

## Chapter 5

# Test-first, by Intention

*Chet and Ron do a small task test first, trying always to express intention in the code rather than algorithm.*

Chet and I wanted to give a short demonstration of test-first programming. We decided to work on an actual problem that I'd encountered at a client location.

There are two key things to watch for in this example. First, we only write new code when we have a test that doesn't work. We call this *test-first programming*.

Second, we don't think much at all about how to do a thing, we think about what we have to do. We call this *programming by intention*. You just write code as if someone had written the hard method for you and you just had to send the message.

The task is this: we have two collections of Sum objects. A Sum has a name (a string) and an amount (a number). The output is to be a single collection of new Sums. If a Sum of the same name appears in each collection, the new one should have the same name, and the total of their amounts. If a Sum appears in only one collection, the new one should have that amount. The order of the output should be the order

of the first collection, followed by any elements from the second that didn't occur in the first. For example,

First	Second	Result
A 1 C 2	A 10 B 3	A 11 C 2 B 3

Chet asked a few questions, mostly about whether we had any code to start with. I said that it had been ugly and didn't work, so we decided to start from scratch.

We were working on a new machine, so we started by defining a simple *Sum* object with a name and an amount. We didn't write tests for it because it was pretty trivial and it was just a foil in our real example. For those who don't speak Smalltalk, we'll give a little commentary.

```
--Sum  
10:25:00
```

The two lines that look like the above just name the class we're putting code into, and the time we did it. They aren't part of Smalltalk, they're just here to give you a sense of how long things took. Here's the class definition:

```
--Sum  
10:25:00  
Object subclass: #Sum  
instance variables: 'name amount'
```

This defines a new class named *Sum*, with instance variables *name* and *amount*. In Smalltalk, you don't have to define the types of the variables.

Now we build the Constructor Method for the class. This is a method definition. Each method definition starts with the name of the method, tabbed out one tab stop. A method definition in Smalltalk has a name which consists of one or more keywords. This one is

*name:amount:*. The method definition also includes the names of the parameters. By convention, these names suggest the types of the parameters, but they are just names to be used in the code that follows.

The method itself starts on the next line, indented. So we're defining *name:amount:*. The definition is

```
^self new  
  setName: aString  
  amount: aNumber
```

which means (*self new*) create a new instance of this class (*Sum*), then send it the message

```
setName: aString  
amount: aNumber
```

which is what we call a Constructor Parameter method. The hat character (^) means "answers"., or "returns".

We start every class definition this same way, with a Constructor and Constructor Parameter method. It gets you going smoothly when you always start a class the same way. Here's the whole method:

```
---Sum class  
10:25:00  
name: aString amount: aNumber  
  ^self new  
    setName: aString  
    amount: aNumber
```

The next step is always to define the Constructor Parameter method we just sent. (Smalltalk doesn't mind if you use a method before you define it, although some versions will give you a warning about it.)

```
---Sum  
10:25:15  
setName: aString amount: aNumber  
  name := aString.  
  amount := aNumber
```

This method just assigns the two parameters to the corresponding instance variables. Now the instance is initialized. We also build accessors for the instance variables, because I happen to know we will need them. This was a mistake, strictly speaking, but I was building this class from memory.

```

---Sum
10:25:30
name
  ^name

```

This method is named *name*, and it answers (returns) the *name* instance variable. This method is an accessor for *name*. Strictly speaking we wouldn't have put this in, except that I was building this class from memory as a "given".

We also built an accessor for *amount* while we were there.

```

---Sum
10:25:40
amount
  ^amount

```

OK, enough warmup. It's time to write our object, which we have decided to call *Summarizer*. We began by making a test class named *SummarizerTest*. Here's the conversation we had as we pair programmed our way along.

Chet: What shall we test first?

Ron: Let's just make an empty one. That should answer an empty collection.

Chet writes the test. He assumes that there is a method already in existence named *emptySummarizer*. This keeps him focused on the immediate task, making an empty one and making sure that its result is empty.

```
--- SummarizerTest
10:32:08
testEmpty
  | summarizer |
  summarizer := self emptySummarizer.
```

This is what we call *programming by intention*. Instead of thinking about how to create an empty *Summarizer* and maybe writing it in line, Chet just expresses his intention, to have an empty *Summarizer*. This makes the code more clear, but more important it keeps you moving smoothly because you don't have to shift gears to think about a subordinate detail while you code the method you're on. You'll see that we do that right along.

Now Chet pauses before writing the first *should*: to discuss what to do.

Chet: How shall we get the answer?

Ron: Let's just send "*summarize*" to the Summarizer.

Chet: How about "*summary*"?

Ron: OK, that's better.

Note that here we are defining a key element of Summarizer's protocol: the method you send to make it actually do its thing. It's great to do this in a test, because you are actually using the object, and that gives you a much better chance of defining a useful and clear interface.

```
--- SummarizerTest
10:32:08
testEmpty
  | summarizer |
  summarizer := self emptySummarizer.
  self should: [summarizer summary isEmpty]
```

The test method above just creates an emptySummarizer (or would if that method existed) and tests to be sure its summary is empty. Naturally, we run the test. It doesn't work, because *emptySummary* isn't defined. We expected that, but we like to run the test every chance we get – it's a good habit to have.

Chet: How do we create one of these deals?

Ron: Let's define the class, give it two instance variables, call them *first* and *second*.

Chet: I hate those names.

Ron: Me too, but I can't think of anything better.

--Summary

10:32:45

Object subclass: #Summarizer

instance variables: 'first second'

We immediately go back to the test, not even making a Constructor Method, because we don't know yet what we want it to look like. Looking at the blank test method for emptySummarizer, we discuss it:

Ron: So we need a Summarizer with two collections of Sums.

Chet: first:second:?

Ron: How about just with:with:?

Chet: OK

The name *with:with:* has good history in Smalltalk when you are creating some object with a couple of items that aren't particularly differentiated except for order. On the other hand, the name isn't very communicative. Ron may have been wrong with this suggestion.

Chet writes the method that way, putting two empty arrays into the Summarizer.

```
--- SummarizerTest
```

```
10:33:06
```

```
emptySummarizer
```

```
  ^Summarizer
```

```
    with: #()
```

```
    with: #()
```

We run the test again. It breaks, of course, because Summarizer doesn't understand the *with:with* method. Chet just types it in, along

with the corresponding Constructor Parameter method. Again, it's a rote thing he just knows how to do because we always do it this way.

```
--- Summarizer class
10:35:03
with: firstCollection with: secondCollection
    ^self new
        setFirstCollection: firstCollection
        secondCollection: secondCollection
```

```
--- Summarizer
10:35:54
setFirstCollection: firstCollection secondCollection: secondCollection
first := firstCollection.
second := secondCollection
```

We run the test again. This time it breaks because the Summarizer doesn't understand "summary". Chet hasn't an idea how to do the method so he just creates one with a halt in it, then runs the test again to get into the debugger.

```
--- Summarizer
10:37:18
summary
    self halt
```

In the debugger we look around a bit.

Chet: OK, the two input collections are empty.

Ron: What should we answer? I don't know what to do next.

Chet: This will work.

```
--- Summarizer
10:38:11
summary
    ^first, second
```

Chet has answered the two inputs collections concatenated together. I don't know why he picked this but I can see it will "work". That is, the result will be an empty collection, so the test will run. This is nearly the simplest thing that could possibly work. Simpler would have been to

just answer a literal empty collection, but Chet was just getting his hands on the variables a bit.

Our first test runs. Even though the object is clearly wrong, we don't have a broken test to make us fix it. So we write another test.

Ron: Let's test the example I wrote down. (The one in the table up above.)

Chet: OK.

Chet starts the method, creates a temp, starts an assignment to it. He pauses and types a left paren. I know that he's about to try to construct the test *Summarizer* right there.

Ron: Just send *abcSummarizer*.

Chet: Right.

Ron: We have to remember to tell them how important this is.

Chet: This is one of the most important things you've taught me. I just write code that assumes that ten seconds ago someone already wrote the method I need, like the *abcSummarizer* method.

Here again, we're talking about *intention*. We just say what we want, not how to do it. When we get there, it's always easy.

```
--- SummarizerTest
10:39:34
testABC
  "self unsafeRun: #testABC"
  | summarizer |
  summarizer := self abcSummarizer.
  self should: [summarizer summary size = 3]
```

So that's what he does. Again, this is programming by intention. Chet just assumes a method *abcSummarizer* that will set up the test object. Then he writes a simple test, that checks to make sure the *summary* send to this one answers back a three-element collection. It's not

enough of a test, but it's enough to break, which is all we need. We go on to build the *Summarizer* to test. Note that we just copied the method – we haven't put the elements in it yet.

```
--- SummarizerTest
10:40:57
abcSummarizer
  ^Summarizer
    with: #()
    with: #()
```

Having built the *Summarizer* object, Chet is ready to enhance the test. I wasn't ready, but he was, so I rode along as he enhanced the test to check all the values.

```
--- SummarizerTest
10:43:06
testABC
  "self unsafeRun: #testABC"
  | summarizer summary |
  summarizer := self abcSummarizer.
  summary := summarizer summary.
  self should: [summary size = 3].
  self should: [summary first name = 'a'].
  self should: [summary first amount = 11].
  self should: [(summary at: 2) name = 'c'].
  self should: [(summary at: 2) amount = 2].
  self should: [summary last name = 'b'].
  self should: [summary last amount = 3].
```

Ron: I guess there's no choice, we're going to have to build the collections now.

Chet: Not quite, we can still do this:

```
--- SummarizerTest
10:44:29
abcSummarizer
  ^Summarizer
    with: self acCollection
    with: self abCollection
```

Chet puts off the inevitable thinking a bit longer by declaring his intention in the method, namely to have a collection with a and c, and one with a and b. Again he assumes that a magic elf has already created them. Programming his intention.

Now it's pretty clear what we need, since we have a name for it. So we type in the methods:

```
--- SummarizerTest
10:45:45
acCollection
  ^OrderedCollection
    with: (Sum
      name: 'a'
      amount: 1)
    with: (Sum
      name: 'c'
      amount: 2)
```

```
--- SummarizerTest
10:45:59
abCollection
  ^OrderedCollection
    with: (Sum
      name: 'a'
      amount: 10)
    with: (Sum
      name: 'b'
      amount: 3)
```

Great, the test is written. We run it. Oops. The original summary method that concatenates the inputs gives back four elements, not three, and the test breaks. We aren't surprised.

Chet: OK, we have to actually do summary. How?

Ron: We could go over the first collection and put all of its elements into a summary collection. Then we could go over the second collection and if the element is in the summary, add it in, otherwise put it in.

Chet: That's weird, the two methods would be sort of alike, but

not quite. How about if we go over each collection, and each time if the item we have is in the summary we add it in, otherwise we create one?

Ron: Good, I like that. So we'll just process first, then process second.

```
--- Summarizer
10:50:29
summary
  self
    processFirst;
    processSecond
```

Chet: Where shall we put the answer?

Ron: Just make a new instance variable, *summarizer*.

Chet: OK, shall I just init it in the Constructor Parameter method?

Ron: OK.

Chet updates the class definition to add the instance variable, and the method to define it as an empty `OrderedCollection`.

```
--- Summarizer
10:51:01
setFirstCollection: firstCollection secondCollection: secondCollection
  first := firstCollection.
  second := secondCollection.
  summary := OrderedCollection new
```

Ron: Duh. We shouldn't have two different process methods, let's just have a process: method that we use twice.

Chet: Duh.

```
--- Summarizer
10:51:55
summary
  self process: first.
  self process: second
```

Ron: OK, let's write process:.

Chet writes a loop over the collection, pausing for a moment to think how to process the items. Then he remembers to just declare his intention (*processItem*.) and go on.

```
--- Summarizer
10:52:32
process: aCollection
    aCollection do: [:each | self processItem: each]
```

It's getting close now. We run the test and notice we don't have *processItem*: defined yet. Chet makes a blank method, then we talk.

```
--- Summarizer
10:52:42
processItem: aSum
```

Ron: OK, rubber meeting road here. What shall we do?

Chet: Well, we just find the matching Sum in the summary ...

Ron: And add our input Sum into it! Do it!

```
--- Summarizer
10:54:22
processItem: aSum
    (self matchingSum: aSum) add: aSum
```

Chet codes just what we said. Our intention is find matching Sum, add our input Sum into it. The code says just that. Now we run the test and, of course, *matchingSum* isn't defined.

Ron: We have to go through the existing summary items and see if we have a matching one ...

Chet: *detect*!

Ron: Yes, do a *detect* ...

Chet: and *ifAbsent*:

Ron: *ifNone*:

Chet: I never can remember which it is.

Ron: Make a new *Sum* and put it in the *summary*

Chet codes it up. This is a pretty standard Smalltalk idiom, so he codes the whole thing in line. The code just says that it'll find a *Sum* in the summary with matching name if there is one, and if not it'll create a new one of that name, and put it in. In either case, it gives back the new or existing matching *Sum*. You could make a case that we should have broken it up, but we discussed it and couldn't find a way we liked better.

```
--- Summarizer
10:56:40
matchingSum: aSum
  ^summary
    detect: [:each | each name = aSum name]
    ifNone: [summary add: (Sum
      name: aSum name
      amount: 0)]
```

We run the test. It doesn't run because *Sum* doesn't understand how to add. We quickly build that:

```
--- Sum
10:58:40
add: aSum
  amount := amount + aSum amount
```

We run the test again. It doesn't work. Instead of getting a collection of *Sums* back, we get back the *Summarizer*. This means we forgot to answer back the result. I recognize this immediately.

Ron: Jeffries error type 1. We didn't answer the collection.  
Where was my partner?

Chet: Driving. You were supposed to help *me*.

Ron: Oh.

```
--- Summarizer
10:59:31
summary
  self process: first.
  self process: second.
  ^summary
```

We run the test. It runs. We celebrate briefly, wishing we had a bell to ring. Then we begin to review the code now that it works, to see if we should clean it up.

Chet: *process*: isn't a very good name.

Ron: We could say *summarize*.

Chet: OK.

--- Summarizer

11:04:45

summary

self summarize: first.

self summarize: second.

^summary

Chet runs the test. It breaks, there's no *summarize*: method. He renames the *process*: method.

--- Summarizer

11:04:53

summarize: aCollection

aCollection do: [:each | self processItem: each]

Chet runs the test and it works. But he doesn't like the new method the way it is.

Chet: Oops, better change *processItem*: to *summarizeItem*: while we're at it.

Ron: Good.

He changes *summarize*: to send *summarizeItem*:

--- Summarizer

11:05:30

summarize: aCollection

aCollection do: [:each | self summarizeItem: each]

Running the test, Chet "discovers" that *summarizeItem*: isn't defined, and renames *processItem*: to *summarizeItem*:

Test-first, by Intention

**--- Summarizer**

**11:05:42**

**summarizeltem: aSum**

**(self matchingSum: aSum) add: aSum**

The test runs again. At this point we look at what we have done and find it good. We stop, and go to lunch at Red Hot and Blue. We each have a Pulled Pig sandwich.

