Object Oriented Testing

Filippo Ricca

DISI, Università di Genova, Italy

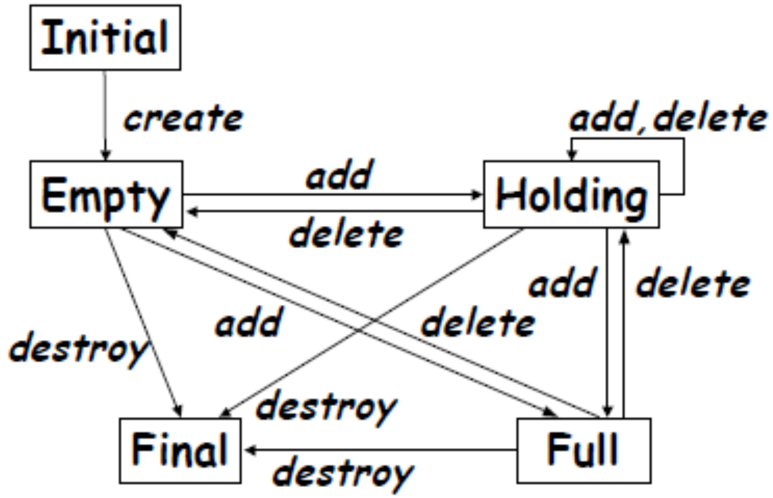
ricca@disi.unige.it

OO Testing

- Research confirms that testing methods proposed for procedural approach are not adequate for OO approach
    - Ex. Statement coverage
- OO software testing poses additional problems due to the distinguishing characteristics of OO
    - Ex. Inheritance
- Testing time for OO software found to be increased compared to testing procedural software

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



They simplify developing but complicate Testing!

OO definitions of unit and integration testing

- Procedural software
  - unit = single program, function, or procedure
- Object oriented software
  - unit = class
  - 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

State-based Testing

State-based Testing (1)

- 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
- Test cases are sequences of method calls that traverse the state machine

State-based Testing (2)

- State machine model can be derived from:
  - specification
  - code
    - also using reverse engineering techniques
  - or both …

- Accessing the state
  - add inspector method, e.g. getState()

FSM derived by code

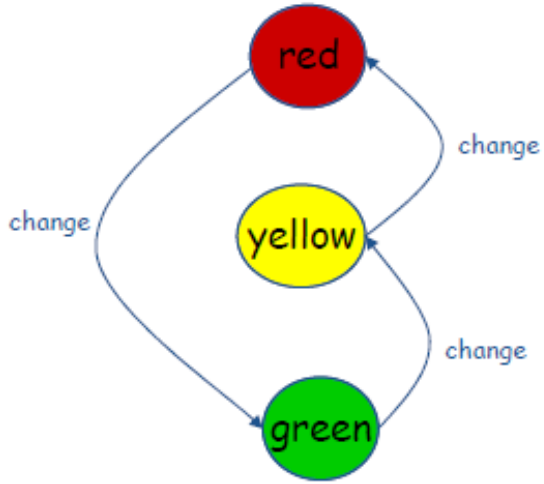
Example

```
class TrafficLight {
        public static final int RED = 0;
        public static final int YELLOW = 1;
        public static final int GREEN = 2;
        private int currentColor = RED;
        public int change() {
                switch (currentColor) {
        case RED:
                currentColor = GREEN;
                break;
        case YELLOW:
                currentColor = RED;
                break;
        case GREEN:
                currentColor = YELLOW;
                break;
                }
        return currentColor;
        }
        public int getCurrentColor() {
                return currentColor;
        }
}
```

TrafficLight

- ***int*** currentColor:{RED, YELLOW, GREEN}

- ***int*** change()
- ***int*** getCurrentColor()



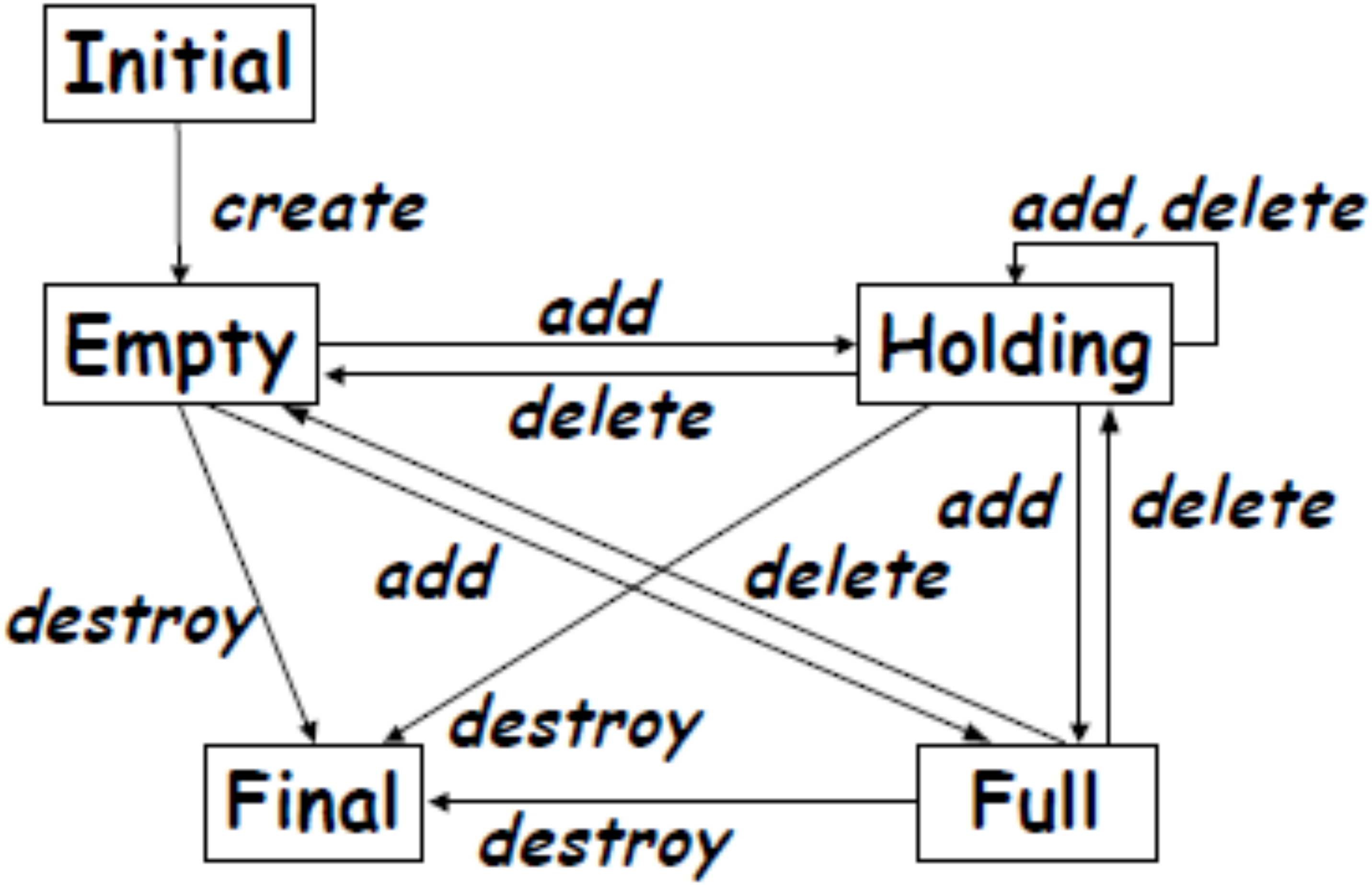
Example Stack

- **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:**
  - create, destroy
  - actions that triggers the transition
    - ex. Add, delete

Examples of transitions

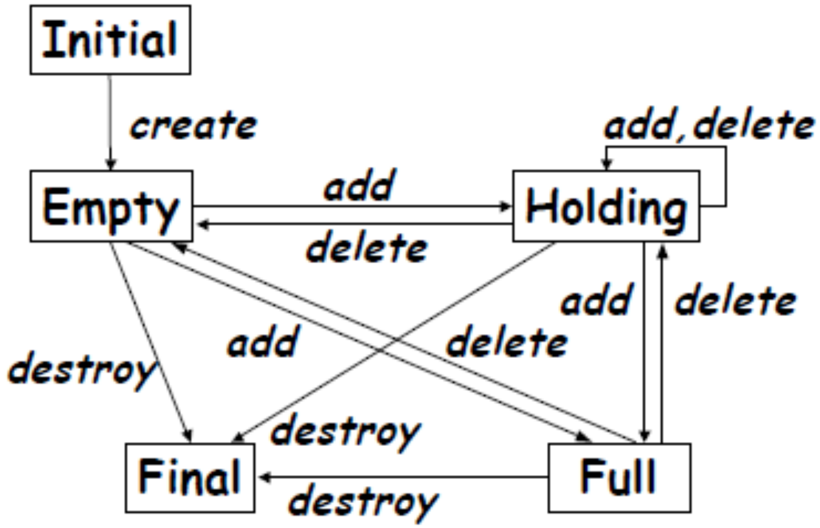
- Initial -> Empty**: action = “create”**
  - **e.g. “s = new Stack()” in Java**
- Empty -> Holding**: action = “add”**
- Empty -> Full**: action = “add”**
  - **if MAXcapacity=1**
- Empty -> Final**: action = “destroy”**
  - **e.g. destructor call in C++, garbage collection in Java**
- Holding -> Empty**: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

Finite State Machine for a Stack



Coverage methods

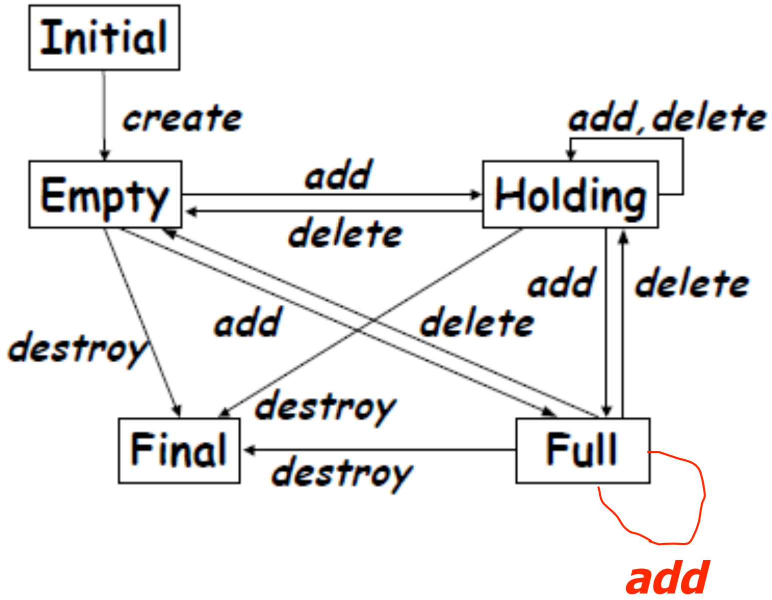
- Writing testcases such that:
  - Each state is covered
  - Each transition is covered
  - Each path is covered
    - Often infeasible



- Ex. State coverage
  - T1: Create, add, add, add [full]
  - T2: Create, destroy [final]

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
    - e.g., getState()



- 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
    - Exception "stack is full"

Junit Testcase: valid transitions

```
// only three elements ...
public void testStackFull() {
    Stack aStack = new Stack();
    assertEqual(“empty”, aStack.getState());
    aStack.push(10);
     assertEqual(“holding”, aStack.getState());
    aStack.push(1);
    aStack.push(7);
    assertEqual(“full”, aStack.getState());
}
```

To have transitions coverage adding other testcases to “drive” the class through all transitions!

Junit Testcase: invalid transition

```
// only three elements ...
public void testStackFull() {
    Stack aStack = new Stack();
    aStack.push(10);
    aStack.push(-4);
    aStack.push(7);
    assertEqual("full", aStack.getState());
    try {
        aStack.push(10)
        fail("method should launch the exception!!");
    } catch(StackFull e){
        assertTrue(true); // OK
    }
}
```

Example 2: Current account

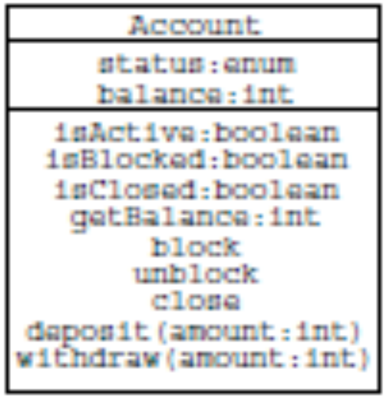


Figure 1: Static view of Account

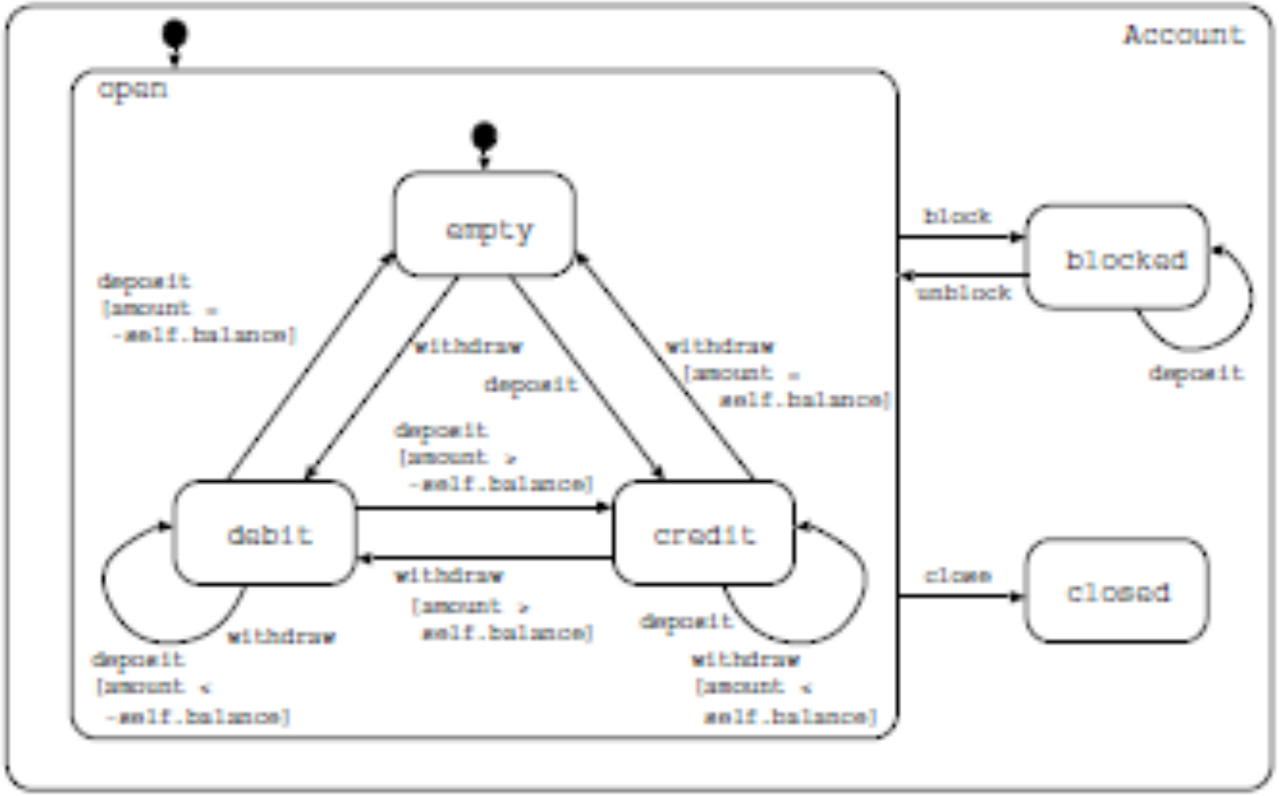
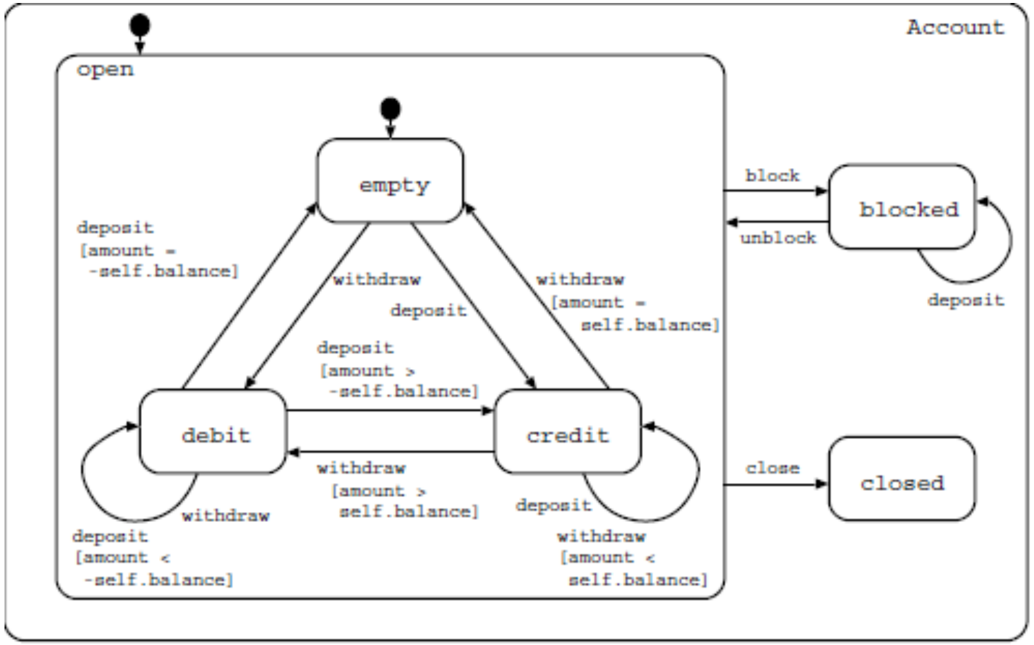


Figure 2: State diagram of class Account

Testcases for account



getState()

TC1: a=account(new); a.withdraw(5); ~~a.close()~~ -- debit

TC2: a=account(new); a.withdraw(5); a.deposit(5); ~~a.close()~~ -- empty

TC3: a=account(new); a.withdraw(2); a.deposit(5); ~~a.close()~~ -- credit

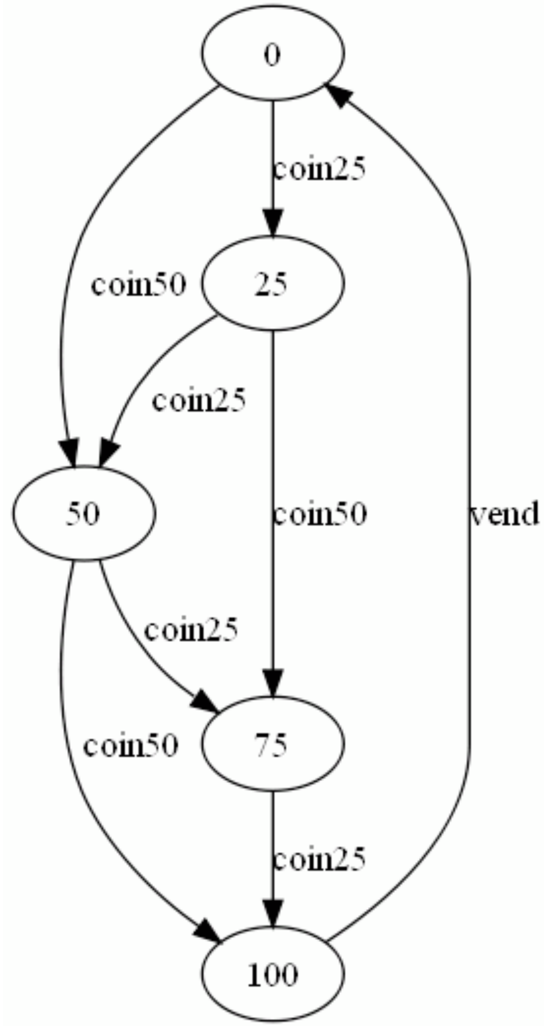
ModelJUnit

- ModelJUnit is a Java library that extends JUnit to support model-based testing
- Helpful for programmers:
  - Write models in Java
  - Focus on unit testing since it integrates well with JUnit
  - Already available test generation algorithms
    - Ex. randomwalk

SUT: simple vending machine

The system under test is illustrated by the following state diagram:

Current State	Event	Action	Next State
0	-		0
	coin25	add25	25
	coin50	add50	50
	-		75
	-		100
25	-		0
	-		25
	coin25	add25	50
	coin50	add50	75
	-		100
...	...	...	...
100	Vend	vend	0
...	...	...	...



To define the model

```
// require junit.jar and modeljunit.jar
import net.sourceforge.czt.modeljunit.*
import net.sourceforge.czt.modeljunit.coverage.*

class VendingMachineModel implements FsmModel {
    def state = 0 // 0,25,50,75,100
    void reset(boolean testing) {state = 0}

     boolean vendGuard() {state == 100}
     @Action void vend() {state = 0}

     boolean coin25Guard() {state <= 75}
     @Action void coin25() {state += 25}

     boolean coin50Guard() {state <= 50}
     @Action void coin50() {state += 50}
}
```

For each state <=75 create a new transition “coin25” going in state+25

Choosing the generation algorithm

Testcases generation

```java
public static void main(String[] aregs){
// create our model and a test generation algorithm
        Tester tester = new RandomTester(new VendingMachineModel());
// build the complete FSM graph for our model, just to ensure that we get accurate model coverage metrics.
       tester.buildGraph();
 // set up our favourite coverage metric
       CoverageMetric trCoverage = new TransitionCoverage();
  tester.addCoverageMetric(trCoverage);
     // A convenience method for adding known listeners and coverage metrics, with printing of messages on the output
        tester.addListener("verbose");
    // Generate some test sequences, with the given total length (sequence of 50 random tests)
       tester.generate(50);
}
```

This example just prints messages as the model is executed.

Output

Metrics Summary:
Action Coverage was 3/3
State Coverage was 5/5
Transition Coverage was 7/8

reset(true)
done (0, coin50, 50)
done (50, coin25, 75)
done (75, coin25, 100)
done (100, vend, 0)

reset(true)
done (0, coin25, 25)
done (25, coin50, 75)
done (75, coin25, 100)
done (100, vend, 0)

reset(true)
done (0, coin25, 25)
done (25, coin50, 75)
done (75, coin25, 100)
done (100, vend, 0)

reset(true)
done (0, coin50, 50)
done (50, coin50, 100)
done (100, vend, 0)

reset(true)
done (25, coin50, 75)
done (75, coin25, 100)
done (100, vend, 0)
....

Tests execution in Junit

- The test generation code within the above main method is usually written within the TestXYZ() methods of JUnit classes
- So that each time you run your Junit test suite, you will generate a suite of tests from your FSM model

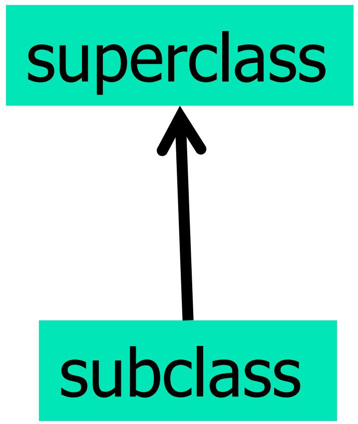
done (0, coin50, 50)
done (50, coin25, 75)
done (75, coin25, 100)
done (100, vend, 0)

```
public void TestVendingMachine() {
    vendingMachine v = new VendingMachine();
    v.reset();
    v.coin50();
    assertEqual(50, v.getState());
    v.coin25();
    assertEqual(75, v.getState());
    ...
}
```

Inheritance

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 without retesting inherited code"
  - **Claim 2**: "A good-quality test suite used for a superclass will also be good for a subclass"



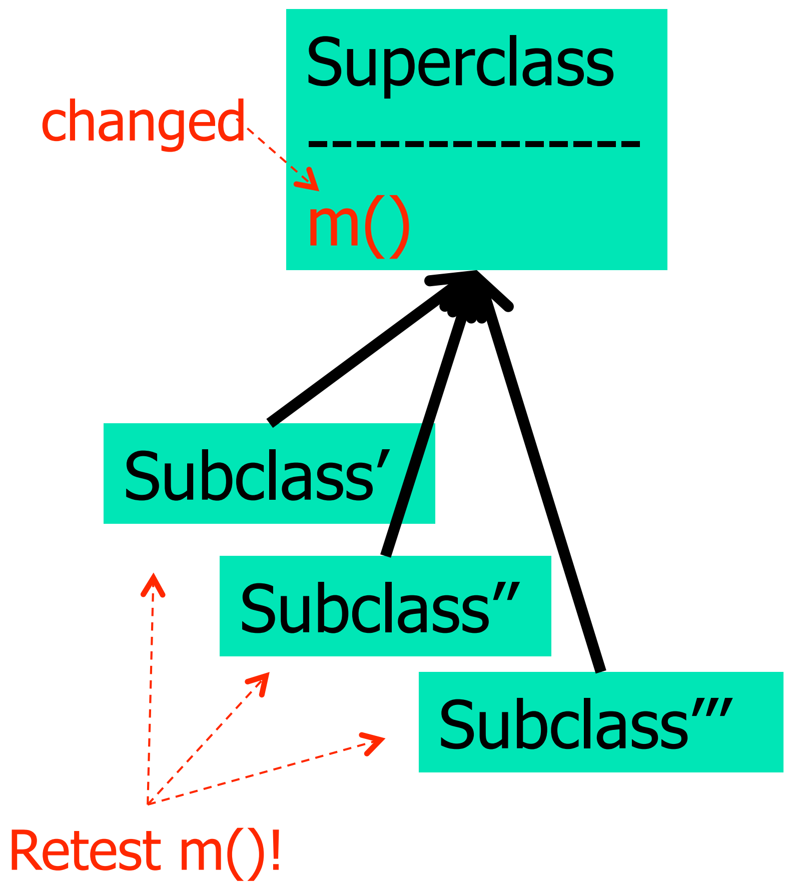
- Both claims are wrong!!!

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
- …

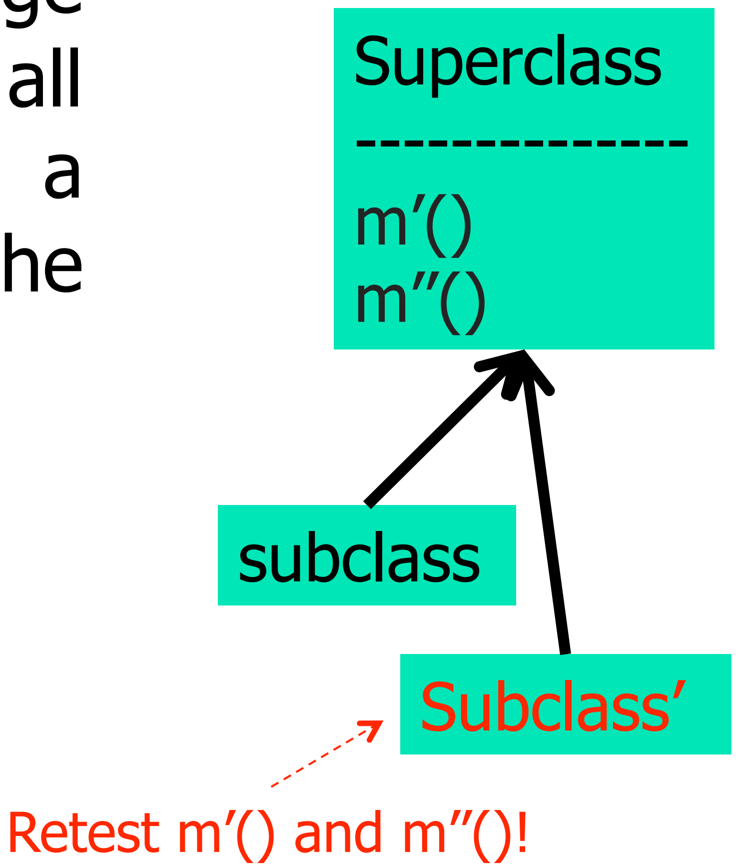
Testing of Inheritance (1)

- **<u>Principle:</u>** 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



Testing of Inheritance (2)

- **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() { … } … }
```

override

- 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 behavioural conformance of the subclass w.r.t. the superclass (LSP)

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

Testcases for m() in A
test m() that call A.m2()

Instead testcases for m() in B
should test the call B.m2()

The interaction is different

Polymorphism and dynamic binding

Combinatorial explosion problem

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

concrete

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

We have to test validateCredit in all the context!!!

There are some techniques to Reduce it …

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

Exception handling

Test of “Exceptions”

- **There are two cases:**
  1. We expect an anomalous behavior and then an exception
  2. We expect a normal behavior and then no exceptions

<u>How to manage exceptions</u>?

```
try {
        throw new AnException("message");
    }
catch (AnException e) { ... }
```

We expect an exception …

```
try {
   // we call the method with wrong parameters
   object.method(null);
  fail("method should launch the exception!!");
} catch(PossibleException e){
    assertTrue(true); // OK
}
```

```
class TheClass {
 public void method(String p)
           throws PossibleException
  { /*... */ }
}
```

"null launch the exception …"

We expect a normal behavior …

```
try {
    // We call the method with correct parameters
    object.method("Parameter");
    assertTrue(true); // OK
} catch(PossibleException e){
    fail ("method should not launch the exception !!!");
}
```

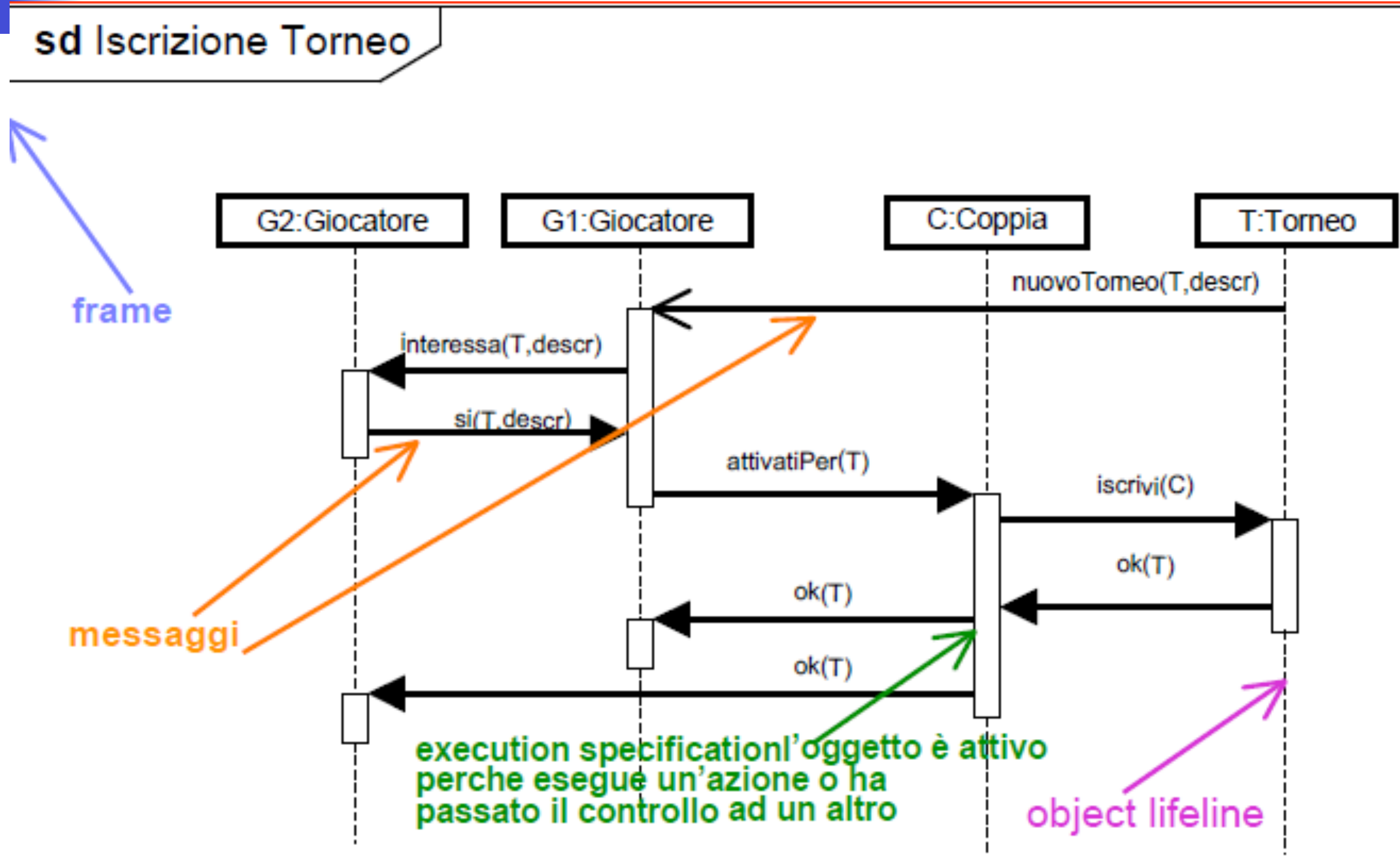
```
class TheClass {
 public void method(String p)
           throws PossibleException
  { /*... */ }
}
```

Integration/interaction Testing

Integration/interaction Testing

- 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
    - Sequence diagrams

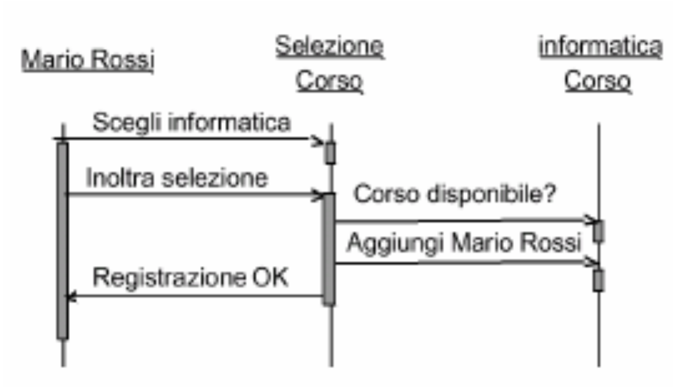
Sequence diagram



V.

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

Normal scenarios and alternatives

- Run enough tests to cover all messages and conditions
  - Normal scenarios
  - Alternatives
- To cover each one: pick a particular path in the diagram and "drive" the objects through that path

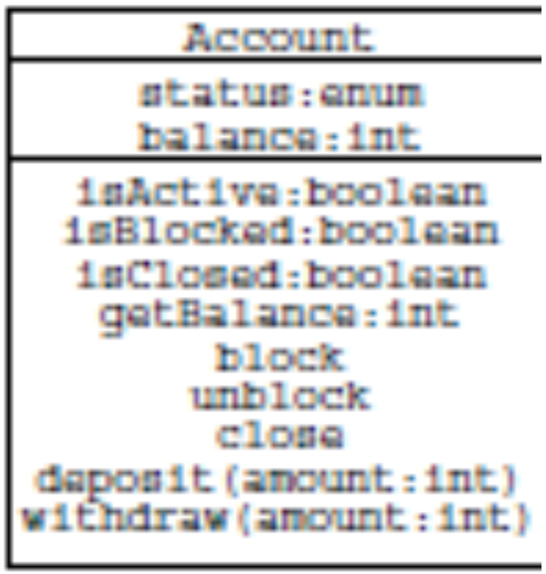
University course registration system

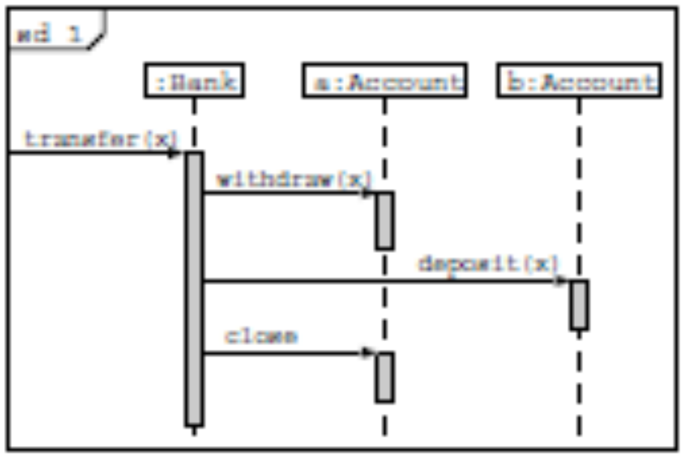
Esempio

Alternative:

- Mario Rossi
  - nome scorretto
  - nome già inserito
  - nome con piano di studi già definitivo
  - .....
- scegli informatica
  - corso diverso da informatica
  - corso inesistente
  - corso non disponibile
  - .....

Integration Testing example





Testing method **transfer** that call two objects Account

Junit Testcase

Bank
----
recoverAccount(...)
transfer(...)
....

```
public class BankTester extends TestCase {

 public void testTransfer() {
     Bank bank = new Bank();
     Account a = bank.recoverAccount("a");
     Account b = bank.recoverAccount("b");
     Euro balanceA = a.getBalance();
     Euro balanceB = b.getBalance();
     bank.transfer(50, a, b);
     assertTrue((balanceA-50).equalTo (a.getBalance()));
     assertTrue((balanceB+50).equalTo (b.getBalance()));
  }
}
```

Class Euro

Possible exercises at the exam

- **Class testing**
  - Given the implementation of a class:
    - Recover the FSM and writing testcases for having state, transition or/and path coverage
  - Given a FSM and the interface of a class (fields+methods)
    - writing Junit testcases to cover valid and invalid transitions
      - Ex. Stack
- **Integration testing**
  - Given some classes (interfaces) and one or more sequence diagrams deriving testcases
    - Valid and alternative sequences
  - Given two/three classes deriving a sequence diagram and writing the testcases

References
(used to prepare these slides)

- Slides of the book "Foundations of software testing" by Aditya P. Mathur
  - http://www.cs.purdue.edu/homes/apm/foundationsBook/InstructorSlides.html
- Slides of Barbara G. Ryder, Rutgers, The State University of New Jersey
  - http://www.cs.rutgers.edu/~ryder/431/f06/lectures/Testing3New-11.pdf
- Generating Test Sequences from UML Sequence Diagrams and State Diagrams
  - *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.5283&rep=rep1&type=pdf*
- Slides of Mauro Pezzè & Michal Young. Ch 15
  - http://ix.cs.uoregon.edu/~michal/book/slides/ppt/PezzeYoung-Ch15-OOTesting.ppt
- Model based testing
  - *http://users.encs.concordia.ca/~y_jarray/COEN345F2008/*