CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

CS608

Testing Object-Oriented Software in More Detail

(Essentials of Software Testing, Chapter 9, Sections 9.4-9.7)

ADVANCED OO TESTING

Overview of Advanced OO Testing

- Inheritance Testing
- UML-Based Testing
- Built-In Testing
- State-Based Testing

INHERITANCE TESTING

Inheritance Testing

- Addresses the Inheritance Fault Model
- Verify classes within an inheritance hierarchy behave correctly
- Simple form verifies that inherited superclass methods continue to work correctly in the context of the subclass
- This can be extended to ensure that all the methods in an entire inheritance tree continue to operate correctly

Inheritance Testing

- Addresses the Inheritance Fault Model
- Verify classes within an inheritance hierarchy behave correctly
- Simple form verifies that inherited superclass methods continue to work correctly in the context of the subclass
- This can be extended to ensure that all the methods in an entire inheritance tree continue to operate correctly
- **Fundamentally a test automation problem:**
  - **How to run XXXTest designed to run against class XXX against a subclass YYY**
- So we will address further under test automation

Note on Test Case Selection: Liskov Substitution Principle

- Not all subclasses fully support their superclass behaviour

Note on Test Case Selection: Liskov Substitution Principle

- Not all subclasses fully support their superclass behaviour
- Such subclasses are referred to as not being *Liskov Substitutable*

Note on Test Case Selection: Liskov Substitution Principle

- Not all subclasses fully support their superclass behaviour
- Such subclasses are referred to as not being *Liskov Substitutable*
- If a subclass is fully substitutable, the superclass tests are executed against each subclass to verify that the subclasses work in superclass context

Note on Test Case Selection: Liskov Substitution Principle

- Not all subclasses fully support their superclass behaviour
- Such subclasses are referred to as not being *Liskov Substitutable*
- If a subclass is fully substitutable, the superclass tests are executed against each subclass to verify that the subclasses work in superclass context
- If a subclass is **not** fully substitutable, then only a subset (and maybe none) of the superclass tests can be reused in the subclass inheritance test. Analysis must be performed to select the applicable tests. This analysis is made more difficult as the standard UML Class Diagram does not specify whether a subclass is fully substitutable or not.

Note on Test Case Selection: Liskov Substitution Principle

- Not all subclasses fully support their superclass behaviour
- Such subclasses are referred to as not being *Liskov Substitutable*
- If a subclass is fully substitutable, the superclass tests are executed against each subclass to verify that the subclasses work in superclass context
- If a subclass is **not** fully substitutable, then only a subset (and maybe none) of the superclass tests can be reused in the subclass inheritance test. Analysis must be performed to select the applicable tests. This analysis is made more difficult as the standard UML Class Diagram does not specify whether a subclass is fully substitutable or not.
- Additional tests must be written to verify the extra subclass features – not really an inheritance test issue

UML-BASED TESTING

UML-Based Testing

- The UML (Unified Modelling Language) is the main analysis and design tool for object oriented software
- There are a large number of diagrams in UML 2.5
- Each of these is a potential source of information and test coverage items for the tester
- Each item on each diagram has a meaning: something to test
- Examples:
    - Class Diagram: relationships between classes and associated multiplicities (one-to-N)
    - Sequence Diagram, Activity Diagram, Interaction Overview Diagram: the interaction between classes and methods

UML-Based Testing

- **Class Diagrams:**
  - Method testing in class context
  - Inheritance testing
- **State Diagrams:**
  - State-based testing
- All of the other diagrams can also be used to generate tests
- Examples:
  - Class Diagram:
    - relationships between classes and associated multiplicities (e.g. 1-to-N)
  - Sequence Diagram, Activity Diagram, Interaction Overview Diagram:
    - the interaction between classes and methods

BUILT-IN TESTING (BIT)

Built-in Testing

- Encapsulation can make testing difficult, especially when trying to access the class attributes in order to verify the actual results match the expected results
  - We have used getters so far, but they are not always available
- Some solutions:
  - Use Java reflection to access private attributes at run-time
  - Use **assertions** for built-in testing (BIT)
    - Available in many languages
    - Referred to as **built-in tests** as the test assertions are built into the code, rather than being located in an external test class
- Example: `assert space>0;`

BIT Issues

- Can be very effective for ensuring that assumptions that the programmer has made are in fact true when required in the code
- Can be effective in verifying that **class invariants** are maintained by asserting them at the end of every method
- In simple terms, a class invariant is a property of a class which must hold true between API calls
  - It must be true when a public method is called
  - It must be true when a public method returns

BIT Issues

- In Java, assertions can be turned on at runtime, so that during testing they are enabled, and in deployment they are disabled
- Less effective in replacing the usual unit tests, as often they need to refer to not only the **current** value of the method variables (attributes, parameters, and local variables) but also the **original** values when the method started execution
- Having the code keep copies of these values for testing can be very inefficient in terms of memory space and execution time. It also increases the effort required to code each method, and increases the chances of making a mistake

SpaceOrder with BIT

“safety conditions”

Used for class invariant

```
public class SpaceOrder {

  boolean special;
  boolean accept=false;
  int acceptedSpace=0; // must always be in [0..1024]

  public SpaceOrder(boolean isSpecial) {
    special = isSpecial;
    assert acceptedSpace >=0 && acceptedSpace <=1024;
  }

  public boolean getSpecial() {
    return special;
  }

  public boolean acceptOrder(int space) {
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    boolean status=true;
    acceptedSpace = 0;
    accept = false;
    if (space<=0) {
      status=false;
    }
    else if (space<=1024 && (space>=16 || special)) {
      accept = true;
      acceptedSpace = space;
    }
    // Check correct result here?
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    return status;
  }

  public boolean getAccept() {
    return accept;
  }

}
```

SpaceOrder with BIT

“safety conditions”

Used for class invariant

Check class invariant (end of constructor)

```
public class SpaceOrder {

  boolean special;
  boolean accept=false;
  int acceptedSpace=0; // must always be in [0..1024]

  public SpaceOrder(boolean isSpecial) {
    special = isSpecial;
    assert acceptedSpace >=0 && acceptedSpace <=1024;
  }

  public boolean getSpecial() {
    return special;
  }

  public boolean acceptOrder(int space) {
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    boolean status=true;
    acceptedSpace = 0;
    accept = false;
    if (space<=0) {
      status=false;
    }
    else if (space<=1024 && (space>=16 || special)) {
      accept = true;
      acceptedSpace = space;
    }
    // Check correct result here?
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    return status;
  }

  public boolean getAccept() {
    return accept;
  }

}
```

SpaceOrder with BIT

"safety conditions"

Used for class invariant

Check class invariant

Check class invariant (start of method)

```
public class SpaceOrder {

  boolean special;
  boolean accept=false;
  int acceptedSpace=0; // must always be in [0..1024]

  public SpaceOrder(boolean isSpecial) {
    special = isSpecial;
    assert acceptedSpace >=0 && acceptedSpace <=1024;
  }

  public boolean getSpecial() {
    return special;
  }

  public boolean acceptOrder(int space) {
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    boolean status=true;
    acceptedSpace = 0;
    accept = false;
    if (space<=0) {
      status=false;
    }
    else if (space<=1024 && (space>=16 || special)) {
      accept = true;
      acceptedSpace = space;
    }
    // Check correct result here?
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    return status;
  }

  public boolean getAccept() {
    return accept;
  }

}
```

SpaceOrder with BIT

"safety conditions"

Used for class invariant

Check class invariant

Check class invariant (start of method)

Check class invariant (end of method)

```
public class SpaceOrder {

  boolean special;
  boolean accept=false;
  int acceptedSpace=0; // must always be in [0..1024]

  public SpaceOrder(boolean isSpecial) {
    special = isSpecial;
    assert acceptedSpace >=0 && acceptedSpace <=1024;
  }

  public boolean getSpecial() {
    return special;
  }

  public boolean acceptOrder(int space) {
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    boolean status=true;
    acceptedSpace = 0;
    accept = false;
    if (space<=0) {
      status=false;
    }
    else if (space<=1024 && (space>=16 || special)) {
      accept = true;
      acceptedSpace = space;
    }
    // Check correct result here?
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    return status;
  }

  public boolean getAccept() {
    return accept;
  }

}
```

Be Careful

- Sometime class invariants must be temporarily broken within a method
- And if you call a method from within that method…
- …it will fail if it checks the invariant
- No real solution for manual BIT
- Except “be careful”
- Note: some formal methods provide solutions to this!

Verify acceptOrder() with BIT?

```
public boolean acceptOrder(int space) {
  assert acceptedSpace >=0 && acceptedSpace <=1024;
  boolean status=true;
  acceptedSpace = 0;
  accept = false;
  if (space<=0) {
    status=false;
  }
  else if (space<=1024 && (space>=16 || special)) {
    accept = true;
    acceptedSpace = space;
  }
  // Check correct result here?
  assert acceptedSpace >=0 && acceptedSpace <=1024;
  return status;
}
```

Check **acceptedSpace** is valid

Does not check **accept** or the **return value** are correct

Verifying Functionality with BIT

- An assertion to verify that the method acceptOrder() has worked correctly cannot always be so easily implemented
- Line 29 checks that acceptedSpace is valid

```
    // Check correct result here?
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    return status;
}
```

- But it does not check that the attribute accept or the return value is correct
- Requires access to the values of the **special** attribute and the **space** parameter when the method was entered
- These may have been changed during the method
- In this case they have not been modified, but a different algorithm or a fault in the code could have modified either of these

Saving Old Values

- As the values at the end of the method cannot be relied on to represent the values at the start of the method, copies need to be made at the start of the method
- In this simple code copies could be easily kept, but this would introduce the possibility of further faults
- And also, in general, complete copies of any referenced objects would be required: this is a non-trivial problem and often unrealistic to implement
- For example, if an array of Counters is passed as an input, then a copy of the entire array would need to be made, including copies of every Counter in the array

Results of Running EP Tests against SpaceOrder with BIT

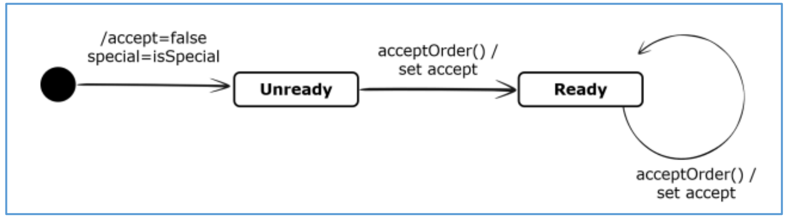
```java
public class SpaceOrder {

  boolean special;
  boolean accept=false;
  int acceptedSpace=0; // must always be in [0..1024]

  public SpaceOrder(boolean isSpecial) {
    special = isSpecial;
    assert acceptedSpace >=0 && acceptedSpace <=1024;
  }

  public boolean getSpecial() {
    return special;
  }

  public boolean acceptOrder(int space) {
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    boolean status=true;
    acceptedSpace = 0;
    accept = false;
    if (space<=0) {
      status=false;
    }
    else if (space<=1024 && (space>=16 || special)) {
      accept = true;
      acceptedSpace = space;
    }
    // Check correct result here?
    assert acceptedSpace >=0 && acceptedSpace <=1024;
    return status;
  }

  public boolean getAccept() {
    return accept;
  }

}
```

```
PASSED: testConstructor("SpaceOrderTest T1", true, true, false)
PASSED: testConstructor("SpaceOrderTest T2", false, false, false)
PASSED: testAcceptOrder("SpaceOrderTest T3", true, 7, true, true)
PASSED: testAcceptOrder("SpaceOrderTest T4", false, 504, true, true)
PASSED: testAcceptOrder("SpaceOrderTest T5", false, 5000, true, false)
PASSED: testAcceptOrder("SpaceOrderTest T6", false, -5000, false, false)
===============================================
Command line suite
Total tests run: 6, Passes: 6, Failures: 0, Skips: 0
===============================================
```

- All the tests pass
- The correct values returned and verified in the EP tests
- All the assertions in the built-in tests have passed

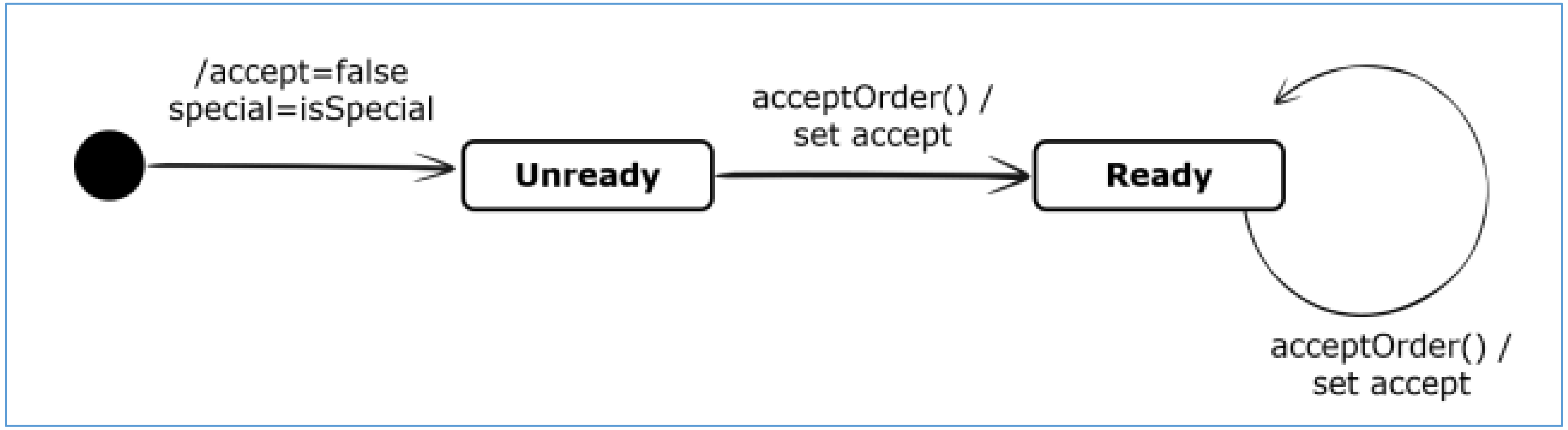
STATE-BASED TESTING

State-Based Testing



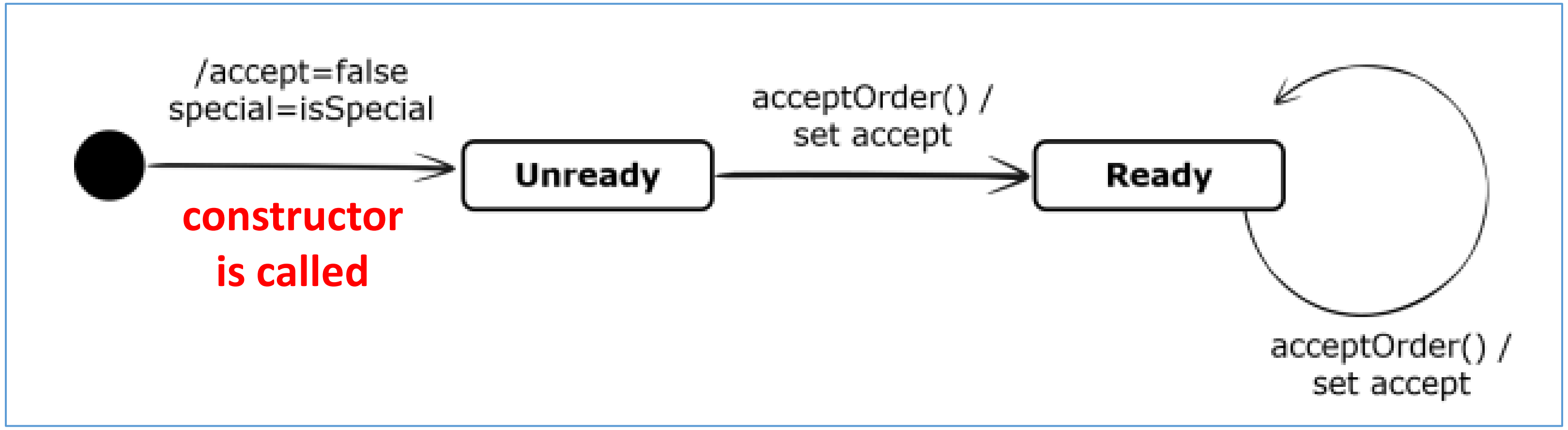
- This form of testing specifically addresses the class encapsulation/state fault model
- The purpose of state-based testing is to verify that a class behaves correctly with regards to its state specification (e.g. **UML State Machine Diagram**)
- A state diagram contains **states** and **transitions** between those states
- State-based testing verifies that the software transitions correctly between the states
- Not unique to OO – non-OO code can retain state between calls

Example State Diagram



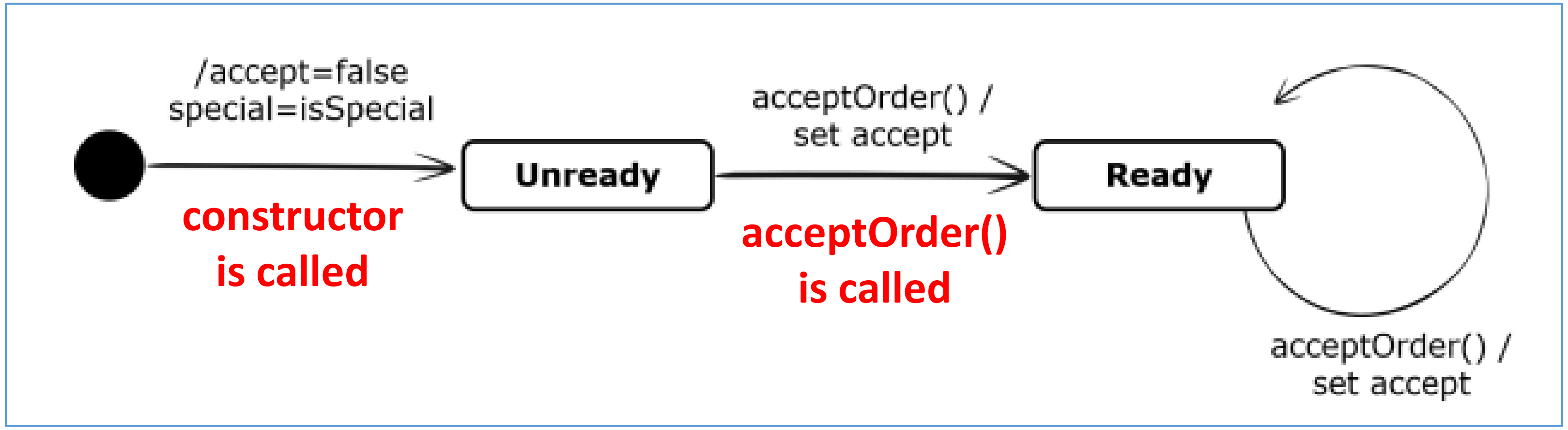
- Objects in the class can be in one of two states:
  1. Unready
  2. Ready
- This state will be represented in class attributes

Example State Diagram



- Objects in the class can be in one of two states:
  1. Unready
  2. Ready

Example State Diagram



- Objects in the class can be in one of two states:
  1. Unready
  2. Ready

Three Simple Test Strategies

- **All transitions** – every transition is verified at least once
- **All end-to-end paths** – every path from the start state to the end state is verified at least once. If there is no end state, which is common in software state diagrams, then every paths from the state to every *terminal state* (no outgoing transitions) can be used
- **All circuits** – every path that starts and ends in the same state is verified
- There are other more complex strategies also (see Binder)…

Transitions

- Each transition is specified by an event/activity pair:
  - The event which causes the transition
  - The accompanying action which should occur

Event / Action Pair

- Each transition is specified by an event/activity pair:
  - The event which causes the transition
  - The accompanying action which should occur

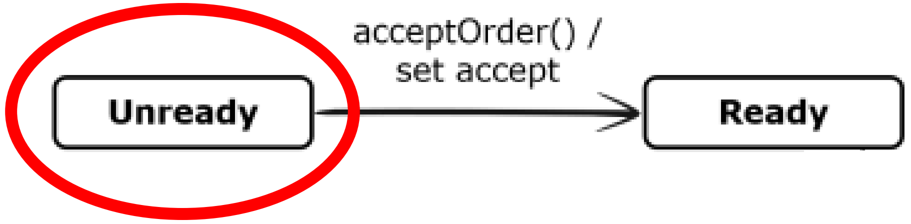
  - Event / Action
  - Example:
    - event=**acceptOrder()**
    - action=**set accept** (to true or false as specified)

Checking a Transition

- Each transition is specified by an event/activity pair:
- The correct operation of a transition is verified by:

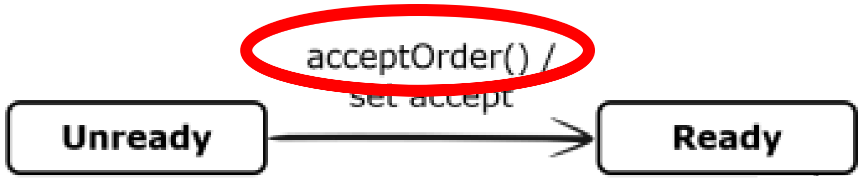
Checking a Transition

- Each transition is specified by an event/activity pair:
- The correct operation of a transition is verified by:
  - Check that the software is in the correct start state (this may have already been done by the verification of the previous transition)



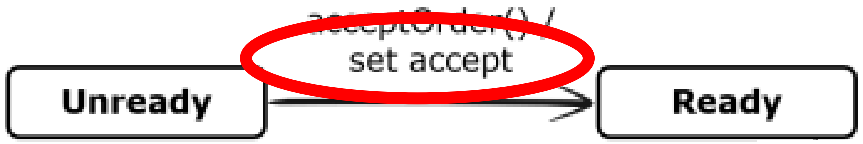
Checking a Transition

- Each transition is specified by an event/activity pair:
- The correct operation of a transition is verified by:
  - Check that the software is in the correct start state (this may have already been done by the verification of the previous transition)
  - Raise the event (method call)



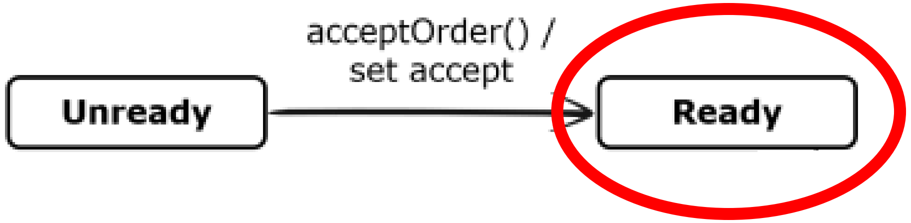
Checking a Transition

- Each transition is specified by an event/activity pair:
- The correct operation of a transition is verified by:
  - Check that the software is in the correct start state (this may have already been done by the verification of the previous transition)
  - Raise the event (method call)
  - Check that the specified activity has taken place correctly



Checking a Transition

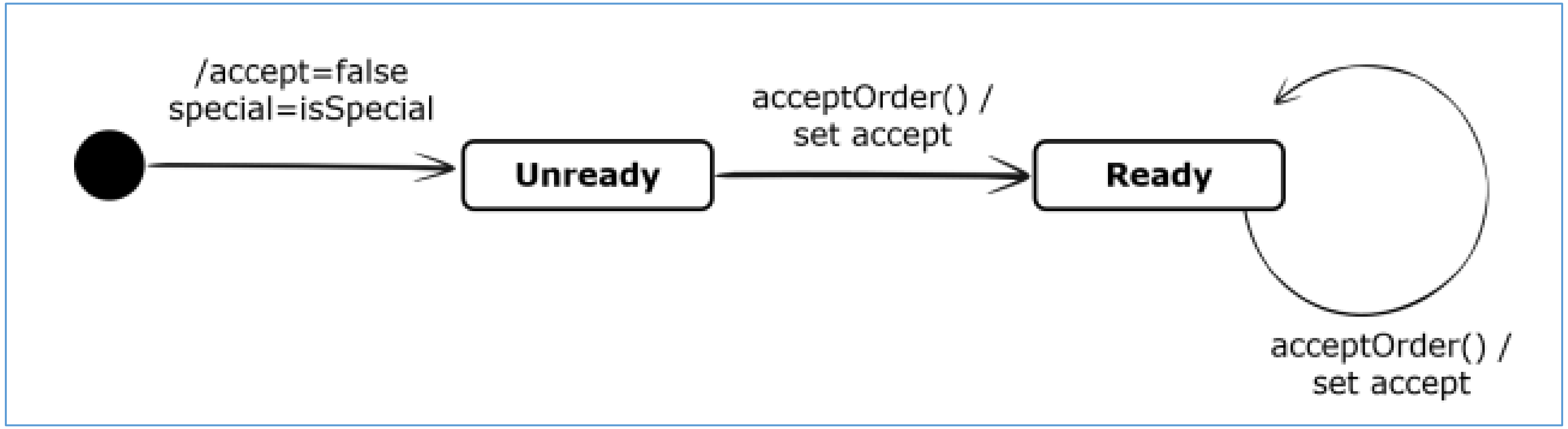
- Each transition is specified by an event/activity pair:
- The correct operation of a transition is verified by:
  - Check that the software is in the correct start state (this may have already been done by the verification of the previous transition)
  - Raise the event (method call)
  - Check that the specified activity has taken place correctly
  - Checking that the software is in the correct end state



Checking a Transition

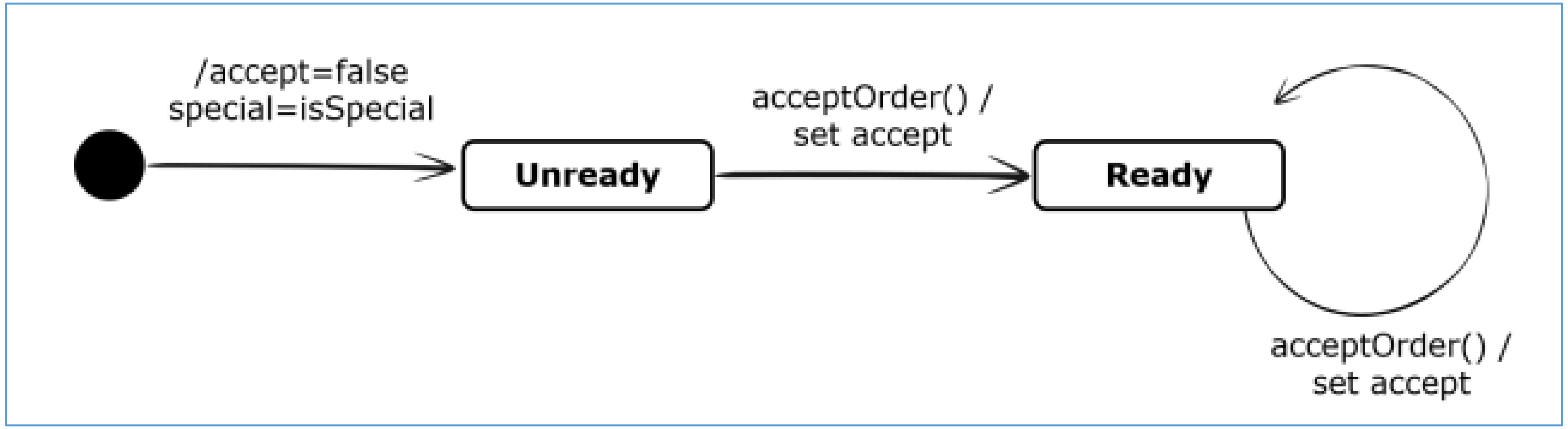
- Each transition is specified by an event/activity pair
- The correct operation of a transition is verified
- Not always possible to **fully** check that:
  - an object is in the correct state
  - or that the correct activity has taken place
- Design for Testability (DFT) issue
- May have to use a partial or “best-effort” check

Example: State Diagram for SpaceOrder



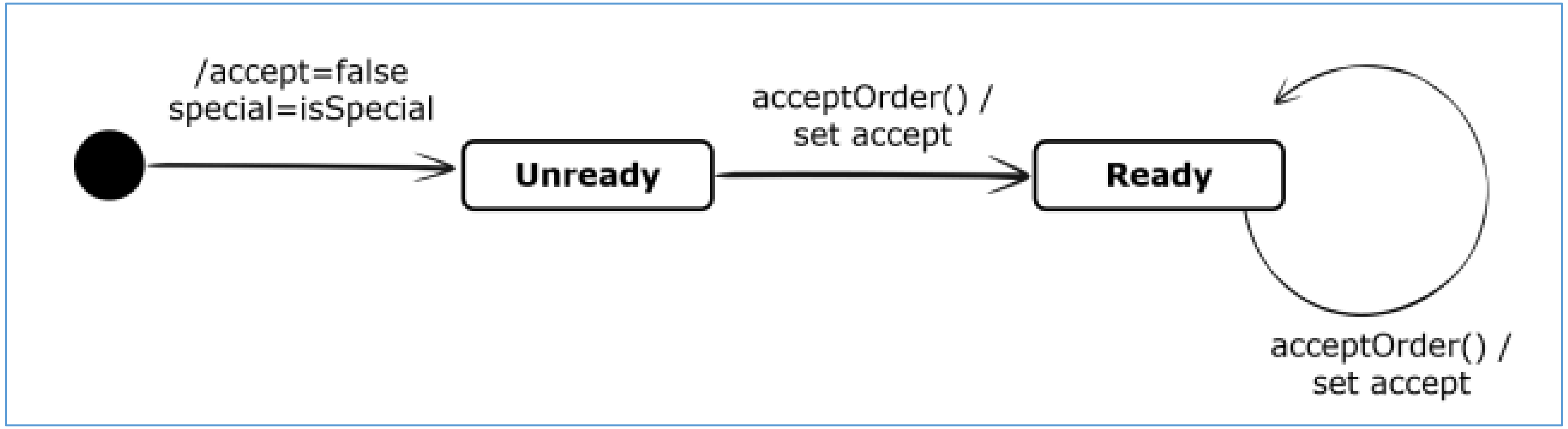
- Two states:
  - **Unready**
  - **Ready**

Example: State Diagram for SpaceOrder



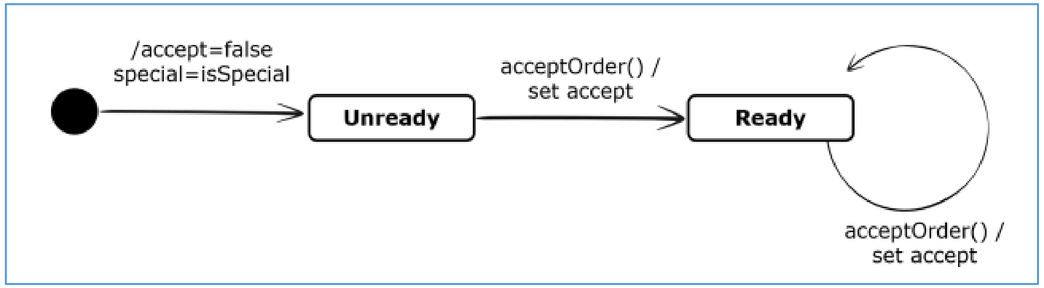
- Three (explicit) transitions:
  - **Constructor**
  - **acceptOrder**() in Unready state
  - **acceptOrder**() in Ready state

Example: State Diagram for SpaceOrder



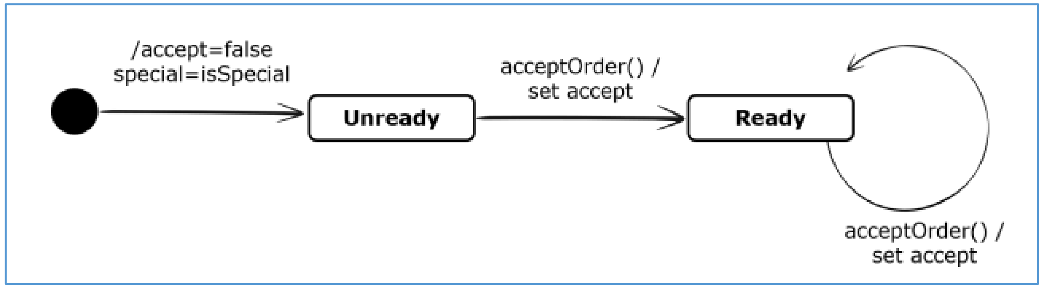
- Four (implicit) transitions – not shown on SD – should have no effect:
  - **getSpecial()** in Unready state
  - **setSpecial()** in Unready state
  - **getAccept**() in Unready state
  - **getAccept**() in Ready state

Example: Transitions for SpaceOrder



Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

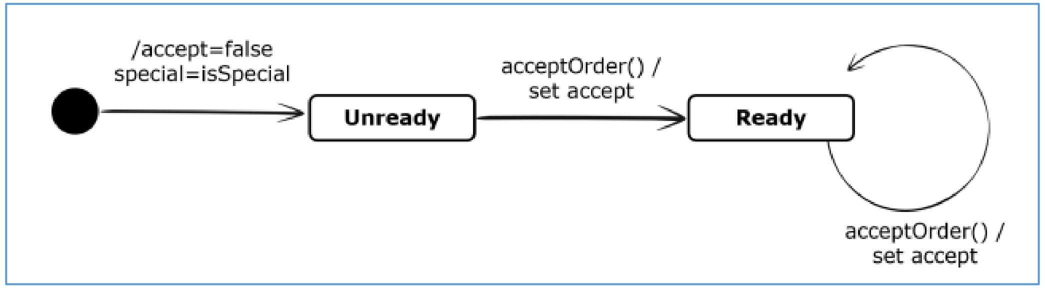
Example: Transitions for SpaceOrder



Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

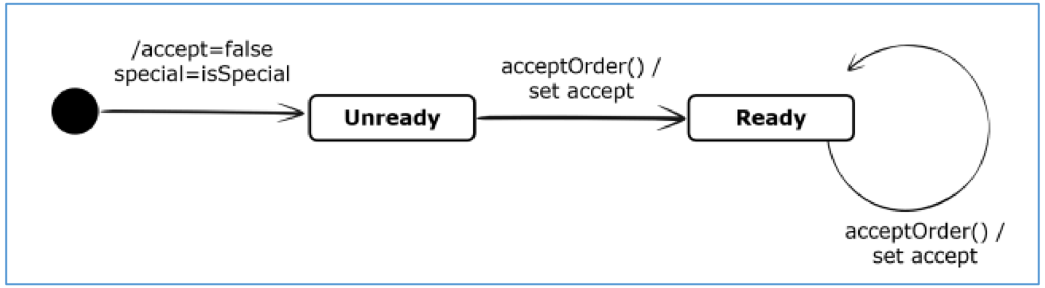
- Using the **all-transitions** strategy results in seven test coverage items: one for each transition

Example: Verifying State Transitions



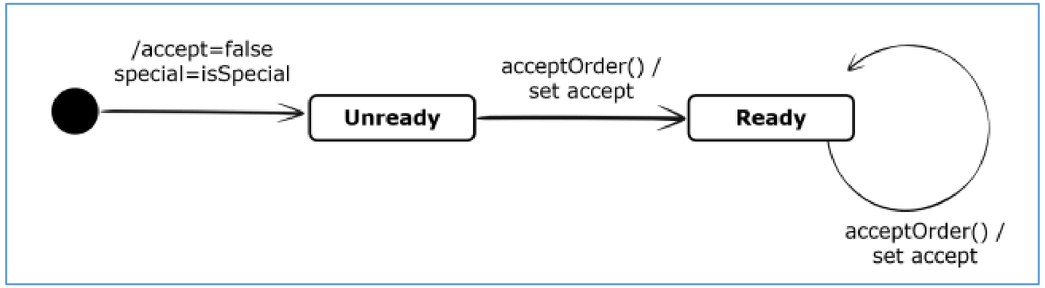
- There is no getState() method
- Have to check the state by using available methods

Example: Verifying State Transitions



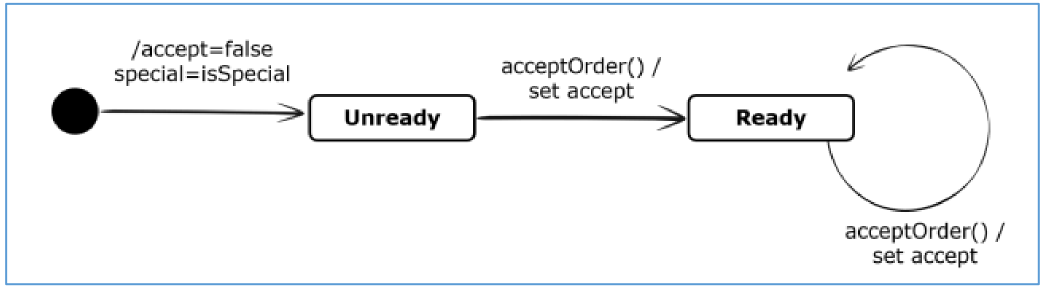
- There is no getState() method
- Have to check the state by using available methods
- Ideally use methods that don't change the state themselves
  - getSpecial()
  - getAccept()

Example: Verifying State Transitions



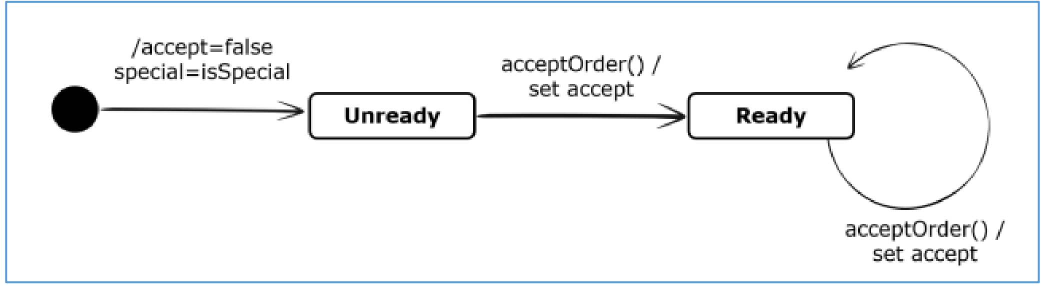
- There is no getState() method
- Have to check the state by using available methods
- Ideally use methods that don't change the state themselves
  - getSpecial()
  - getAccept()
- These are used to produce what is called a "**state signature**"
  - A sequence (or vector) of values returned by method calls that is unique to the state the object is in
  - If you call these methods, and get these return values, then the object was in the state associated with the state signature

Example: Verifying State Transitions



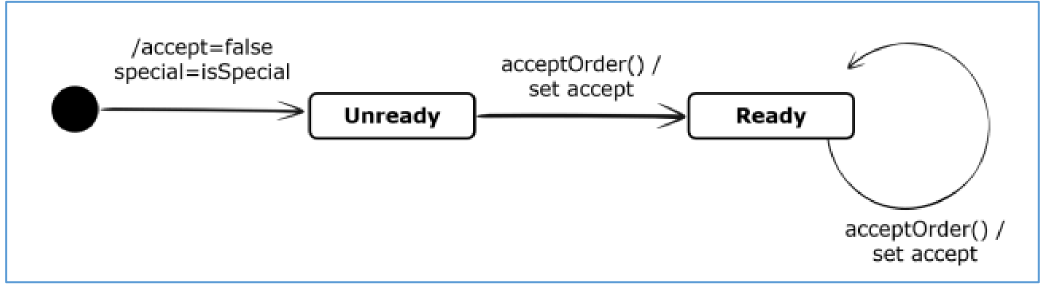
- You cannot fully verify that the software is in the **Unready** state: the best partial check is that getAccept() returns false, and getSpecial() returns the provided value of isSpecial

Example: Verifying State Transitions



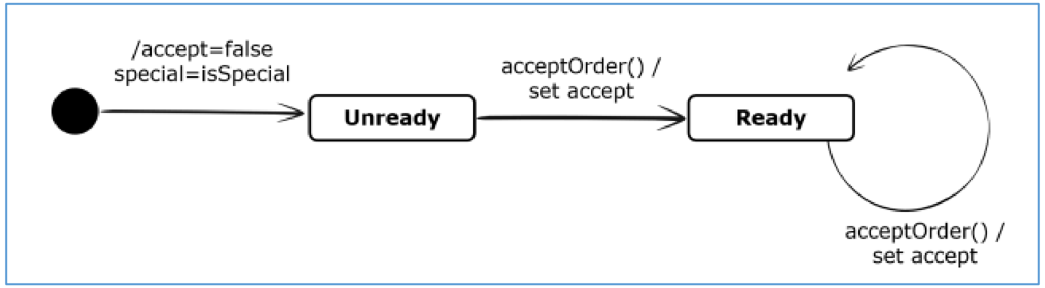
- You cannot fully verify that the software is in the **Unready** state: the best partial check is that getAccept() returns false, and getSpecial() returns the provided value of isSpecial
- You can verify that the software is in the **Ready** state if acceptOrder() has set accept to true by calling getAccept(), which should return true, and getSpecial() which should return the provided value of isSpecial

Example: Verifying State Transitions



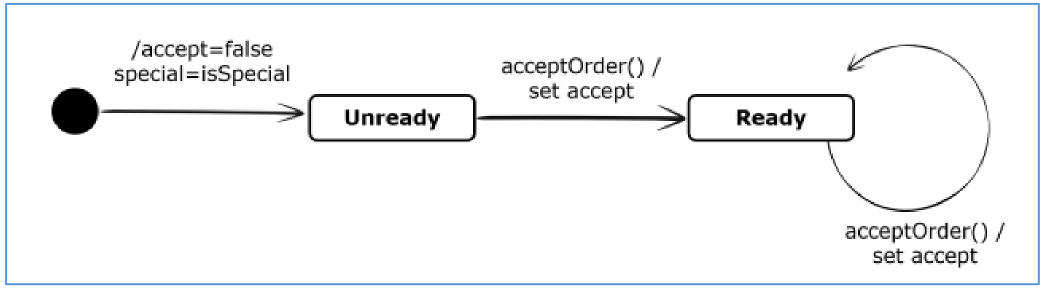
- You cannot fully verify that the software is in the **Unready** state: the best partial check is that getAccept() returns false, and getSpecial() returns the provided value of isSpecial
- You can verify that the software is in the **Ready** state if acceptOrder() has set accept to true by calling getAccept(), which should return true, and getSpecial() which should return the provided value of isSpecial
- You cannot verify that the software is in the **Ready** state if acceptOrder() has set accept to false
  - As in **Unready** state, getAccept() returns false
  - And getSpecial returns provided value

Continued…



- The transition from Unready to Ready should be checked by setting values for special and space that should set accept to true
- This allows both the Unready and Ready states to be uniquely identified (if the software is working correctly)

Continued…



- The transition from Unready to Ready should be checked by setting values for special and space that should set accept to true
- This allows both the Unready and Ready states to be uniquely identified (if the software is working correctly)
- The transition from Ready to Ready by calling acceptOrder() can only be (partially) verified by using a value for space that causes accept to be set to false

Implementation

```java
public class SpaceOrderStateTest {

  @Test
  public void allTransitionsTest() {
    // transition 1
    SpaceOrder o = new SpaceOrder(false);
```

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- First: cause T1

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- Then use the state signature for **Start** to check the state

```
public class SpaceOrderStateTest {

  @Test
  public void allTransitionsTest() {
    // transition 1
    SpaceOrder o = new SpaceOrder(false);
    // check activity and state for t1
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
```

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- The state signature has caused transitions T2 and T3
- So use the state signature again to check still in Unready

```
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
```

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- Now cause transition T4

```
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // transition 4
      o.acceptOrder(1000);
```

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- And check in state Ready using the state signature

```
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // transition 4
      o.acceptOrder(1000);
      // check activity and state for t4
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
```

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- The state signature has caused transitions T5 and T6
- So use the state signature again to check still in Ready

```
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // transition 4
      o.acceptOrder(1000);
      // check activity and state for t4
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // check activity and state for t5 and t6
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
```

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- Next, cause transition T7

```
public class SpaceOrderStateTest {

  @Test
  public void allTransitionsTest() {
    // transition 1
    SpaceOrder o = new SpaceOrder(false);
    // check activity and state for t1
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
    // check activity and state for t2 and t3
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
    // transition 4
    o.acceptOrder(1000);
    // check activity and state for t4
    assertFalse(o.getSpecial());
    assertTrue(o.getAccept());
    // check activity and state for t5 and t6
    assertFalse(o.getSpecial());
    assertTrue(o.getAccept());
    // transition 7
    o.acceptOrder(2000);
```

Implementation

- Tests for transitions T1, T2, T3, T4, T5, T6, T7 in order

Number	Start State	Event	End State
1	Start	Constructor	Unready
2	Unready	getSpecial()	Unready
3	Unready	getAccept()	Unready
4	Unready	acceptOrder()	Ready
5	Ready	getSpecial()	Ready
6	Ready	getAccept()	Ready
7	Ready	acceptOrder()	Ready

- And finally, use the state signature for T7 to check still in that state

```java
public class SpaceOrderStateTest {

  @Test
  public void allTransitionsTest() {
    // transition 1
    SpaceOrder o = new SpaceOrder(false);
    // check activity and state for t1
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
    // check activity and state for t2 and t3
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
    // transition 4
    o.acceptOrder(1000);
    // check activity and state for t4
    assertFalse(o.getSpecial());
    assertTrue(o.getAccept());
    // check activity and state for t5 and t6
    assertFalse(o.getSpecial());
    assertTrue(o.getAccept());
    // transition 7
    o.acceptOrder(2000);
    // check activity and state for t7
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
  }
```

Implementation

- Note that the checks to verify the software is in the correct state often cause transitions themselves, which must also be checked for in turn
- For example: checking the state after T1 causes transitions T2 and T3 to occur
- So check these next

```java
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // transition 4
      o.acceptOrder(1000);
      // check activity and state for t4
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // check activity and state for t5 and t6
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // transition 7
      o.acceptOrder(2000);
      // check activity and state for t7
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
   }

}
```

Implementation

- Note that the checks to verify the software is in the correct state often cause transitions themselves, which must also be checked for in turn
- For example: checking the state after T1 causes transitions T2 and T3 to occur
- So check these next

```
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // transition 4
      o.acceptOrder(1000);
      // check activity and state for t4
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // check activity and state for t5 and t6
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // transition 7
      o.acceptOrder(2000);
      // check activity and state for t7
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
   }

}
```

Implementation

- Note that the checks to verify the software is in the correct state often cause transitions themselves, which must also be checked for in turn
- This makes writing state-based tests quite challenging

```
public class SpaceOrderStateTest {

   @Test
   public void allTransitionsTest() {
      // transition 1
      SpaceOrder o = new SpaceOrder(false);
      // check activity and state for t1
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // check activity and state for t2 and t3
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
      // transition 4
      o.acceptOrder(1000);
      // check activity and state for t4
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // check activity and state for t5 and t6
      assertFalse(o.getSpecial());
      assertTrue(o.getAccept());
      // transition 7
      o.acceptOrder(2000);
      // check activity and state for t7
      assertFalse(o.getSpecial());
      assertFalse(o.getAccept());
   }

}
```

Order of Testing

- The order in which transitions are tested is important. It would be possible to put each transition test into a separate test method and use dependencies to force them to execute in the correct order
- Or use a complex test method and a data provider
- The other approach, as shown, is to test all the transitions in a single method
- This is much clearer and simpler to implement, but has the disadvantage that it is more difficult to debug a failed test, as the test covers multiple transitions

```java
public class SpaceOrderStateTest {

  @Test
  public void allTransitionsTest() {
    // transition 1
    SpaceOrder o = new SpaceOrder(false);
    // check activity and state for t1
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
    // check activity and state for t2 and t3
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
    // transition 4
    o.acceptOrder(1000);
    // check activity and state for t4
    assertFalse(o.getSpecial());
    assertTrue(o.getAccept());
    // check activity and state for t5 and t6
    assertFalse(o.getSpecial());
    assertTrue(o.getAccept());
    // transition 7
    o.acceptOrder(2000);
    // check activity and state for t7
    assertFalse(o.getSpecial());
    assertFalse(o.getAccept());
  }

}
```

All Transition Test Results

```
PASSED: allTransitionsTest
===============================================
Command line suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

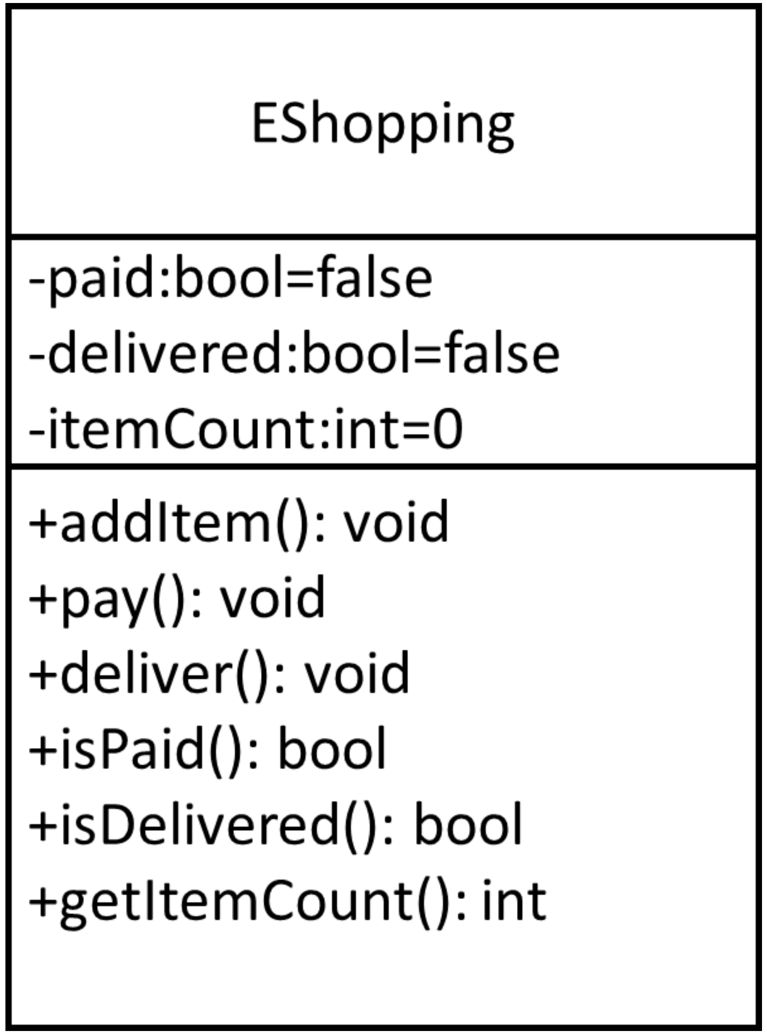
- All the tests have passed, showing as far as is possible that that each transition in the state diagram works
- Test effectiveness is limited by being unable to access the object state
- A tester can only test what is possible

This Afternoon’s Lab

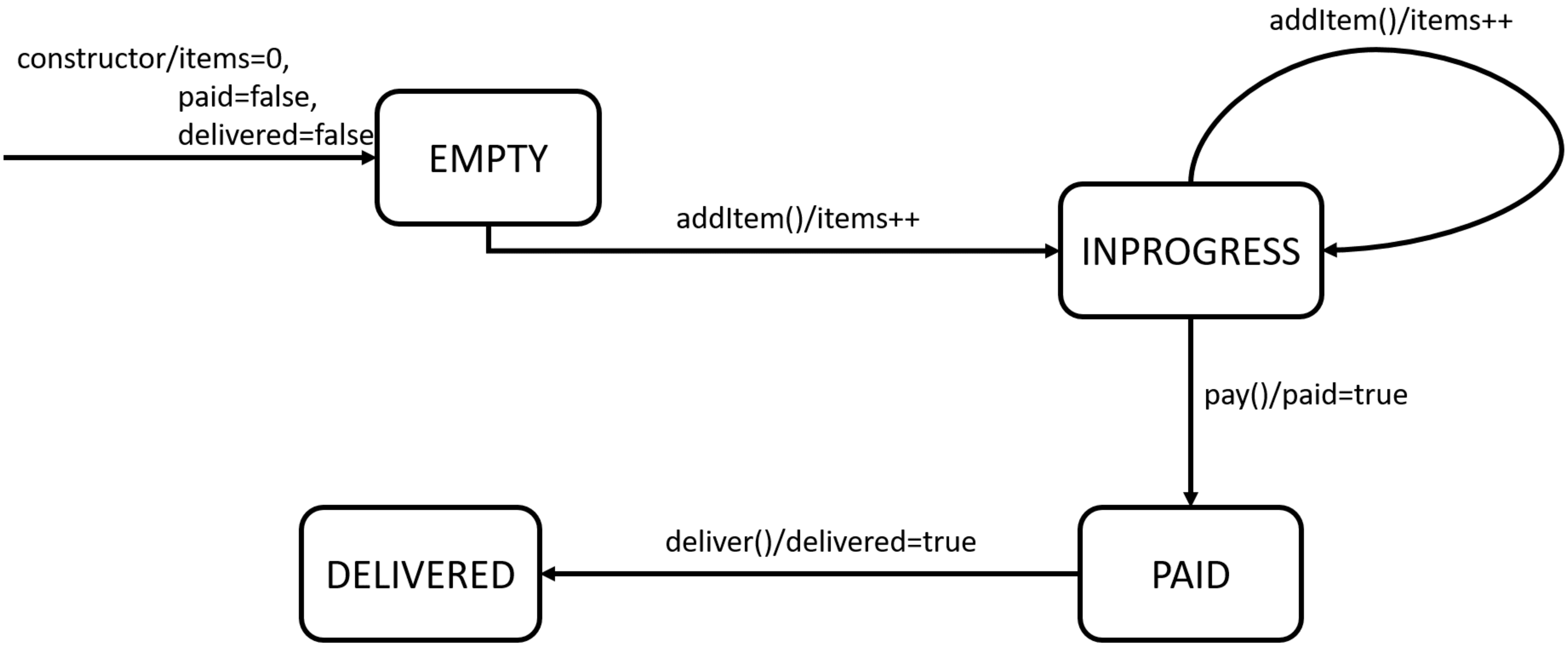
This Afternoon’s Lab

- State-Based Testing
- Class: **EShopping**
- Make sure to test each state transition at least once
- Quiz
- Next: examine the state diagrams for EShopping

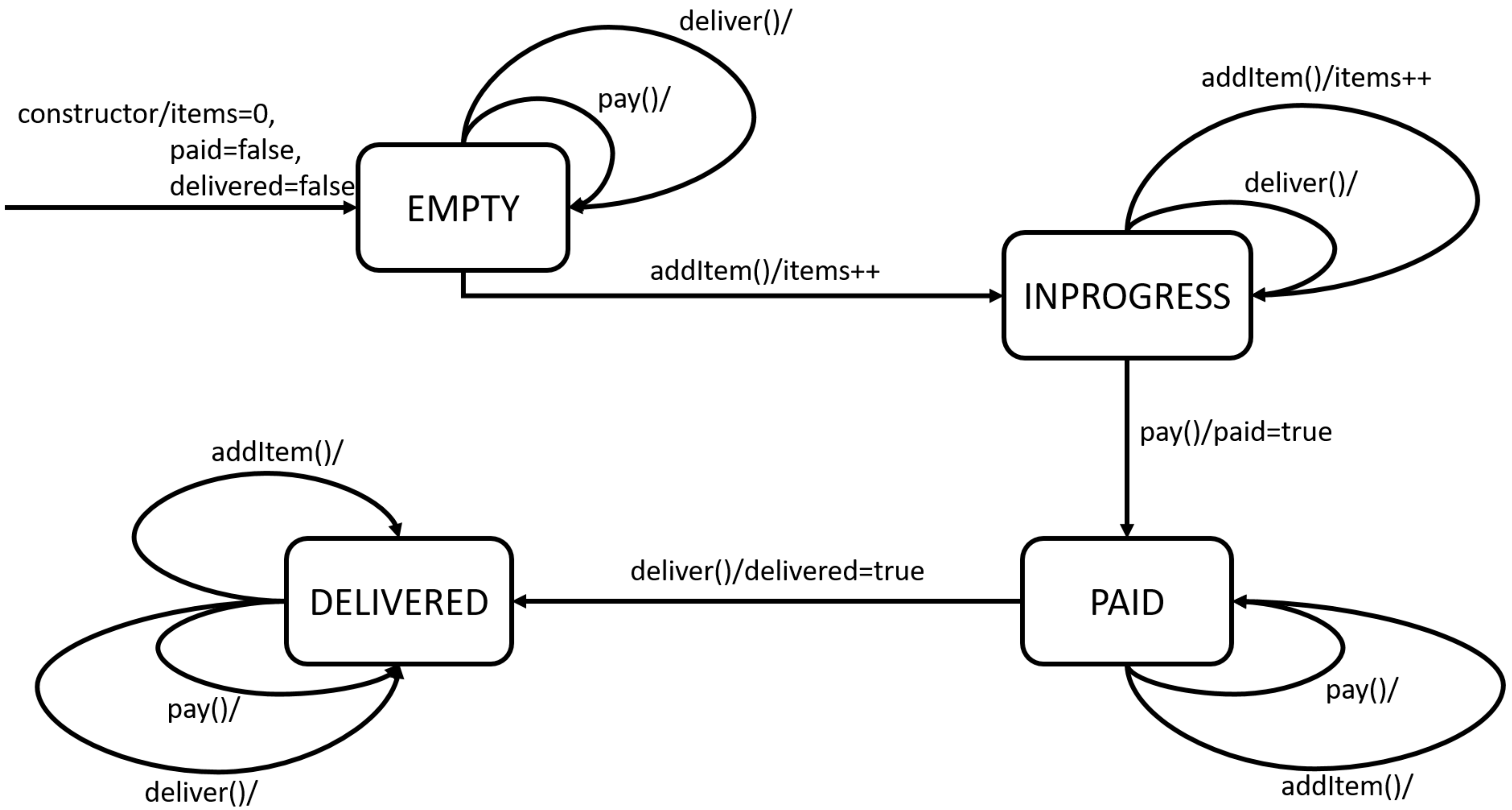
Class EShopping



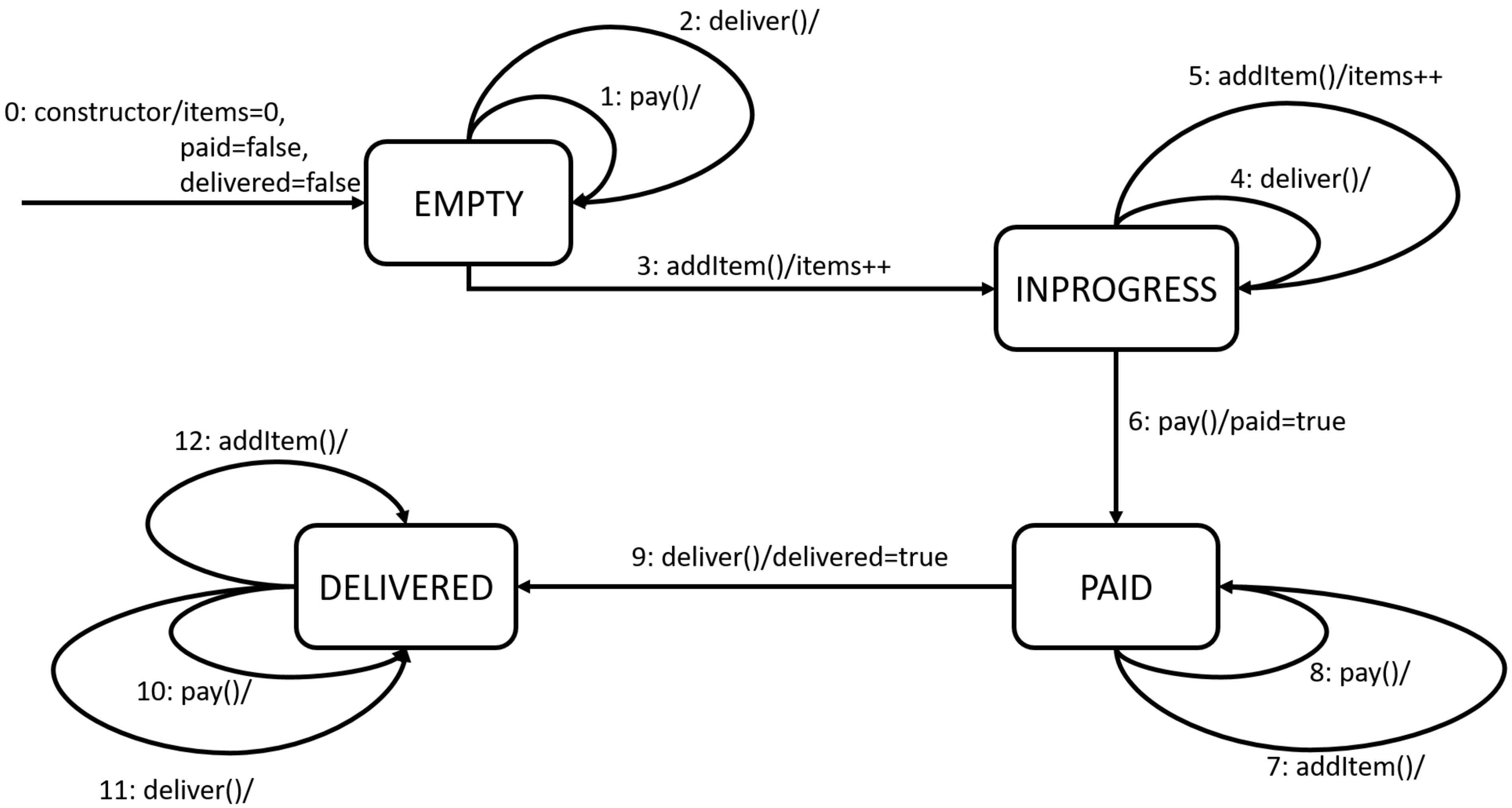
LAB7 UML State Diagram



Expanded SD (implicit transitions)



Numbered SD


2: deliver()/
1: pay()/
0: constructor/items=0,
paid=false,
delivered=false
EMPTY
5: addItem()/items++
4: deliver()/
3: addItem()/items++
INPROGRESS
6: pay()/paid=true
12: addItem()/
9: deliver()/delivered=true
DELIVERED
PAID
10: pay()/
8: pay()/
11: deliver()/
7: addItem()/