SE 552: Lecture 7

Overview

State dependence

Pessimistic and optimistic programming

Examples

Nested monitor problem

State dependence

Objects:

- have *internal state* (the current values of fields)
- respond to *external messages* (method calls)

Sometimes these may conflict:

Stack	*Empty*	*Full*	*Neither*
push	OK	??	OK
pop	??	OK	OK
size	OK	OK	OK

What should we do in the ?? cases?

Is this a problem for immutable classes?

State dependence

How do we try to avoid error states?

Example: avoiding gridlocks on traffic circles.

In the UK, traffic on the circle has right of way.

In France, traffic entering the circle has right of way.

How do these two systems deal with avoiding gridlock?

State dependence

Three possible ways to deal with failure:

- *Baulking*: throw an exception.
- *Guarding*: wait for another thread to change the object's state.
- *Timeout*: guard for some time, then baulk.

Two possible ways to detect failure:

- *Optimistically* (or *liberally*, or *try-and-see*): try calling the method and see if anything bad happens.
- *Pessimistically* (or *conservatively*, or *check-and-act*): make sure no errors can happen before proceeding.

Example: a stack

```
class Stack {
  ImmutableList contents = ImmutableList.factory.build ();
  public void push (final Object o) {
    contents = contents.cons (o);
  }
  public Object pop () {
    return contents.pop ();
  }
  public int size () {
    return contents.size ();
  }
}
```

What do we need to do to this class to make it thread-safe?

What needs done to the class to make it use baulking? guarding? timeout?

Is this pessimistic or optimistic?

Example: an optimistic stack

```
class Stack {
  ImmutableList contents = ImmutableList.factory.build ();
  public void push (final Object o) {
    final ImmutableList oldState = contents;
    final ImmutableList newState = oldState.cons (o);
    if (commit (oldState, newState)) { return; } else { error recovery }
  }
  public Object pop () { ... }
  public int size () { return contents.size (); }
  public boolean commit
    (final ImmutableList oldState, final ImmutableList newState) {
    ...
  }
}
```

What should the code for pop be?

What should the code for commit be?

Pattern for pessimistic programming

Using the *objects as state* pattern:

```
class Pessimistic {
  ImmutableState state = new ImmutableState ();
  final Object lock = new Object ();
  Result method (Argument arg) { synchronize (lock) {
    if (calling this method in the current state is ok...) {
      ...perform some computation...
      final ImmutableState newState = ...compute new state...;
      final Result result = ...compute result...;
      state = newState;
      return result;
    } else {
      error recovery
    }
  } }
}
```

What should the error recovery code be?

What if we replace ImmutableState by MutableState?

Pattern for optimistic programming

```
class Optimistic {
  ImmutableState state = new ImmutableState ();
  final Object lock = new Object ();
  Result method (Argument arg) {
    try {
      ...perform some computation...
      final ImmutableState newState = ...compute new state...;
      final Result result = ...compute result...;
      if (commit (oldState, newState)) { return result; } else { error recovery }
    } catch (final SomeException ex) {
      more error recovery
    }
  }
  boolean commit (final ImmutableState oldState, final ImmutableState newState) {
    some code goes here
  }
}
```

What should the error recovery code be?

What if we replace ImmutableState by MutableState?

Nested monitor problem

```
class Outer {
  protected final Object lock = new Object ();
  protected final Inner delegate = new Inner ();
  void waitForTrue () throws InterruptedException { synchronized (lock) {
    delegate.waitForTrue ();
  } }
  void setValue (boolean b) { synchronized (lock) {
    delegate.setValue (b);
  } }
}
class Inner {
  protected boolean value = false;
  protected final Object lock = new Object ();
  void waitForTrue () throws InterrupedException { synchronized (lock) {
    while (!value) { wait (); }
  } }
  void setValue (boolean b) { synchronized (lock) {
    value = b; lock.notifyAll ();
  } }
}
```

Nested monitor problem

What happens if:

```
final Outer outer = new Outer ();
Thread t1 = new Thread () { public void run () { outer.waitForTrue (); } }
Thread t2 = new Thread () { public void run () { outer.setValue (true); } }
t1.start (); t2.start (); t1.join ();
```

Oops... this is called the *nested monitor problem*. What can we do about it?

Summary

State causes headaches!

One headache is failure: not all operations may be supported in all states.

Possible ways to deal with failure are: baulking, guarding and timeout.

Possible ways to detect failure are: optimistic or pessimistic.

Be careful with combining guarding and baulking: the nested monitor problem awaits!

Next week: more of Chapter 3.