<u>E24 - TPP</u>

Written by

Robert C. Martin
(Uncle Bob)

The theme is Transformation. Caterpillars into butterflies. Tadpoles into Frogs, ugly-duckling into swan. Original series Star Trek Enterprise into the current movie version. Thank kind of thing.

Segment 1 - Welcome

1 FRONT DOOR 1

UNCLE BOB

Welcome, welcome, to Episode 24 of Clean Code. The Transformation Priority Premise.

Here, let me take your hat.

Remember in the previous episode, episode 23, we talked about mocks. It was a two-parter.

We began with the little puppet show about the addition of a new feature to a system. We explored the architecture of that new feature, and showed how Mocks can be used to help us test at the boundaries of our systems.

Then we took a deep dive into the classification structure of Test Doubles. We described Dummies, Stubs, Spies, Mocks, and fakes.

Then we examined the two schools of test driven development thought: Mockism and Statism. We showed how each has it's own particular problems leading to the uncertainty principle of TDD.

Then we got all pragmatic and studied some common unit testing patterns like Self-shunt, and Humble Object.

Finally, we talked about mocking frameworks, their advantages, disadvantages, and why I don't use them very much.

2 OFFICE 2

UNCLE BOB

Now, in this episode we're going to talk about The Transformation Priority Premise; which represents an outpost on the front lines of TDD research. Welcome to the bleeding edge!

What you are about to learn is an observation, not a theory, a premise, not a law. It might be valuable. It might be dead wrong. But I find it...

3 GS 3

SPOCK

Fascinating.

4 OFFICE 4

UNCLE BOB

We'll begin by studying the concept of transformations; which are the counterpart to refactorings. Transformations are small changes to the code that alter behavior in specific ways.

Next we'll compile a list of transformations, and walk through them one by one.

Then we'll use that list, along with TDD and Refactoring, to build a simple Sort algorithm.

After that, we'll use the list again, but with a subtle difference, to build another Sort algorithm. And we'll compare the two.

Finally, we'll define the premise that there is a priority to the ordering of that list that leads to superior algorithms.

5 MINECRAFT 5

By enchantment table.

ENGINEER
So put your thinking caps on and make sure your XP is good and high, because we're about to cast one mighty strong spell as we study the remarkable world of The Transformation Priority Premise.

Segment 2 - HR Diagrams

6 WORKSTATION 6

Note: J hertzspring.jpg, and russell.jpg when mentioned.

UNCLE BOB
In 1905, Ejnar Hertzsprung and Henry Norris Russell did something pretty amazing. They managed to determine the distances to many stars, and thereby compute their absolute magintudes -- their true luminosity. And then they plotted the luminosity of those stars against their spectral class, their color. The result was astonishing. We call it a Hertsprung-Russel, or HR diagram.

7 GS 7

Panelled Room.

Note: J 1917HR.jpg

UNCLE BOB
Here is the HR diagram that Russell published seven years later in 1917. Look across the top, you can see the spectral classes BAFGKM. Stars on the left are blue, and on the right they are red. In the vertical scale on the left you can see the magnitude. Very bright stars are at the top, very dim stars at the bottom.

Look at that grouping of stars! Look how it forms a kind of thick line from upper left to lower right. That grouping is called the main sequence.

8 WORKSTATION 8

UNCLE BOB
Imagine the chill that must have gone down their spines when that line showed up on their plots. What was it’s significance? Why is it that luminosity and color should be related in such a simple way?

9 GS 9

Note: J HRDiagram.png (credit Richard Powell).

UNCLE BOB
Here’s what that HR diagram looks like when you include 22,000 stars, whose distances, colors, and absolute magnitudes are known with accuracies that Hertzsprung and Russell could only have dreamed about.

Notice that the color scale has been related to temperature. That’s the surface temperature of the star.

Look at the main sequence. It’s got a few wiggles in it; but it’s still that same line that Hertzsprung and Russell saw over a century ago. It shows that luminosity is positively correlated to temperature.

But then there’s that branch off to the right where temperature and luminosity seem to be negatively correlated.

And then there’s that strange curved line at the bottom where luminosity is very low yet temperatures are very high.

What’s going on with all that?

10 WORKSTATION 10

Note: J HRDiagram.png

UNCLE BOB
The main sequence is composed of stars that are fusing hydrogen to helium in their cores. Since that's what most stars are doing most of the time, the main sequence has the most stars in it.

11 GS 11

Panelled Room.

Note: J 1917HR.jpg

UNCLE BOB
Hertzsprung and Russell knew nothing of nuclear reactions when they first drew that simple scatter plot. Nevertheless, they found the signature of hydrogen fusion in that line that went from the top left to the bottom right.

12 WORKSTATION 12

Note: J HRDiagram.png

UNCLE BOB
But, as we've learned. Stars don't fuse hydrogen for their entire lives. When the hydrogen runs out, they start fusing Helium into Carbon. That's what that branch off to the right is all about. Stars over there have run out of hydrogen and have begun to fuse Helium.

Of course that means our sun, which sits right here on the main sequence, will one day move over to that branch on the right. So stars move through this diagram as they age.

13 GS 13

Note: J HRDiagram.png

Note: VO

Animate a line being drawn as I describe it. See HRDiagramPath.jpg For the line and reference points.

UNCLE BOB

Let’s track the sun as it wends its way through the HR diagram. It starts here (A) on the main sequence, happily fusing hydrogen to helium in it’s core. Our sun will spend 10 - 12 billion years happily sitting right there. Oh, it’ll gradually slide a bit over to the left as it’s temperature slowly rises; but that motion is insignificant on this scale.

The big change comes when it runs out of hydrogen in it’s core. The core will contract until a shell of hydrogen surrounding the helium core starts to fuse. The outer envelope will expand by a factor of 100. And our Sun will quickly slide over here, (B) to join all the other red giants.

But that hydrogen shell keeps dumping more and more helium on the core. And eventually the pressures and temperatures in the helium core are sufficient to cause it to ignite and start fusing Helium into carbon. And the Sun slides horizontally over to join all the other end-stage hot red giants (C).

Now things start happening pretty rapidly. As the helium in the core is exhausted, the core contracts again until helium near the carbon core can fuse in a shell. The Sun slides horizontally back towards the cooler red giants (D). But it can linger there only for a short time because the core has become unstable, and has started to eject the envelope of gas that surrounds it.

As the ejected gasses begin to disperse, the sun will shine forth as a planetary nebula.

(MORE)

UNCLE BOB (CONT'D)
But also the intensely high temperatures in the tiny core will begin to leak out, and the sun rapidly moves horizontally to the left. (E) As the envelope disperses even more the sun becomes less and less luminous, so it begins its downward slide to join the white dwarfs (F)

Now, with the envelope completely dispersed, the core, now a diamond the size of the Earth, begins its long slow slide down that final curve as it gradually cools.

14 GS 14

Note: J HRDiagram.png

UNCLE BOB
Isn’t it amazing just how much you can learn from a simply scatter plot? The relationship between two seemingly unrelated variables, like color and brightness, can hold a wealth of information.

Segment 3 - Transformation

15 WORKSTATION 15

UNCLE BOB
Danny, do you know what a refactoring is?

16 GS 16

DANNY DOTNET
Gosh, yes, Uncle Bob. A refactoring, as defined by Martin Fowler in his book: 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.

17 WORKSTATION 17

UNCLE BOB
Excellent! And we could paraphrase that by saying that a refactoring is a change in structure, but not in behavior, right?

18 GS 18

DANNY DOTNET
Yes, that sounds right Uncle Bob. A refactoring changes the structure of the code, but not what the code does.

19 WORKSTATION 19

UNCLE BOB
Good. Now, Ruby, is it true that refactorings don't change behavior -- even a little bit?

20 GS 20

RUBY ROD
Well, like y'know, some don't. Like rename.
(MORE)

RUBY ROD (CONT'D)
Renaming a variable doesn’t change any behavior at all. But there are others, like extract method, that can add a few nanoseconds to the execution time.

21 WORKSTATION 21

UNCLE BOB
OK, so then is it fair to say that a refactoring is a change in structure without significant change in behavior?

22 GS 22

RUBY ROD
Well, yeah -- but it’s kinda wordy.

23 WORKSTATION 23

UNCLE BOB
OK, yes, it’s wordy. But bear with me for a minute. Now, Jerry, let’s invert that statement. What would you call something that changes behavior without significant change in structure?

24 GS 24

JERRY JAVA
Yeah, yeah, changes behavior but not structure, yeah. Uh. Impossible?

25 GS 25

RUBY ROD
Whoa, like yeah, Uncle Bob, like, man, if you change the behavior of the code, you’ve got to change the structure too? Don’t you?

26 WORKSTATION 26

UNCLE BOB
What if you simply changed an addition operator to a subtraction operator, a + to a -. Does that change structure?

27 GS 27

JERRY JAVA
Yeah, + to -. Yeah. No. No structure changes. No.

28 WORKSTATION 28

UNCLE BOB
What if I surround a single statement with a while loop? Does that change structure -- significantly?

29 GS 29

RUBY ROD
Well, that depends on what you think is significant.

30 GS 30

DANNY DOTNET
Oh, I get it! What you're saying, Uncle Bob, is that there are operations on the code that change the behavior without significantly changing the structure.

31 GS 31

RUBY ROD
Yeah, but like I said, it depends on what you think is significant.

32 WORKSTATION 32

UNCLE BOB
OK, so let's simply stipulate that for some definition of significance, there are operations that are the inverse of refactorings. They change behavior without changing structure -- significantly. Let's call these new operations: Transformations.

33 GS 33

RUBY ROD
Whoa, Transformations. I saw all those movies. They were like really rad.

34 WORKSTATION 34

UNCLE BOB
Now, Danny, where do transformations fit in the red-green-refactor cycle of TDD?

35 GS 35

DANNY DOTNET
Gosh, Uncle Bob, we'd have to use them to pass the tests. So cycle would be: Red, Transform to green, Refactor.

36 GS 36

JERRY JAVA
Yeah, transform-to-green, yeah. But that means you can't change structure.

37 WORKSTATION 37

UNCLE BOB
Right! When you are making a test pass, you change behavior but not structure -- you transform. Then you refactor by changing structure but not behavior.

38 GS 38

RUBY ROD

So like, man, you keep the two operations like totaly separate.

39 WORKSTATION 39

UNCLE BOB

Right! When a test is failing you preserve structure while improving behavior to make it pass. Then you switch hats, and preserve behavior while improving structure.

40 GS 40

RUBY ROD

Man, like that's so symmetrical.

41 WORKSTATION 41

UNCLE BOB

Indeed it is! But there's also a lurking asymmetry. Jerry, what is the rule for making a test pass? What kinds of changes must you make?

42 GS 42

JERRY JAVA

Yeah, yeah, rule, heh heh. As the tests get more specific, the code gets more generic.

43 GS 43

Note: S split screen, simultaneous. (Steal it from a previous episode).

JERRY JAVA

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

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

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

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

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

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

44 WORKSTATION 44

UNCLE BOB
Uh, right. Yes. And, so tell me, what direction must these transformations move the code in?

45 GS 45

RUBY ROD
Whoa, since, like, the transformations are used to make the tests pass; they gotta make the code more general. The transformations are generalizers.

46 WORKSTATION 46

UNCLE BOB
Right again! So transformations are small changes to the code that generalize behavior without affecting structure.

47 GS 47

DANNY DOTNET
Transformations generalize behavior without changing structure! Gosh, Uncle Bob, I'm so excited. What are these transformations?

48 WORKSTATION 48

UNCLE BOB
Well, Danny, I could give you a list; but it would probably be better to walk step by step through a familiar kata and try to identify them.

49 GS 49

DANNY DOTNET
Oh, do the prime factors Kata, Uncle Bob. That's my favorite.

50 WORKSTATION 50

UNCLE BOB
OK, Danny. Prime factors coming up.

51 SCREENCAST 51

UNCLE BOB
We'll begin with the basis structure of the kata in place. We'll put all the tests here in this function, just to keep them together. We'll put the implementation in this function for the same reason. That way we won't be flipping back and forth between files. Finally, as you can see, we're using the hamcrest matchers to aid in readability.

Our first test is that the prime factors of 1 is the empty set. And you can see that this fails because our primeFactorsOf function is returning null. We can make this pass by transforming that null into the a constant -- an empty list.

That's our first transformation. Null to constant.

52 GS 52

RUBY ROD
Whoa, like isn't null already a constant?

53 WORKSTATION 53

UNCLE BOB
It is; but it's a special kind of constant. It's a constant that has no meaningful type and no meaningful value. So I consider it different from a constant that has both a type and a value. But let's continue.

54 SCREENCAST 54

UNCLE BOB
Notice how this transformation, Null to Constant, makes the code more general. Null is specific and immutable; but, as Spock is fond of saying, with an ArrayList there are always possibilities.

55 GS 55

SPOCK
Indeed.

56 SCREENCAST 56

UNCLE BOB
Now for the next test, we'll assert that the prime factors of 2 is the list that contains 2. And this fails, of course.

Now, to make this pass we need to generalize that list we're returning. So first we'll transform it from a constant to a variable named factors.

That's our second transformation: constant to variable.
(MORE)

UNCLE BOB (CONT'D)
And, ironically, we'll use the 'Extract Variable' refactoring to achieve it.

57 GS 57

RUBY ROD
But like, hold your horses there Uncle Bob. I mean, changing a constant to a variable doesn't change the behavior does it -- so like its not really a transformation is it.

58 GS 58

JERRY JAVA
Yeah, yeah, it's a refactoring.

59 WORKSTATION 59

UNCLE BOB
You're right Ruby, at least partly. Changing a constant to a variable doesn't directly alter behavior; but it does enable it. Now that we have the variable, we can change it's state. So transforming a constant to a variable can't stand alone as a transformation; but it is a necessary part of one.

60 SCREENCAST 60

UNCLE BOB
Here's the other part. We'll use this 'if' statement to split the flow so that we can add a 2 to the list.

That's the third of our transformations: Split flow.

61 GS 61

DANNY DOTNET
Oh, but Uncle Bob, doesn't that 'if' statement make the code more specific?
(MORE)

DANNY DOTNET (CONT'D)
I mean, the behavior isn't really more general now is it?

62 SCREENCAST 62

UNCLE BOB
Actually, Danny, yes it is. Here, if I changed the if statement to look like this (n==2), then it would be as specific as the test. But n>1 is, in fact, much more general, because it enables -- possibilities.

63 GS 63

SPOCK
Logical.

64 SCREENCAST 64

UNCLE BOB
OK, so our next test will be 3. The prime factors of three is just the list containing a 3. And of course this fails.

Now to make this pass we simply transform the constant 2 into the variable n. That's just the constant to variable transformation again. And the tests pass.

65 GS 65

RUBY ROD
Whoa! I mean like this time that transformation did change the behavior.

66 WORKSTATION 66

UNCLE BOB
Right, Ruby, transformations that enable behavior change, like constant to variable does, can also cause behavior changes if the transformation couples to something that's already changing -- like n.

67 GS 67

RUBY ROD
Wow, like, this blows my mind!

68 SCREENCAST 68

UNCLE BOB
Mine too! So the next test is a little harder. The prime factors of 4 is the list with two 2s in it. To make this pass we need to split the flow again. This time we'll split it based on whether n is divisible by 2. If it is, we'll add a 2 to the list, and then reduce n by the factor 2.

So that's just the split flow transformation again.

And of course this fails because, even though it passed the 4 case, it broke the 2 case returning a list with a 2 and a 1 in it.

It's easy to see why. 2 is divisible by 2, and so a 2 is put in the list, and n gets reduced to 1. Then 1 is put in the list.

So we'll split the flow one more time to prevent a 1 from being put in the list.

And now this passes.

69 GS 69

DANNY DOTNET
Gosh, Uncle Bob, you're sure splitting the flow a lot. And look you've split it twice in n>1.

70 WORKSTATION 70

UNCLE BOB
Keep an eye on those splits Danny; they won't be there much longer.

71 SCREENCAST 71

UNCLE BOB

But I do take your point about the n>1 splits. In fact, the second one is inside the first. We can at least fix that.

So now we add three more tests, and they all pass. The 5 case passes because 5 isn't divisible by 2. The fact that the 6 case passes might be a it surprising, but six divided by 2 is just 3, so it works. And the seven case passes for the same reason 5 passed.

And that leads us to 8. The prime factors of 8 is the list with three 2s in it. Of course this fails.

To make it pass, we can transform this if to a while. Voila! It passes.

This is our fourth transformation: if to while.

72 GS 72

JERRY JAVA

Yeah, yeah, if to while, that's really strange.

73 WORKSTATION 73

UNCLE BOB

It is pretty cool isn't it. But then, a while is just a general form of an if; and an if is just a specific form of a while.

74 GS 74

RUBY ROD

Whoa! I didn't know they were even related.

75 SCREENCAST 75

UNCLE BOB
Oh they are, ruby, they are. But now let's refactor that while loop into a for loop, and get rid of those horrible braces.

There we go, nice and clean, and everythign passes.

Now, one more test. 9. The prime factors of 9 is the list with two 3s in it.

Of course this fails.

We can make this pass by duplicating the for loop and changing all the 2s the 3s. See, that passes.

76 GS 76

MONK
Duplicate Code!

77 GS 77

DANNY DOTNET
Yeah, Uncle Bob, I mean, no disrespect or anything, but we're not supposed to duplicate code like that, are we?

78 WORKSTATION 78

UNCLE BOB
No, Danny, we're not. Or rather, we're not supposed to check duplicate code in. There's nothing wrong with taking a peek like this.

But that peek is revealing! That code is specific, isn't it. It's specific to the test. If we keep passing tests by duplicating this code, we'll never get the algorithm done.

(MORE)

UNCLE BOB (CONT'D)
So that duplication was not a transformation because it didn't make the code more general. So we should undo it, and do something general, like a loop.

79 WORKSTATION 79

UNCLE BOB
Before we do that, I just want to point out that this is one of the reasons we don't like duplicate code. Duplicate code is always specific; it's never general.

80 SCREENCAST 80

UNCLE BOB
OK, so we can make this pass by transforming the constant 2 to a variable named divisor. Then we'll move that variable up above the if. Next we transform the if to a while, and then finally increment the divisor at the end of that while loop.

And, of course, it passes.

Now a little refactoring to get rid of the braces, and we see the end state of the prime factors Kata.

81 GS 81

DANNY DOTNET
I just love the way that kata ends. Those three little lines are so ... cute.

82 WORKSTATION 82

UNCLE BOB
That's not a word I usually associate with Code, Danny. In any case, in this kata we saw four different transformations. Null to Constant, Constant to Variable, split flow, and if to while.

(MORE)

UNCLE BOB (CONT'D)
Of course that’s not the whole list. There are quite a few more transformations. So, now, Let’s look at the list.

Segment 4 - The Transformation List

83 GS 83

Note: J Transformations.graffle

ALBERT
Here is the list of transformations as we currently understand it. We do not consider this list to be complete. We may add or remove items as we gain better understanding. For now, however, this list is a good first approximation.

84 GS 84

Note: J Transformations.graffle

ENGINEER
Folks, what he’s tryin’ to tell ya, is that we don’t know much more about this than amounts to a hill of beans.

85 GS 85

Note: J Transformations.graffle (#1 in red).

ALBERT
The first transformation is “NULL”. It represents the starting implementation of every function. A NULL function does nothing, and if it returns, it returns NULL or zero.

86 GS 86

ENGINEER
This’n here tranformation is the first one you use. You transform a function that don’t exist into one as does; but don’t do nuttin’

87 SCREENCAST 87

UNCLE BOB
Here's where we used it in the Prime Factors kata. We've got a test for the primeFactorsOf function, but the function doesn't exist yet. We simply ask the IDE to create it, and voila! The Null Transformation.

88 GS 88

Note: J Transformations.graffle (#1 in grey, #2 in red)

ALBERT
Transformation #2 is Null to Constant. It transforms a NULL to a static constant of some kind.

89 GS 89

ENGINEER
Rightcha are Albert. You use this here Null to Constant tranformation when ya got a function that returns a null, and ya want it to return something better'n that.

90 SCREENCAST 90

UNCLE BOB
We saw the Null to Contant transformation used in the Prime Factors Kata in order to pass the very first test. We tranformed the NULL that was being returned into an empty ArrayList.

91 GS 91

Note: J Transformations.graffle (1-2 grey, #3 red).

ALBERT
The third transformation is Constant to variable. It transforms a static contant of some kind into a variable or argument of the function.
(MORE)

ALBERT (CONT'D)
Often that new variable holds the value of the constant.

92 GS 92

ENGINEER
Yep, when ya got a constant what ya don't want to be a constant no more, ya can use this here Constant to Variable transformation to change that constant into a variable.

93 SCREENCAST 93

UNCLE BOB
We saw the Constant to Variable transformation used in the Prime Factors Kata when we transformed the empty ArrayList into a variable named factors; and again when we transformed the constant 2 into the argument n, and one more time when we transformed all the 2s in this for loop into the variable named divisor.

94 GS 94

Note: J Transformation.graffle (1-3 grey, 4 red)

ALBERT
The fourth transformation is Add Computation. This transformation adds one or two simple computations, and may even initialize a variable; but it never assigns a variable that already has a value.

95 GS 95

ENGINEER
What he's sayin' is that the Add Computation transformation can do math and stuff; it can even call other functions, but it can't change the state of any existing variables.

96 SCREENCAST 96

UNCLE BOB
There were three uses of Add Computation in the prime factors kata; but they weren't very significant. They were the three predicates of the if statements.

97 GS 97

Note: J transformations.graffle (1-4 grey, 5 red)

ALBERT
The fifth transformation is Split Flow. This transformation usually involves an 'if' statement or something equivalent. It splits the flow of execution into two, and only two, paths.

98 GS 98

ENGINEER
That last little bit that Albert said is reeeeel important. That there split flow transformation don't breed switch statements like horny rabbits. It's more like milk cows that only have one calf. Split flow only splits the flow into two paths. Just two.

99 SCREENCAST 99

UNCLE BOB
We saw the split flow transformation in the Prime Factors kata three times. The first in order to pass the 2 test, and the next two were used to pass the 4 test.

100 GS 100

Note: J transformation.graffle (1-5 grey, 6 red)

ALBERT
The sixth transformation is Variable to Array.
(MORE)

ALBERT (CONT'D)
You use this transformation when you have just one of something, and you need to generalize the behavior to deal with more than one of the same thing.

101 GS 101

ENGINEER
If'n ya harken back to Episode 22 you'll remember that there One-to-Many gizmo. That's what this here transformation is all about. First'n ya deal with one of a thing; and then ya deal with a passle.

102 SCREENCAST 102

UNCLE BOB
We also saw the Variable to Array transformation way back in Episode 4 when we looked at the Stack kata. We got the stack to work with just one element; and then we generalized it to work with many.

103 GS 103

Note: J transformations.graffle (1-6 grey, 7 Red)

ALBERT
The seventh transformation is Array to Container. We use this in order to generalize a simple list into a more complex data structure like a dictionary or a set.

104 GS 104

ENGINEER
Uh. Yup. That's what is is alright. You got an array, an' it needs to be a set or a stack, you transform the array.

105 SCREENCAST 105

UNCLE BOB
In Java this can be a pretty ugly transformation since the syntax for arrays is different from the syntax for containers. You have to replace all the square brackets with calls to get, add, put, or whatever the appropriate function call it.

106 GS 106

Note: J transformations.graffle (1-7 grey, 8 red).

ALBERT
The eigth transformation is If to While. We use this transformation when we realize that a flow that has been split needs to also be repeated.

107 GS 107

ENGINEER
Yeah, you see that there if to while transformation all the time, 'specially after that variable to array transformation. There'll be this 'if' statement that worked fine for the variable; but for an array, it needs to be a while.

108 SCREENCAST 108

UNCLE BOB
We saw if to while twice in the Prime Factors Kata. The first time for the 8 test, and the second time for the 9 test. In both cases the word 'if' was simply changed to the word 'while'. The predicate was preserved.

109 GS 109

Note: J transformations.graffle (1-8 grey, 9 red)

ALBERT
The ninth transformation is "recurse". We use this transformation when we want to repeat a computation contained by a function. We simply have the function call itself.

110 GS 110

ENGINEER
By cracky this is my favorite transformation. Ya got some operation you need to perform over and over again, and so you put that operation into a function and then, like a puppy dog chasing it's tail, you get that function to call itself over and over again.

111 WORKSTATION 111

UNCLE BOB
The Recurse transformation is the neglected transformation. Most Java, C, C++, and C# programmers forget that recursion is usually simpler than iteration. Indeed, many programmers don't even think about recursion and just go straight to a for loop.

112 SCREENCAST 112

UNCLE BOB
We saw the recurse transformation back in Episode 19 while we were solving the Word Wrap problem. It might have come to you as a surprise back then; but a word wrapped string is simply the first word break followed by the wordwrap of the rest of the string.

113 WORKSTATION 113

UNCLE BOB
Again, this is a transformation that is under-used by the vast majority of programmers.

114 GS 114

JERRY JAVA
Yeah, yeah, under-used. That's cuz it blows the stack. Boom! Hahahaha.

115 GS 115

DANNY DOTNET
Yeah, Uncle Bob, I mean, Gosh, functions that call themselves can consume an awful lot of stack space.

116 WORKSTATION 116

UNCLE BOB
Yes, it's true. Our 21st century enterprise platforms have not yet integrated the lessons learned in the 1950s, and so don't do tail call optimization. And so in Java and C# you have to be a little careful with recursion. But there's a difference between care and neglect. A little recursion, in just the right places, can make a function much easier to read and understand.

117 GS 117

SPOCK
And perhaps one day, the powers that be will stop their futile bickering, catch up to the 1950s, and implement the simple and logical expedient of tail-call-optimization.

118 GS 118

Note J: transformations.graffle (1-9 grey, 10 red)

ALBERT
The tenth transformation is 'Iterate'. We use this transformation when some computation needs to be repeated, and we do not wish to use recursion. Generally we use a for loop.

119 GS 119

ENGINEER
Yeah, when you want to do something over and over again, there's nothing quite like a for loop to get that done.

120 SCREENCAST 120

UNCLE BOB
Right, I mean, sometimes a loop is just the obvious solution.

But take a minute to appreciate the elegance of the three component for loop.

As far as I can tell this was invented by Ken Thompson and Denis Ritchie, as part of the C language. It is perhaps one of the most beautiful loop abstractions ever created.

121 GS 121

Note J: transformations.graffle (1-10 grey, 11 red)

ALBERT
The eleventh transformation is 'Assign'. We use this transformation when we want to alter the value of some already existing variable.

122 GS 122

ENGINEER

Yep. When you've already got a variable with some value in it; and you want to change the value of that variable; you can use an assignment to do it. But take care now, because this transformation is only used when you are _changing_ the value of an existing variable. When you initialize a variable, ya ain't using the 'assign' transformation.

123 WORKSTATION 123

UNCLE BOB

Assignment is one of those operations that programmers take for granted. It's so common, so pervasive, so intuitive, that hardly any thought is ever given to it. And yet it's the cause of most of the really difficult problems we encounter in complex systems. But we'll come back to that another time.

124 SCREENCAST 124

UNCLE BOB

We used assignment three times in the Prime Factors Kata. The first time was when we added that 2 to the factors list. That changed the state of the list. So that was an assignment.

We used it again when we reduced the value of n by 2. Finally, we used it when we incremented the divisor.

Notice that none of these assignments used the traditional = operator. Notice also that the traditional = operator is used twice to initialize variables, but not to assign them. Keep this in mind.

(MORE)

UNCLE BOB (CONT'D)
You use the assign transformation whenever you change the state of an existing variable. The assign transformation is not perfectly correlated with the = operator.

125 GS 125

Note: J transformations.graffle (1-11 grey, 12 red)

ALBERT
I have the feeling I should be singing this in a song.

The twelfth transformation is ‘Add Case’. We use this transformation when we have a split flow, and we want to split it further.

126 GS 126

ENGINEER
Yeah, this here ‘add case’ transformation is when you add a else-if to an existing if statement, or a case to a switch statement. This is when you breed your execution paths like rabbits, or Gerbils, or flies.

127 SCREENCAST 127

UNCLE BOB
The ‘add case’ transformation is used when you need to split the flow more than once. It usually means some kind of switch statement or if/else statement.

We saw this used in the Bowling Game Kata back in Episode 6 (part 2). In this case the flow gets split into three distinct paths. The strike path, the spare path, and the no-marks path.

128 WORKSTATION 128

UNCLE BOB

So, those are the transformations. Just twelve -- at least so far. And they're all pretty simple to understand.

You may have noticed that the last few transformations felt ugly. If you did, you've begun to feel the notion of priority. It's just possible that there's an ordering to these transformations.

But we should leave that until later in the episode.

Segment 5 - Sort.

129 WORKSTATION 129

UNCLE BOB
Are you ready? Let's do a thought experiment together. Let's use TDD to write an algorithm that sorts a list of integers.

130 SCREENCAST 130

UNCLE BOB
We begin by creating an executable test harness as usual.

Then we pose the most trivial test case we can think of. What would that trivial test case be?

131 GS 131

DANNY DOTNET
Oh, I know, I know. The most trivial test case for a sort algorithm is sorting an empty list!

132 SCREENCAST 132

UNCLE BOB
That sounds good to me. So we can pose this test case this way. We can get it all to compile using the Null transformation. Then we can clean it up just a bit -- and, of course, it fails.

Now, how do we get this to pass?

133 GS 133

JERRY JAVA
Yeah, yeah. Pass. That's simple. Return an empty list.

134 SCREENCAST 134

UNCLE BOB
Well done Jerry; so we use the null to constant transformation, just like in the Prime Factors kata; and of course it passes.

Now, what's the next most trivial test case?

135 GS 135

RUBY ROD
Yeah, man, y'know, that's an easy one. A list with just one number is, like, already sorted.

136 SCREENCAST 136

UNCLE BOB
Good choice Ruby. That's a real easy test to write. And of course it fails. Now, how do we get it to pass -- which transformation should we use?

137 GS 137

DATA
It would appear that the constant to variable transformation would work for this case. You should simply return the input argument.

138 SCREENCAST 138

UNCLE BOB
Right you are, Data! And that makes the test pass nicely.

But now we have some duplication, so let's clean the tests up a bit.

Now for a hard one. The what's the next test case? Jerry?

139 GS 139

JERRY JAVA
Two Integers. Yeah, yeah, a list with two integers, yeah.

140 WORKSTATION 140

UNCLE BOB
And what order should those two integers be in.

141 GS 141

SPOCK
If the two integers are in order, the test will pass. In order to make the test fail, the two integers should be out of order.

142 SCREENCAST 142

UNCLE BOB
Yes, that seems clear. So we'll pass in two integers out of order. And that fails as it should. Now, how do we make that pass?

143 GS 143

DANNY DOTNET
Uncle Bob! Can I answer? If the first element is greater than the second, we should swap the two elements.

144 WORKSTATION 144

UNCLE BOB
OK, Danny, and what transformation is that?

145 GS 145

DANNY DOTNET
That's a split flow transformation Uncle Bob.

146 SCREENCAST 146

UNCLE BOB
Right again. So we can split the flow based on the ordering. To do that we'll have use the add computation transformation to compute the ordering. Then we save the first element in a temp, which is another add computation transformation. Next we swap first and second elements using assignment transformations.

Ooops, that fails for an ArrayIndexOutOfBounds exception. That's because the previous two tests didn't have two elements to swap. So we need to split the flow one more time. And that passes.

147 WORKSTATION 147

UNCLE BOB
OK, now it starts to get interesting. What's the next test case?

148 GS 148

KIRK
Three Elements!

149 WORKSTATION 149

UNCLE BOB
What order should they be in?

150 GS 150

DATA
The permutations of three things taken three at a time is three factorial, or 6. Two of these permutations already pass. The one where the three numbers are already in order, and the one where only the first two are out of order.
(MORE)

DATA (CONT'D)
So the next permutation to try is the one where just the second two are out of order.

151 SCREENCAST 151

UNCLE BOB
OK, that sounds reasonable. So let's phrase the test this way, with the second and third integers out of order. Now, how do we get this one to pass?

152 GS 152

RUBY ROD
Well, like, you already compared and swapped the first two; so you should just compare and swap the second two.

153 SCREENCAST 153

UNCLE BOB
You mean like this, right? We just copy the if statement, and then change the zeros to 1s and the 1s to twos.

Oh, but that fails for arrays of size two. So we have to copy the outer if statement too.

There, that passes.

154 GS 154

MONK
Duplicate Code! Too Specific! You must refactor my children!

155 SCREENCAST 155

UNCLE BOB
Yeah, that's just like the prime factors Kata isn't it. So let's get rid of that duplicate code and make a loop instead.

(MORE)

UNCLE BOB (CONT'D)
First let's transform the constants 0 and 1 to use a variable named index that starts at 0. Next we'll invert that if statement so that it's in a position to be transformed into a while, and eliminate the extraneous else. Now we can rephrase the predicate so that it's a function of index. This still fails for the same reason, so now let's transform the if to a while and increment the index. Voila! It passes. So now let's clean it up a bit.

156 GS 156

RUBY ROD
Whoa, that was, like heavy duty.

157 WORKSTATION 157

UNCLE BOB
Yes, we did a lot of transformation and refactoring for that last step; and I rather like the result. But we're not done yet. It's still possible to pose a test case that fails. Any guesses?

158 GS 158

SPOCK
The two remaining permutations that fail are those that end with what, when sorted, will be the first element.

159 GS 159

JERRY JAVA
Yeah, yeah, the first is last.

160 SCREENCAST 160

UNCLE BOB
OK, so let's phrase the test this way.
(MORE)

UNCLE BOB (CONT'D)
We’ll start with the list in reverse order, so that the last elmement is the smallest. And, you’re right. It fails.

Now, how do we make this pass?

161 GS 161

ALBERT
The problem as I see it is that the current loop ripples larger elements towards the end of the list. The largest of all elements will wind up at the end of the list. But the smallest element only moves one place towards it’s goal. So we must repeat that loop over and over until the smallest item makes it to the start of the list.

162 GS 162

ENGINEER
Aw crimeny Albert, All ya gotta do is just put the loop in another loop.

163 SCREENCAST 163

UNCLE BOB
OK, let’s try this. We’ll just use the Iterate transformation to repeat the loop, but with an ever decreasing length. And, voila! It passes.

164 GS 164

DANNY DOTNET
Is that the whole agorithm Uncle Bob?

165 WORKSTATION 165

UNCLE BOB
Let’s see, Danny. Let’s try a list of a thousand elements.

166 SCREENCAST 166

UNCLE BOB
We can do that. We'll just create a function that checks an array of length N, and we'll call it with 1000. Our function will simply load N random numbers into the list, sort it, and then make sure all the elements are in order.

There. That works!

167 GS 167

SHERLOCK
Oh for heaven's sake, you don't have any idea what you are saying. Work? Work? Try sorting a list of 50000 elements why don't you?

168 GS 168

RUBY ROD
Yeah, that's, like a real good idea. Let's see how long that takes.

169 SCREENCAST 169

Note: vo Switch to ruby's voice while recording. Add fan noise at appropriate time.

UNCLE BOB
OK, I can do that.

RUBY ROD
Woah, man, that's taking a long time. I mean, holy CFM Batman, like your fan is starting to really make noise. Hey, get back everybody, I think that computer is about to explode.

Oh, there it is. Whoa, that took like 14 seconds!

170 GS 170

SHERLOCK
Oh, did you find that surprising? If you'd like another surprise try sorting a list that's half that size.

171 SCREENCAST 171

Note: vo Switch to ruby's voice while recording.

UNCLE BOB
OK, that's easy. 3, 2, 1, Go!

172 GS 172

RUBY ROD
Like, I don't trust this. I think we should run for the hills.

Whoa, it's done. And it only took 3 seconds. Like, that's a quarter of the time.

173 GS 173

SHERLOCK
And what kind of algorithm quadruples it's time when the input doubles? An n squared algorithm! You ninnies have built a bubble sort!

174 GS 174

RUBY ROD
Gosh, and we did this with TDD. So, like, maybe TDD is a really good way to make really bad algorithms.

Segment 6 - Sort II.

175 WORKSTATION 175

UNCLE BOB
Well, hold on now. Did you notice that we didn’t pay any attention to that feeling that we talked about earlier, that the transformations have a priority? How would our result have been different if we had?

176 GS 176

DANNY DOTNET
Gosh, Uncle Bob, what do you mean? How could we have used the priority of the transformations? Every step we took was pretty obvious.

177 BATHROOM 177

UNCLE BOB
Go back and consider each of those test cases and ask yourself this: Could a different transformation have been used to make that test case pass.

178 GS 178

SPOCK
Interesting. You are implying that there may have been a fork in the road, a different way to make the tests pass, and that we followed the wrong path.

179 WORKSTATION 179

UNCLE BOB
That’s exactly what I’m saying. There was one critical moment, in the development of that algorithm, where we had a choice of transformations; and we made the wrong choice.
(MORE)

UNCLE BOB (CONT'D)

Here, let me show you.

180 SCREENCAST 180

UNCLE BOB

Let’s go back in time to the third test case. The test case where we had two elements out of order. We made that pass by doing a compare and swap. We changed the input list by reassigning it’s values. To do that, we used the Assign transformation.

But there was another way to make that test pass. Look here.

Instead of changing the existing list, we can create a new list. If the size of the input list is less than 2 we simply copy it to the output list and return it.

Otherwise if the two elements are out of order we add them to the new list in the right order. If the input is in order, we simply copy it to the output list.

And that passes; and we used the Add Computation transformation rather than the Assignment transformation.

181 GS 181

DANNY DOTNET

Wow, Uncle Bob, I never thought of that. You’re right! There was a fork in the road. But what happens next?

182 SCREENCAST 182

UNCLE BOB

Well, Danny, the next failing test is three elements, where the first two are out of order, like this. See how it fails. Now, how can we make this pass?

183 GS 183

DATA
Since there are three elements, and since the first two are out of order, the first element in the list must be the middle element. So simply determine which elements are greater and less, and put them into the array in the right order.

184 SCREENCAST 184

UNCLE BOB
OK, so, you mean I should split the flow to add a new case for when the size is three, and just add the computation that assumes that the first element is in the middle? That works, but...

185 GS 185

MONK
Duplicate Code! Specific Code! Refactor, children, Refactor!

186 SCREENCAST 186

UNCLE BOB
OK, so, yeah, these two else clauses are pretty similar. But how do I merge them? One adds two to the list, the other adds three. How can I generalize those pieces of code?

187 GS 187

SPOCK
Look carefully. There are three cases. All similar, yet they all differ. The difference is in the number of elements being added. This implies a loop.

188 SCREENCAST 188

UNCLE BOB
Three cases. Yes, here they are. And yet, each differs only by the number, and order, of elements placed in the list. So, yes, that implies some kind of loop. And the predicate in each case is the length of the input list. And that implies that I want to loop over the input list.

189 GS 189

RUBY ROD
Woah! Man, Like, I've got an idea. Can I drive?

190 WORKSTATION 190

UNCLE BOB
Sure, go for it.

191 SCREENCAST 191

RUBY ROD
OK, like so, look at these three lines. We add the smallest, the middle, and the greatest. So let's name 'em l, m, and h for low, medium, and high.

OK, now those three lines are truth, we just have to get the variables right. And we can do that with this loop.

See, that passes.

192 GS 192

DANNY DOTNET
Oh! Uncle Bob! I see it! I see it! Can I drive please?

193 WORKSTATION 193

UNCLE BOB
Sure Danny, drive away.

194 SCREENCAST 194

DANNY DOTNET
Look, Uncle Bob, we can get rid of this second case now because, well, see that zero on the end.

Look what if we make l and h Objects instead of integers. Objects that can be null. Look!

See, now there are nulls in the list, and now we can just put these if statement in and; see; that works!

195 GS 195

JERRY JAVA
Lemme drive! Lemme drive!

196 WORKSTATION 196

UNCLE BOB
Why not? Everybody else is.

197 SCREENCAST 197

JERRY JAVA
Kill the special case! Ha ha ha ha ha ha.

198 GS 198

SPOCK
Impressive! But there are two cases that will fail. May I?

199 WORKSTATION 199

UNCLE BOB
Oh please!

200 SCREENCAST 200

SPOCK
This case, where the first element is smaller than the others must fail because the others would both have to be placed into h. Note the 3 got lost.

Also this case, where the first element is larger than the others, since both others would have to go into l. Note that the 2 got lost.

201 GS 201

DATA
I believe I can resolve that dilemma -- with your permission Uncle Bob.

202 WORKSTATION 202

UNCLE BOB
Don't mind me. I'm just the author.

203 SCREENCAST 203

DATA
We can replace h and l with lists, and then just add the lists to the output array.

Oh, my, that fails.

204 GS 204

SHERLOCK
Well, of course it fails you ninny. Here, let me.

205 SCREENCAST 205

SHERLOCK
You forgot to sort the two new lists. If we do that, then, of course it passes.

206 GS 206

RUBY ROD
Whoa, that’s like so cool. Does it work for all cases?

207 WORKSTATION 207

UNCLE BOB
Do you guys mind if I take back the keyboard now?

208 SCREENCAST 208

UNCLE BOB
So lets put our general test function in and fix it to work with this new sort function. Now let’s call it with 1000 elements.

Yeah, that works.

209 GS 209

JERRY JAVA
Yeah, yeah, but how fast is it? I mean, look at all that copying, and creating new lists, and -- I bet it’s really slow.

210 SCREENCAST 210

UNCLE BOB
Well, let’s try 50,000 elements like before. Hmm. About a tenth of a second. That’s a lot better than 14 seconds. Let’s try 100,000. Uh, gee, about one sixth of a second. OK, let’s try a million. Wow, still less than a second! It would have taken the bubble sort a little over an hour and a half to sort such a big list.

211 GS 211

DANNY DOTNET
Wow, that’s really fast!

212 GS 212

SHERLOCK
Well, of course it's fast you silly nillies. What you are looking at is the quicksort algorithm. But the way you've written it, it's got a pretty serious flaw. If you'll allow me just one more time.

213 SCREENCAST 213

SHERLOCK
Look what happens if we sort, a relatively small array that's already in order. That's a rather unsatisfactory result, wouldn't you say?

214 COUCH 214

UNCLE BOB
Yes, you have to be careful with recursion. I'll leave the explanation, and the solution, for you, as a homework problem.

215 215

Segment 7 - The Premise

216 COUCH 216

UNCLE BOB
So what happened there? The first time we wound up at the bubble sort, possibly the worse sort algorithm imaginable. The second time we would up at a quicksort, one of the best possible sort algorithms.

In neither case did we know our destination. We were just making tests pass by using our transformations. Yet one pathway led to a bad algorithm, and the other pathway led to a good algorithm.

217 GS 217

SPOCK
Clearly we encountered a fork in the road, one led to bubble sort, the other to quick sort. The difference was in the path we decided to take at the fork.

218 GS 218

DATA
Certainly we will encounter more forks like this as we write our applications. What we need is some way to help us decide which path to take so that we don’t wind up at bad algorithms.

219 WHITEBOARD 219

Note: J transformations.graffle

UNCLE BOB
And that’s where the priorities come in. Here you see the transformations listed in order of those priorities.
(MORE)

UNCLE BOB (CONT'D)
The transformations at the top are high priority transformations. Those at the bottom are low priority transformations.

The premise is, that when you are at a fork in the road, when you can pass the test with two or more transformations, then if you pick the transformation that's highest on the list, you'll wind up at a good algorithm.

220 GS 220

Note: J transformations.graffle

ALBERT
Ja, ja, when we chose the Assignment transformation, we wound up at a bubble sort. But when we chose the add computation transformation we wound up at quicksort. So we should always choose the path with the highest priority.

221 GS 221

JERRY JAVA
Yeah, yeah, highest priority. That feels a lot like magic.

222 GS 222

DANNY DOTNET
Yeah, Uncle Bob, I mean it's kind of hard to believe that something as simple as preferring computation to assignment could lead to algorithms as different as bubble sort and quick sort.

223 WHITEBOARD 223

Note: J tranformations.graffle

UNCLE BOB
And yet that's exactly what we saw, isn't it?

224 GS 224

RUBY ROD

Well, yeah, but, I mean, does it work in every case?

225 WHITEBOARD 225

Note: J transformation.graffle

UNCLE BOB

I honestly don’t know. That’s why I call it a premise and not a theory. But I can tell you this. So far, I haven’t been disappointed. I haven’t found an effective counter example.

So consider that as homework. Come up with an effective counter example to this premise.

And while you are doing that, consider something else...

226 WORKSTATION 226

Note: J transformation.graffle

UNCLE BOB

It should be clear is that these transformations are arranged in the order of their complexity. Simple ones, that have a low impact on the code are at the top, complex transformations that have significant impact on the code are at the bottom.

So if we strive to make our systems pass by using only high priority transformations, our code will be simpler and better.

Whether or not high priority transformations select for better algorithms, it would appear that they do select for better code.

227 KITCHEN 227

Note: 2

UNCLE BOB
And one last point. Remember back in Episode 19, the first in this series of Advanced TDD, we talked about the problem of “Getting Stuck”.

UNCLE BOB (CONT’D)
Remember we said that getting stuck was when, in order to get the current test to pass, you had to write too much code.

228 WORKSTATION 228

UNCLE BOB
We said that if you find yourself stuck, you should try to find a simpler test to pass.

229 IMAC 229

Note: J transformations.graffle

UNCLE BOB
Well, now we can define getting stuck in a different way. Getting stuck is when, in order to get the current test to pass, you have to use a low priority transformation, like Iterate, Assignment, or Add Case.

230 SUN ROOM 230

UNCLE BOB
The solution is to try to find a test that can be passed with a higher priority transformation.

Segment 8 - Conclusion

231 FRONT DOOR 231

UNCLE BOB

So that's it. That's Episode 24. That's the Transformation Priority Premise.

And, just to caution you one more time. This is not a principle, not a theory, not even a hypothesis. It's a premise. It's something I find curious, and that might -- might be important. Perhaps one of you will determine whether that's true.

232 GS 232

SHERLOCK

We began by studying the concept of transformations; the small changes that alter behavior in specific ways.

233 GS 233

SPOCK

Next we studied a list of transformations, and then studied each in turn.

234 GS 234

DATA

We used that list, along with TDD and Refactoring, to build a simple Sort algorithm; that turned out to be a bubble sort.

235 GS 235

ENGINEER

Yeah, and then we used them transformations again, but we took a different path an we built a Quick Sort algorithm.

236 GS 236

ALBERT

Finally, we stated the premise that there is a priority to the ordering of that list, and that following that priority leads to better code, and possibly even better algorithms.

237 FRONT DOOR 237

With dogs.

UNCLE BOB

So, we all hope you that this episode gave you something to think about. But we've still got a lot of ground left to cover. The next episode will be a case study of a simple application written with TDD. Then we'll head off into the fascinating world of Design Patterns. And after that, well, there's still acceptance testing, professionalism, functional programming, and lots, lots more.

You won't want to miss the next exciting episode of Clean Code. Episode 25. The TDD Case Study.

Come on you dogs.