The Virtual Clock test pattern

Author: Paolo Perrotta
Version: Draft-1(request for comments)
Date: December 23, 2002
E-mail: pperrotta@inwind.it

We can have a hard time unit-testing our code when it depends on the system clock. This paper describes both the problem and a common, repeatable solution.

The problem

Let's say that we want to write a to-do list. We must be able to insert tasks into the list. A task can be anything – maybe something like "reply to a client message", or "do this computation". When we feel like it, we can ask the list to execute all the tasks. As an added complication, tasks that are older than a given age must expire. The list must ignore them and only execute "young" tasks.

Let's start by defining an interface for tasks:

```
public interface Task {
    public long getTimeOfBirth();
    public void execute();
}
```

Now let's write the to-do list:

```
import java.util.*;

public class ToDoList {

    public static final long MAX_TASK_AGE_MILLIS = 100;

    private List _tasks = new LinkedList();

    public void addTask(Task t) {
        _tasks.add(t);
    }
    public void processTasks() {
        Iterator i = _tasks.iterator();
        while(i.hasNext()) {
            Task next = (Task)i.next();
            long taskAge = System.currentTimeMillis() - next.getTimeOfBirth();
            if(taskAge <= MAX_TASK_AGE_MILLIS)
                next.execute();
        }
    }
}
```

We can insert *Tasks* into the list with *addTask()*. When we call *processTasks()*, the list goes through all the tasks, executes those that are younger than 0.1 seconds and just ignores the rest. (Note that this code smells of Feature Envy: the *Tasks* should probably calculate their own age themselves, instead of leaving this to the *ToDoList*. But this is just an example, so bear with me).

© 2002, Paolo Perrotta – All rights reserved

We just noticed that we should have written a test for the *ToDoList*. We haven't been good Test-First Designers, have we? Better late then never, so let's focus on the test now. We need something concrete to test with, so let's write a *Task* implementation first:

```
class MockTask implements Task {

    private long _timeOfBirth;
    private boolean _executed = false;

    MockTask(long timeOfBirth) {
        _timeOfBirth = timeOfBirth;
    }
    public long getTimeOfBirth() {
        return _timeOfBirth;
    }
    public void execute() {
        _executed = true;
    }
    boolean isExecuted() {
        return _executed;
    }
}
```

To be more flexible in our testing, we can also set the "time of birth" of each task. Our test will create some tasks with the desired birth dates and insert them into a *ToDoList*. Then it will check that "young" tasks are executed, and "old" ones are ignored.

But now we face a problem: to write a good test we must be sure that some tasks expire before we ask the list to process them, and some others don't. What would happen if the garbage collector or anything else slowed the system down for more than 0.1 seconds between "task insertion" and "task execution"? All our tasks would expire before we even get a chance to execute them! Unfortunately, we cannot control system time. Or, do we?

The solution

We need more control over time. We might write a controlled Virtual Clock and use it instead of the uncontrolled system clock. Let's implement this as a Singleton:

```
public class VirtualClock {

    private static VirtualClock _instance = new VirtualClock();
    private boolean _frozen = false;
    private long _time = 0;

    private VirtualClock() {}

    public static VirtualClock getInstance() {
      return _instance;
    }

    public void freeze() {
      _frozen = true;
      _time = System.currentTimeMillis();
    }

    public void restart() {
      _frozen = false;
    }

    public long getTime() {
        if(_frozen)
            return _time;
        return System.currentTimeMillis();
    }
}
```

Now we can modify *ToDoList* to read the Virtual Clock instead of the system clock:

```
public void processTasks() {
    Iterator i = _tasks.iterator();
```

© 2002, Paolo Perrotta – All rights reserved

```
    while(i.hasNext()) {
        Task next = (Task)i.next();
        long taskAge = VirtualClock.getInstance().getTime() - next.getTimeOfBirth();
        if(taskAge <= MAX_TASK_AGE_MILLIS)
            next.execute();
    }
}
```

That's all it took. We can write a decent test now:

```
import junit.framework.TestCase;

public class ToDoListTest extends TestCase {

    private VirtualClock _clk = VirtualClock.getInstance();

    public void setUp() {
        _clk.freeze();
    }

    public void testProcessTasks() {
        final long oldAge = _clk.getTime() - ToDoList.MAX_TASK_AGE_MILLIS;
        MockTask barelyYoungEnough = new MockTask(oldAge);
        MockTask alreadyTooOld = new MockTask(oldAge - 1);

        ToDoList list = new ToDoList();
        list.addTask(barelyYoungEnough);
        list.addTask(alreadyTooOld);
        list.processTasks();

        assertTrue(barelyYoungEnough.isExecuted());
        assertTrue(!alreadyTooOld.isExecuted());
    }

    public void tearDown() {
        _clk.restart();
    }

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(ToDoListTest.class);
    }
}
```

Behold the Green Bar!

It all boils down to…

When you write unit tests, you need lots of control. If a test requires that a certain object be set in a certain way, so be it. You must be able to set up the environment the way you want.

Some times this is easy: you set up a test fixture and call in your test.

Some times this is difficult: you might need to use techniques such as Mock Objects or specialized testing frameworks.

Some other times... well, you can't.

The system clock is the most common example. It's an important system input, but you can't control it. The Virtual Clock pattern gets around this problem by replacing the system clock with something that you can control:

Don't access the system clock directly. Instead, use a Virtual Clock - a controlled proxy for the system clock. You can switch to a "controlled time input" for testing, and then switch back to "real time input" in production code.

Implementation highs and lows

The Virtual Clock I wrote has two problems:

© 2002, Paolo Perrotta – All rights reserved

- *It's limited.* I bet it won't take long before your tests need even more control over the all-important "time" variable. When that happens, just implement a *setTime(long)* method. In my experience, *incTime(long)* can also be useful.
- *It's a Singleton.* Singletons can be evil (http://c2.com/cgi/wiki?SingletonsAreEvil). If you have troubles sharing your Virtual Clock amongst different clients, you might go for a different and more object-oriented approach. *VirtualClock* might become an abstract class with two concrete subclasses: a *RealTimeClock* and a *ControlledClock*. Then you might pass the desired concrete clock to whomever needs it. In our example, we would pass a *RealTimeClock* to the *ToDoList* in our production code, but we still use a *ControlledClock* for testing. You could safely implement *RealTimeClock* as a Singleton, since it's a read-only object.

More ways to use it

The Virtual Clock pattern allows you to decouple "time as an input" from "real time". This can be useful for other things besides testing. The most obvious example: you might want to simulate the system over a long time span. Or maybe you might want to trick an algorythm that processes hystorical data into believing that it's running over "some other period".

I also had this idea of freezing the clock just to read it multiple times. I thought that this would be useful when (1) my code was reading the clock so often that it could slow down the system, or (2) my code was assuming constant time over a number of instructions. Thinking about it, this was a terrible case of "too much pattern thinking". The first situation would probably mean that my code sucks and needs some refactoring. As for the second situation, it can be dealt with by using a simple temporary variable.

The dark side

Be wary of two possible Virtual Clock bug patterns:

Frozen time bug: You stop the clock, then you forget to restart it. This happened to me more often that I'd be happy to admit (hint to self: double-check your *setUp()* and *tearDown()* methods). Luckily, this bug is easy to detect and fix.

Shared clock bugs: If you implement the Virtual Clock as a Singleton, you'll be potentially exposed to the usual shared resource problems. See above for a solution.

Known uses

I used the Virtual Clock on more or less all my projects during the latest two years. I barely ever access the system clock anymore.

Martin Fowler mentioned that he always uses indirection on the system clock (*http://martinfowler.com/ap2/timePoint.html*) *(draft note: check whether Martin's new book includes the pattern)*.

Real-time coders routinely simulate time *(draft note: is it really as common as I think?)*.

John Carmack from id Software reportedly uses the technique to test his Quake 2 game engine. (draft note: Unfortunately, I can't find that document on the Internet anymore).

Future directions

This pattern might be extended to any volatile input of the system besides time. If you have any experiences of applying a similar pattern to some other kind of system input, please drop me a line.

Kent Beck suggested that this pattern might complement those described by Michael J. Pont into his "Patterns for Time-Triggered Embedded Systems" (Addison-Wesley, 2002). I didn't read the book yet. *(draft note: read it, or get more info – should I quote the book in "Known uses"?)*

Bibliography

(draft note: write this)

© 2002, Paolo Perrotta – All rights reserved