Tao of Test-Driven Development

http://github.com/sl4mmy/presentations/tree/master

By Kent R. Spillner <kspillner@acm.org>
Copyright (c) 2009, Kent R. Spillner
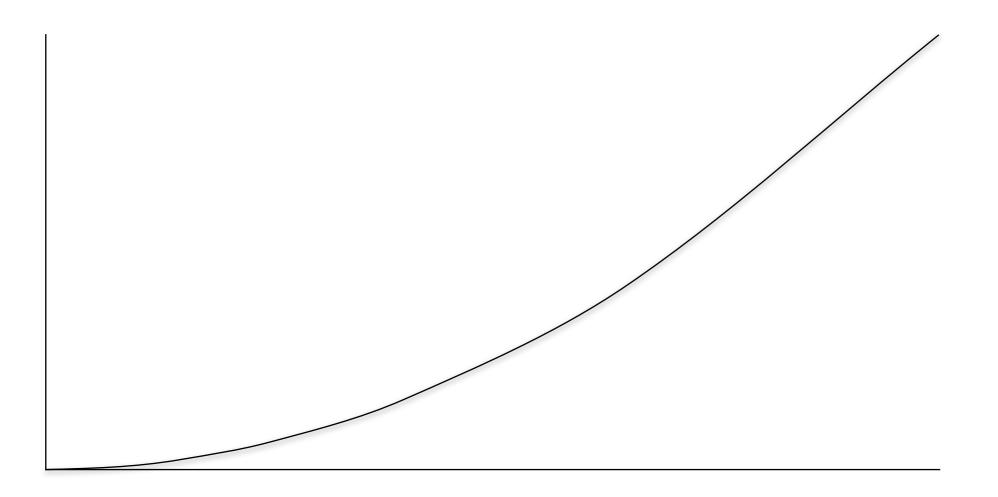
Kent Beck

The Problem

— Cost of Fixing Bugs vs. Time

Worst-Case Scenario

If you even test at all!

As one of the last things before releasing, testing is usually one of the first things cut!

A Simple Hypothesis:

Testing first finds more bugs, more quickly allowing you to fix them more affordably

Enter Test-Driven Development

RED

Write a test to verify the correctness of the next feature you're about to add

Run it, watch it fail

"Testing shows the presence, not the absence of bugs"

-Edsger W. Dijkstra

Always, always, always run that test and watch it fail!

GREEN

Do the simplest thing that can possibly work

Run it again, watch it pass

Hard-code values, allocate single storage instead of collections, perform single calculations instead of looping

REFACTOR

Refactor mercilessly to eliminate duplication and technical debt

Clean your code before continuing on

LATHER, RINSE REPEAT

Pleasant Surprises

Lower costs of fixing bugs is just the tip of the iceberg!

More confident: comprehensive tests let you experiment boldly

More productive: more time spent testing means more time spent programming, less debugging

More reliable: you're writing regression tests that keep you from taking 2 steps forward, 1 step back

More agile: you can turn on a dime at a moment's notice!

More Than Just Tests

Test-driven development goes beyond traditional testing

Better code, simpler designs

Automated programmer tests don't just verify behavior, they also exercise your APIs

Passing tests are executable documentation that is never (rarely) out-of-date

And if you give your tests descriptive names, they will also communicate *why* the system behaves a particular way

Tests as Communication Tools

Tests are the best place for new team members to start

Tests Make Good Bookmarks

Failing tests are natural stopping places
When you return to the code, pick up where you left off by turning that red test to green

More Bang for Your Buck

Red Before Green

Ensure your tests are correct
Read and re-read your tests to confirm they
actually verify the correct behavior

Know What You're Doing

Read and re-read tests before modifying or deleting them

Test Everything

Nothing is "too simple" to be tested
Test any code that could break

If code is simple to write, it should be simple to test
Don't be lazy!

Give your tests descriptive names that explain both what is being tested, and why

Tests Drive the Code

Write a test to verify the trivial "happy path"
Then verify the "sad path"
Then verify the border conditions

Keep Tests Focused

Don't test too much at once
Prefer many short tests to a few long ones

Baby Steps

Short tests, simple code
Repeat, refactor

Listen to Your Tests

Tests are the first users of production code
If tests are long and difficult to read, your APIs are probably complex and difficult to use

Q: What's wrong with modifying production code
to make it easier to test?

A: Nothing!

Failure Doesn't Just Happen

Tests fail for a reason
Listen to failing tests: investigate that reason!

\<break /\>

Dojo Time!

The Rules are Simple:

Using your favorite programming tools, write tests to drive the implementation of an elevator management system which satisfies the following requirements

You will have 10 minutes for each requirement
You may pair with another person

There are no "right" or "wrong" implementations

Force yourself to follow and practice what we just learned

Ready... Set...

When the "call button" is pressed, the elevator should travel to that floor if it's not already there

When the "call button" is pressed and the elevator is available on that floor, it should allow those passengers to board

When the passenger pushes a floor's button, the elevator should travel to that floor

When the elevator drops off its last passenger, it should wait on that floor until another passenger calls for an elevator

When the elevator is traveling to a floor and another passenger pushes the call button on a different floor, the elevator should drop-off the first passenger and then go pick-up the second

When the elevator is traveling to a floor and another passenger pushes the call button on a different floor that is on the way to the first floor, the elevator should stop and pick-up the second passenger and then drop-off the first passenger

Discussion:

At this point, do you model time in your implementation? Why or why not? If you do, how do you model time?

When passengers push the buttons for different floors, the elevator should stop at every selected floor

(Assume all the passengers get on at the same time, and no new passengers get on until after everyone is dropped off)

When passengers push the buttons for different floors, the elevator should stop at every selected floor it passes on its way to the next drop-off

(Assume all passengers are traveling in the same "direction," including new passengers that get on *before* everyone is dropped off)

Discussion:

At this point, do you model floors in your implementation? Why or why not? If you do, how do you model floors?

When the elevator is traveling to a floor and another passenger pushes the call button on a different floor that is in the opposite direction, the elevator should drop-off all the passengers traveling in the first direction before returning to pick-up the other passengers

When passengers push the buttons for different floors, the elevator should continue traveling in the same direction until it has dropped everyone off before traveling in the opposite direction

When the elevator is traveling to a floor and another passenger pushes the call button on a different floor that is in the same direction but further away, the elevator should drop-off all the passengers traveling in *both* directions before returning to pick-up the new passengers

Discussion:

At this point, how do you model elevator "requests"? How do you track the current request? Pending requests?

Discussion:

What is the most challenging aspect of test-driven development? Have you experienced any benefits of test-driven development yet? If yes, what is the most surprising benefit?

Now let's add support for multiple elevators...

When the "call button" is pressed, the closest available elevator should travel to that floor if it's not already there

When the "call button" is pressed and every elevator is currently traveling to other floors to drop-off passengers, the next available elevator should travel to that floor to pick-up the new passengers

Challenge:

What does your implementation currently do in the following situation: one elevator is currently traveling to floor 2 then 3 then 4, the other elevator is currently traveling to floor 4 then 3 then 2, and a passenger on floor 1 calls an elevator and then a passenger on floor 5 calls an elevator?

When the previous situation occurs, the first elevator (traveling to floor 2 then 3 then 4) should be scheduled to pick-up the passengers calling for the elevator on floor 4 and the other elevator (traveling to floor 4 then 3 then 2) should be scheduled to pick-up the passengers calling for the elevator on floor 1

Discussion:

How simple and flexible is your current design? Has test-driving this implementation improved the design? If yes, how?

Challenge:

The elevator manager is receiving complaints from passengers that the elevators take too long to arrive when called. How much work is involved in replacing the scheduling logic so that the manager can experiment with different scheduling algorithms?

When an elevator is called or a floor's button is pressed, the elevator should stop at the next closest scheduled floor

Discussion:

Now the elevator manager is receiving complaints from passengers that the elevators are taking too long to drop passengers off! How much work will it be this time to change scheduling algorithms again?

When an elevator is called or a floor's button is pressed, the elevator should stop at the next closest scheduled floor but should only stop at any floor a maximum of 1 time before fulfilling all other pending requests

When an elevator is called or a floor's button is pressed, the elevator should stop at the next closest scheduled floor and should stop at each floor at most 1 time and should not stop at more than 3 consecutive floors

(Assume that after stopping at 3 consecutive floors, the elevator automatically *falls back* to the original scheduling algorithm)

Despite tweaking the scheduling algorithm, some passengers are still complaining about long wait times.

The elevator manager has already added extra elevator shafts, but there is not enough space to add more...

In desparation, they turn to you and ask:

How much work will it be to support piggy-backing multiple elevators in a single shaft?

When an elevator is called from the *lower* lobby, only the closest (or next available) *lower* elevator should travel to that floor if it's not already there

When an elevator is called from the *upper* lobby, only the closest (or next available) *upper* elevator should travel to that floor if it's not already there

When the *lower* elevator is traveling *upwards*, it should only be able to stop at *odd*-numbered floors

When the *upper* elevator is traveling *upwards*, it should only be able to stop at *even*-numbered floors

When the *lower* elevator is traveling *upwards*, it should never be able to pass the *upper* elevator

When the *lower* elevator is traveling *downwards*, it should be able to stop at *any* floor

When the *upper* elevator is traveling *downwards*, it should be able to stop at *any* floor *except* the lower lobby

When the *upper* elevator is traveling *downwards*, it should never be able to pass the *lower* elevator

Discussion:

How much work was it to change the details of the scheduling algorithm itself (versus the amount of work required to support changing the algorithm itself)? How did test-driven development make you more productive while implementing these changes? How did test-driven development complicate the implementation of these changes?

Discussion:

How much work was it to support the requirements that the upper and lower elevators never pass each other? Were you able to use your existing implementation of the "next closest available" scheduling logic? How confident are you that your software will prevent elevator crashes from happening in production?

What Did We Learn Today?

RED - GREEN - REFACTOR

Don't forget to run that new failing test you just wrote!!!

Keep it simple!
(Both the tests, and the code)

Baby steps!
Let the tests drive the code

Descriptive test names are priceless!

And always, always, always
listen to the tests!