9

The Clean Method

How do we write software? Nowadays, of course, most of us sit at our laptop and type into an IDE. We operate in a loop.

1. We write a small bit of code (with the possible help of an AI).
2. We compile and test it.
3. We go back to step 1.

That loop might take anywhere from five seconds to an hour depending upon the discipline you follow.

Of course, that's not how it used to be. In the earliest days of programming, the loop was more like this.

1. We wrote a large bit of code by hand.
2. We desk-checked it by hand.
3. We submitted it to be encoded—by hand—into a machine-readable form.

4. We submitted it for compilation and test.

That loop often took between one and three days. The difference is stark. In the past, the investment per line of code was enormous. Nowadays, that cost is trivial.

When costs are high, we spend a lot of time planning up front in order to avoid waste. When costs are low, it makes more sense to explore by failing. In both cases, the idea of cleaning the code is generally ignored.

Make It Right

In the early days, cleaning code was inconvenient because the code was written by hand on sheets of paper. Erasing, cutting, and pasting sheets of paper is very inefficient, and just not fun—especially when small changes can spread through the body of the code in very inconvenient ways.

Nowadays, cleaning the code has been made much more convenient by our IDEs and refactoring tools. But we often bypass cleaning the code because that tight loop allows us to make progress so quickly that pausing to clean the code seems like a misuse of time. We drive hard to make the code work—and then we check the code in and move on to the next problem.

Over two decades ago, Kent Beck stated the simple rule that guides this book.

First, make it work.

Then, make it right.

There's more to this rule that we'll talk about later. For now, just look at those two lines. Most of us do the first part because we consider it to be the reason we were hired. We were hired to make code work. But many of us bypass the second part because it seems like a waste of time. After all, if we are good programmers, we should be writing right code all the time. We shouldn't need to have to stop and make it right.

Perhaps you are that good. Perhaps the code that comes out of your fingers is right the first time and every time. If so, you can put this book down.

However, most of us are not particularly adept at keeping two goals in our head at the same time. The primary goal of making the code work pushes aside the secondary goal of making the code right. Most of us find that the mental effort required to get the code to work is so great that we have no mental effort left to simultaneously make the code right.

Thus, by the time we get the code to work, the code is also a mess. And that's when the second line of Beck's rule comes into play. Once we have the code working, then we will have the mental capacity to clean it—to make it right.

Of course, we'd better have a way to ensure that in making the code right we don't break the fact that it works. It is not our goal to create clean code that doesn't work. How are we to get that assurance? How will we make sure that the act of cleaning does not break the code?

We will talk about testing disciplines in a later chapter. Suffice it to say that if we don't have a test suite that we can run quickly and that we implicitly trust to show that the code works, then we are not going to do a lot of cleaning.

Clean code depends on fast, convenient, and comprehensive tests.

So how do we make the code right? What's the process? It's another tight loop similar to writing the code.

1. We clean one little thing.
2. We run the tests.
3. If the tests fail, we revert.
4. Go back to step 1.

The timing of this loop is typically between five seconds and two minutes. The name of this loop is *refactoring*, and each cleaning step is *a refactoring*.

To quote Martin Fowler, refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."[1]

1. [Refactoring], p. 45.

How often should we transition between the writing loop and the refactoring loop? As often as possible. This implies an outer loop that includes both the loop to make it work and the loop to make it right.

1. Get something small to work.
    a. We write a small bit of code along with tests.
    b. We compile and test it.
    c. If it's not too messy, we go back to step 1a. Otherwise …
2. Clean it up.
    a. We clean one little thing.
    b. We run the tests.
    c. If the tests fail, we revert and go to step 2a.
    d. If there is more to clean, we go back to step 2a. Otherwise …
3. Go back to step 1.

The timing of this loop depends on your testing discipline and is on the order of a few minutes to an hour or so.

The bottom line is that, unless you are a multitalented mega being, writing clean code means that you are going to have to sit in that outer loop, patiently cleaning up after yourself, and never allowing the mess to get too big to clean.

Be like the sushi chef who never stops making sushi and who never stops cleaning the implements and the environment.

Example

Perhaps the best way to understand how to write clean code is to watch someone do it. So, in this chapter, we will walk through the procedure for writing a clean module. The technique will involve

- The testing discipline of TDD
- Refactoring
- Design principles
- Design patterns

We will begin with an imaginary Python application: the Tax Calculator for the mythical state of Bobolia. It's been a long time since I've written any Python, so bear with me.

Here's the first test.

```
——bobolia_taxes_test.py——
import unittest
import bobolia_taxes

class bobolia_tax_tests(unittest.TestCase):
    def test_simplest_case(self):
        tax_return = {"income": {"salary": 50000}}
        tax_calculator = bobolia_taxes.tax_calculator
        self.assertEqual(7500, tax_calculator.get_tax

if __name__ == '__main__':
    unittest.main()
```

In the Great State of Bobolia, the base tax rate is 15%. So someone earning 50,000 bobbles must pay 7,500 bobbles in tax. Note that the tax return is a simple map with an `income` key that contains yet another map with a `salary` key.

We can get this to pass with a simple degenerate module.

```
——bobolia_taxes.py——
class tax_calculator:
    def __init__(self, tax_return):
        self.tax_return = tax_return

    def get_tax(self):
        return 7500
```

Actually, I made it fail first by returning `0`, and then I made it pass by returning `7500`. This is the TDD way. First we see the test fail, and then we make it pass. If that sounds like a waste of time, let me just say that this simple practice has saved me a lot of debugging time.

But let's move on to the next tax law. Citizens of Bobolia who make no more than 30,000 bobbles per year do not have to pay any tax at all.

```
——bobolia_taxes_test.py——
class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalcul

    def test_simplest_case(self):
        tax_return = {"income": {"salary": 50000}}
        self.assertEqual(7500, self.tax_calculator.ge

    def test_30K_limit(self):
        tax_return = {"income": {"salary": 30000}}
        self.assertEqual(0, self.tax_calculator.get_t
```

Here you can see I've already begun to make some design changes. Rather than construct the `TaxCalculator` with the `tax_return`, I decided to pass the `tax_return` into the `get_tax` function. I also changed the names of the classes to comport with the PEP 8 standard.

Making these tests pass was pretty simple.

```
——bobolia_taxes.py——
class TaxCalculator:
    def __init__(self):
        pass

    def get_tax(self, tax_return):
        total_income = tax_return["income"]["salary"]
        if total_income > 30000:
            return total_income * 0.15
        return 0
```

Now, clearly this isn't fair. You can't have a citizen who earns 30,001 bobbles pay 4,500 in tax. Therefore, no citizen's after-tax income should be less than 30,000.

```
——bobolia_taxes_test.py——
class BoboliaTaxTests(unittest.TestCase):
```

```
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalcul

    def test_15pct_tax_rate(self):
        tax_return = {"income": {"salary": 50000}}
        self.assertEqual(7500, self.tax_calculator.ge

    def test_no_tax_for_30K_and_below(self):
        tax_return = {"income": {"salary": 30000}}
        self.assertEqual(0, self.tax_calculator.get_t

    def test_no_after_tax_income_less_than_30K(self):
        tax_return = {"income": {"salary": 30001}}
        self.assertEqual(1, self.tax_calculator.get_t
        tax_return = {"income": {"salary": 35000}}
        self.assertEqual(5000, self.tax_calculator.ge
```

No surprise here. I improved the names of the tests. Name improvement is something that often takes place once you know more about the application you are writing. Getting these tests to pass was pretty easy, though it forced me to move most of the older code around a bit.

```
———bobolia_taxes.py———
def get_tax(self, tax_return):
    total_income = tax_return["income"]["salary"]
    tax = total_income * 0.15
    after_tax_income = total_income - tax
    if after_tax_income < 30000:
        return total_income - 30000
    else:
        return tax
```

Now then, let's talk about the social demerit system. Bobolia is a free state. We believe that what our neighbors do is none of our darn business. However, there are behaviors that are detrimental to the state of Bobolia, and we believe citizens who engage in these behavior should compensate society for the damage they've done. So all citizens earn badbob points through the course of a year, and they must report these badbobs on their tax returns. (Don't cheat! We have ways …)

Badbob points are used in calculating taxes. So we need to add a `badbobs` key to the tax return. Oh, but that means we have to go back and

change all the tests.

Anytime you have to go back and change a bunch of tests, it means there's a flaw in the design of your tests. Let's fix that flaw before we proceed.

```
——bobolia_taxes_test.py——
def make_return(args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": 0}
    tax_return.update(args)
    return tax_return


class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalcul

    def test_15pct_tax_rate(self):
        tax_return = make_return({"income": {"salary"
        self.assertEqual(7500, self.tax_calculator.ge

    def test_no_tax_for_30K_and_below(self):
        tax_return = make_return({"income": {"salary"
        self.assertEqual(0, self.tax_calculator.get_t

    def test_no_after_tax_income_less_than_30K(self):
        tax_return = make_return({"income": {"salary"
        self.assertEqual(1, self.tax_calculator.get_t
        tax_return = make_return({"income": {"salary"
        self.assertEqual(5000, self.tax_calculator.ge
```

This simple change has decoupled the tests from the expectations of the production code. The `make_return` function will ensure that any omitted fields are given a reasonable default value. Thus, when new fields are added to the tax return, the existing tests are not likely to require changes.

Now, on with the badbob processing.

The purchase of alcoholic beverages is detrimental to society. So, if you spend 100 bobbles on whiskey, you should earn 100 badbobs. Alcohol badbobs are of class 1 and will be added to your tax at a rate of 50%. So

that 100-bobble bottle of whiskey will cost you another 50 bobbles at tax time.

```
——bobolia_taxes_test.py——
def make_return(args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": {"class1": ()}}
    tax_return.update(args)
    return tax_return


class BoboliaTaxTests(unittest.TestCase):
    …
    def test_class_1_badbobs(self):
        tax_return = make_return({"income": {"salary"
                                  "badbobs": {"class1
                                 )
        self.assertEqual(7550, self.tax_calculator.ge
```

While writing this test, I realized that the structure of the `badbobs` that I had placed in `make_return` was insufficient. So I changed that to add the `class1` key and the empty tuple. The test shows the extra `50` bobbles in tax for the 100 class 1 badbobs. Making this pass was easy, but a little unsettling.

```
——bobolia_taxes.py——
def get_tax(self, tax_return):
    total_income = tax_return["income"]["salary"]
    tax = total_income * 0.15
    after_tax_income = total_income - tax
    if after_tax_income < 30000:
        tax = total_income - 30000
    tax += sum(tax_return["badbobs"]["class1"]) / 2
    return tax
```

This function is getting messy. We could at least break it up into a couple of nicer functions.

```
——bobolia_taxes.py——
def get_tax(self, tax_return):
```

```
    total_income = tax_return["income"]["salary"]
    tax = self.determine_base_tax(total_income)
    tax += self.determine_badbob_adjustment(tax_retur
    return tax

def determine_badbob_adjustment(self, tax_return):
    return sum(tax_return["badbobs"]["class1"]) / 2

def determine_base_tax(self, total_income):
    tax = total_income * 0.15
    after_tax_income = total_income - tax
    if after_tax_income < 30000:
        tax = total_income - 30000
    return tax
```

That's a bit better. I'm not sure why I put these functions into a class. The class is just forcing me to add `self` everywhere. Maybe I should get rid of that class—but I've got a nagging feeling that the class is going to be important, even if for nothing more than the name space.

Perhaps that class will come in handy for allowing the methods to communicate through instance variables rather than arguments. Let's try that.

```
——bobolia_taxes.py——
class TaxCalculator:
    def __init__(self):
        self.tax_return = None

    def get_tax(self, tax_return):
        self.tax_return = tax_return
        tax = self.determine_base_tax()
        tax += self.determine_badbob_adjustment()
        return tax

    def determine_badbob_adjustment(self):
        return sum(self.tax_return["badbobs"]["class1

    def determine_base_tax(self):
        total_income = self.tax_return["income"]["sal
        tax = total_income * 0.15
        after_tax_income = total_income - tax
        if after_tax_income < 30000:
```

```
            tax = total_income - 30000
        return tax
```

I think that's a little better. I'm going to keep the class.

The Great State of Bobolia considers the purchase of gasoline-powered vehicles to be very detrimental to society. Such purchases earn class 2 badbobs at a rate of one badbob per bobble. Class 2 badbobs increase taxes according to the following table:

Class 2 badbobs	% of total income
1,001–10,000	5%
10,001–50,000	10%
>50,000	15%

——bobolia_taxes_test.py——

```
def make_return(args):
    tax_return = {"income": {"salary": 0},
                  "badbobs": {"class1": (),
                              "class2": ()}}
    if "income" in args:
        tax_return["income"].update(args["income"])
    if "badbobs" in args:
        tax_return["badbobs"].update(args["badbobs"])
    return tax_return


class BoboliaTaxTests(unittest.TestCase):
    …
    def test_class_2_badbob_purchases_under_1001(self
        tax_return = make_return({"income": {"salary"
                                  "badbobs": {"class2
                                 )
        self.assertEqual(7500, self.tax_calculator.ge

    def test_class_2_badbob_purchases_under_10001(sel
        tax_return = make_return({"income": {"salary"
```

```
                                    "badbobs": {"class2
                                   )
        self.assertEqual(7500 + 2500,
                         self.tax_calculator.get_tax(

    def test_class_2_badbob_purchases_under_50001(sel
        tax_return = make_return({"income": {"salary"
                                    "badbobs": {"class2
                                   )
        self.assertEqual(7500 + 5000,
                         self.tax_calculator.get_tax(

    def test_class_2_badbob_purchases_over_50000(self
        tax_return = make_return(
          {"income": {"salary": 50000},
           "badbobs": {"class2": (10000,40000,1000)}}
          )
        self.assertEqual(7500 + 7500,
                         self.tax_calculator.get_tax(
```

These tests show the four table entries pretty clearly. I made these tests pass one at a time and ended up with the following production code.

```
——bobolia_taxes.py——
class TaxCalculator:
    def __init__(self):
        self.tax_return = None
        self.total_income = None

    def get_tax(self, tax_return):
        self.tax_return = tax_return
        tax = self.determine_base_tax()
        tax += self.determine_badbob_adjustment()
        return tax

    def determine_badbob_adjustment(self):
        class1 = sum(self.tax_return["badbobs"]["clas
                 class1_adjustment = class1 /2
                 class2 = sum(self.tax_return["badbob
        if class2<1001:
            class2_adjustment = 0
        elif class2<10001:
            class2_adjustment = self.total_income * 0
        elif class2<50001:
```

```
            class2_adjustment = self.total_income * 0
        else:
            class2_adjustment = self.total_income * 0
        return class1_adjustment + class2_adjustment

    def determine_base_tax(self):
        self.total_income = self.tax_return["income"]
        tax = self.total_income * 0.15
        after_tax_income = self.total_income - tax
        if after_tax_income < 30000:
            tax = self.total_income - 30000
        return tax
```

This is pretty awful. I expect all those numbers to change over time. I even expect the number of rows in the table to change. Maintaining this is going to be labor intensive and error prone. We can probably do better.

```
——bobolia_taxes.py——

class TaxCalculator:
    …

    def get_tax(self, tax_return):
        self.tax_return = tax_return
        tax = self.determine_base_tax()
        tax += BadBobAdjuster(self).get_adjustment()
        return tax

    def determine_base_tax(self):
        self.total_income = self.tax_return["income"]
        tax = self.total_income * 0.15
        after_tax_income = self.total_income - tax
        if after_tax_income < 30000:
            tax = self.total_income - 30000
        return tax


class BadBobAdjuster:
    def __init__(self, tax_calculator):
        self.tax_calculator = tax_calculator

    def get_adjustment(self):
        class1_adjustment = self.class1_adjustment()
        class2_adjustment = self.get_class2_badbob_ad
```

```
            return class1_adjustment + class2_adjustment

        def get_class2_badbob_adjustment(self):
            class2 = sum(self.tax_calculator.tax_return["
            class2_rate = self.get_class2_rate(class2)

            return class2_rate * self.tax_calculator.tota

        def get_class2_rate(self, class2):
            class2_rate = 0
            class2_table = ((1001, 0),
                            (10001, 0.05),
                            (50001, 0.1),
                            ("max", 0.15))
            for row in class2_table:
                if row[0] == "max" or class2 < row[0]:
                    class2_rate = row[1]
                    break
            return class2_rate

        def class1_adjustment(self):
            class1 = sum(self.tax_calculator.tax_return["
            class1_adjustment = class1 / 2
            return class1_adjustment
```

So I created the table and the loop to extract the rate. Then, I thought that the whole badbob idea should be moved into its own class, just to have a namespace for it.

Notice what's happening here. The production code is evolving and morphing to become more generic. Parts of it are being peeled off into their own classes and separated from each other. Meanwhile, the tests are simply growing in a linear fashion. With each new test case the test class becomes more specific.

As the tests get more specific, the code gets more generic.

This strategy keeps the tests from knowing too much about the production code. When changes to the production code are made, fewer changes need to be made to the tests.

But now we have a problem. If a citizen saves up to buy an expensive car, they might end up with a tax rate that reduces their after-tax income to

significantly less than 30,000 bobbles. In one sense, this is fair. Citizens who harm society ought to compensate for that harm. On the other hand, forcing a citizen to live on less than, say, 20,000 bobbles a year would be cruel. So let's limit the effect of the badbobs so that no badbob adjustment will reduce after-tax income to below 20,000 bobbles.

```
——bobolia_taxes_test.py——
…
def test_badbobs_do_not_reduce_after_tax_income_below
    tax_return = make_return({"income": {"salary": 20
                               "badbobs": {"class2": (
                              )
    self.assertEqual(0, self.tax_calculator.get_tax(t
```

This failed in an unexpected way. It returned a tax of −9,000 bobbles. Negative tax! I bet the citizens would love that. On the other hand, the BRS (Bobolia Revenue Service) would not be pleased. This was a bug. I fixed it by putting in a guard.

```
——bobolia_taxes.py——
…
def determine_base_tax(self):
    self.total_income = self.tax_return["income"]["sa
    if self.total_income <= 30000:
        return 0
…
```

Yes, TDD is not a magic bullet. You will still write some bugs. Fortunately, as shown in this case, other tests are likely to uncover those bugs. With that fixed, I could get the test to pass pretty easily.

```
——bobolia_taxes.py——
…
def get_tax(self, tax_return):
    self.tax_return = tax_return
    base_tax = self.determine_base_tax()
    badbob_adjustment = BadBobAdjuster(self).get_adju
    total_tax = base_tax + badbob_adjustment
    after_tax_income = self.total_income - total_tax
```

```
        if self.total_income < 20000:
            return 0
        elif after_tax_income < 20000:
            total_tax = self.total_income - 20000
        return total_tax
```

Those two guards that I just added are a bit ugly. I think I can just use the `max` function instead.

```
——bobolia_taxes.py——
…
    def get_tax(self, tax_return):
        self.tax_return = tax_return
        base_tax = self.determine_base_tax()
        badbob_adjustment = BadBobAdjuster(self).get_
        total_tax = base_tax + badbob_adjustment
        after_tax_income = self.total_income - total_
        if after_tax_income < 20000:
            total_tax = max(0, self.total_income - 20
        return total_tax

    def determine_base_tax(self):
        self.total_income = self.tax_return["income"]
        tax = self.total_income * 0.15
        after_tax_income = self.total_income - tax
        if after_tax_income < 30000:
            tax = max (0, self.total_income - 30000)
        return tax
```

That's better.

Now then … A year has gone by, and everyone is paying their taxes. But after some riots and really mean tweets, we have decided that here in the Great State of Bobolia income inequality is a social harm. Low earners were paying too much, and high earners were paying too little. Those who earn too many bobbles should simply pay a higher tax rate than those who earn less. Not just more taxes, but a higher rate. We've determined that the following table would represent "their fair share":

Income	Tax rate
0–30,000	0%
30,001–100,000	15% of the amount over 30,000
100,001–250,000	10,500 + 20% of the amount over 100,000
250,001–500,000	40,500 + 30% of the amount over 250,000
>500,000	115,500 + 40% of the amount over 500,000

Of course, this changes the way taxes are calculated, so many of our previous tests are going to need deleting or changing. Since most of our tests use the salary of 50,000, let's create a constant for the tax on 50,000, named `tax_50K`, and then we'll use that in all the tests. That way, if the tax calculation changes again, we only have to change the constant instead of all the tests.

```
——bobolia_taxes_test.py——
…
tax_50K = 3000


class BoboliaTaxTests(unittest.TestCase):
    def setUp(self):
        self.tax_calculator = bobolia_taxes.TaxCalcul
        self.tax_return = None

    def make_return(self, args):
        tax_return = {"income": {"salary": 0},
                      "badbobs": {"class1": (),
                                  "class2": ()}}
        if "income" in args:
            tax_return["income"].update(args["income"
        if "badbobs" in args:
            tax_return["badbobs"].update(args["badbob
        self.tax_return = tax_return

    def make_simple_return(self, salary):
```

```
        self.make_return({"income": {"salary": salary

    def get_tax(self):
        return self.tax_calculator.get_tax(self.tax_r

    def test_class_1_badbobs(self):
        self.make_return({"income": {"salary": 50000}
                          "badbobs": {"class1": (100,
                         )
        self.assertEqual(tax_50K + 50, self.get_tax()

    def test_class_2_badbob_purchases_under_1001(self
        self.make_return({"income": {"salary": 50000}
                          "badbobs": {"class2": (400,
                         )
        self.assertEqual(tax_50K, self.get_tax())

    def test_class_2_badbob_purchases_under_10001(sel
        self.make_return({"income": {"salary": 50000}
                          "badbobs": {"class2": (1000
                         )
        self.assertEqual(tax_50K + 2500, self.get_tax

    def test_class_2_badbob_purchases_under_50001(sel
        self.make_return({"income": {"salary": 50000}
                          "badbobs": {"class2": (1000
                         )
        self.assertEqual(tax_50K + 5000, self.get_tax

    def test_class_2_badbob_purchases_over_50000(self
        self.make_return({"income": {"salary": 50000}
                          "badbobs": {"class2": (1000
                         )
        self.assertEqual(tax_50K + 7500, self.get_tax

    def test_badbobs_do_not_reduce_after_tax_income_b
        self.make_return({"income": {"salary": 20000}
                          "badbobs": {"class2": (1000
                         )
        self.assertEqual(0, self.get_tax())

    def test_0_30_tax_bracket(self):
        self.make_simple_return(15000)
        self.assertEqual(0, self.get_tax())
        self.make_simple_return(30000)
```

```
            self.assertEqual(0, self.get_tax())

        def test_30_100_tax_bracket(self):
            self.make_simple_return(30001)
            self.assertEqual(0, self.get_tax())
            self.make_simple_return(50000)
            self.assertEqual(tax_50K, self.get_tax())
            self.make_simple_return(100000)
            self.assertEqual(10500, self.get_tax())

        def test_100_250_tax_bracket(self):
            self.make_simple_return(100001)
            self.assertEqual(10500, self.get_tax())
            self.make_simple_return(200000)
            self.assertEqual(30500, self.get_tax())
            self.make_simple_return(250000)
            self.assertEqual(40500, self.get_tax())

        def test_250_500_tax_bracket(self):
            self.make_simple_return(250001)
            self.assertEqual(40500, self.get_tax())
            self.make_simple_return(500000)
            self.assertEqual(115500, self.get_tax())

        def test_500_up_tax_bracket(self):
            self.make_simple_return(500001)
            self.assertEqual(115500, self.get_tax())
            self.make_simple_return(1000000)
            self.assertEqual(315500, self.get_tax())
```

I made quite a few changes to the tests to try to keep the noise down. I moved the `make_return` method into the test class, and I added `make_simple_return` and `get_tax`. I also made the `tax_return` into an instance variable of the test class so that I didn't have to keep assigning and passing it.

The production code is a relatively simple change to the `determine_base_tax` method.

```
———bobolia_taxes.py———
…
def determine_base_tax(self):
    self.total_income = self.tax_return["income"]["sa
```

```
        if self.total_income <= 30000:
            return 0
        elif self.total_income <= 100000:
            return round(0.15 * (self.total_income - 3000
        elif self.total_income <= 250000:
            return round(0.2 * (self.total_income - 10000
        elif self.total_income <= 500000:
            return round(0.3 * (self.total_income - 25000
        else:
            return round(0.4 * (self.total_income - 50000
    …
```

Those tests are pretty ugly. Let's clean them up by using composed assertions.

```
    ——bobolia_taxes_test.py——
        …

        def assert_tax_with_badbobs(self, income, badbobs
            self.make_return({"income": {"salary": income
                              "badbobs": badbobs})
            self.assertEqual(tax, self.get_tax())

        def test_class_1_badbobs(self):
            self.assert_tax_with_badbobs(50000,
                                         {"class1": (100,
                                         tax_50K + 50)

        def test_class_2_badbob_purchases_under_1001(self
            self.assert_tax_with_badbobs(50000,
                                         {"class2": (400,
                                         tax_50K)

        def test_class_2_badbob_purchases_under_10001(sel
            self.assert_tax_with_badbobs(50000,
                                         {"class2": (1000
                                         tax_50K + 2500)

        def test_class_2_badbob_purchases_under_50001(sel
            self.assert_tax_with_badbobs(50000,
                                         {"class2": (1000
                                         tax_50K + 5000)

        def test_class_2_badbob_purchases_over_50000(self
```

```
        self.assert_tax_with_badbobs(50000,
                                     {"class2": (1000
                                     tax_50K + 7500)

    def test_badbobs_do_not_reduce_after_tax_income_b
        self.assert_tax_with_badbobs(20000,
                                     {"class2": (1000
                                     0)

    def assert_tax_for(self, income, tax):
        self.make_simple_return(income)
        self.assertEqual(tax, self.get_tax())

    def test_0_30_tax_bracket(self):
        self.assert_tax_for(15000, 0)
        self.assert_tax_for(30000, 0)

    def test_30_100_tax_bracket(self):
        self.assert_tax_for(30001, 0)
        self.assert_tax_for(50000, tax_50K)
        self.assert_tax_for(100000, 10500)

    def test_100_250_tax_bracket(self):
        self.assert_tax_for(100001, 10500)
        self.assert_tax_for(200000, 30500)
        self.assert_tax_for(250000, 40500)

    def test_250_500_tax_bracket(self):
        self.assert_tax_for(250001, 40500)
        self.assert_tax_for(500000, 115500)

    def test_500_up_tax_bracket(self):
        self.assert_tax_for(500001, 115500)
        self.assert_tax_for(1000000, 315500)
    …
```

That's much better. Those tests are now in a position to become parametric (table driven) should the need arise—but I don't think we're there yet.

On the other hand, the production code is crying out for a table-driven approach. You don't think we've seen the last of the tax brackets, do you? So let's take care of that.

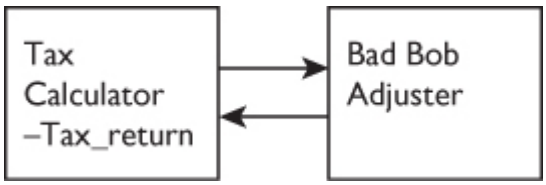
```
——bobolia_taxes.py——
…
def determine_base_tax(self):
    tax_brackets = ((30000, 0),
                    (100000, .15),
                    (250000, .20),
                    (500000, .30),
                    (None, .40))

    self.total_income = self.tax_return["income"]["sa

    previous_maximum = 0
    offset = 0
    for maximum, rate in tax_brackets:
        if maximum is None or self.total_income <= ma
            return round(
              rate *
              (self.total_income - previous_maximum)
              offset)
        else:
            offset = (maximum - previous_maximum) * r
            previous_maximum = maximum
…
```

OK, that's better. Any new bracket can be added easily.

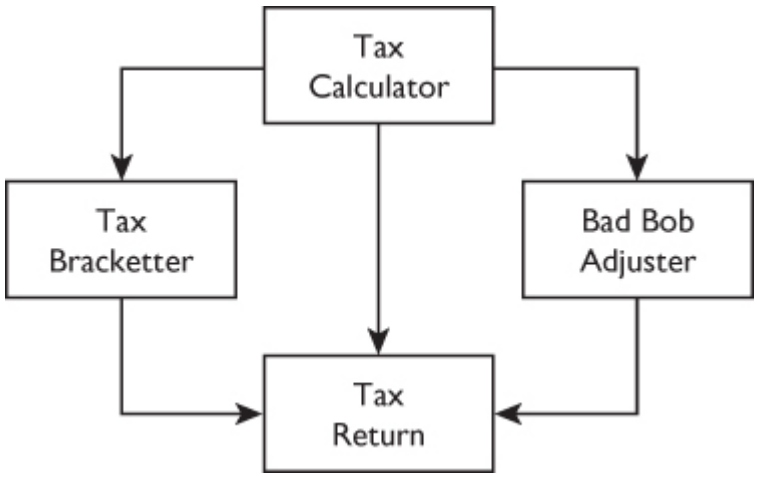
At this point, you might ask whether the tests ought to use the same `tax_brackets` table. I'm usually not in favor of allowing the tests to know the details of the production code. The whole point of tests is that they are a second statement of intent. It makes little sense to use the production code to test the production code. So I prefer to do the calculations manually and enter them into the tests.

Considering the Design and Architecture

Now let's take a step back. So far you have watched me write a small module using a process that keeps it "clean" and tested as it grows and is changed. I hope the process has been illuminating. The module has grown into two classes that have the following relationships.



First of all, I don't like that reciprocal relationship. It only exists because the `BadBobAdjuster` uses the `tax_return` field of the `TaxCalculator`. Second, I think this is a bit asymmetric. Why does the `BadBobAdjuster` deserve its own special class whereas the tax bracket calculations are buried in the `TaxCalculator`? Let's clean that up a bit. I'd like it to look like this.



This change requires only minor changes to the tests, so our test design is holding up.

```
——bobolia_taxes_test.py——
…
def make_return(self, args):
    tax_return_data = {"income": {"salary": 0},
                       "badbobs": {"class1": (),
                                   "class2": ()}}
    if "income" in args:
        tax_return_data["income"].update(args["income
    if "badbobs" in args:
        tax_return_data["badbobs"].update(args["badbo
    self.tax_return = bobolia_taxes.TaxReturn(tax_ret
…

——bobolia_taxes.py——
class TaxCalculator:
    def get_tax(self, tax_return):
        total_income = tax_return.get_total_income()
```

```
        base_tax = TaxBracketter(tax_return).determir
        badbob_adjustment = BadBobAdjuster(tax_returr
        total_tax = base_tax + badbob_adjustment
        after_tax_income = total_income - total_tax
        if after_tax_income < 20000:
            total_tax = max(0, total_income - 20000)
        return total_tax


class TaxBracketter:
    tax_brackets = ((30000, 0),
                    (100000, .15),
                    (250000, .20),
                    (500000, .30),
                    (None, .40))

    def __init__(self, tax_return):
        self.tax_return = tax_return

    def determine_base_tax(self):
        total_income = self.tax_return.get_total_incc

        previous_maximum = 0
        offset = 0
        for maximum, rate in self.tax_brackets:
            if maximum is None or total_income <= max
                return round(rate *
                             (total_income - previous_
                             offset)
            else:
                offset = (maximum - previous_maximum)
                previous_maximum = maximum


class BadBobAdjuster:
    def __init__(self, tax_return):
        self.tax_return = tax_return
        self.total_income = tax_return.get_total_incc

    def get_adjustment(self):
        class1_adjustment = self.class1_adjustment()
        class2_adjustment = self.get_class2_badbob_ad
        return class1_adjustment + class2_adjustment

    def get_class2_badbob_adjustment(self):
```

```
            class2 = self.tax_return.get_total_class2_bac
            class2_rate = self.get_class2_rate(class2)
            return class2_rate * self.total_income

        class2_table = ((1001, 0),
                        (10001, 0.05),
                        (50001, 0.1),
                        (None, 0.15))

        def get_class2_rate(self, class2):
            for limit, rate in self.class2_table:
                if limit is None or class2 < limit:
                    return rate
            return None

        def class1_adjustment(self):
            class1 = self.tax_return.get_total_class1_bac
            return class1 / 2


    class TaxReturn:
        def __init__(self, tax_return_data):
            self.tax_return_data = tax_return_data

        def get_total_income(self):
            return self.tax_return_data["income"]["salary

        def get_total_class1_badbobs(self):
            return sum(self.tax_return_data["badbobs"]["c

        def get_total_class2_badbobs(self):
            return sum(self.tax_return_data["badbobs"]["c
```

At this point, you might be accusing this design of *classitis*[2]—the tendency to create too many classes and thereby complicate the design. I disagree. The three new classes do not add any complexity that wasn't already there. They just move that complexity into nicely named places.

---

2. [APOD], p. 26.

A place for everything, and everything in its place!

One major advantage of the `TaxReturn` class is that it hides the internal data structure of the `tax_return_data` from the rest of the application. When there are changes to that data structure, the other modules in the application will not need to be changed.

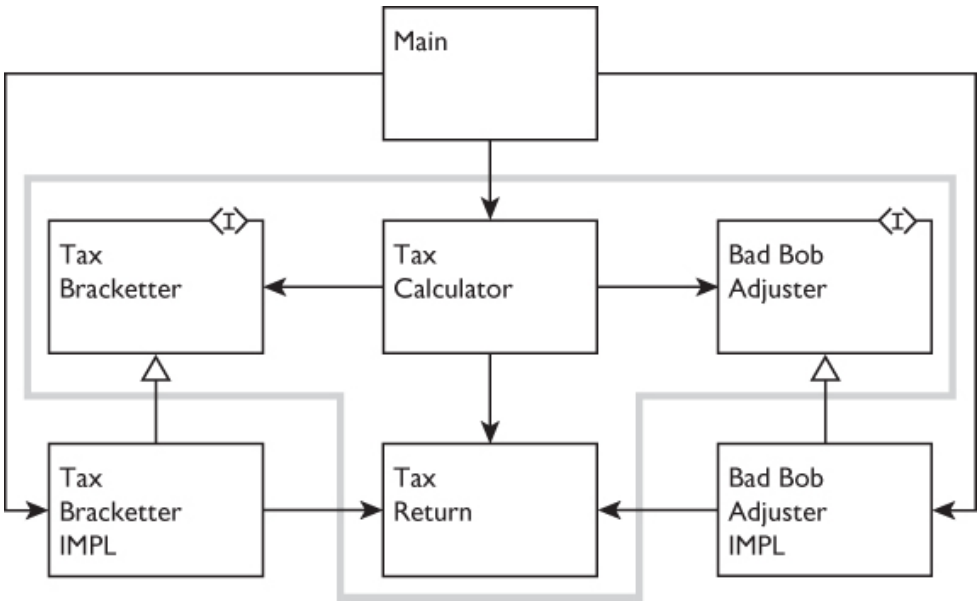
In any case, the advantage of those classes should become evident soon; because it's time for us to take yet another step back and think about the overall architecture of this system.

It seems to me that this system has two axes of change so far. There's the computation of the base tax, which is done by the `TaxBracketter` ; and then there's the computation of the social demerit score, accomplished by the `BadBobAdjuster` . If we expect this system to grow (and what tax system doesn't grow?), then we may want to establish these axes of change within the architecture of our system. Or, to say that differently, we may wish to draw architectural boundaries that isolate and protect them.

We'll use the Single Responsibility Principle (SRP) to do that, and we'll use the Dependency Inversion Principle (DIP) to create a structure like this.[3]

---

3. See Chapter 19, "The SOLID Principles."



Note the architectural boundary drawn around the four central classes. Notice also that every arrow that crosses that boundary points *inward*, following the Dependency Rule[4] of architecture. Inside that boundary is the

high-level policy of this system. Outside are lower-level details. The `main` component is the lowest level in the system. It creates the implementations of the `TaxBracketter` and `BadBobAdjuster` and hands them to the `TaxCalculator` prior to any taxes being calculated.

---

4. See Chapter 26, "The Clean Architecture."

The `TaxCalculator` uses the Strategy[5] pattern to delegate the bracket and demerit calculations to the appropriate objects. The canonical form of Strategy calls for the strategy implementations to derive from interfaces. However, in Python, interfaces do not need to exist as independent source code entities. Python is dynamically typed, and so polymorphic dispatch is through duck types. So the "interfaces" shown above are just the method signatures that the `TaxCalculator` expects to invoke.

---

5. [GOF95].

The role of the main component will be played by our tests.

```
——bobolia_taxes_test.py——
def setUp(self):
    self.tax_bracketter = tax_bracketter.TaxBrackette
    self.badbob_adjuster = badbob_adjuster.BadBobAdju
    self.tax_calculator = tax_calculator.TaxCalculato
        self.tax_bracketter, self.badbob_adjuster)
    self.tax_return = None
```

The `TaxCalculator` and `TaxReturn` exist in their own module.

```
——tax_calculator.py——
class TaxCalculator:
    def __init__(self, tax_bracketter, badbob_adjuste
        self.tax_bracketter = tax_bracketter
        self.badbob_adjuster = badbob_adjuster

    def get_tax(self, tax_return):
        total_income = tax_return.get_total_income()
        base_tax = self.tax_bracketter.determine_base
```

```
        badbob_adjustment =
          self.badbob_adjuster.get_adjustment(tax_ret
        total_tax = base_tax + badbob_adjustment
        after_tax_income = total_income - total_tax
        if after_tax_income < 20000:
            total_tax = max(0, total_income - 20000)
        return total_tax

class TaxReturn:
    def __init__(self, tax_return_data):
        self.tax_return_data = tax_return_data

    def get_total_income(self):
        return self.tax_return_data["income"]["salary

    def get_total_class1_badbobs(self):
        return sum(self.tax_return_data["badbobs"]["c

    def get_total_class2_badbobs(self):
        return sum(self.tax_return_data["badbobs"]["c
```

Then come the `TaxBracketter` and `BadBobAdjuster` modules.

```
———tax_bracketter.py———
class TaxBracketter:
    tax_brackets = ((30000, 0),
                    (100000, .15),
                    (250000, .20),
                    (500000, .30),
                    (None, .40))

    def determine_base_tax(self, tax_return):
        total_income = tax_return.get_total_income()

        previous_maximum = 0
        offset = 0
        for maximum, rate in self.tax_brackets:
            if maximum is None or total_income <= max
                return round(rate *
                             (total_income - previous
                             offset)
            else:
                offset = (maximum - previous_maximum)
                previous_maximum = maximum
```

```
——badbob_adjuster.py——
class BadBobAdjuster:
    def __init__(self):
        self.total_income = None
        self.tax_return = None

    def get_adjustment(self, tax_return):
        self.tax_return = tax_return
        self.total_income = tax_return.get_total_incc
        class1_adjustment = self.class1_adjustment()
        class2_adjustment = self.get_class2_badbob_ad
        return class1_adjustment + class2_adjustment

    def get_class2_badbob_adjustment(self):
        class2 = self.tax_return.get_total_class2_bad
        class2_rate = self.get_class2_rate(class2)
        return class2_rate * self.total_income

    class2_table = ((1001, 0),
                    (10001, 0.05),
                    (50001, 0.1),
                    (None, 0.15))

    def get_class2_rate(self, class2):
        for limit, rate in self.class2_table:
            if limit is None or class2 < limit:
                return rate
        return None

    def class1_adjustment(self):
        class1 = self.tax_return.get_total_class1_bad
        return class1 / 2
```

Conclusion

Is this clean? I'm pretty happy with it, though I'm not a Python programmer, so I've likely violated a lot of conventions and probably have not used the most efficient techniques. Still, I think some major goals have been accomplished. We've separated the system into at least four major components: `main`, `TaxCalculator`, `TaxBracketter`, and

`BadBobAdjuster`. And we placed those components behind appropriate architectural boundaries.

Our tests have survived a significant amount of change and have showed themselves to be resilient. The same is true of our basic algorithms. Changes to one are very unlikely to affect the others. We've also built in some flexibility by using tables to drive the algorithms.

As the tests grew in specificity, the production code grew in generality.

Some folks might complain about the names. I dunno, I kinda like `TaxBracketter`[6] and `BadBobAdjuster`. But if you play with this code, you might find some better names.

---

6. Despite the spelling error.

I have to say that I was pleased with IntelliJ's Python plug-in and Copilot's ability to help someone who hadn't written Python in over 20 years. I also have to say that I was impressed with the responsiveness of the tests. I'm used to JVM startup times, and it's a joy to have the test results pop up on the screen before my fingers have left the test button.