

Only you can see this message



This story's distribution setting is on. You're in the Partner Program, so this story is eligible to earn money. [Learn more](#)

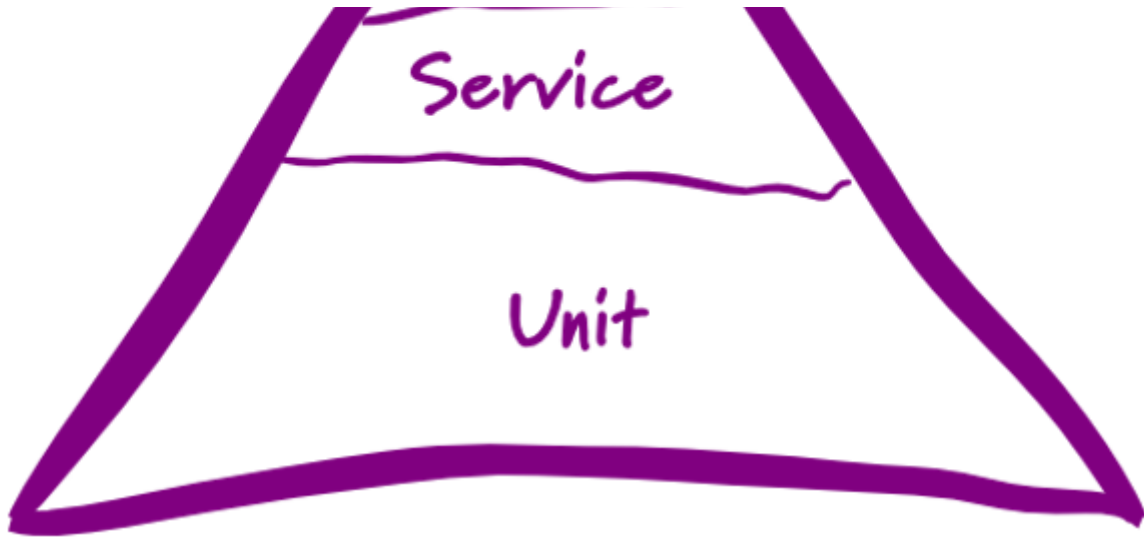
ATDD — working with the acceptance test development paradigm



Craig Godden-Payne

Oct 26, 2018 · 6 min read ★



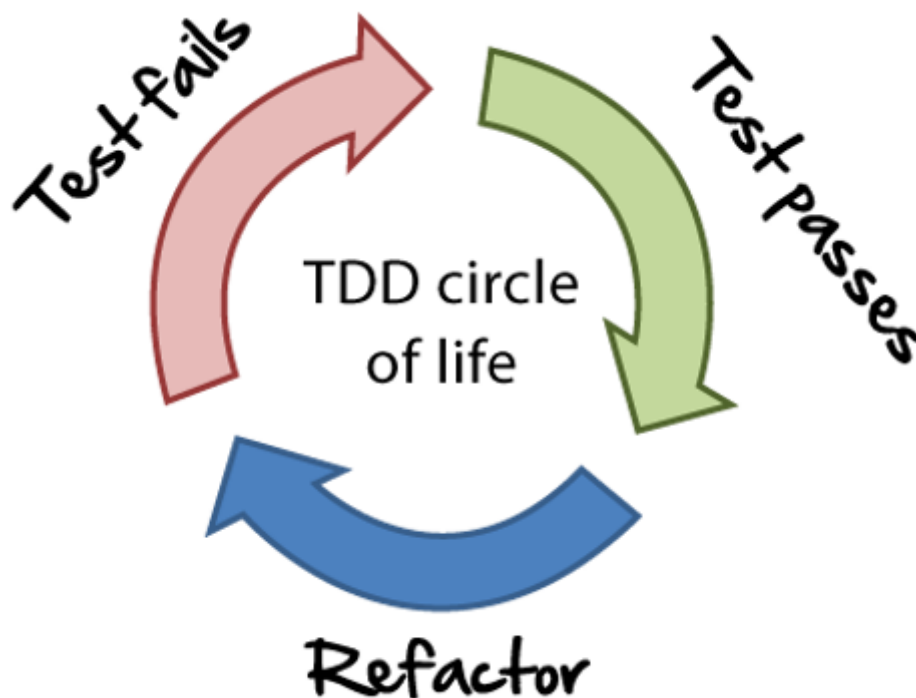


First of all, it's worth looking at the testing pyramid.

The further you go up the pyramid, the less tests you should have.

This is because the tests are more brittle, harder to maintain, and more complicated.

Next it's worth touching on Test Driven Development as a concept. TDD is a software development process that relies on the repetition of a very short development cycle.



You should write an (initially failing) automated test case that defines a new feature. You then produce the minimum amount of code to pass that test, and then refactor the

new code to acceptable standards.

As with all software development practices, there are pros and cons to working in TDD.

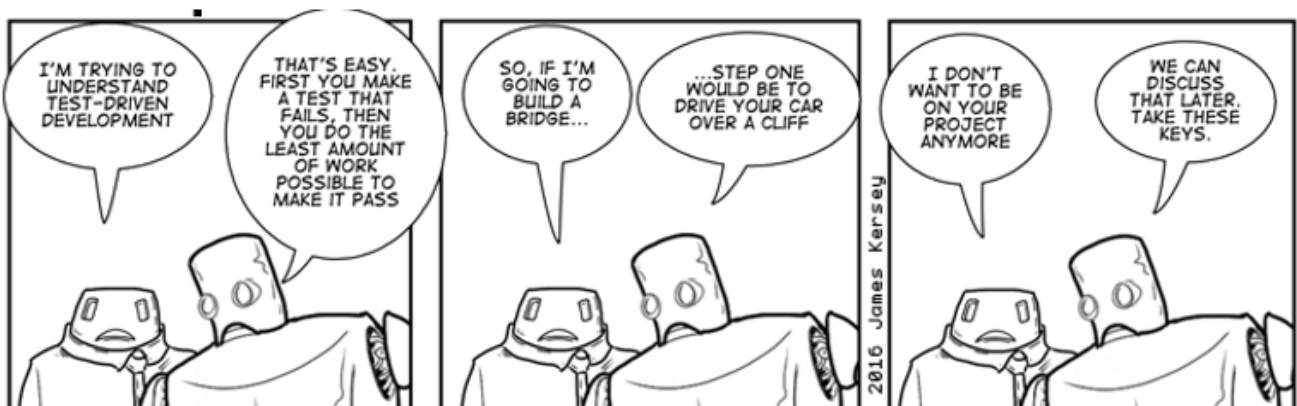
Pros

- Easy to iterate on code when the suite of tests back you up.
- Makes code easier to maintain and refactor.
- You never run out of time, and end up missing out the tests
- Forces your code to be more modular (otherwise they'd be hard to test against).
- Documents your code better than documentation (it doesn't go out of date, since you're running it all the time).
- Creates a common code pattern that can be followed by other devs
- Never get carried away and write more code than is necessary

Cons

- The test suite itself has to be maintained; tests are not completely deterministic (i.e. reliant on external dependencies).
- Can slow down development, compared to non tested code.
- Can over complicate code structure if over thought
- Writing good unit tests is an art form. Don't focus too much on metrics like code coverage. They do not tell you about the quality of the unit tests.

Is Test Driven Development worth the effort?



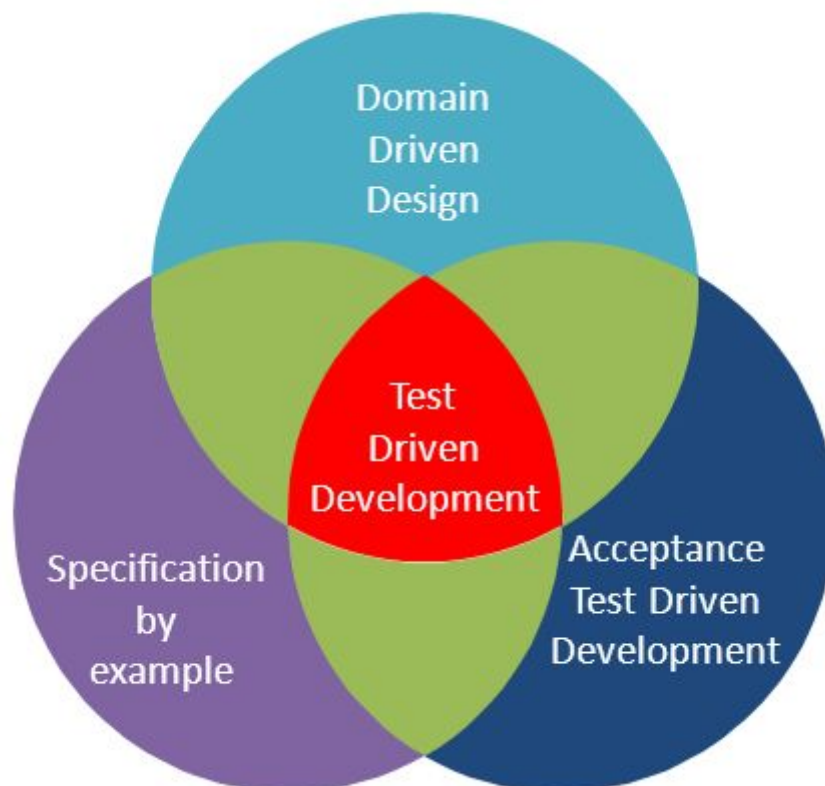


Unit Tests allow you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again. This saves hours.

TDD helps you to realise when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.

The tests and the code work together to achieve better code. Your code could be bad / buggy. Your TEST could be bad / buggy. In TDD you are banking on the chances of both being bad / buggy being low. Often it's the test that needs fixing but that's still a good outcome.

TDD helps with coding constipation. When faced with a large and daunting piece of work ahead writing the tests will get you moving quickly.



Unit Tests help you to really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.

Unit Tests give you instant visual feedback, we all like the feeling of all those green lights when we're done. It's very satisfying. It's also much easier to pick up where you left off after an interruption because you can see where you got to — that next red light that needs fixing.

Contrary to popular belief unit testing does not mean writing twice as much code, or coding slower. It's faster and more robust than coding without tests once you've got the hang of it. Test code itself is usually relatively trivial and doesn't add a big overhead to what you're doing.

“Imperfect tests, run frequently, are much better than perfect tests that are never written at all”. If pushed for time, write tests where I think they'll be most useful even if the rest of the code coverage is woefully incomplete.

Good unit tests can help document and define what something is supposed to do

```
Executed 1 of 5 SUCCESS (0 secs / 0.026 secs
Executed 2 of 5 SUCCESS (0 secs / 0.028 secs
Executed 4 of 5 SUCCESS (0 secs / 0.031 secs
Executed 5 of 5 SUCCESS (0 secs / 0.032 secs
Executed 5 of 5 SUCCESS (0.043 secs / 0.032
```

How do acceptance tests differ?

Acceptance tests are also used as regression tests prior to a production release

The name acceptance tests was changed from functional tests. This better reflects the intent, which is to guarantee that a customer's requirements have been met and the system is acceptable





A user story is not considered complete until it has passed its acceptance tests

Acceptance tests usually are just black box system tests.

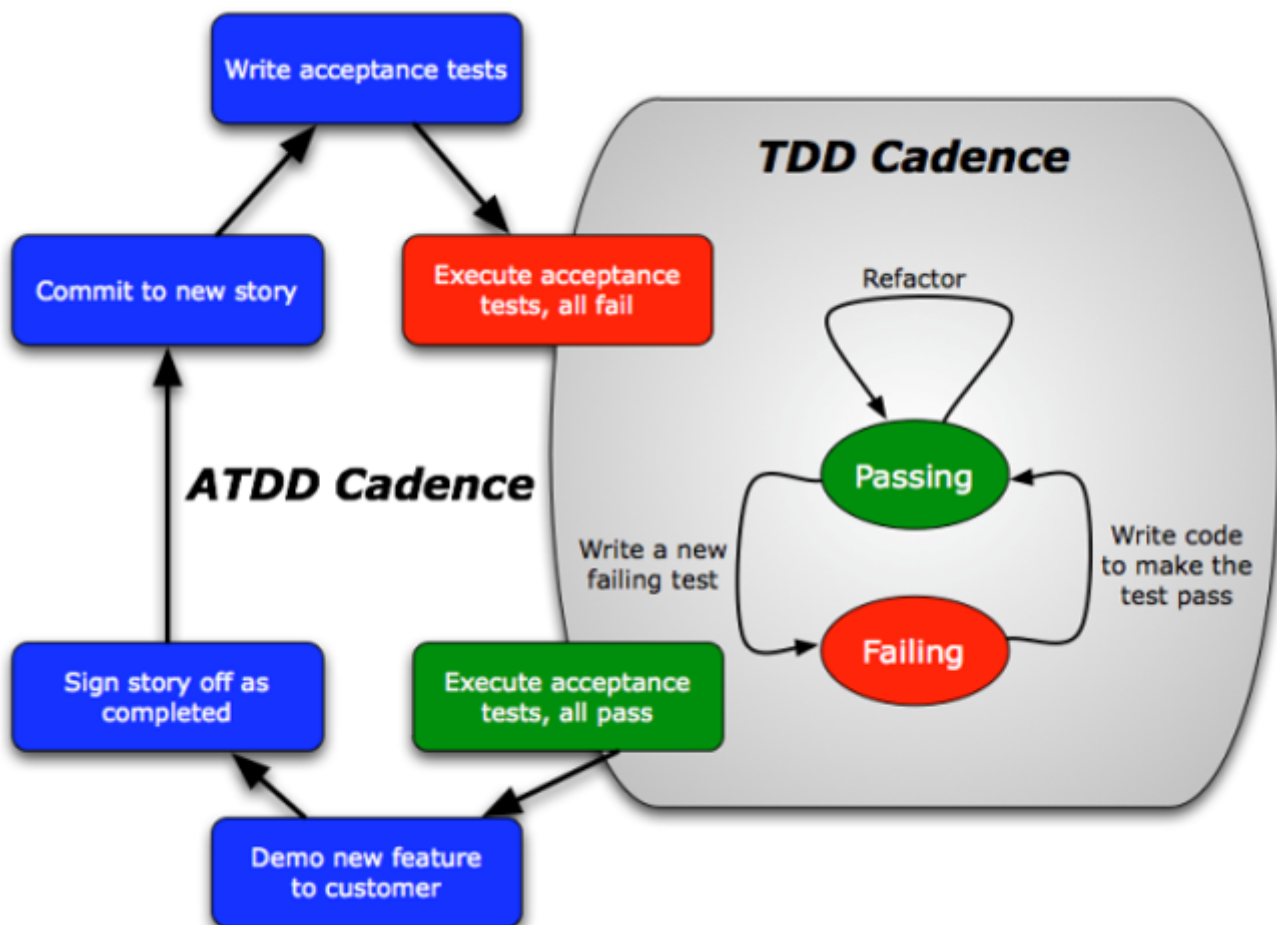
Acceptance tests should be automated so they can be run often.

Combining the two approaches

Acceptance Test Driven Development (ATDD) is a practice in which the whole team collaboratively discusses acceptance criteria, with examples, and then distills them into a set of concrete acceptance tests before development begins.

It's the best way to ensure that we all have the same shared understanding of what it is we're actually building.

It's also a good way to ensure everyone has a shared definition of Done.



Where's the best place to start writing in a TDD style?

Use appropriate design patterns — This should start to split your code into chunks of testable behaviours e.g. Single Responsibility Principle — every component, class or function should have a well-defined, single responsibility and only one reason to change. If you try to describe to someone what a function does (its responsibility) and use words such as or, and, also, besides etc. then your class most likely has more than one responsibility and therefore violates SRP.

Keep changes to the UI separate from logic (use a pattern such as MVC). It's quite difficult to assert changes to the UI in a unit test, but its really simple to assert the result of a function that performs some logic, such as a calculation, or validation logic.

If you struggle writing TDD, think about the problem and how you want to tackle it. If it's complicated, split it down further and write the smallest part of what you need to do, then write your test. Using this approach is not proper TDD, but it will get you thinking in the right mind-set. The key is to start scoping out scenarios and covering the code with tests as you go, which will prove invaluable when it comes to refactoring or future code changes.



You can use modular units, that we can segregate the functionality to different modules * that can look after one thing.

```
function Calculator(){}
Calculator.prototype = (function(){
  var calculateInterest = function(amount, months, interestRate){
    return (amount * (interestRate * .01)) / months;
  };

  var calculatePayment = function(amount, months, interest){
    return ((amount / months) + interest).toFixed(2);
  };

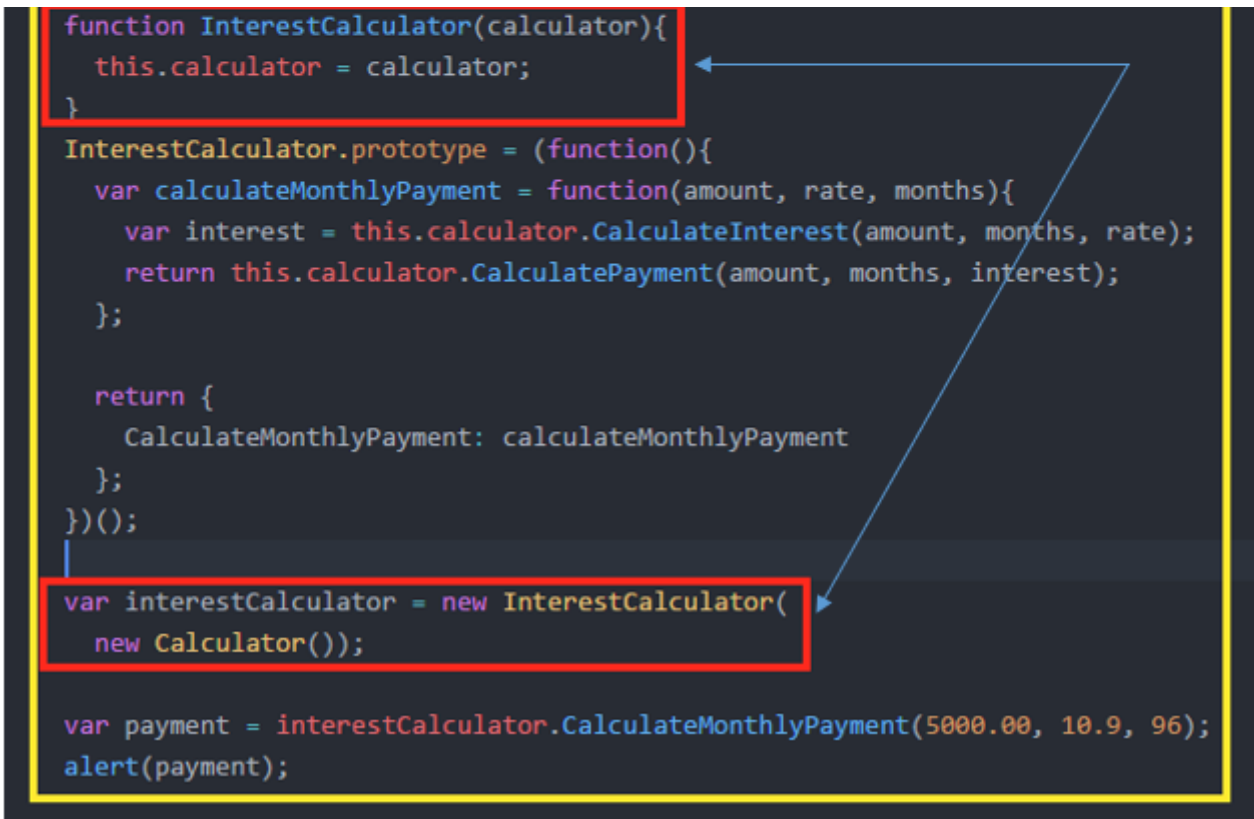
  return {
    CalculateInterest: calculateInterest,
    CalculatePayment: calculatePayment
  }
})();
```

```
})();  
  
function InterestCalculator(calculator){  
  this.calculator = calculator;  
}  
InterestCalculator.prototype = (function(){  
  var calculateMonthlyPayment = function(amount, rate, months){  
    var interest = this.calculator.CalculateInterest(amount, months, rate);  
    return this.calculator.CalculatePayment(amount, months, interest);  
  };  
  
  return {  
    CalculateMonthlyPayment: calculateMonthlyPayment  
  };  
})();  
  
var interestCalculator = new InterestCalculator(  
  new Calculator());  
  
var payment = interestCalculator.CalculateMonthlyPayment(5000.00, 10.9, 96);  
alert(payment);
```

Dependencies are injected in, which means we can mock (fake/spy) on the dependency, and we only test a small subset of the functionality, rather than the end to end path.

An Example of this, if calling CalculateMonthlyPayment, we only need to assert that we called CalculateInterest and CalculatePayment, rather than the actual functionality inside the methods.

```
function Calculator(){}  
Calculator.prototype = (function(){  
  var calculateInterest = function(amount, months, interestRate){  
    return (amount * (interestRate * .01)) / months;  
  };  
  
  var calculatePayment = function(amount, months, interest){  
    return ((amount / months) + interest).toFixed(2);  
  };  
  
  return {  
    CalculateInterest: calculateInterest,  
    CalculatePayment: calculatePayment  
  }  
})();
```

```
function InterestCalculator(calculator){
  this.calculator = calculator;
}

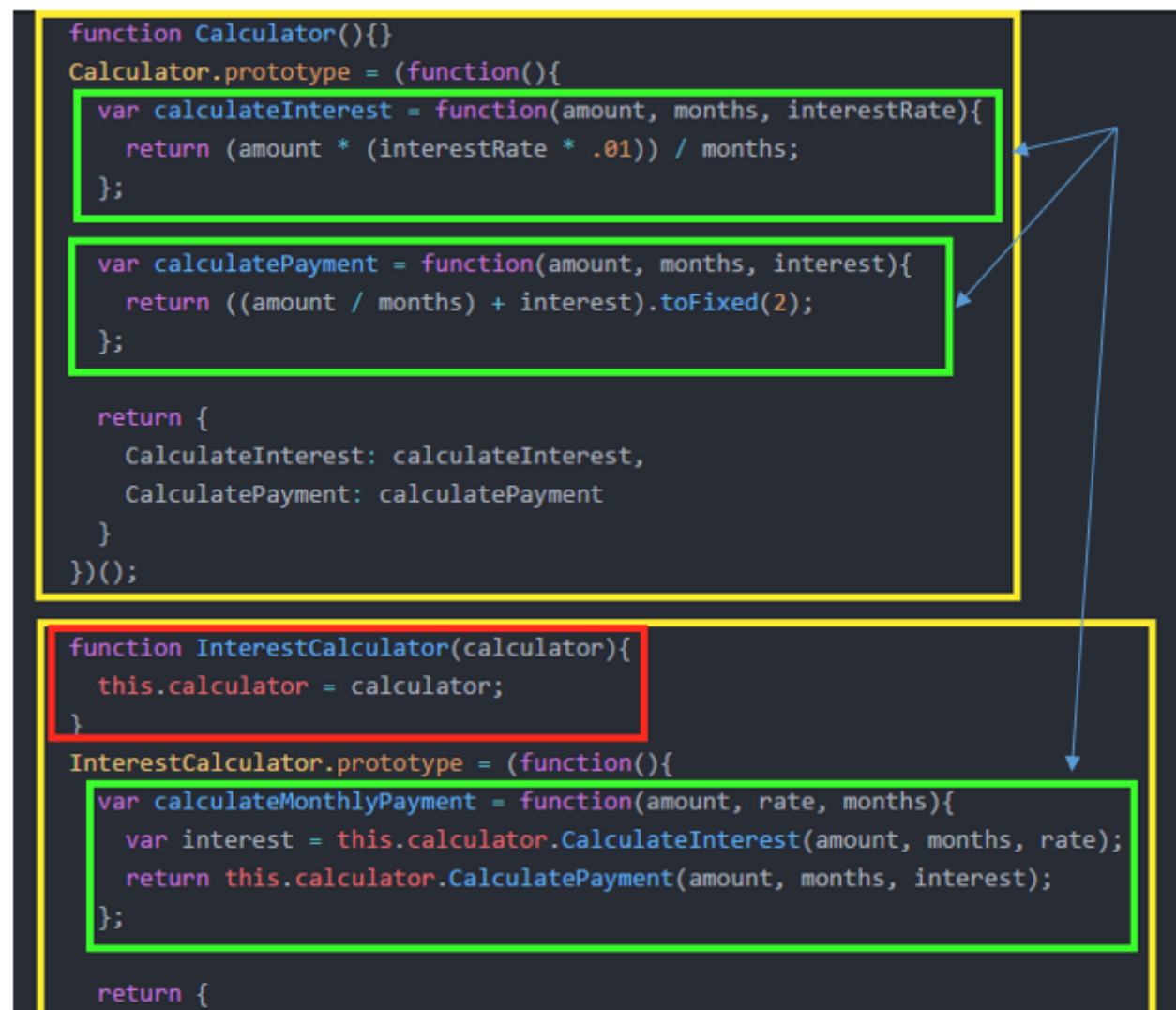
InterestCalculator.prototype = (function(){
  var calculateMonthlyPayment = function(amount, rate, months){
    var interest = this.calculator.CalculateInterest(amount, months, rate);
    return this.calculator.CalculatePayment(amount, months, interest);
  };

  return {
    CalculateMonthlyPayment: calculateMonthlyPayment
  };
})();

var interestCalculator = new InterestCalculator(
  new Calculator());

var payment = interestCalculator.CalculateMonthlyPayment(5000.00, 10.9, 96);
alert(payment);
```

Small, easily testable units of code



```
function Calculator(){}
Calculator.prototype = (function(){
  var calculateInterest = function(amount, months, interestRate){
    return (amount * (interestRate * .01)) / months;
  };

  var calculatePayment = function(amount, months, interest){
    return ((amount / months) + interest).toFixed(2);
  };

  return {
    CalculateInterest: calculateInterest,
    CalculatePayment: calculatePayment
  }
})();

function InterestCalculator(calculator){
  this.calculator = calculator;
}

InterestCalculator.prototype = (function(){
  var calculateMonthlyPayment = function(amount, rate, months){
    var interest = this.calculator.CalculateInterest(amount, months, rate);
    return this.calculator.CalculatePayment(amount, months, interest);
  };

  return {
```

```
    CalculateMonthlyPayment: calculateMonthlyPayment
  };
})();
|
var interestCalculator = new InterestCalculator(
  new Calculator());

var payment = interestCalculator.CalculateMonthlyPayment(5000.00, 10.9, 96);
alert(payment);
```

In these examples, we can pass pre-defined inputs, and assert that the output is the true result of the inputs.

```
describe("When I call CalculateMonthlyPayment", function() {
  var calculator, interestCalculator, result, amount, interestRate, months;
  var calculatePaymentResult = 1;
  var calculateInterestResult = 2;

  beforeEach(function() {
    amount = 5000; interestRate = 10.9; months = 96;
    calculator = {};

    spyOn(calculator, 'CalculateInterest').and.returnValue(calculateInterestResult);
    spyOn(calculator, 'CalculatePayment').and.returnValue(calculatePaymentResult);

    interestCalculator = new InterestCalculator(calculator);
    result = interestCalculator.CalculateMonthlyPayment(amount, interestRate, months);
  });

  it("should call CalculateInterest on the calculator", function() {
    expect(calculator.CalculateInterest).toHaveBeenCalledWith(amount, months, rate);
  });

  it("should call CalculatePayment on the calculator", function() {
    expect(calculator.CalculatePayment).toHaveBeenCalledWith(amount, months, calculatePaymentResult);
  });

  it("should return the result of calculate payment", function() {
    expect(result).toEqual(calculateInterestResult);
  });
});
```

Testing Tdd Acceptance Tests Software Development Test Automation

About Help Legal

Get the Medium app

