SOFTENG 251 2002 – Digital Simulator

© Rick Mugridge

September 17, 2002

Introduction

This example shows the development of a simulation system that is event-driven. It shows how to develop such systems using Test Driven Development (TDD)[1], and how to build suitable abstractions and modularity. It shows how interfaces, subtypes and inheritance are introduced as a part of the development process, and hence illustrates *design in action*. It incorporates a general approach to simulation, a useful technique for understanding and analysing complex systems.

A digital circuit[2] consists of one or more *gates*, such as *inverters* and *and-gates*, connected together with *wires*. A wire carries a digital signal, either *0* or *1*. Gates compute their output signal based on their inputs, after some delay. Such circuits can be used to create abstract gates, such as for binary addition. This can be done by composing (wiring together) elementary (and other abstract) gates.

An inverter, an And-gate and an Or-gate are shown in Fig. 1. A half adder circuit is shown in Fig. 2 and a full adder is shown in Fig. 3.

Step 1. Getting Started (01)

We can set a major goal in our to-do list, and break it down somewhat:

> TO DO:
> Build a digital circuit simulator
> • Simulate the half-adder circuit shown above

It's often far from easy to decide on the best way to begin, as you may be just starting to really understand what is required and there may be lots of issues which all seem to interact. So don't bite off too much at once. If in doubt, make the first (and subsequent) steps as simple as possible. If the approach is not working, consider another way.

In this case, we have to deal with wires, several types of gates, delays and propagation of values. A good place to start is with the parts that seem more straightforward; for example, dealing with things that are easily modelled as objects, such as wires and gates. How to manage *time* is less clear, although we could use the simulation ideas from `Bounce`.

[1] *Test Driven Development: By Example*, Kent Beck, Addison-Wesley November 2002.
[2] *Structure and Interpretation of Computer Programs*, Abelson, Sussman and Sussman, MIT Press, 1996.

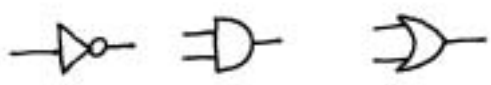

Figure 1: Inverter, And, Or Gates

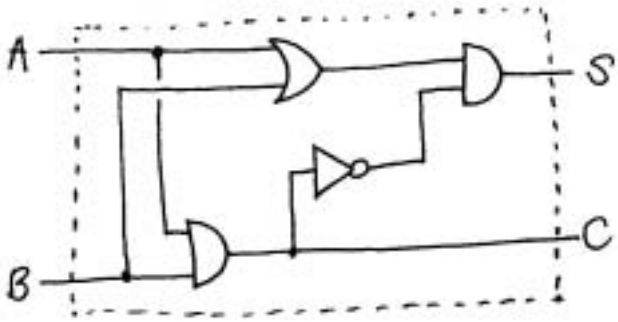

Figure 2: Half Adder

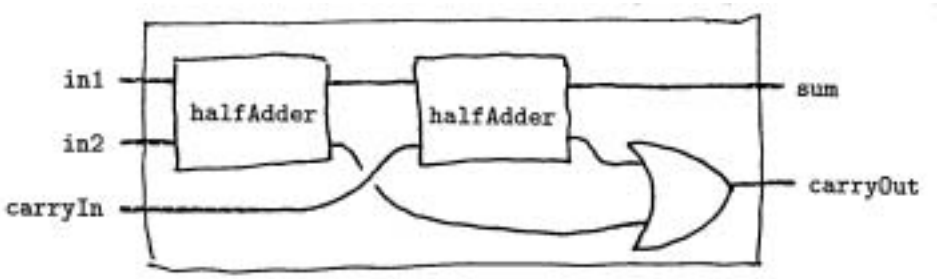

Figure 3: Full Adder

We begin with a simplified *inverter* gate, and *wires* by writing the first test; it ignores delays for now. There's no need to worry about the other types of gates; we can add them in later. That expands our stack of things to do:

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
• • Simulate an inverter
• • Simulate an inverter, ignoring delays
• • • Simulate an inverter with no signal change

In writing the test, we start making design decisions, such as the names of the classes we'll use and their initial interfaces (public methods, etc). The first test introduces a class `Wire`, with a notion of a boolean signal (`getSignal()` and `setSignal()`), and a class `Inverter`, which takes two `Wires` as arguments to the constructor. The interfaces of these classes are sure to change as we proceed, but the names seem reasonable.

TestSimulator.java (01)

```
public class TestSimulator extends TestCase {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestSimulator.class);
    }
    public TestSimulator(String name) {
        super(name);
    }
    public void testInvert1() {
        Wire in = new Wire();
        in.setSignal(true);
        Wire out = new Wire();
        Inverter invert = new Inverter(in,out);
        assertTrue(!out.getSignal());
    }
}
```

We kept it simple by representing a signal value as a boolean.

It took some thinking to write this first test, as I was making several decisions: how to make the step small enough, what to focus on first, what the class names should be and what their interfaces should be initially.

Inverter.java (01)

```
/** An inverter is a boolean NOT, eventually with some delay.
    The simplest thing possible to meet the tests.
    @author rick mugridge, july 2002
 */
public class Inverter {
    public Inverter(Wire in, Wire out) {
    }
}
```

We do the simplest thing that will work. Why? Because we don't want to guess about the future – we wait until a test drives us to write the code. This may seem trivial here, but it's a powerful technique when things are more complex or unclear. The best way to prepare for *then* is to use the technique all of the time *now*.

You should end up with better code – as long as you refactor so that the design of your code is always explicit and of high quality. Surprisingly, you'll also go faster over all, because you'll spend less time debugging and fixing errors.

Wire.java (01)

```
/** A Wire passes signals between devices.
    The simplest thing possible to meet the tests.
    @author rick mugridge, july 2002
 */
public class Wire {
    public void setSignal(boolean signal) {
    }
    public boolean getSignal() {
        return false;
    }
}
```

Our first test now passes with a green bar!

Notice that we can pass the test quickly by returning `false` from the method `getSignal()`. We will generalise the code as our tests drive the development.

Also notice that I don't have any JavaDoc comments for the two methods here. That's because they follow a standard Java approach, of a getter/setter pair for a *property* called `signal`. This notion comes from JavaBeans[3] (which will be covered at some stage).

Step 2: Invert 0 (02)

```
    public void testInvert0() {
        Wire in = new Wire();
        Wire out = new Wire();
        in.setSignal(false);
        Inverter invert = new Inverter(in,out);
        assertTrue(out.getSignal());
    }
```

This step drives the generalisation of `Invert`, by adding a test case in `TestSimulator.java` with an input value of 0.

Why not do this as a part of the first step? We had enough to think about in getting started, working out the test, the new classes, etc. And it was great to get some quick positive feedback with a green bar, before moving on to this step.

Inverter.java (02)

```
public class Inverter {
    public Inverter(Wire in, Wire out) {
        out.setSignal(!in.getSignal());
    }
}
```

Again, we do no more than necessary to pass the tests (because we want the tests to drive the development).

[3] And before that, Delphi

Wire.java (02)

```
public class Wire {
    private boolean signal = false;

    public void setSignal(boolean signal) {
        this.signal = signal;
    }
    public boolean getSignal() {
        return signal;
    }
}
```

A green bar again, with two successes! So we can tick off the last item on the to-do list:

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
• • Simulate an inverter
• • Simulate an inverter, ignoring delays
⋆ ⋆ ⋆ √ Simulate an inverter with no signal change

Step 3: Refactor the tests (03)

```
public class TestSimulator extends TestCase {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestSimulator.class);
    }
    private Wire in, out;

    public TestSimulator(String name) {
        super(name);
    }
    public void setUp() {
        in = new Wire();
        out = new Wire();
    }
    public void testInvert1() {
        in.setSignal(true);
        Inverter invert = new Inverter(in,out);
        assertTrue(!out.getSignal());
    }
    public void testInvert0() {
        in.setSignal(false);
        Inverter invert = new Inverter(in,out);
        assertTrue(out.getSignal());
    }
}
```

The tests contained some redundancy, so we used `setUp()` to initialise common values (*Extract Method*). The tests are just as important as the other code, so we want them to be clean and clear.

The tests pass. Why did we not initialise the `Inverter` in `setUp()` too?

Step 4: Test Two Input Signal Changes (04)

We want to push the refinement of `Inverter`; it is currently limited because it only changes its output wire when it is first created. So now we make an input signal change after we have wired it up.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
• • Simulate an inverter
• • Simulate an inverter, ignoring delays
⋆ ⋆ ⋆ √ Simulate an inverter with no signal change
• • • Simulate an inverter with signal changes

```
public class TestSimulator extends TestCase {
...
    public void testInvert1Then0() {
        in.setSignal(true);
        Inverter invert = new Inverter(in,out);
        assertTrue(!out.getSignal());
        in.setSignal(false);
        assertTrue(out.getSignal());
    }
}
```

You'll get a good nose for picking such steps after a while. It's good to build skill in this, driving your understanding by refining examples (tests) to push the current "theory". It's related to a general testing skill – looking for interesting edge cases.

Inverter.java (04)

```
public class Inverter implements ChangeListener {
    private Wire in, out;

    public Inverter(Wire in, Wire out) {
        this.in = in;
        this.out = out;
        in.addChangeListener(this);
    }
    public void stateChanged(ChangeEvent ev) {
        out.setSignal(!in.getSignal());
    }
}
```

An `Inverter` has to respond to a change to its input wire at any time.

We used an *event-driven* approach here, which allows a signal to propagate through the wires and gates of a circuit. Notice how we minimise the coupling between the gate and the wire here by having an anonymous object register for notification when the signal value changes; this gives us good modularity. We follow the Java naming convention of methods for registering listeners, which keeps it familiar and therefore simple. We use an existing Java listener, as it fits our purpose well.

There's a small smell here – an `Inverter` is a `ChangeListener` for purely local reasons; this Java `interface` and the method `stateChanged()` shouldn't really be a part of its public interface. We expect that only `Wire` objects will call `stateChanged()`.

The lastest test fails; that requires a change to `Wire`, as covered in Step 5.

Step 5: TestWire

As the change to `Wire` to pass the last test is non-trivial, it makes sense to develop it test-first.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
• • Simulate an inverter
• • Simulate an inverter, ignoring delays
⋆ ⋆ ⋆ √ Simulate an inverter with no signal change
• • • Simulate an inverter with signal changes
• • • • Wire fires an event on a signal change

This means a delay in satisfying the test added in Step 4 while we write `TestWire`. We could either comment out that previous test (and comment out the corresponding change to `Inverter`), until we're ready to satisfy it, or we could just leave it hanging, as a reminder that we need to get back to it.

> This often occurs during test-first development: you write some code that needs some more detailed code elsewhere. I prefer to develop code inwards, which often means having some tests unsatisfied while you get another part working. I'm happy if a small number of tests are failing while I work on another part. Kent Beck prefers to comment out those tests so that he gets a green bar at each step; he uses the to-do list to keep track of what he's got to do[4].
>
> I didn't include this intermediate step initially, because I expected to succeed with all of this in one step – I have introduced such listeners many times before. But there's no point in taking bigger steps if you're not sure what you're doing and don't expect to succeed. Experiment with the size of your steps and see how small you can make them if you need to.

[4] *Test Driven Development: By Example*, Kent Beck, Addison-Wesley November 2002.

TestWire.java (05)

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

    public TestWire(String name) {
        super(name);
    }
    int calls = 0;

    public void testEventFired() {
        Wire wire = new Wire();
        wire.addChangeListener(new ChangeListener() {
                public void stateChanged(ChangeEvent ev) {
                    calls++;
                }
            });
        assertEquals(1,calls);
        wire.setSignal(true);
        assertEquals(2,calls);
        wire.setSignal(true); // Should have no affect
        assertEquals(2,calls);
    }
}
```

Notice how we embed the `ChangeListener` within the `TestCase`. If we had more than one test, we'd need to use `setUp()` to initialise the instance variable `calls` to zero.

The `Wire` fires an event as soon as a listener is registered with it, so that the listener can update its state according to the current state of the wire.

Wire.java (05)

```
public class Wire {
    private boolean signal = false;
    private List listeners = new ArrayList();

    public void setSignal(boolean signal) {
        boolean changed = this.signal != signal;
        this.signal = signal;
        if (changed)
            fireEvent();
    }
    protected void fireEvent() {
        ChangeEvent event = new ChangeEvent(this);
        for (Iterator it = listeners.iterator(); it.hasNext(); )
            ((ChangeListener)it.next()).stateChanged(event);
    }
    public boolean getSignal() {
        return signal;
    }
    public void addChangeListener(ChangeListener cl) {
        listeners.add(cl);
        cl.stateChanged(new ChangeEvent(this));
    }
    public void removeChangeListener(ChangeListener cl) {
        listeners.remove(cl);
    }
}
```

As a `Wire` fires `Changeevents`, we followed the Java tradition of providing the methods `addXXListener()` and `removeXXListener()` for a listener named `XXListener` with corresponding event `XXEvent`.

Now that the test in `TestWire` works, we can check that all the tests in `TestSimulator` now work; they do. With running the two tests, we have two green bars with 3 and 1 successes.

TestAll.java (05)

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

    public static Test suite() {
        TestSuite suite = new TestSuite("All Tests");
        suite.addTestSuite(TestWire.class);
        suite.addTestSuite(TestSimulator.class);
        return suite;
    }
}
```

As we've now got more than one test file, it will be handy to run them both at once. So we introduced a class `TestAll`:

So running that, we now have a green bars with 4 successes.

Step 6. Modelling Time

So we can now simulate an inverter, ignoring time delays. Next, let's handle a sequence of changes to the inputs of a gate. So we need to capture that history in some way, to check that things occurred in the correct order.

To make the first step towards that, we introduce a `Probe`, that can be placed on a `Wire` to record changes. Initially, it just reports the current state of the wire. That's an easy step.

We'll change our tests to use a `Probe`.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
•• Simulate an inverter
⋆⋆ √ Simulate an inverter, ignoring delays
⋆⋆⋆√ Simulate an inverter with no signal change
⋆⋆⋆√ Simulate an inverter with signal changes
⋆⋆⋆⋆ √ Wire fires an event on a signal change
•• Simulate an inverter with explicit time delays
• • • Introduce a current-state Probe
• • • Probe records history

TestSimulator.java (06)

```
public class TestSimulator extends TestCase {
    ...
    private Wire in, out;
    private Probe probe;

    public void setUp() {
        in = new Wire();
        out = new Wire();
        probe = new Probe(out);
    }
    ...
    public void testInvert1Then0() {
        in.setSignal(true);
        Inverter invert = new Inverter(in,out);
        assertTrue(!probe.getSignal());
        in.setSignal(false);
        assertTrue(probe.getSignal());
    }
}
```

The other tests are changed similarly.

Probe.java (06)

```
/** A probe listens to the input wire, to record its state.
    At this stage, it can just refer to the wire for its state.
    @author rick mugridge, july 2002
 */
public class Probe {
    private Wire in;

    public Probe(Wire in) {
        this.in = in;
    }
    public boolean getSignal() {
        return in.getSignal();
    }
}
```

Back to a green bar again, with 4 successes.

Step 7: Probe Log in TestSimulator.java

We now want to record the history of the wire that the probe is on. We use a standard trick: record the history in a String[5].

We also need to be clear about the state of a wire initially, so we add another test. This prompts me to create the `Inverter` in the `setUp()` too. (Note that the order of wiring things up impacts on the `Probe` result.)

```
public class TestSimulator extends TestCase {
    private Wire in, out;
    private Probe probe;
    private Inverter invert;

    public void setUp() {
        in = new Wire();
        out = new Wire();
        invert = new Inverter(in,out);
        probe = new Probe(out);
    }
    public void testInvertInitial() {
        assertEquals("(true)",probe.getLog());
    }
    public void testInvert0() {
        in.setSignal(false);
        assertEquals("(true)",probe.getLog());
    }
    public void testInvert1() {
        in.setSignal(true);
        assertEquals("(true)(false)",probe.getLog());
    }
    public void testInvert1Then0() {
        in.setSignal(true);
        in.setSignal(false);
        assertEquals("(true)(false)(true)",probe.getLog());
    }
}
```

[5] As we did with `Bounce`

Probe.java (07)

Add a log capability to the probe.

```
public class Probe implements ChangeListener {
    private Wire in;
    private String log = "";

    public Probe(Wire in) {
        this.in = in;
        in.addChangeListener(this);
    }
    public String getLog() {
        return log;
    }
    public void stateChanged(ChangeEvent ev) {
        log += "("+in.getSignal()+")";
    }
}
```

A green bar with 5 successes. We've making progress.

Step 8: Delays

It's not obvious how to proceed, so we look for a small step[6]. We need to add new tests to drive the development in small steps, without having to guess what's required.

Let's now incorporate delays in an inverted gate. We assume that an inverter has a delay of 4. A wire and a probe have no delay. This requires little changes to most of the classes.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
•• Simulate an inverter
⋆⋆ √ Simulate an inverter, ignoring delays
•• Simulate an inverter with explicit time delays
⋆⋆⋆√ Introduce a current-state Probe
⋆⋆⋆√ Probe records history
• • • Probe records time delays in history

[6] I actually tried a way from here that didn't drive the development so well, so I backtracked. I started on an agenda for the simulator, before knowing exactly what was required of it. The agenda is now developed in Step 12!

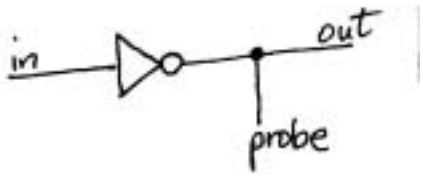

Figure 4: Inverter Circuit

```
public class TestSimulator extends TestCase {
    ...
    public void setUp() {
        in = new Wire();
        out = new Wire();
        probe = new Probe(out);
        invert = new Inverter(in,out);
    }
    public void testInvertInitial() {
        assertEquals("(0,false)(4,true)",probe.log());
    }
    public void testInvert0() {
        in.setSignal(0,false);
        assertEquals("(0,false)(4,true)",probe.log());
    }
    public void testInvert1() { // Invert delay is 4
        in.setSignal(5,true);
        assertEquals("(0,false)(4,true)(9,false)",probe.log());
    }
    public void testInvert1Then0() { // Invert delay is 4
        in.setSignal(5,true);
        in.setSignal(13,false);
        assertEquals("(0,false)(4,true)(9,false)(17,true)",probe.log());
    }
```

The inverter used in `TestSimulator` is shown in Fig. 4.

This involves an important design decision – to pass the time into `setSignal()`[7]. Now a signal is changed at a particular (simulation) time. A probe log now includes the time that the signal state changed.

It's clearer to put the probe on the `out` wire before wiring in the inverter.

The timing diagram for `testInvert1Then0` is shown in Fig. 5.

[7] In retrospect, this decision made it easier to understand the program later.

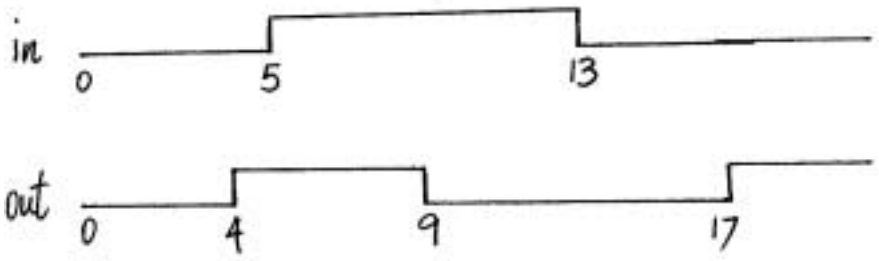

Figure 5: Inverter Timing

Wire.java (08)

```
public class Wire {
    private boolean signal = false;
    private List listeners = new ArrayList();

    public void setSignal(int time, boolean signal) {
        boolean changed = this.signal != signal;
        this.signal = signal;
        if (changed)
            fireEvent(time);
    }
    protected void fireEvent(int time) {
        SignalChangeEvent event = new SignalChangeEvent(this,time);
        for (Iterator it = listeners.iterator(); it.hasNext(); )
            ((SignalChangeListener)it.next()).signalChanged(event);
    }
    public boolean getSignal() {
        return signal;
    }
    /** Assumes all wiring is made at time 0 */
    public void addSignalChangeListener(SignalChangeListener scl) {
        listeners.add(scl);
        scl.signalChanged(new SignalChangeEvent(this,0));
    }
    public void removeSignalChangeListener(SignalChangeListener scl) {
        listeners.remove(scl);
    }
}
```

The method `setSignal()` is changed to include the time parameter. Because we want the simulation time passed when changes are propagated, we introduce a new event, a `SignalChangeEvent`, with associated listener.

SignalChange (08)

```
/** Provides timing information when a signal is changed.
    @author rick mugridge july 2002
 */
public class SignalChangeEvent extends java.util.EventObject {
    private int time = 0;
    public SignalChangeEvent(Object source, int time) {
        super(source);
        this.time = time;
    }
    public int getTime() {
        return time;
    }
}

public interface SignalChangeListener extends java.util.EventListener {
    public void signalChanged(SignalChangeEvent ev);
}
```

Inverter.java (08)

An `Inverter` now has a delay of 4.

```
public class Inverter implements SignalChangeListener {
    private static int DELAY = 4;
    private Wire in, out;

    public Inverter(Wire in, Wire out) {
        this.in = in;
        this.out = out;
        in.addSignalChangeListener(this);
    }
    public void signalChanged(SignalChangeEvent ev) {
        out.setSignal(ev.getTime()+DELAY,!in.getSignal());
    }
}
```

Probe.java (08)

```
/** A probe listens to the input wire, to record its state.
    It records a timed-based log of signal changes.
    @author rick mugridge, july 2002
 */
public class Probe implements SignalChangeListener {
    private Wire in;
    private String log = "";

    public Probe(Wire in) {
        this.in = in;
        in.addSignalChangeListener(this);
    }
    public String log() {
        return log;
    }
    public void signalChanged(SignalChangeEvent ev) {
        log += "("+ev.getTime()+","+in.getSignal()+")";
    }
}
```

A green bar again, with five successes.

That step introduced a lot of changes, due to the time being passed when a signal change was propagated. Such "plumbing" or "wiring" changes to the code are common when evolving programs (regardless of the programming language used).

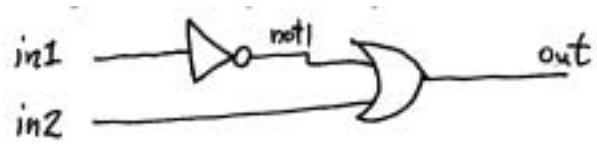

Figure 6: !in1 or in2 Circuit

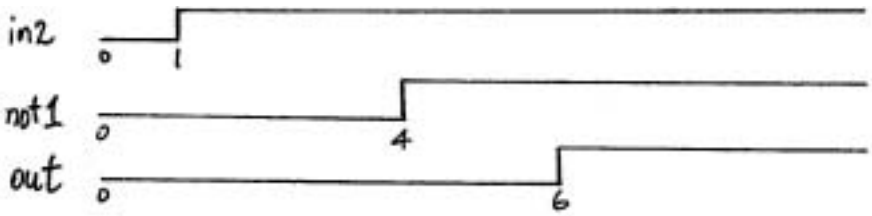

Figure 7: Required Timing for circuit !in1 or in2

Step 9. Or Gate

We have been able to simply propagate times immediately, so far, because we have a single chain. A good next step is to introduce a circuit that doesn't work properly with this approach. We need a gate with two inputs, such as an OR-gate, so that we can have two separate paths to the same wire, with the first path "longer".

The circuit shown in Fig. 6 is for (!in1 or in2). The required timing is shown in Fig. 7. Expected timing for this circuit (given the implementation as at Step 8) is shown in Fig. 8.

We first develop the OR-gate:

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
•• Simulate an inverter
⋆⋆ √ Simulate an inverter with explicit time delays
⋆⋆⋆√ Introduce a current-state Probe
⋆⋆⋆√ Probe records history
⋆⋆⋆√ Probe records time delays in history
•• A circuit with two paths to a wire (first path longer)
• • • Simulate an OR-gate

The class `TestSimulator` was actually just testing inverters. Tests for OR gates don't fit well with the `setUp()`, so it's time for a change. We rename `TestSimulator` to be `TestInverter` and

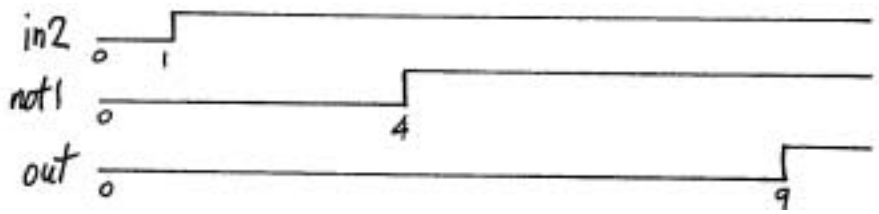

Figure 8: Expected Timing (as of Step 8) for circuit !in1 or in2

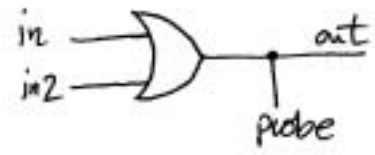

Figure 9: Or Test Circuit

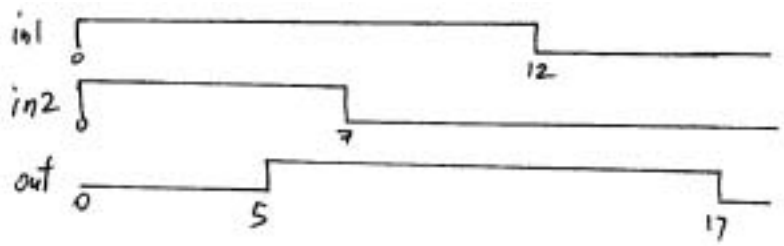

Figure 10: Timing for Or Test Circuit

introduce a new test class, `TestOrGate`.

The test circuit is shown in Fig. 9 and its timing diagram is shown in Fig. 10..

TestOrGate.java (09)

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

    public TestOrGate(String name) {
        super(name);
    }
    public void testOrGate() { // Or-gate delay is 5
        Wire out = new Wire();
        Probe probe = new Probe(out);
        Wire in1 = new Wire();
        Wire in2 = new Wire();
        new OrGate(in1,in2,out);
        in1.setSignal(0,true);
        in2.setSignal(0,true);
        in2.setSignal(7,false);
        in1.setSignal(12,false);
        assertEquals("(0,false)(5,true)(17,false)",probe.getLog());
    }
}
```

OrGate.java (09)

```
public class OrGate implements SignalChangeListener {
    private static int DELAY = 5;
    private Wire in1, in2, out;

    public OrGate(Wire in1, Wire in2, Wire out) {
        this.in1 = in1;
        this.in2 = in2;
        this.out = out;
        in1.addSignalChangeListener(this);
        in2.addSignalChangeListener(this);
    }
    public void signalChanged(SignalChangeEvent ev) {
        out.setSignal(ev.getTime()+DELAY,in1.getSignal() || in2.getSignal());
    }
}
```

A green bar with 7 successes.

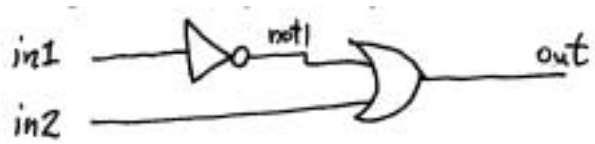

Figure 11: !in1 or in2 Circuit

Step 10. Circuit with two paths

So now we can build the circuit shown in Fig. 11. We add the test into `TestOrGate` because it fits easily; that means refactoring that class.

TestOrGate.java (10)

```
public class TestOrGate extends TestCase {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestOrGate.class);
    }
    private Wire in1, in2, out;
    private Probe probe;

    public TestOrGate(String name) {
        super(name);
    }
    public void setUp() {
        out = new Wire();
        probe = new Probe(out);
        in1 = new Wire();
        in2 = new Wire();
    }
    public void testOrGate() { // Or-gate delay is 5
        new OrGate(in1,in2,out);
        in1.setSignal(0,true);
        in2.setSignal(0,true);
        in2.setSignal(7,false);
        in1.setSignal(12,false);
        assertEquals("(0,false)(5,true)(17,false)",probe.getLog());
    }
    public void testOrGateWithInverter() { // Or-gate delay is 5
        // We need to wire it up backwards, so that gates haven't already acted
        Wire not1 = new Wire();
        new OrGate(not1,in2,out);
        new Inverter(in1,not1);
        in2.setSignal(1,true);
        assertEquals("(0,false)(6,true)",probe.getLog());
    }
}
```

The new test is valid, but it fails, giving "(0,false)(9,true)" instead of "(0,false)(6,true)".

It's because the signal propagation through the inverter is calculated immediately. The wire `not1` is set to `1` at time 4 (due to the inverter) and then the wire `out` given the value `1` at time 9

(due to the Or gate). Then the wire `in2` is changed at time 1, causing the Or gate to recalculate its output – which remains unchanged. See p?? for the timing diagrams.

The changes are propagated immediately, rather than in proper (simulation time) sequence. Hence we have a good test to drive the development further.

Step 11: Introduce Agenda

We need a way of scheduling time - the role of a simulator. So let's introduce an `Agenda`, which keeps track of the tasks that need to be done and when.

That means changing the gates so that they can add a `Task` to the Agenda to schedule the propagation of the signals with delays. So we need to pass them an `Agenda`. So let's first change the tests to incorporate the `Agenda`.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
⋆⋆ √ Simulate an inverter
•• A circuit with two paths to a wire (first path longer)
⋆⋆⋆√ Simulate an OR-gate
• • • Agenda schedules event propagation
• • •• Agenda passed to gates

TestOrGate.java (11)

```
public class TestOrGate extends TestCase {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestOrGate.class);
    }
    private Wire in1, in2, out;
    private Probe probe;
    private Agenda agenda;

    public TestOrGate(String name) {
        super(name);
    }
    public void setUp() {
        agenda = new Agenda();
        out = new Wire();
        probe = new Probe(out);
        in1 = new Wire();
        in2 = new Wire();
    }
    public void testOrGate() { // Or-gate delay is 5
        new OrGate(in1,in2,out,agenda);
        in1.setSignal(0,true);
        in2.setSignal(0,true);
        in2.setSignal(7,false);
        in1.setSignal(12,false);
        agenda.run();
        assertEquals("(0,false)(5,true)(17,false)",probe.getLog());
    }
    public void testOrGateWithInverter() { // Or-gate delay is 5
        // We need to wire it up backwards, so that gates haven't already acted
        Wire not1 = new Wire();
        new OrGate(not1,in2,out,agenda);
        new Inverter(in1,not1,agenda);
        agenda.run();
        in2.setSignal(1,true);
        assertEquals("(0,false)(6,true)",probe.getLog());
    }
}
```

That's only some of the code, to show the addition of the `Agenda`.

Inverter.java (11)

```
public class Inverter implements SignalChangeListener, Task {
    private Agenda agenda;
    private static int DELAY = 4;
    private Wire in, out;

    public Inverter(Wire in, Wire out, Agenda agenda) {
        this.in = in;
        this.out = out;
        this.agenda = agenda;
        in.addSignalChangeListener(this);
    }
    public void signalChanged(SignalChangeEvent ev) {
        agenda.schedule(ev.getTime()+DELAY,this);
    }
    public void run(int time) {
        out.setSignal(time,!in.getSignal());
    }
}
```

When an input signal changes for the `Inverter`, it schedules a task with the `Agenda` to run later. The `Agenda` calls the method `run()` of the interface `Task` once the delay is up.

Notice that the output is computed based on the value of the signal of the input wire that exists at the end of the delay. That's clearly wrong, so we need to add an item to our to-do list: "Output of gate depends on input values before delay". We don't try and fix the code immediately – if we did, how would we know that we got it right? And we don't want to add a new test at this stage, when we still have a failing test.

OrGate.java (11)

```
public class OrGate implements SignalChangeListener, Task {
    private static int DELAY = 5;
    private Wire in1, in2, out;
    private Agenda agenda;

    public OrGate(Wire in1, Wire in2, Wire out, Agenda agenda) {
        this.in1 = in1;
        this.in2 = in2;
        this.out = out;
        this.agenda = agenda;
        in1.addSignalChangeListener(this);
        in2.addSignalChangeListener(this);
    }
    public void signalChanged(SignalChangeEvent ev) {
        agenda.schedule(ev.getTime()+DELAY,this);
    }
    public void run(int time) {
        out.setSignal(time,in1.getSignal() || in2.getSignal());
    }
}
```

This is changed in the same way as the `Inverter`. It has the same smell, of advertising the gate as a `SignalChangeListener`, when that's only relevant to it responding to wire signal changes, and as a `Task`, when that's only relevant to it handling the `run()`. Notice also that there is common code between the `OrGate` and `Inverter` classes.

We'll add those to our to-do list: "Gates share code" and "Gates shouldn't advertise extra interfaces".

Agenda.java and Task.java (11)

```
/** Manages the scheduling of Tasks in the simulation.
    For now, we'll just run them immediately.
    @author rick mugridge july 2002
*/
public class Agenda {
    public void schedule(int time, Task task) {
        task.run(time);
    }
    public void run() {
    }
}

/** A Task is scheduled to run at a certain time.
    @author rick mugridge july 2002
*/
public interface Task {
    public void run(int time);
}
```

Notice how we managed to do the least possible by simply having the `Agenda` run the `Task` immediately. This allows us to check that 7 of our 8 tests pass – the last one was already failing. So now we've worked out the interface to `Agenda`; we can develop it properly.

Step 12. Agenda

We've sorted out the interface to `Agenda`, so now it's time to develop it test-first. An `Agenda` keeps track of the current time, and a list of tasks in time order. The simulation gets the first task from the agenda and carries it out. This may led to new tasks being added.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
⋆⋆ √ Simulate an inverter
•• A circuit with two paths to a wire (first path longer)
⋆⋆⋆√ Simulate an OR-gate
• • • Agenda schedules event propagation
⋆⋆⋆⋆ √ Agenda passed to gates
• • •• Agenda runs Task

• Gates share code
• Gates shouldn't advertise extra interfaces
• Output of gate depends on input values before delay

TestAgenda.java (12)

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

    private String taskLog;

    public TestAgenda(String name) {
        super(name);
    }
    public void testOneTask() {
        taskLog = "";
        Agenda agenda = new Agenda();
        agenda.schedule(12, new Task(){
                public void run(int time) {
                    taskLog += "("+time+")";
                }
            });
        agenda.run();
        assertEquals("(12)",taskLog);
    }
}
```

As expected, we immediately get a green bar with 1 success with `TestAgenda`. `TestAll` still has 1 fail, also as expected.

Step 13: Schedule two tasks out of order

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

    private String taskLog;
    private  Agenda agenda;

    public TestAgenda(String name) {
        super(name);
    }
    public void setUp() {
        taskLog = "";
        agenda = new Agenda();
    }
    public void testOneTask() {
        agenda.schedule(12, new MockTask("A"));
        agenda.run();
        assertEquals("(12,A)",taskLog);
    }
    public void testTwoTasksOutOfOrder() {
        agenda.schedule(15, new MockTask("B"));
        agenda.schedule(12, new MockTask("A"));
        agenda.run();
        assertEquals("(12,A)(15,B)",taskLog);
    }
    private class MockTask implements Task {
        String name;
        public MockTask(String name) {
            this.name = name;
        }
        public void run(int time) {
            taskLog += "("+time+","+name+")";
        }
    }
}
```

As usual, we proceed in small steps. Here we add two tasks out of time order to drive the development of class `Agenda`.

Notice how we get the benefit here of defining the general interface `Task`. The `Agenda` is completely independent of gates, wires, etc. This means that we can test it separately and we can reuse it more easily. Modularity is a great thing!

To avoid repetition, we use a `setUp()` and introduce a `MockTask`. We run the test and check that it fails, as expected.

TestAgenda.java (13B)

So now we need a way of keeping a time-ordered list of tasks. We could keep the tasks in a list, along with their times. A better approach would be to use an ordered tree. Looking through the Java Collection classes, the best one looks to be `TreeSet`, but that requires the elements to be unique, as well as `Comparable`.

So we need to develop a class to hold the time-task pair, where each object is unique and they can be compared. What's the best way of developing that? Test-first, of course. So let's comment out that last test for now and add a test for class `TimedTask` in `TestAgenda`:

```
/*    public void testTwoTasksOutOfOrder() {
        agenda.schedule(15,new MockTask("B"));
        agenda.schedule(12,new MockTask("A"));
        agenda.run();
        assertEquals("(12,A)(15,B)",taskLog);
    }
*/
    public void testTimedTask() {
        TimedTask tt1 = new TimedTask(12,new MockTask("A"));
        TimedTask tt2 = new TimedTask(13,new MockTask("B"));
        assertTrue(tt1.compareTo(tt2) < 0);
        assertTrue(tt2.compareTo(tt1) > 0);
    }
```

TimedTask.java (13B)

```
public class TimedTask implements Comparable {
    private final int time;
    private final Task task;

    public TimedTask(int time, Task task) {
        this.time = time;
        this.task = task;
    }
    public int compareTo(Object other) {
        TimedTask that = (TimedTask)other;
        return this.time - that.time;
    }
}
```

TestAgenda.java (13C)

And now we want to ensure that two tasks that are scheduled at the same time are not equal (otherwise, only one will end up in the `TreeSet`).

```
    public void testTimedTasksUnique() {
        TimedTask tt1 = new TimedTask(12,new MockTask("A"));
        TimedTask tt2 = new TimedTask(12,new MockTask("A"));
        assertTrue(tt1.compareTo(tt2) != 0);
    }
```

TimedTask.java (13C)

```
public class TimedTask implements Comparable {
    private final int time;
    private final Task task;
    private static int COUNT = 0;
    private int id = COUNT++;

    public TimedTask(int time, Task task) {
        this.time = time;
        this.task = task;
    }
    public int compareTo(Object other) {
        TimedTask that = (TimedTask)other;
        if (time == that.time)
            return id - that.id;
        else
            return this.time - that.time;
    }
}
```

TestAgenda.java (13D)

And now we check `TimedTasks` with a `TreeSet`:

```
    public void testTimedTasksInTreeSet() {
        TimedTask tt1 = new TimedTask(12,new MockTask("A"));
        TimedTask tt2 = new TimedTask(12,new MockTask("A"));
        TreeSet set = new TreeSet();
        set.add(tt2);
        set.add(tt1);
        assertEquals(tt1,set.first());
        set.remove(tt1);
        assertEquals(tt2,set.first());
        set.remove(tt2);
        assertTrue(set.isEmpty());
    }
```

Agenda.java (13E)

So now we can uncomment that test in `TestAgenda` and satisfy it:

```
public class Agenda {
    private SortedSet tasks = new TreeSet();

    public void schedule(int time, Task task) {
        tasks.add(new TimedTask(time,task));
    }
    public void run() {
        while (!tasks.isEmpty()) {
            TimedTask timedTask = (TimedTask)tasks.first();
            tasks.remove(timedTask);
            timedTask.run();
        }
    }
}
```

Can this result in an infinite loop? Yes, if a task keeps adding itself to the agenda.

TimedTask.java (13E)

Finally, we need to add a `run()` method here:

```
public class TimedTask implements Comparable {
    private final int time;
    private final Task task;
    private static int COUNT = 0;
    private int id = COUNT++;

    public TimedTask(int time, Task task) {
        this.time = time;
        this.task = task;
    }
    public int compareTo(Object other) {
        TimedTask that = (TimedTask)other;
        if (time == that.time)
            return id - that.id;
        else
            return this.time - that.time;
    }
    public void run() {
        task.run(time);
    }
}
```

So we finally have the 5 tests in `TestAgenda` all passing. But `TestAll` now has four failures.

Step 14. Update Other Tests

The change to method `schedule()` in class `Agenda` breaks five of the tests in `TestInverter` and `TestOrGate`. That's because we'd assumed that we can change an input signal in-line, but now they need to be scheduled.

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
⋆⋆ √ Simulate an inverter
•• A circuit with two paths to a wire (first path longer)
⋆⋆⋆√ Agenda schedules event propagation
⋆⋆⋆⋆ √ Agenda passed to gates
⋆⋆⋆⋆ √ Agenda runs Task
••• Older tests failing – schedule input changes
••• Output of gate depends on input values before delay

• Gates share code
• Gates shouldn't advertise extra interfaces

TestOrGate.java (14)

We'll do it with `scheduleSignal()`.

```
public class TestOrGate extends TestCase {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestOrGate.class);
    }
    private Wire in1, in2, out;
    private Probe probe;
    private Agenda agenda;

    public TestOrGate(String name) {
        super(name);
    }
    public void setUp() {
        agenda = new Agenda();
        out = new Wire();
        probe = new Probe(out);
        in1 = new Wire();
        in2 = new Wire();
    }
    public void testOrGate() { // Or-gate delay is 5
        new OrGate(in1,in2,out,agenda);
        scheduleSignal(in1,0,true);
        scheduleSignal(in2,0,true);
        scheduleSignal(in2,7,false);
        scheduleSignal(in1,12,false);
        agenda.run();
        assertEquals("(0,false)(5,true)(17,false)",probe.getLog());
    }
    public void testOrGateWithInverter() { // Or-gate delay is 5
        // We need to wire it up backwards, so that gates haven't already acted
        Wire not1 = new Wire();
```

```
        new OrGate(not1,in2,out,agenda);
        new Inverter(in1,not1,agenda);
        scheduleSignal(in2,1,true);
        agenda.run();
        assertEquals("(0,false)(6,true)",probe.getLog());
    }
    private void scheduleSignal(Wire wire, int time, boolean value) {
        agenda.schedule(time,new WireChangeTask(wire,value));
    }
}
```

WireChangeTask.java (14)

```
public class WireChangeTask implements Task {
    private Wire wire;
    private boolean value;

    WireChangeTask(Wire wire, boolean value) {
        this.wire = wire;
        this.value = value;
    }
    public void run(int time) {
        wire.setSignal(time,value);
    }
}
```

Two of the tests still fail. It looks like it's because of the gates computing their output based on their input values after the delay.

TestInverter.java (14)

So let's write a simple test that will check whether it is the already-identified problem with the output value of the gate being determined by the values of the inputs after the delay rather than before:

```
    public void testInvertQuickly() { // Inverter delay is 4
        scheduleSignal(in,1,true);
        agenda.run();
        assertEquals("(0,false)(4,true)(5,false)",probe.getLog());
    }
```

The result log of "(0,false)" seems to confirm it. When the `in` wire changed the gate would've scheduled a task to update the output. After the gate's delay (at time = 4), it appears to have calculated the output (to `out` wire) based on an `in` signal of 1 rather than of signal of 0. Hence the `out` wire's signal is unchanged. So now that we have a clear, failing test, we can update class `Inverter`:

Inverter.java (14)

```
    private Agenda agenda;
    private static int DELAY = 4;
    private Wire in, out;

    public Inverter(Wire in, Wire out, Agenda agenda) {
        this.in = in;
        this.out = out;
        this.agenda = agenda;
        in.addSignalChangeListener(this);
    }
    public void signalChanged(SignalChangeEvent ev) {
        agenda.schedule(ev.getTime()+DELAY,
                        new WireChangeTask(out,!in.getSignal()));
    }
}
```

So the above test (`testInvertQuickly()`) now passes. Let's do the same for Or gates. First, we'll write a simple confirming test:

TestOrGate.java (14)

```
    public void testOrQuickly() { // Or-gate delay is 5
        scheduleSignal(in1,1,true);
        scheduleSignal(in1,2,false);
        agenda.run();
        assertEquals("(0,false)(6,true)(7,false)",probe.getLog());
    }
```

As expected, given the current implementation of `OrGate`, the result is "(0,false)". After the gate's delay (at time = 5), it appears to have calculated the output (to `out` wire) based on an `in` signal of 0 rather than of signal of 1 (and hence the `out` wire's signal is unchanged).

Notice the common code between the tests creeping in here, which we need to refactor out; we'll add that to our to-do list.

OrGate.java (14)

```
public class OrGate implements SignalChangeListener {
    private static int DELAY = 5;
    private Wire in1, in2, out;
    private Agenda agenda;

    public OrGate(Wire in1, Wire in2, Wire out, Agenda agenda) {
        this.in1 = in1;
        this.in2 = in2;
        this.out = out;
        this.agenda = agenda;
        in1.addSignalChangeListener(this);
        in2.addSignalChangeListener(this);
    }
    public void signalChanged(SignalChangeEvent ev) {
        agenda.schedule(ev.getTime()+DELAY,
            new WireChangeTask(out,in1.getSignal() || in2.getSignal()));
    }
}
```

So we expect that change to pass the test `testOrQuickly()`, but it doesn't! What's going wrong? After a few minutes, I discover that the test is wrong – it needs to create an `OrGate`.

TestOrGate.java (14B)

```
public void testOrQuickly() { // Or-gate delay is 5
    new OrGate(in1,in2,out,agenda);
    scheduleSignal(in1,1,true);
    scheduleSignal(in1,2,false);
    agenda.run();
    assertEquals("(0,false)(6,true)(7,false)",probe.getLog());
}
```

Yes, that's the problem,. So we now have a green bar with 15 successes.

Step 15. Refactor Gates

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
⋆⋆√ A circuit with two paths to a wire (first path longer)
⋆⋆⋆√ Agenda schedules event propagation
⋆⋆⋆√ Older tests failing – schedule input changes
⋆⋆⋆√ Output of gate depends on input values before delay
•• Gates share code
•• Gates shouldn't advertise extra interfaces
•• Tests share code

Gate.java (15)

```
public abstract class Gate {
    protected final Wire in;
    private final Wire out;
    private final Agenda agenda;
    private final int delay;
    private final Handler handler = new Handler();

    public Gate(Wire in, Wire out, Agenda agenda, int delay) {
        this.in = in;
        this.out = out;
        this.agenda = agenda;
        this.delay = delay;
    }
    protected void registerWithWire(Wire wire) {
        wire.addSignalChangeListener(handler);
    }

    protected abstract boolean getOutSignal();

    class Handler implements SignalChangeListener {
        public void signalChanged(SignalChangeEvent ev) {
            agenda.schedule(ev.getTime()+delay,
                            new WireChangeTask(out,getOutSignal()));
        }
    }
}
```

Inverter.java (15)

```
public class Inverter extends Gate {
    private final static int DELAY = 4;

    public Inverter(Wire in, Wire out, Agenda agenda) {
        super(in,out,agenda,DELAY);
        registerWithWire(in);
    }
    protected boolean getOutSignal() {
        return !in.getSignal();
    }
}
```

OrGate.java (15)

```
public class OrGate extends Gate {
    private final static int DELAY = 5;
    private final Wire in2;

    public OrGate(Wire in1, Wire in2, Wire out, Agenda agenda) {
        super(in1,out,agenda,DELAY);
        this.in2 = in2;
        registerWithWire(in);
        registerWithWire(in2);
    }
    protected boolean getOutSignal() {
        return in.getSignal() || in2.getSignal();
    }
}
```

After this refactoring the tests still all pass. Let's now refactor the tests, by extracting a common superclass of `TestInverter` and `TestOrGate`.

DigitalTestCase.java (15)

```
public abstract class DigitalTestCase extends TestCase {
    protected Wire out;
    protected Probe probe;
    protected Agenda agenda;

    public DigitalTestCase(String name) {
        super(name);
    }
    public void setUp() {
        agenda = new Agenda();
        out = new Wire();
        probe = new Probe(out);
    }
    protected void scheduleSignal(Wire wire, int time, boolean value) {
        agenda.schedule(time,new WireChangeTask(wire,value));
    }
}
```

TestInverter.java (15)

```
public class TestInverter extends DigitalTestCase {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestInverter.class);
    }
    private Wire in;
    private Inverter invert;

    public TestInverter(String name) {
        super(name);
    }
    public void setUp() {
        super.setUp();
        in = new Wire();
        invert = new Inverter(in,out,agenda);
    }
   ...
```

And the tests all pass again.

Conclusion

So we are well underway in the development of the digital simulation. We can tick off what's been done and plan the next step:

TO DO:
Build a digital circuit simulator
• Simulate the half-adder circuit shown above
$\star\star\surd$ A circuit with two paths to a wire (first path longer)
$\star\star\surd$ Gates share code
$\star\star\surd$ Gates shouldn't advertise extra interfaces
$\star\star\surd$ Tests share code

We've also seen how tests can be designed to drive the development of a program. It's not always easy to choose the next step, to decide what's a good move, and then to write a good test. But our efforts led us to clarity and correctness, and the tests leave a legacy for ensuring that things don't break as we evolve the program further.

In time to come, programmers will wonder how high-quality programs could have been developed without using this approach. Just as today we wonder how programmers managed in the past with goto's and without for-loops and parameter passing (ie, pre-"structured programming"), and object-oriented programming.