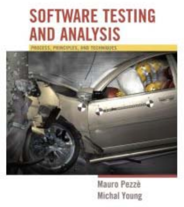


Note: we may have to break loops and generate stubs

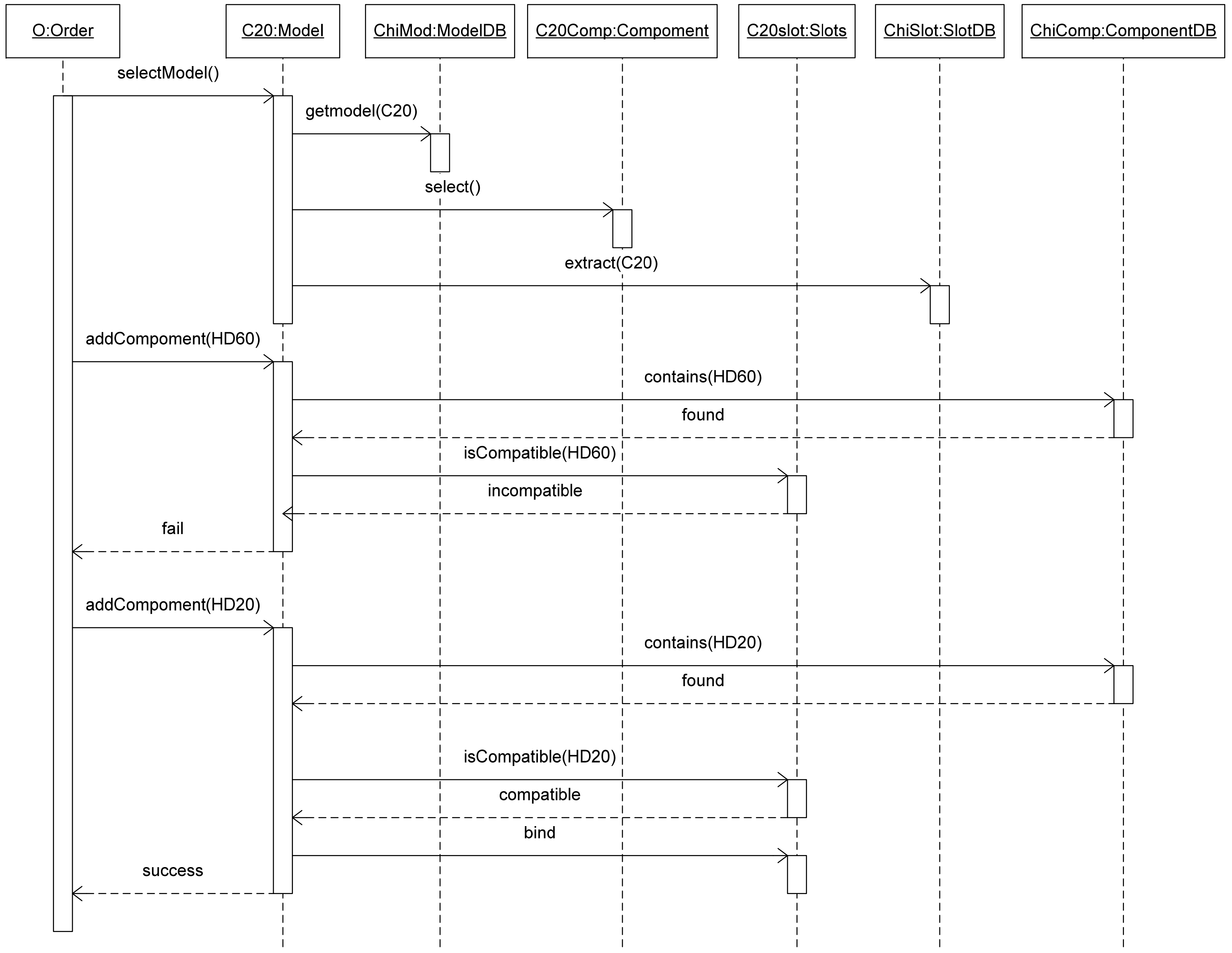


Interactions in Interclass Tests

- 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

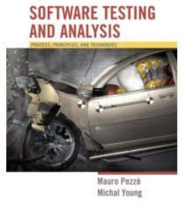


sequence diagram



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

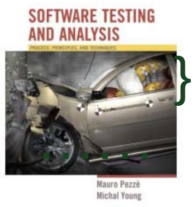


From the implementation ...

```
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;
       } ...}  ...  }
```

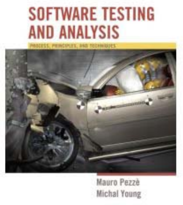
private instance variable

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

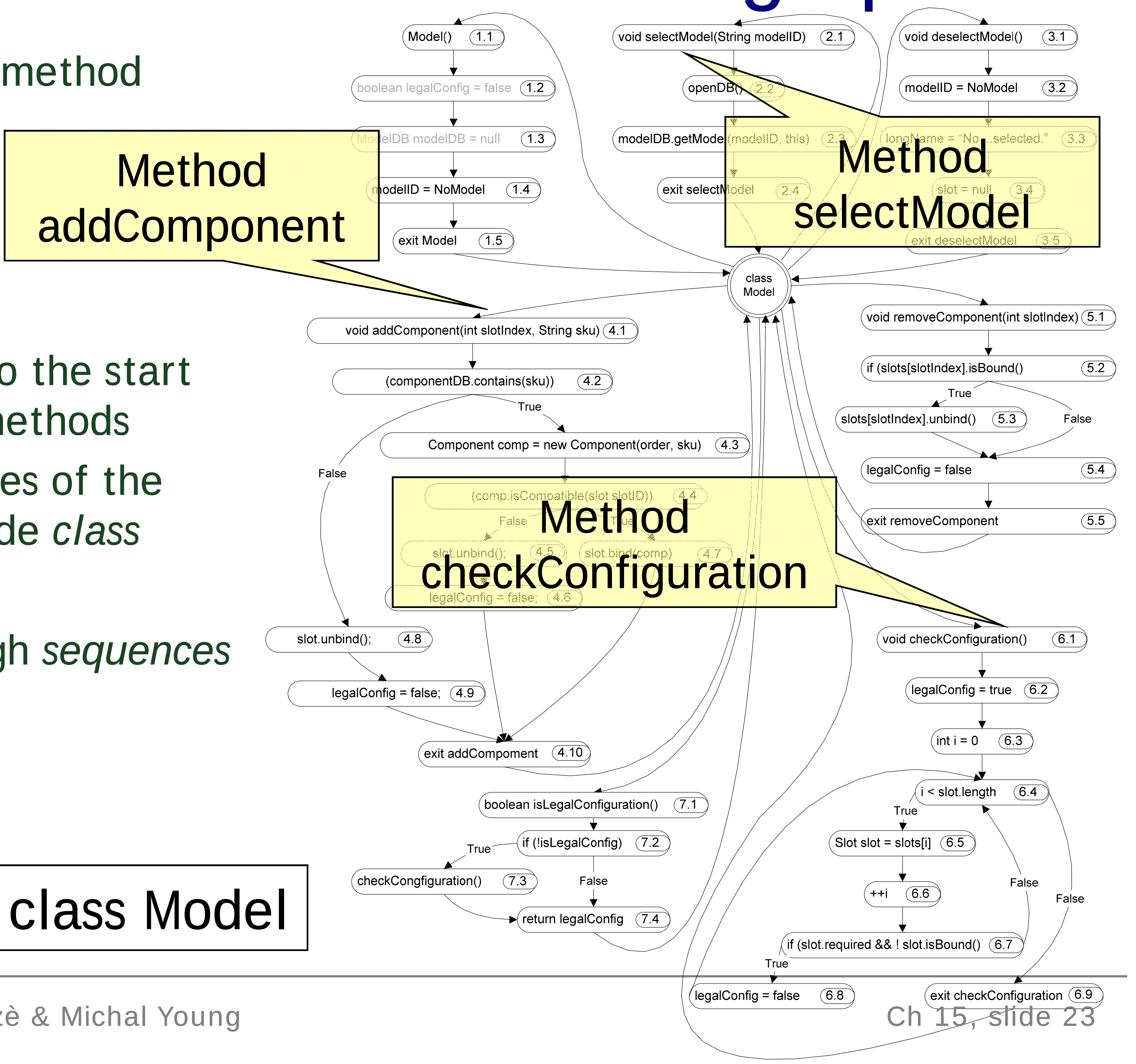
+

edges

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

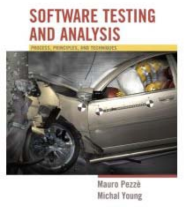
from the end nodes of the methods to node *class*

=> control flow through *sequences* of method calls



Method addComponent
Method selectModel
Method checkConfiguration
Model() 1.1
boolean legalConfig = false 1.2
ModelDB modelDB = null 1.3
modelID = NoModel 1.4
exit Model 1.5
void selectModel(String modelID) 2.1
openDB() 2.2
modelDB.getModel(modelID, this) 2.3
exit selectModel 2.4
void deselectModel() 3.1
modelID = NoModel 3.2
longName = "No ... selected." 3.3
slot = null 3.4
exit deselectModel 3.5
class Model
void addComponent(int slotIndex, String sku) 4.1
(componentDB.contains(sku)) 4.2
True
False
Component comp = new Component(order, sku) 4.3
(comp.isCompatible(slot.slotID)) 4.4
False
True
slot.unbind(); 4.5
slot.bind(comp) 4.7
legalConfig = false; 4.6
slot.unbind(); 4.8
legalConfig = false; 4.9
exit addCompoment 4.10
void removeComponent(int slotIndex) 5.1
if (slots[slotIndex].isBound() 5.2
True
False
slots[slotIndex].unbind() 5.3
legalConfig = false 5.4
exit removeComponent 5.5
void checkConfiguration() 6.1
legalConfig = true 6.2
int i = 0 6.3
i < slot.length 6.4
True
False
Slot slot = slots[i] 6.5
++i 6.6
False
if (slot.required && ! slot.isBound() 6.7
True
legalConfig = false 6.8
exit checkConfiguration 6.9
boolean isLegalConfiguration() 7.1
if (!isLegalConfig) 7.2
True
False
checkCongfiguration() 7.3
return legalConfig 7.4

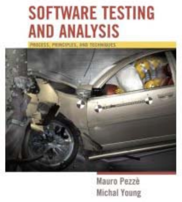
class Model



Interclass structural testing

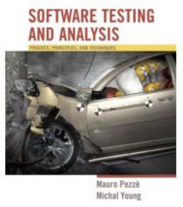
- Working “bottom up” in dependence hierarchy
    - Dependence is not the same as class hierarchy; not always the same as call or inclusion relation.
    - May match bottom-up build order
  - Starting from leaf classes, then classes that use leaf classes, …

- Summarize effect of each method: Changing or using object state, or both
  - Treating a whole object as a variable (not just primitive types)



Polymorphism and dynamic binding

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

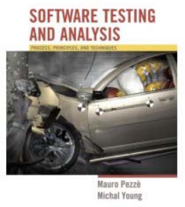


“Isolated” calls: the combinatorial explosion problem

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

EduCredit	USAccount	VISACard
BizCredit	UKAccount	AmExpCard
IndividualCredit	EUAccount	StoreCard
	JPAccount	
	OtherAccount	

The combinatorial problem: 3 x 5 x 3 = 45 possible combinations of dynamic bindings (just for this one method!)



The combinatorial approach

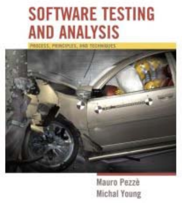
Identify a set of combinations that cover all pairwise combinations of dynamic bindings

Same motivation as pairwise specification-based testing

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

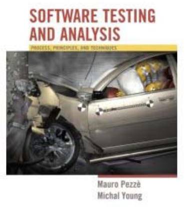
Inheritance

- 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!

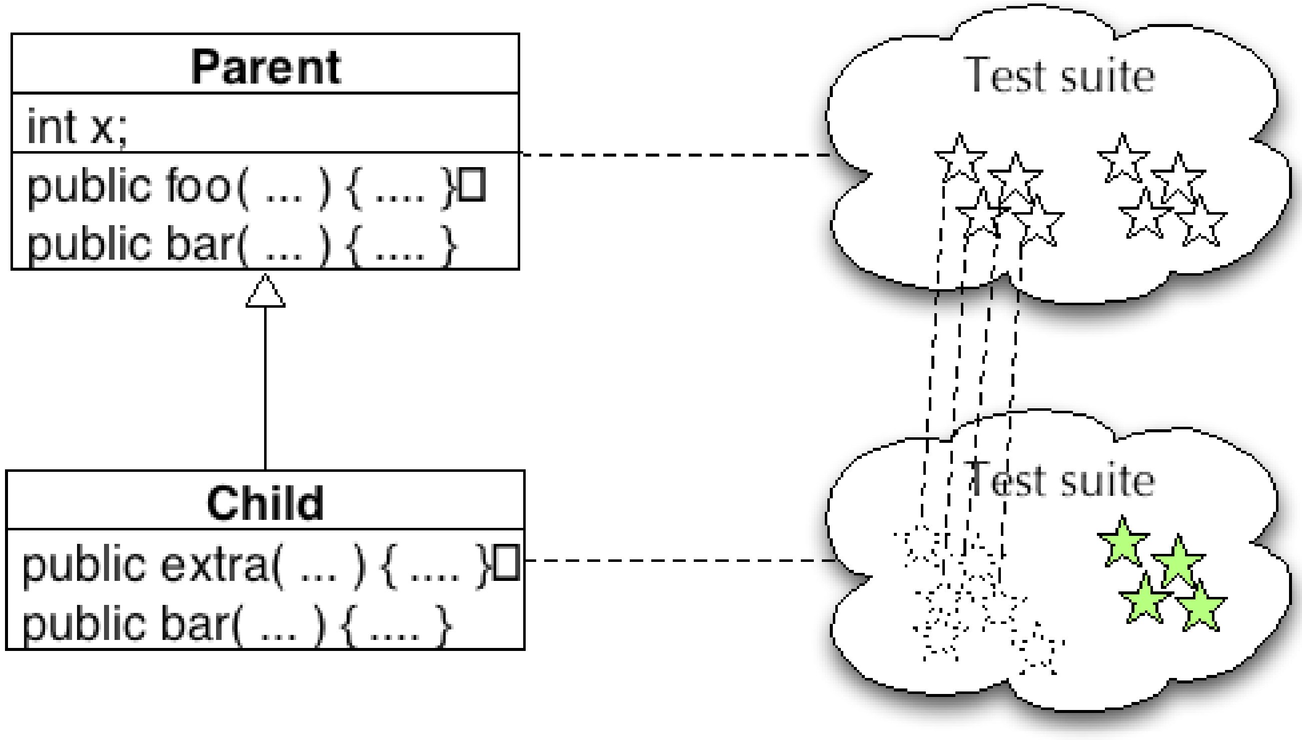


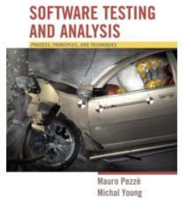
Reusing Tests with the Testing History Approach

- 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

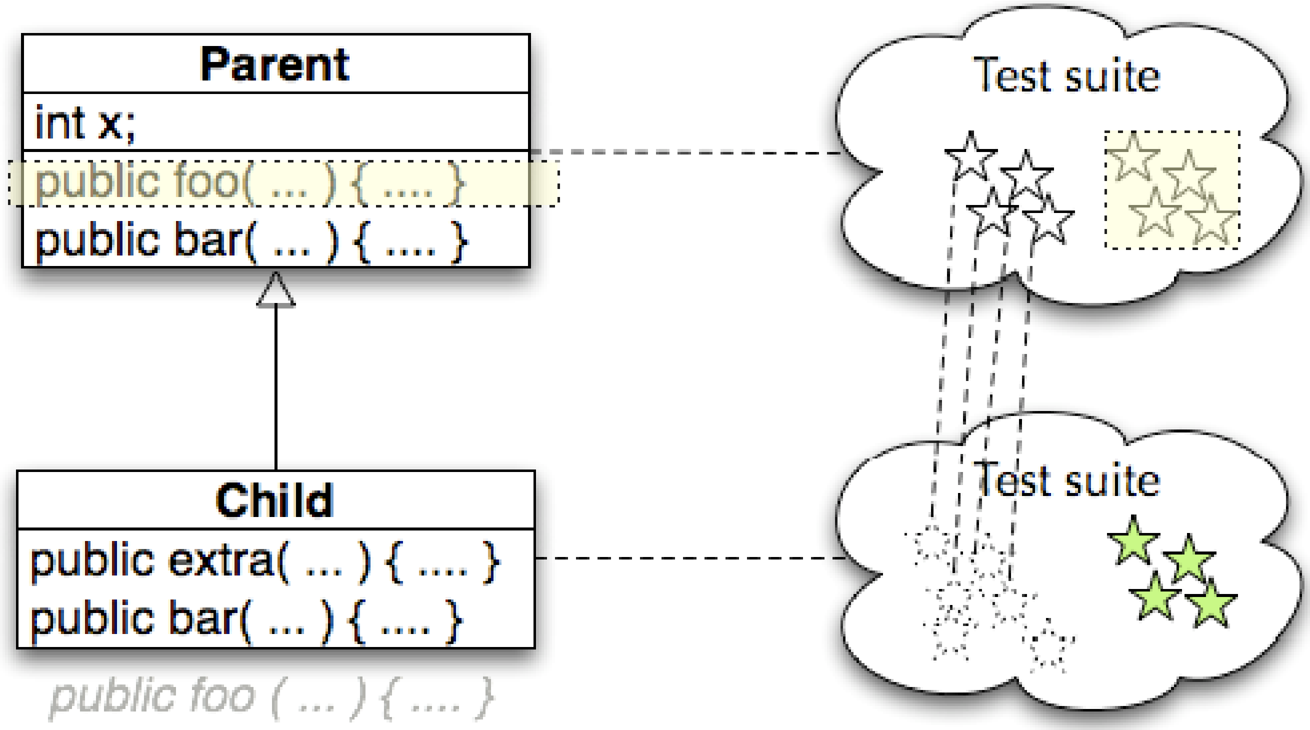


Testing history

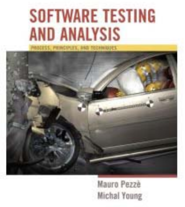

Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite



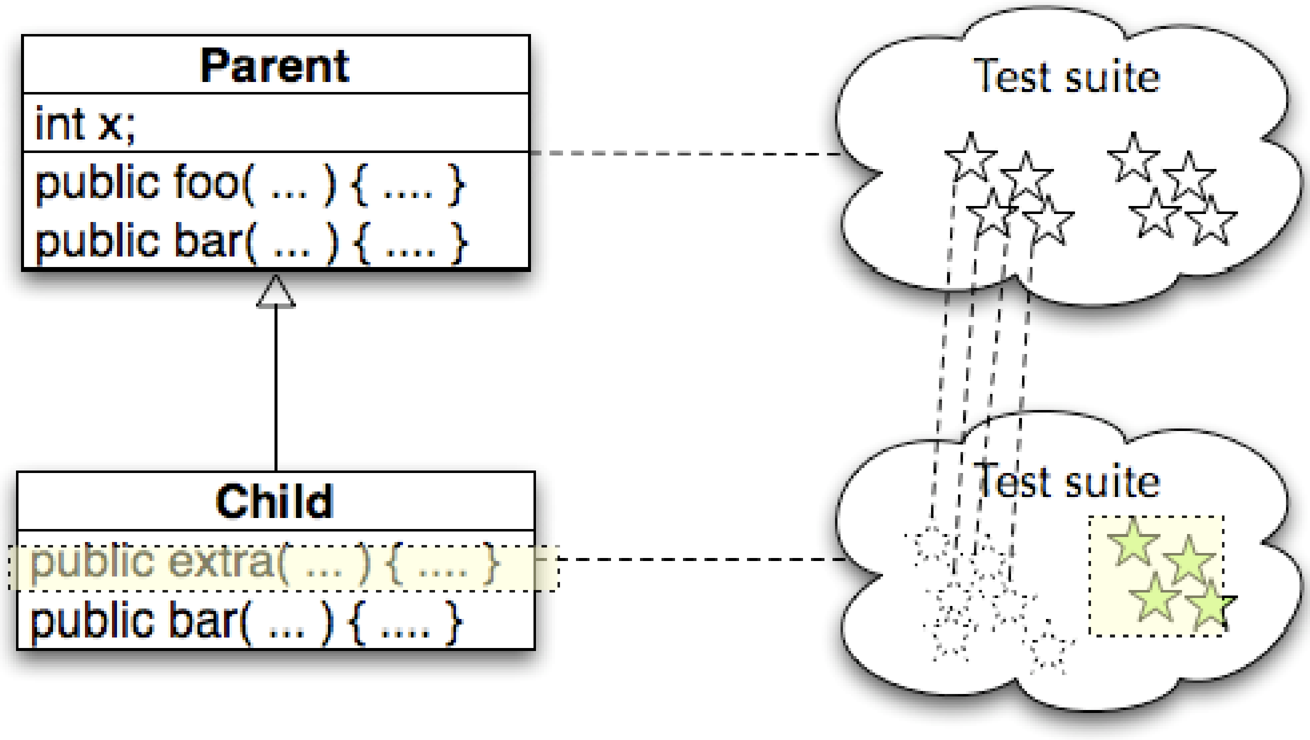
Inherited, unchanged


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
public foo (...) { .... }
Test suite
Test suite

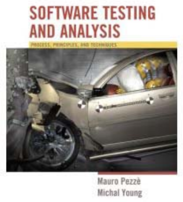
Inherited, unchanged ("recursive"):
No need to re-test



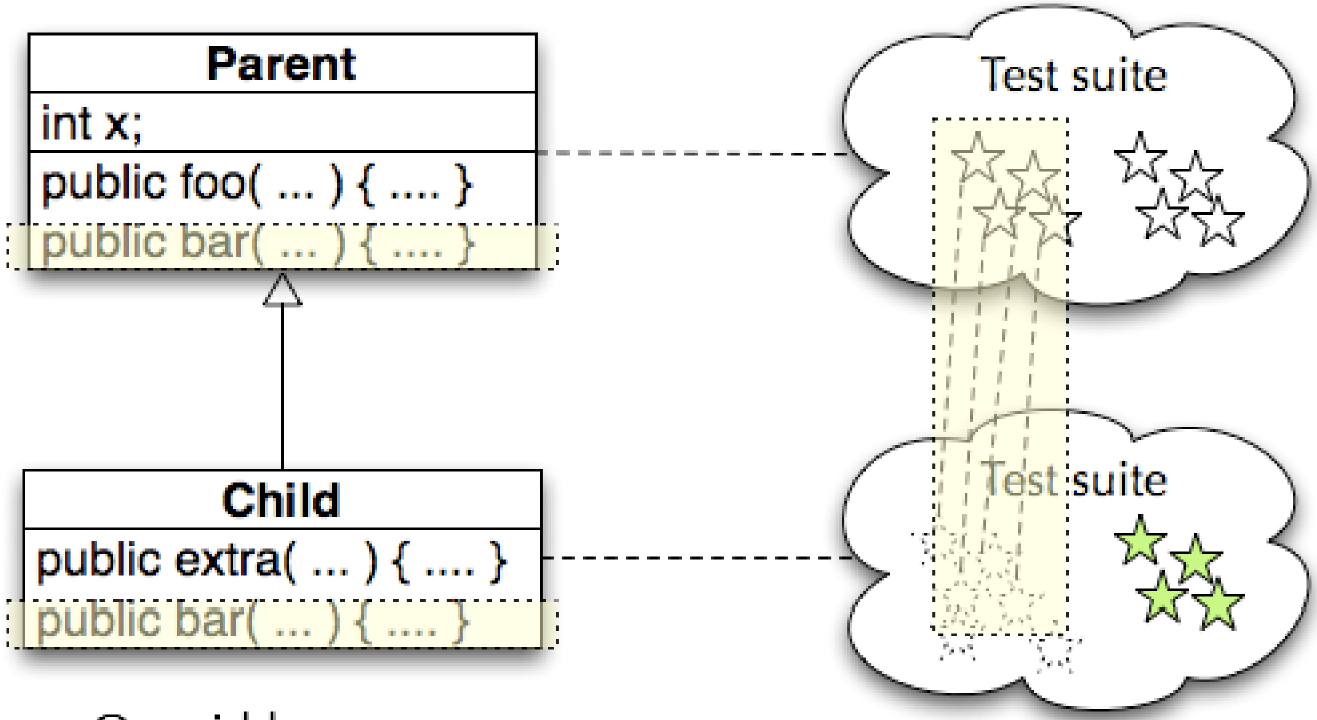
Newly introduced methods


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

New:
Design and execute new test cases

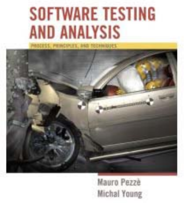


Overridden methods


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

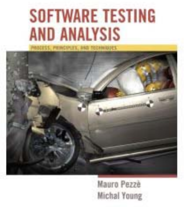
Overridden:

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

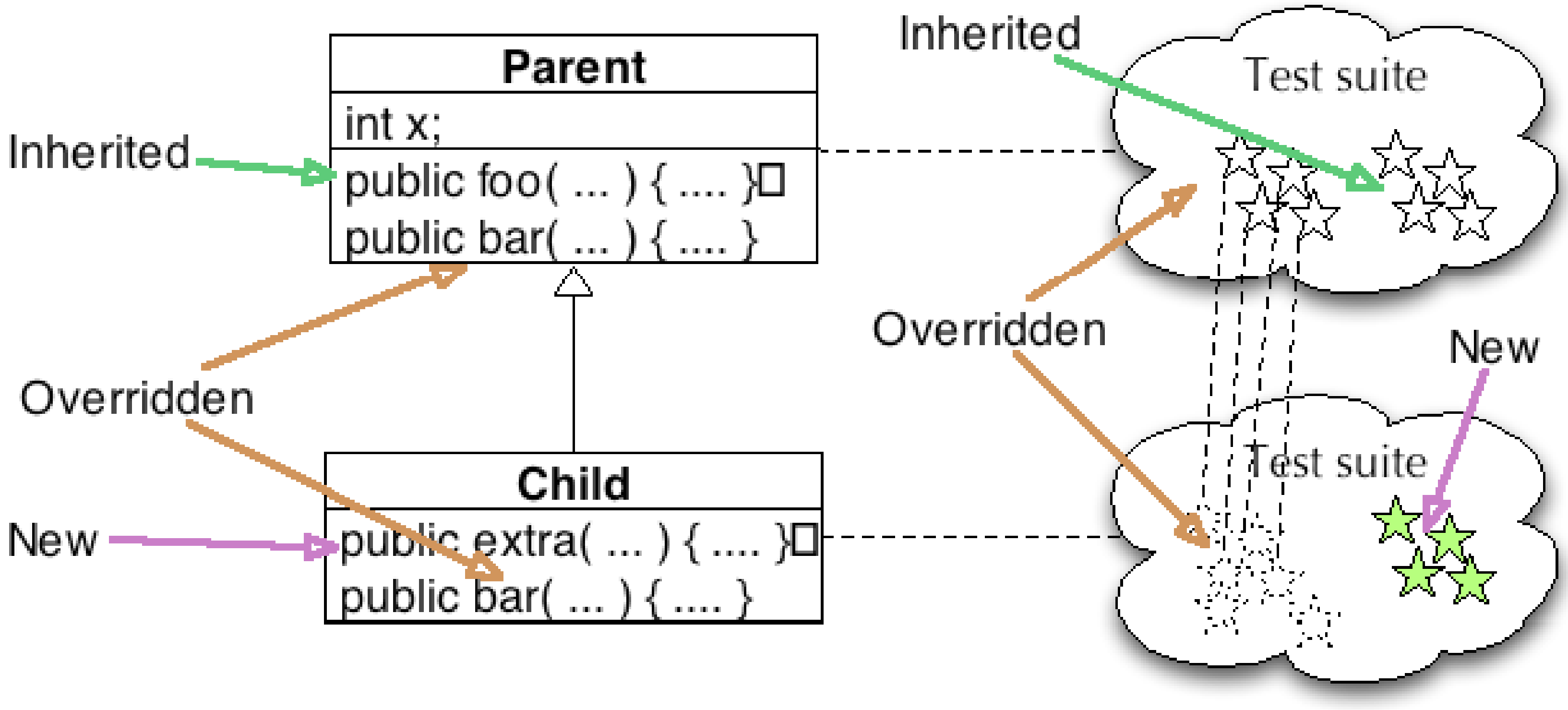


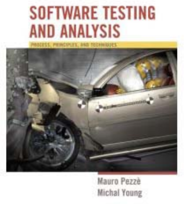
Testing History – some details

- 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



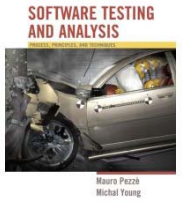
Testing History - Summary


Inherited
Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Overridden
Child
New
public extra(...) { .... }
public bar(...) { .... }
Inherited
Test suite
Overridden
New
Test suite



Does testing history help?

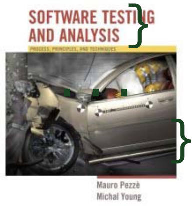
- Executing test cases should (usually) be cheap
  - It may be simpler to re-execute the full test suite of the parent class
  - … but still add to it for the same reasons
- But sometimes execution is not cheap …
  - Example: Control of physical devices
  - Or very large test suites
    - Ex: Some Microsoft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



Exception handling

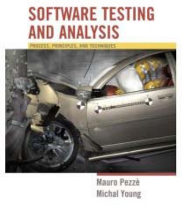
```
void addCustomer(Customer theCust) {
   customers.add(theCust);
     }
     public static Account
   newAccount(...)
   throws InvalidRegionException
     {
   Account thisAccount = null;
   String regionAbbrev = Regions.regionOfCountry(
                             mailAddress.getCountry());
   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



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