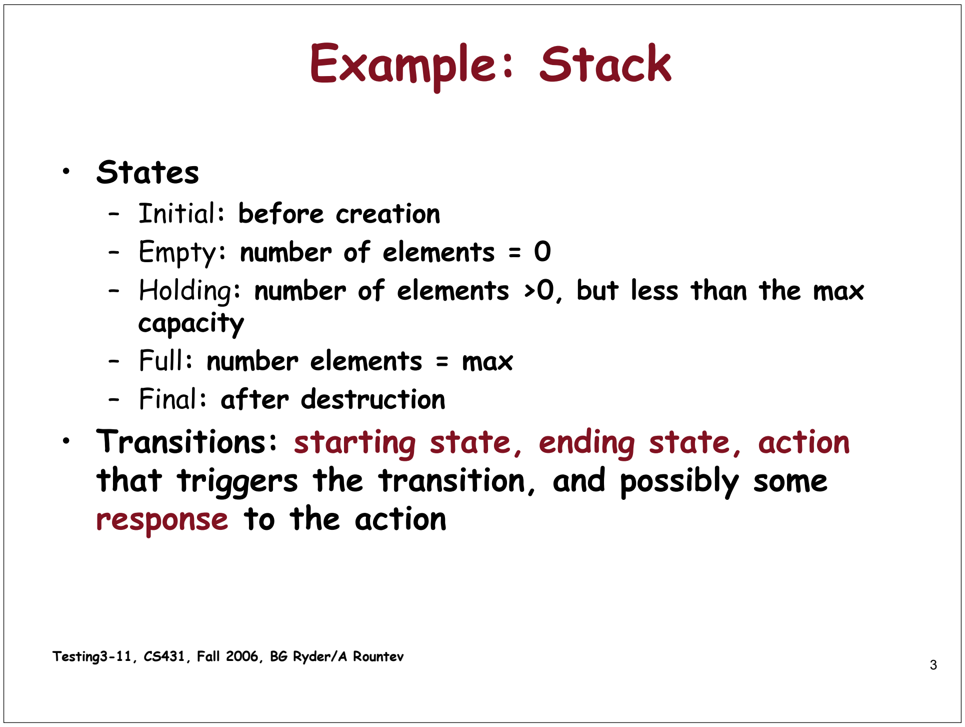
Testing3

- State-based testing
- Inheritance
- Testing interacting classes
  - Communication diagrams
  - Object relation graph (ORD)
- Regression testing
- GUI Testing

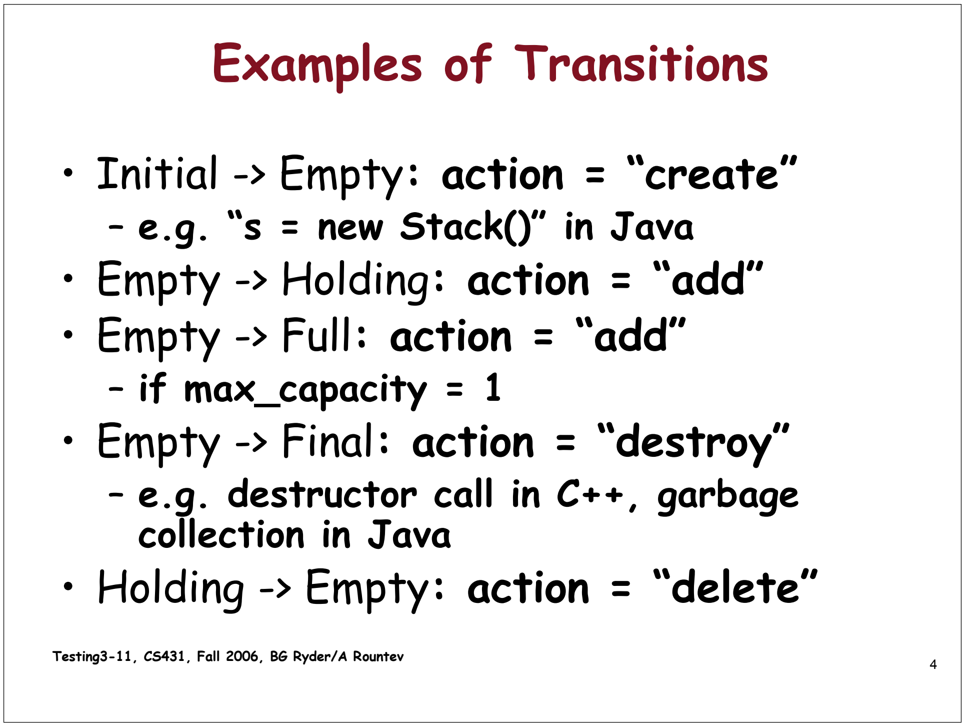
State-based Testing

- Natural representation with finite-state machines
  - States correspond to certain values of the attributes
  - Transitions correspond to methods
- FSM can be used as basis for testing
  - e.g. “drive” the class through all transitions, and verify the response and the resulting state



Example: Stack

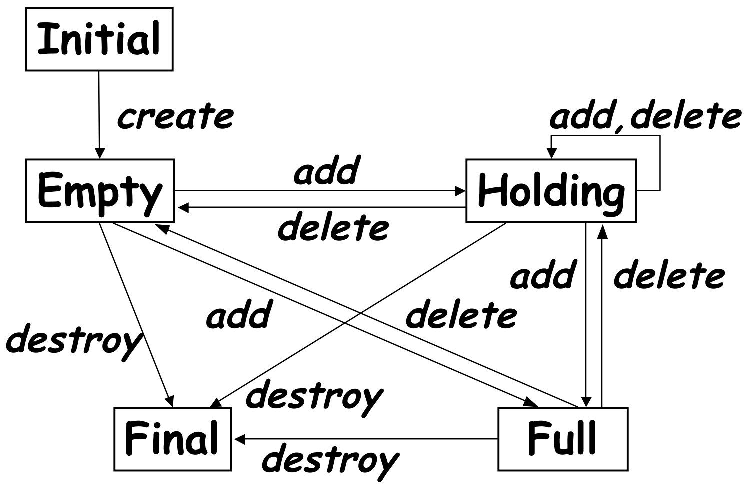
- **States**
  - Initial: **before creation**
  - Empty: **number of elements = 0**
  - Holding: **number of elements >0, but less than the max capacity**
  - Full: **number elements = max**
  - Final: **after destruction**
- **Transitions: starting state, ending state, action that triggers the transition, and possibly some response to the action**



Examples of Transitions

- Initial -> Empty: **action = "create"**
  - **e.g. "s = new Stack()" in Java**
- Empty -> Holding: **action = "add"**
- Empty -> Full: **action = "add"**
  - **if max_capacity = 1**
- Empty -> Final: **action = "destroy"**
  - **e.g. destructor call in C++, garbage collection in Java**
- Holding -> Empty: **action = "delete"**

Finite State Machine for a Stack

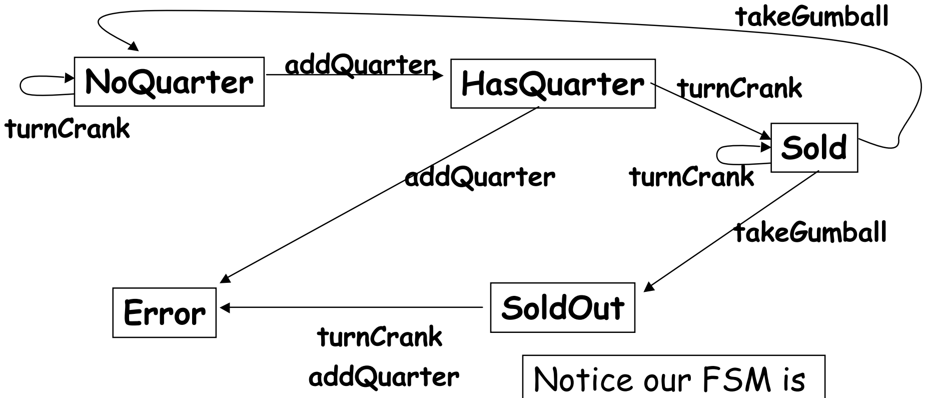


FSM-based Testing

- Each valid transition should be tested
  - Verify the resulting state using a state inspector that has access to the internals of the class
- Each invalid transition should be tested to ensure that it is rejected and the state does not change
  - e.g. Full -> Full is not allowed: we should call *add* on a full stack

Example 2

- Gumball machine from our State pattern
  - States: NoQuarter, HasQuarter, Sold, SoldOut
  - Transitions: turnCrank(), addQuarter(), takeGumball(), halt()



takeGumball

NoQuarter

addQuarter

HasQuarter

turnCrank

turnCrank

Sold

addQuarter

turnCrank

takeGumball

Error

SoldOut

turnCrank

addQuarter

Notice our FSM is non-deterministic

Inheritance

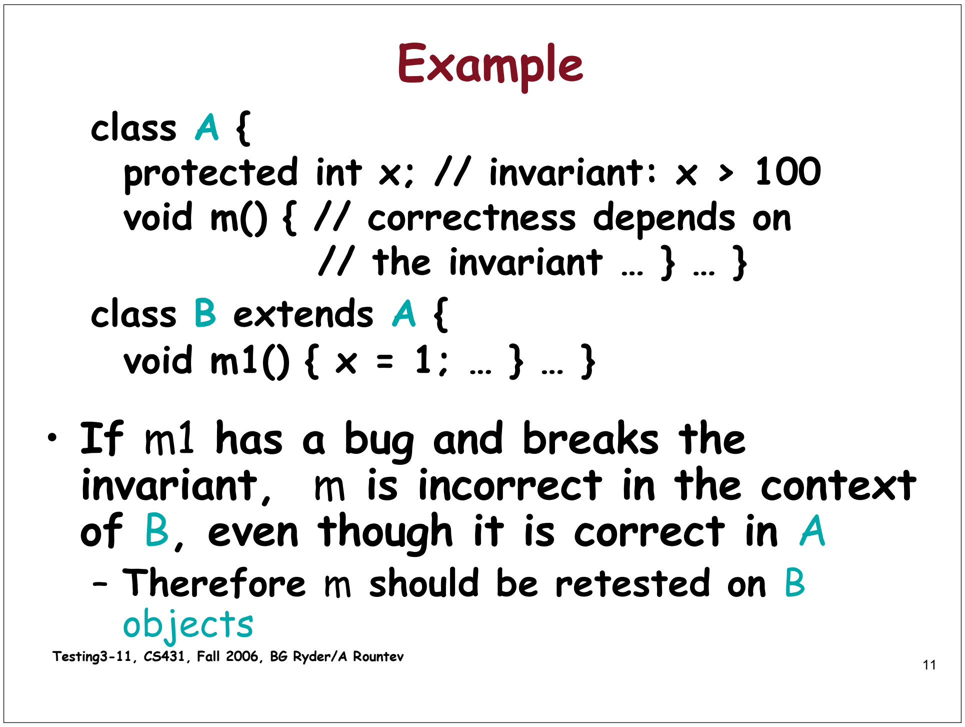
- People thought that inheritance will reduce the need for testing
  - Claim 1: "If we have a well-tested superclass, we can reuse its code (in subclasses, through inheritance) without retesting inherited code"
  - Claim 2: "A good-quality test suite used for a superclass will also be good for a subclass"
- Both claims are wrong

Problems with Inheritance

- Incorrect initialization of superclass attributes by the subclass
- Missing overriding methods
  - Typical example: equals and clone
- Direct access to superclass fields from the subclass code
  - Can create subtle side effects that break unsuspecting superclass methods
- A subclass violates an invariant from the superclass, or creates an invalid state

Testing of Inheritance

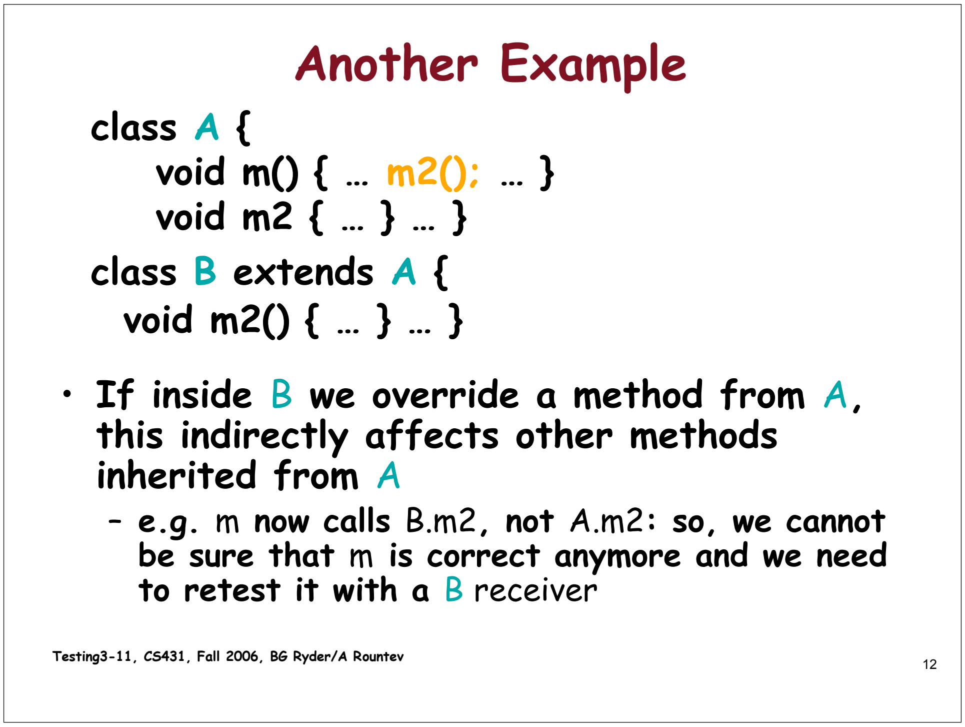
- Principle: inherited methods should be retested in the context of a subclass
- Example 1: if we change some method m() in a superclass, we need to retest m() inside all subclasses that inherit it
- Example 2: if we add or change a subclass, we need to retest all methods inherited from a superclass in the context of the new/changed subclass



Example

```
class A {
  protected int x; // invariant: x > 100
  void m() { // correctness depends on
             // the invariant … } … }
class B extends A {
  void m1() { x = 1; … } … }
```

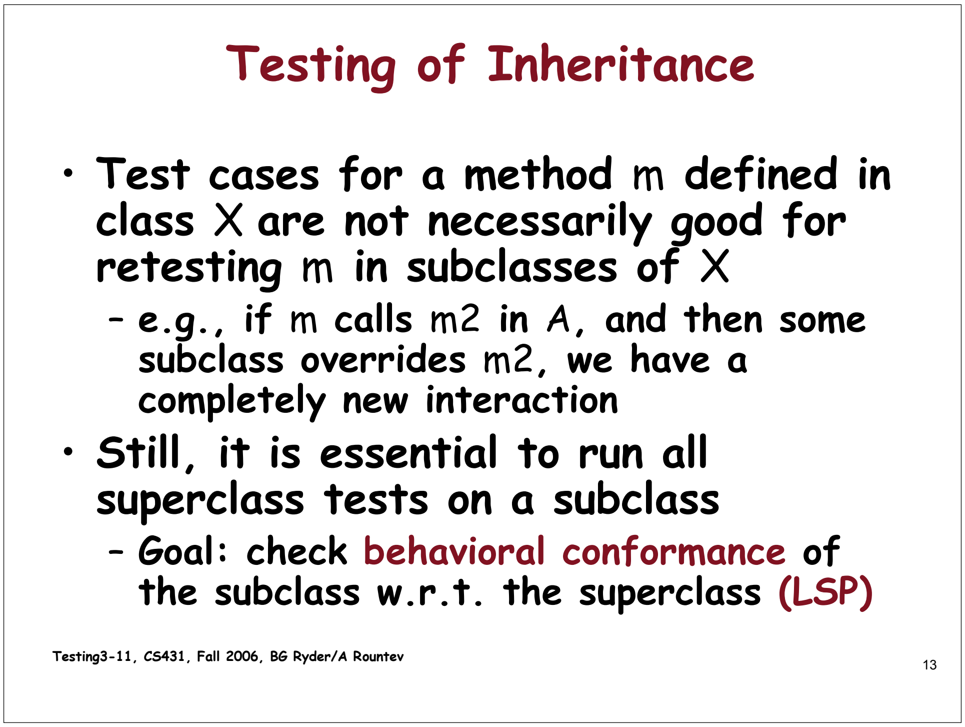
- If m1 has a bug and breaks the invariant, m is incorrect in the context of B, even though it is correct in A
  - Therefore m should be retested on B objects



Another Example

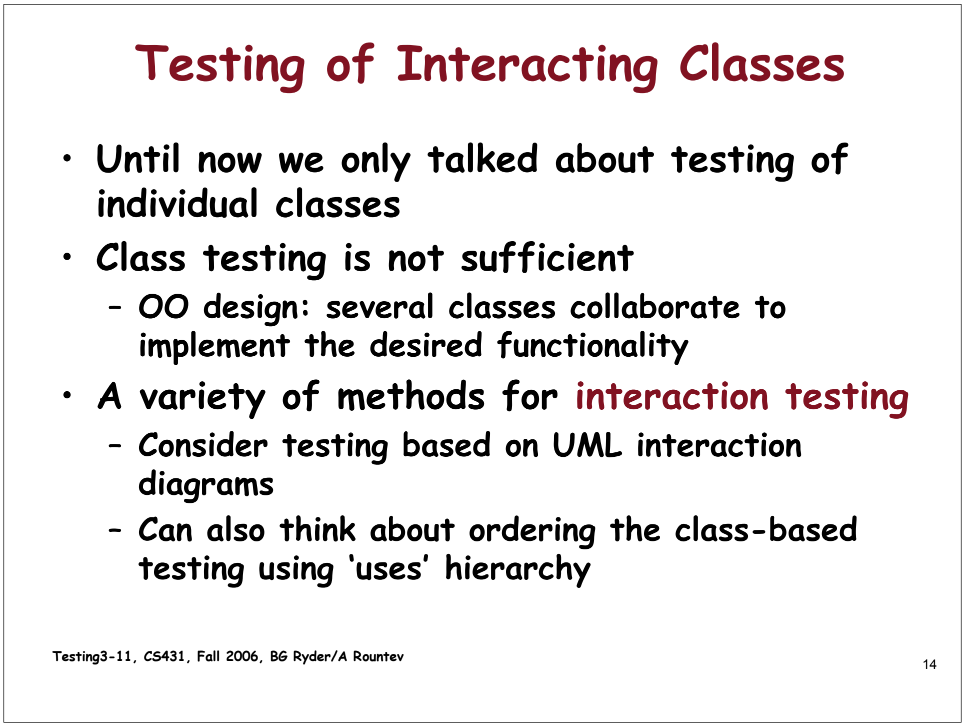
```
class A {
    void m() { … m2(); … }
    void m2 { … } … }
class B extends A {
  void m2() { … } … }
```

- If inside B we override a method from A, this indirectly affects other methods inherited from A
  - e.g. m now calls B.m2, not A.m2: so, we cannot be sure that m is correct anymore and we need to retest it with a B receiver



Testing of Inheritance

- Test cases for a method m defined in class X are not necessarily good for retesting m in subclasses of X
  - e.g., if m calls m2 in A, and then some subclass overrides m2, we have a completely new interaction
- Still, it is essential to run all superclass tests on a subclass
  - Goal: check behavioral conformance of the subclass w.r.t. the superclass (LSP)



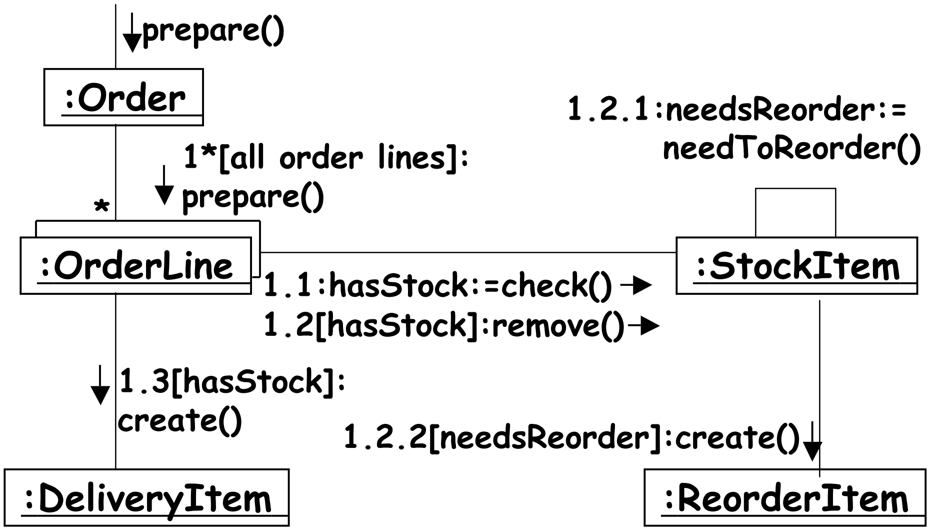
Testing of Interacting Classes

- Until now we only talked about testing of individual classes
- Class testing is not sufficient
  - OO design: several classes collaborate to implement the desired functionality
- A variety of methods for interaction testing
  - Consider testing based on UML interaction diagrams
  - Can also think about ordering the class-based testing using ‘uses’ hierarchy

UML Interaction Diagrams for Testing

- UML interaction diagrams: sequences of messages among a set of objects
  - There may be several diagrams showing different variations of the interaction
- Basic idea: run tests that cover all diagrams, and all messages and conditions inside each diagram
  - If a diagram does not have conditions and iteration, it contains only one path

Communication Diagram



prepare()

:Order

1*[all order lines]: prepare()

*

:OrderLine

1.1:hasStock:=check()

1.2[hasStock]:remove()

:StockItem

1.2.1:needsReorder:= needToReorder()

1.3[hasStock]: create()

1.2.2[needsReorder]:create()

:DeliveryItem

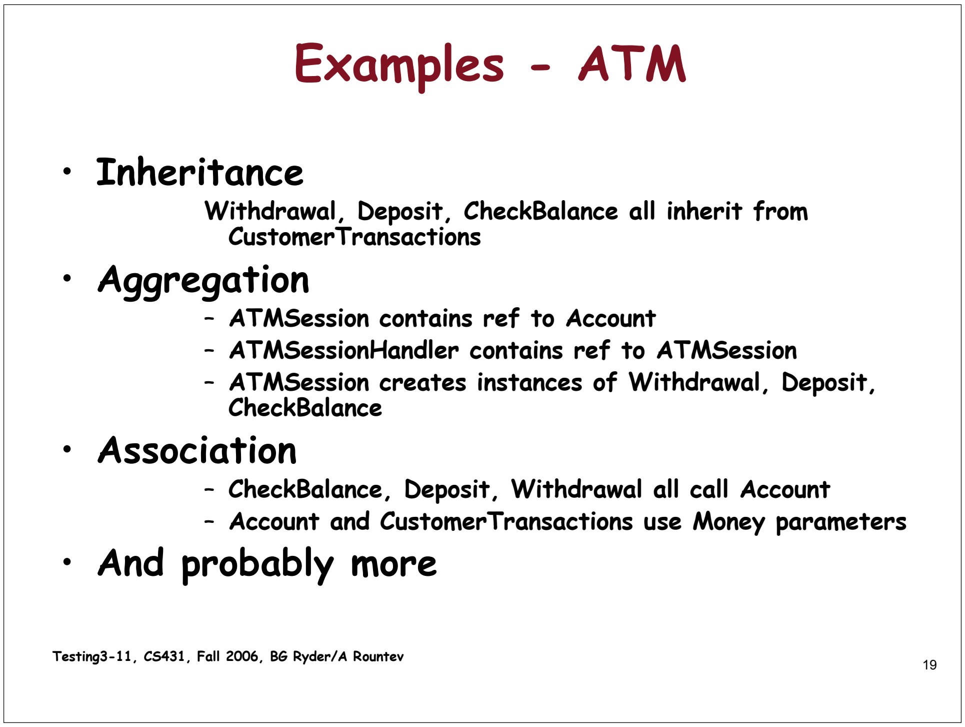
:ReorderItem

Coverage Requirements

- Run enough tests to cover all messages and conditions
  - test with 0 loop iterations and >=1 iterations
  - test with hasStock=true and hasStock=false
  - test with needsReorder=true and needsReorder=false
- To cover each one: pick a particular path in the diagram and “drive” the objects through that path

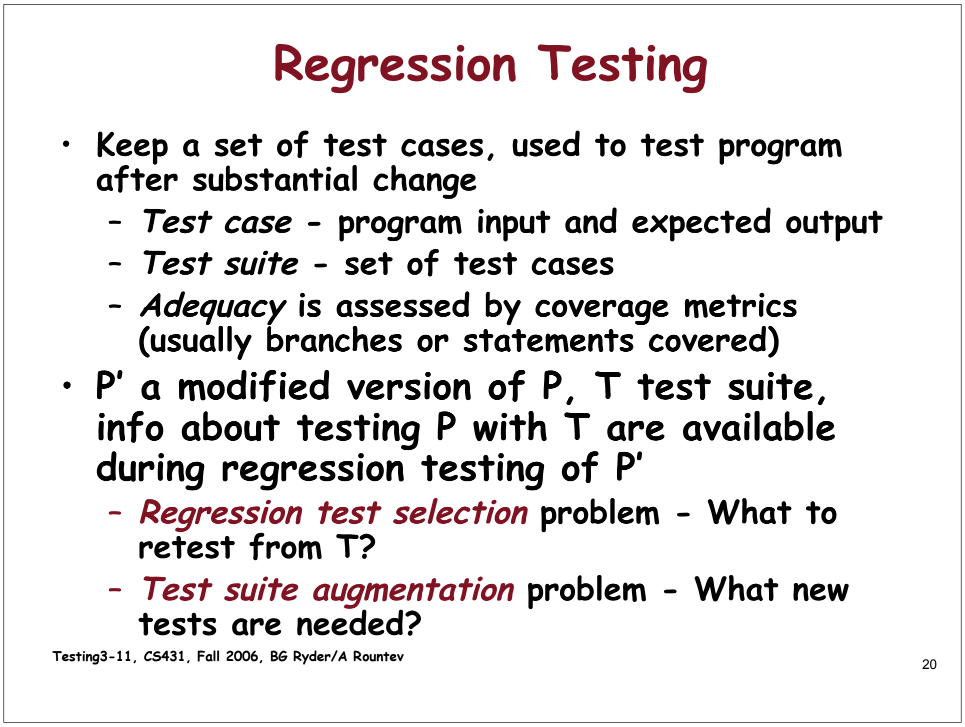
Object Relation Diagram

- ORD represents relationships between classes
  - Inheritance
  - *Aggregation* - describes relation between an aggregate object and its constituent parts
    - Objects of class B declared as instance or static fields of class A
    - Objects of class B dynamically created by methods in A
  - *Association* - 2 independent classes associate with each other (e.g., data or control dependence, message passing)
    - Class A uses data members of class B
    - Class B’s methods are invoked by a method in class A
    - Class B’s objects are formal parameters of a method in A



Examples - ATM

- Inheritance
  - Withdrawal, Deposit, CheckBalance all inherit from CustomerTransactions
- Aggregation
  - ATMSession contains ref to Account
  - ATMSessionHandler contains ref to ATMSession
  - ATMSession creates instances of Withdrawal, Deposit, CheckBalance
- Association
  - CheckBalance, Deposit, Withdrawal all call Account
  - Account and CustomerTransactions use Money parameters
- And probably more



Regression Testing

- Keep a set of test cases, used to test program after substantial change
  - *Test case* - program input and expected output
  - *Test suite* - set of test cases
  - *Adequacy* is assessed by coverage metrics (usually branches or statements covered)
- P' a modified version of P, T test suite, info about testing P with T are available during regression testing of P'
  - *Regression test selection* problem - What to retest from T?
  - *Test suite augmentation* problem - What new tests are needed?

Selective Regression Testing

- Only need to rerun tests which might be affected by program changes
    - Requires tool support for analysis
    - Need to know which tests 'cover' which edges/nodes in CFG
    - Need to know where the original P and edited program P' CFGs first differ on paths from method entry
  - Idea: do parallel traversal of CFG(P) and CFG(P'); when targets of like-labeled edges differed, then use coverage matrix to find tests that will exercise that edge
  - Q: Does this approach scale?

GUI Testing

P. Gerrard, "Testing GUI Apps",EuroSTAR'97

- Forms-based interfaces
    - Hierarchical
    - One-at-a-time
    - Sometimes in tabbed order
- GUIs
    - Allow multiple windows at once
    - Allow access thru; menu bars, buttons, keyboard short-cuts
    - No order constraints
    - User free to access system functionality in their own preferred manner

GUI Testing - Difficulties

- Challenges:
  - Event-driven system
    - Too many possible user inputs
    - Hard to anticipate context in which event handlers execute
  - Unsolicited events can occur
  - OO with large number of objects
  - Hidden synchronization and dependences
    - Many times objects depend on one another
    - E.g., if user selects check box then a text field is made invisible

GUI Testing -- Difficulties

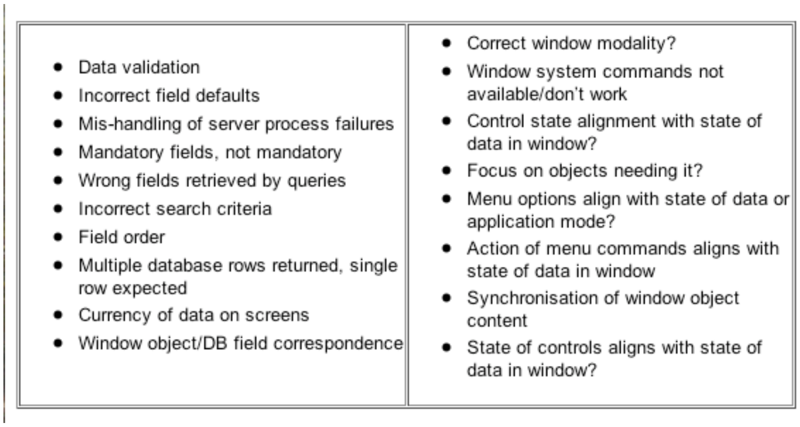
- Challenges, cont.
  - Infinite input domain
    - User can click anywhere on screen and enter data in any order
  - Many ways in and out
    - Many 'ways in' to reach the same point in application; do all need testing?
    - Many 'ways out' by using keyboard shortcuts, mouse, function keys; do all need testing?
  - Window management
    - Do we need to test O/S handling of window behavior (e.g., resizing, closing); which 'normal' window controls need testing?

Testing Strategies

- Oriented towards black-box testing
- Focus on categorizing errors into types
  - Test each type, thus adopting a divide and conquer approach
- Reuse traditional black-box testing of forms input, where possible
- Test in stages
  - Test lowest levels of detail first, then integrate components and test, then integrate entire application and tests
  - Build testing in trusted layers
- Automate wherever possible

Kinds of GUI Errors

P. Gerrard, "Testing GUI Apps",EuroSTAR'97



- Data validation
- Incorrect field defaults
- Mis-handling of server process failures
- Mandatory fields, not mandatory
- Wrong fields retrieved by queries
- Incorrect search criteria
- Field order
- Multiple database rows returned, single row expected
- Currency of data on screens
- Window object/DB field correspondence

- Correct window modality?
- Window system commands not available/don't work
- Control state alignment with state of data in window?
- Focus on objects needing it?
- Menu options align with state of data or application mode?
- Action of menu commands aligns with state of data in window
- Synchronisation of window object content
- State of controls aligns with state of data in window?

GUI Testing Stages

- Low-level (~unit)
    - Checklist
    - Navigation (reqs application backbone to simulate calls to window under test; window to invoke WUT; windows to be invoked by WUT
- Application(~unit or func-system test)
    - Focus on behavior of objects w/i windows- traditional techniques)
        - Equivalence partitioning and boundary value analysis
        - Decision tables
        - State transition testing

GUI Testing Stages

- Integration (func-syst test)
    - Interesting Q's: Dialogue vs 1 direct call? Info passed in 1 dirn or both dirns? Is call context-sensitive? Are there diff message types?
    - Kinds: Client/Server communication; Synchronization)
- Non-functional (non-func-syst test)
    - Soak tests - exercise app for long time to see memory-leak type errors
    - Compatibility - exercise app, switch to other apps, switch back - looks for resource problems
    - Platform configuration/environment