

[CLASS ID]

## **Clean Code 2: More Tips for Writing Clear and Concise Code**

Ben Rand  
Job Industrial Services, Inc.

### **Learning Objectives**

- Learn how to write more-cohesive, focused classes and methods
- Learn how to use dependency injection to write more-testable classes
- Learn how to name classes, methods, and variables meaningfully
- Learn how to apply refactorings to improve code

### **Description**

Have you ever been frustrated by code—likely mostly your own—that’s difficult to read or understand? Or spent hours wading through hundreds of lines of code to fix a bug? Or had a hard time changing your code because you’re afraid you’ll introduce some new bug? If you write code, you’ve probably written your share of bad code—who hasn’t? This follow up to last year’s highly rated course “Clean Code” has been expanded with even more ways to elevate your coding skills. We’ll explore the importance of naming, what functions and classes should do, why comments are (mostly) evil, and how proper formatting is a critical part of writing clean code. You’ll learn about the important skill of refactoring your code, and how unit tests can help you safely refactor. Whatever language you’re programming in, and whatever Autodesk product you focus on, you’ll learn something valuable in this class.

### **Speaker(s)**

Ben Rand has been using AutoCAD software since Release 12. He learned to program using LISP in AutoCAD, worked his way up through VBA to VB.NET, and now spends most of his days programming in C# (occasionally still in AutoCAD!). He has worked in the Industrial Engineering field for more than 18 years as a CAD Manager, developer and IT Director. In 2013, he was the 2013 Top DAUG overall winner at AU, and he served as a mentor for the AutoCAD Mentor All-Star team. Ben has been presenting at AU since 2015, and was honored to appear on the Top Rated Speaker list in 2017 and 2018, and is a Pluralsight Author. Ben is the proud father of four children and enjoys reading and playing a variety of sports including pickleball, volleyball, and tennis. In 2018, Ben was a member of a USTA men's league team that won a national championship.

## What is Clean Code? – A story

As I sat down at my desk one day to work on this course, I realized that my chair wasn't rolling properly. This wasn't new information to me, it had been rolling poorly for several months. I flipped the chair over and examined each of the wheels. None were missing, they were all round...they all had the appearance of performing the function of a wheel. But as I started looking closer, I realized that one of the wheels had a bunch of string wrapped tightly around the inside portion. So, I started unwinding the string, sometimes resorting to scissors to cut jammed portions free.



FIGURE 1 JAMMED WHEEL

At this point, you might be thinking, "Wow, this guy is a world class procrastinator! He'll do anything to avoid getting his work done." And you would not be entirely incorrect. In fact, that's what I thought this little story was going to be about. But as I diligently chopped away at all this string preventing the wheels of my chair from functioning properly, I realized that this is a pretty good analogy for why writing *clean code* is so important.

Probably everyone in this room has written code that works. It does what you intended it to do. But bad code, as opposed to clean code, has a way of jamming us up when we want to make improvements or fix bugs. It's hard to read, easy to get lost, difficult to understand and to reason about. Bad code makes it difficult to find and fix problems, and eventually makes us afraid to touch anything for fear of breaking something else in the system.

This class draws a lot from Robert C. Martin's book, *Clean Code: A Handbook of Agile Software Craftsmanship*. If you don't own a copy, get on Amazon right now after this class, and order a copy. And read it when it arrives. If you have a copy and haven't read it in a while, read it again. I've drawn some of the examples from years of experience mentoring colleagues, teaching at an Autodesk Training Center, and tutoring my sons as they started their own journey into programming in high school and college. But most of all, I'm drawing from my own painful, years-long experience of writing and having to maintain my own code.

## Bad Code – A Definition

Before I define Clean Code, I should probably confess something. My name is Ben Rand, and I have written bad code. Professionally. I've written bad code in many different languages: Lisp, VBA, VB and C#, and probably some in Java, Ruby and JavaScript as well. In fact, I still write bad code. Professionally. But I'm getting better at recognizing it when I do. And most importantly, doing something about it when it happens.

---

*I'm not a great programmer; I'm just a good programmer with great habits.  
— Kent Beck*

---

So, what are some of the characteristics of bad code?

- Bad code **works** BUT
- Is hard to read
- Is hard to reason about
- Tries to do too much
- Uses confusing names
- Relies on magic numbers/strings
- Is hard to maintain
- Is poorly formatted
- *Lacks unit tests*

Wait, bad code *works*? Yes, just like my jammed-up chair wheels, bad code frequently works. We programmers are a persistent bunch, and we'll continue to copy and paste, add more nested if/else statements, and willfully ignore our ever-growing spaghetti functions until finally our Frankenstein baby of a project takes its first painful steps into the world and, surprisingly, actually does something close to what we intended it to do.

The problem with most bad code is **not** that it doesn't work. The problem with bad code is that it's so very difficult to understand it, debug it, and change later on.

## Clean Code – A Definition

What is "clean code"? Here are some of the characteristics of clean code.

- Clean code **works** AND
- Is easy to read
- Is easy to reason about
- Is easy to extend
- Is easy to understand, including several weeks or months from now
- Is easy to maintain
- *Has unit tests*

Both clean and bad code tend to *work*.

So, what's the problem? How often have you gotten a program to work, and then the users come back and say, "That's nice, now can you make it do this?" You scratch your head, add a few more levels of nested if/else statements and *voila!* It works. For a while. Until suddenly it doesn't.

I recently had to make what I figured would be a relatively easy change to a large project I've been working on for several years. While the actual change took maybe 2 minutes to code, I spent more than an hour searching through my own code, trying to find where the change needed to happen. Bad code does this to you again and again and again.

---

*Inside every large program, there is a small program trying to get out.*  
--C.A.R. Hoare

---

It's important to understand that writing code that works and writing clean code are two distinct activities. We first write code that works, then we refactor to clean code.

### **The importance of Unit Tests**

You may have also noticed that one of the characteristics that differentiates clean code from bad code is the presence (or absence) of unit tests. Why is this so important?

Without unit tests, your code becomes fragile and difficult to change. How do you know whether or not a new change you are making a) fixed the problem you intended to fix AND b) doesn't cause a problem somewhere else? Without unit tests, you really don't. Eventually your code *rots* to the point that you are afraid of making changes because you don't know what you might break as you "fix" the system. Don't be afraid of your code. Write unit tests.

### **Elements of Clean Code**

In this class, we're going to explore several elements of clean code. Along the way, we'll look at many examples in a variety of languages you might encounter in the Autodesk ecosphere to illustrate these concepts. I felt like it was important to show you that bad code is not confined to a certain language (I'm looking at you, VBA!). Oh no, we can write bad code in ANY language (even you, Ruby!). And while the tooling to clean up our messes may vary between IDEs, the principles of clean code are pretty well language agnostic.

On the next pages, I humbly submit before you a single function I once wrote in VBA. Go ahead and feast your eyes. I realize it's too small to really read, but the fact that it took 5 separate screenshots to capture a single function should tell you a lot. In retrospect, this single function may violate nearly every principle of writing clean code. It worked, but it was a nightmare to test. And eventually I was afraid of making changes. But at least my indentations are good!

```

Public Sub Straight()
    Dim MS As AcadModelSpace
    Dim Util As AcadUtility
    Dim varStartPt As Variant
    Dim varEndPt As Variant
    Dim objStraight As cStraight
    Dim dLength As Double
    Dim LF As CLastFitting
    Dim dStraightLength As Double
    Dim numStraights As Double
    Dim dLeftOverLength As Double
    Dim I As Integer
    Dim dAng As Double
    Dim tmpEnd As Variant
    Dim tmpStart As Variant
    Dim ini As CIni
    Dim objUCS As AcadUCS
    Dim inputString As String
    Dim dElevation As Double
    Dim tmpPt(0 To 2) As Double
    Dim objLine As AcadLine
    Dim dHyp As Double
    Dim dSideA As Double
    Dim dCosa As Double
    Dim dAngOffXY As Double
    Dim dTrayOffset As Double

    If Not cetoolbox.PSOrMS = "MS"
        MsgBox MSMESSAGE
        Exit Sub
    End If
    ThisDrawing.SendCommand "UCS

    Set ini = New CIni
    With ThisDrawing
        Set MS = .ModelSpace
        Set Util = .Utility
    End With

    On Error Resume Next
    Set LF = New CLastFitting
    dStraightLength = LF.Length *

    cetoolbox.IsLayer LF.Layer, 1
    ThisDrawing.ActiveLayer = This

    Err.Clear
    With Util
        .InitializeUserInput 0, "Offset"
        varStartPt = .GetPoint(, vbCr & "Pick start point [Offset] or <last point>: ")
        If Err Then
            inputString = ThisDrawing.Utility.GetInput
            Err.Clear
            Select Case inputString
                Case "Offset"
                    dTrayOffset = LF.TrayOffset
                    varStartPt = .GetPoint(, vbCr & "Pick start point or <last point>: ")
                    If Err.Description = ERR_PICKPOINT Then
                        varStartPt = LF.EndPt
                    ElseIf Err.Description = ERR_GETPOINT Then
                        .Prompt vbCr & "User cancelled." & vbLf
                        Exit Sub
                    End If
                Case vbNullString
                    varStartPt = LF.EndPt
            End Select
        End If
        If IsEmpty(varStartPt) Then
            MsgBox "Error picking first point. Please try the command again."
            GoSub Cleanup
        End If

        Err.Clear
        varStartPt(2) = varStartPt(2) + dTrayOffset
        varStartPt = .TranslateCoordinates(varStartPt, acWorld, acUCS, False)

        .InitializeUserInput 0, "Slope World Relative Elevation"
        varEndPt = .GetPoint(varStartPt, vbCr & "Pick end point or [Slope/World/Re]
        If Err Then
            If StrComp(Err.Description, "User input is a keyword", 1) = 0 Then
                ' One of the keywords was entered
                inputString = ThisDrawing.Utility.GetInput
                Select Case inputString
                    Case "Slope"
                        Dim bSlope As Boolean
                        tmpEnd = .GetPoint(, vbCr & "Pick end point of slope: ")
                        bSlope = True
                    Case "Elevation"
                        tmpEnd = .GetPoint(, vbCr & "Pick point at end point eleva
                        If Err.Description = ERR_GETPOINT Then
                            .Prompt vbCr & "User cancelled." & vbLf
                            Exit Sub
                        End If
                    End If
                End If
            End If
        End If
    End With

```

Figure 2 A very long method (VBA)--but wait, there's more!

```
: ")
```

```

End If
tmpEnd(0) = varStartPt(0): tmpEnd(1) = varStartPt(1)
Case "World"
dElevation = .GetDistance(, vbCr & "Enter world elevation:
tmpEnd = varStartPt
tmpEnd(2) = dElevation
Case "Relative"
dElevation = .GetDistance(, vbCr & "Enter relative elevation:
tmpEnd = varStartPt
tmpEnd(2) = tmpEnd(2) + dElevation
Case vbNullString
.Prompt vbCr & "User cancelled." & vbLf
Exit Sub
End Select
varEndPt = tmpEnd
ElseIf Err.Description = ERR_PICKPOINT Or Err.Description = ERR_GETPOINT
.Prompt vbCr & "User cancelled." & vbLf
Exit Sub
Else
.Prompt vbCr & "User cancelled." & vbLf
Exit Sub
End If
End If

Err.Clear

tmpStart = varStartPt
varStartPt = .TranslateCoordinates(varStartPt, acUCS, acWorld, False)
If bSlope = False Then
If (Abs(varEndPt(2) - varStartPt(2)) > 0.00001) Then
If varEndPt(2) < varStartPt(2) Then
tmpStart = varEndPt
varEndPt = varStartPt
varStartPt = tmpStart
End If
tmpPt(0) = varEndPt(0): tmpPt(1) = varEndPt(1): tmpPt(2) = varStartPt(2)
Set objLine = MS.AddLine(varStartPt, tmpPt)
dSideA = objLine.Length
dHyp = cetoolbox.GetDist(varStartPt, varEndPt)
dCosA = dSideA / dHyp
dAngOffXY = cetoolbox.RtoD(cetoolbox.ArcCos(dCosA))
objLine.Delete
End If
End If
End With

On Error GoTo 0

```

```

tmpEnd = varEndPt
dLength = cetoolbox.GetDist(varStartPt, varEndPt)
dAng = cetoolbox.getang(varStartPt, varEndPt)

If dLength < dStraightLength Then
dLeftOverLength = dLength
GoSub DoShort
Else
If dStraightLength = 0 Then
dStraightLength = 144
End If
numStraights = dLength \ dStraightLength
For I = 1 To numStraights
Set objStraight = New cStraight
With objStraight
.StartPoint = tmpStart
tmpEnd = Util.PolarPoint(tmpStart, dAng, dStraightLength)
.EndPoint = tmpEnd
.Length = dStraightLength
.ITray_Draw

tmpStart = tmpEnd
tmpEnd = Util.PolarPoint(tmpStart, dAng, dStraightLength)
End With
Set objStraight = Nothing
Next I
If dLength Mod dStraightLength > 0 Then
dLeftOverLength = dLength - (dStraightLength * numStraights)
tmpEnd = Util.PolarPoint(tmpStart, dAng, dLeftOverLength)
GoSub DoShort
End If
End If

ELC2.UpdateCableID2
GoSub Cleanup

DoShort:
Set objStraight = New cStraight
With objStraight
.StartPoint = tmpStart
.EndPoint = tmpEnd
.Length = dLeftOverLength
.ITray_Draw
End With

Cleanup:
With LF
.EndPt = tmpEnd
.Elevation = varStartPt(2)
End With
Set MS = Nothing
Set Util = Nothing
Set ini = Nothing
Set LF = Nothing

End Sub

```

Figure 3 A very long function, continued



Keep in mind that clean code is more of a process, a journey, than it is a final destination. I know you're all going to hate this analogy, but you should look at writing code the same way you did writing papers in high school and college. You write a first rough draft, then review and revise, and revise some more, until you arrive at something better, cleaner, than your first draft. Your second, third, and fourth drafts of a paper were incrementally better than the rough draft. It is the same with code, we just have a different term for it: *refactoring*.

---

*Good software and good writing requires that every line has been rewritten, on average, at least 10 times.<sup>1</sup>*  
--Yevgeniy Brikman

---

Some elements of clean code that I'll discuss in this class are certainly debatable. For example, should indents be 2 spaces, or 4? Should maximum line length be 80 characters, or 120? If you program by yourself, those are somewhat personal decisions. If you're in a team, they're team decisions. Neither is right or wrong, as long as you indent (more on this later), or think about *why* untamed line lengths make code unreadable. There is no "One True Way"™ to write clean code, but there are important principles and guidelines. What we're really after in learning to write clean code is to put more *thought* into the various elements of your code than you may have previously done.

## Naming

I would nominate naming (of classes, methods, variables, etc) as possibly the single most important thing you can do to bring clarity to your code.

Names of classes, methods, and variables should:

- Reveal intent
- Be pronounceable
- Avoid disinformation
- Use nouns for classes
- Use verbs (or verb phrases) for functions
- Avoid single letter variables
- Use long descriptive names instead of short, cryptic names
- Avoid "Hungarian" notation prefixes

It's *hard* to get names right the first time. I usually have to start with something I know is temporary then use a **Rename** refactoring later to rename classes, methods and variables as the intent of my code becomes clearer to me—as *I'm coding*. I use that refactoring *frequently*.

It often helps to figure out what story your code is trying to tell, what problem it is trying to solve. Writing out a simple narrative can be useful in identifying the primary *actors* (i.e. classes) in the

---

<sup>1</sup> Yevgeniy Brikman: <https://www.ybrikman.com/writing/2018/08/12/the-10-to-1-rule-of-writing-and-programming/>

story, what information each actor holds (i.e. properties, fields), and how these actors should interact with one another (i.e. methods).

Let's start with variable names. Consider the following VB code which you might see in any number of AutoCAD samples.

```
Using tr As Transaction = db.TransactionManager.StartTransaction()  
    'Do stuff  
    tr.Commit()  
End Using
```

*Sample 1 Variable names are important too (VB)*

Is the variable *tr* well-named? How do we pronounce that when reasoning about the code—'truh'? 'Tee Arrrrr'? It's completely understandable why "tr" might be used. It's less to type, and we programmers are a lazy bunch. But if you program in a good IDE, you probably only have to type 2-3 characters of a variable name anyway before the IDE suggests an auto completion.

Worse though, is that "tr" is not particularly intention revealing. If *tr.Commit()* is several lines (or screenfuls!) of code away from where the variable is declared, our eyes have to flick back and forth to establish that *tr* is a *Transaction* object. What about common variants such as "tran" or "trans"? These are somewhat better, maybe. But what's wrong with simply naming the variable "transaction"? No more ambiguity.

There are certainly cases where your team might decide that a certain shortcut naming scheme makes sense for well-known objects, stuff you interact with all the time. In the AutoCAD API, "tran" or "trans", "db", "ed", and "doc" are all pretty commonly found in online samples. If your team agrees to adhere to standard variable names like these, that's fine. But be consistent about it. It's pretty easy to wind up with "tran" here, "trans" over there, and "tr" somewhere else. And without some sort of shared knowledgebase, these short names just aren't as explicit as "transaction", or "database", or "editor", or "document".

In times hopefully past, Hungarian notation mandated type prefixes on the front of variable names. In a C# Windows Form application, you'd see things like:

```
TextBox txtFirstName;  
TextBox txtLastName;  
ComboBox cboDepartments;
```

*Sample 2 Hungarian notation adds a prefix to variables to indicate type*

Adding type prefixes just isn't necessary, and makes it harder to reason about the code.

```
TextBox firstName;  
TextBox lastName;  
ComboBox departments;
```

*Sample 3 Type prefixes aren't very useful*

Likewise adding suffixes such as *List* to the end of a variable name, for example *employeeList*, is unnecessary. If you're working with multiple employee objects, just pluralize the variable name: *employees*.



## Magic Strings and Numbers

A common problem is the use of “magic” numbers. Magic numbers clearly have some important meaning in the system, but unless you hold the magic decoder ring to decipher the meaning of that number in the code, it’s hard to know what’s going on. I’ll use an example from AutoLISP to illustrate.

```
(setvar "osmode" 512)
(setq "osmode" 29)
```

*Sample 4: Magic numbers obscure meaning (AutoLISP)*

What does the value 512 represent? What about 29? If you know your way around Autocad, you might know this. But what if you don’t? Maybe adding a “crutch” comment will help.

```
(setvar "osmode" 512) ;set near on
```

*Sample 5: Comments help--but have to be maintained (AutoLISP)*

While this comment helps, it means that we have to read beyond the code to figure out what ‘512’ means. A better alternative is to stop using “magic” numbers altogether and replace them with well-named variables.

```
(setq END_SNAP 1
      MID_SNAP 2
      CENTER_SNAP 4
      NODE_SNAP 8
      QUADRANT_SNAP 16)
```

*Sample 6: Assign named variables to magic numbers (AutoLISP)*

This does a few things. Most importantly, we provide a meaningful (readable) name for the underlying value. And it provides a single place we need to go to make changes if the “magic number” should need to change.

Now compare the sample below to Sample 4. Which code would you rather try to make sense of, or change? The value “29” is difficult to unpack to figure out what it does. Using explicitly named variables makes it pretty clear which snaps are being turned on, without the need for comments that also need to be maintained. The code below is essentially “self documenting.”

```
(setq "osmode" (+ END_SNAP CENTER_SNAP NODE_SNAP QUADRANT_SNAP))
```

*Sample 7: Replacing magic numbers with well-named variables (AutoLISP)*

Writing clean code does not always mean writing code with the fewest number of characters. It means writing code in a way that is easy to understand and maintain, both today while you’re thinking about it, and several months from now when you (or a colleague) has to come back and figure out what the code is doing.

---

*Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*  
--Martin Golding

---

Strings are not immune to this either. A colleague of mine loves to copy and paste code like this:

```
;CS-2
(cond ((and (= block "CS-2") (< 8.1 (distof cadworxalpha 4)))
      (alert "Not suggested for this size pipe!")))
```

*Sample 8: Watch for repeating text (AutoLISP)*

The alert message, “Not suggested for this size pipe!” appeared 28 times in one code file I recently reviewed! That’s 28 places that potentially need to change when it’s decided that the message should change. Consider using variables for repeating strings.

```
(setq NOT_SUGGESTED_MSG "Not suggested for this size pipe!")
...
(alert NOT_SUGGESTED_MSG)
```

*Sample 9: Replace magic strings with named variables (AutoLISP)*

If naming of variables is important, naming of methods and classes is even more so. One of my sons was struggling with an assignment in his Java programming class. In his solution, I found this critical statement:

```
if (valid (numbersDoubledSum (doubler (numSeparator (userInput))),
    numberSum (numSeparator (userInput))) != false)
{ //do something }

    // Rule 1: Double every 2nd digit from right to left, starting with second digit
    from right
    //          If doubling the digit results in two-digit number, add the two digits
    to get a
    //          single digit number.
    // Rule 2: Add together all the resulting single digit numbers from Rule 1.
    // Rule 3: Add every other digit from right to left starting from the right-most
    digit.
    // Rule 4: Sum the result from step 2 and step 3.
    //          If the result is divisible by 10, card number is valid, otherwise it
    is invalid.
```

*Sample 10: Bad names hide intent (Java)*

Hopefully you’re as puzzled by this as I was. Without the comments containing the “rules” for validating a credit card—oh...that’s what this is supposed to do?—it’s nearly impossible to guess what the code is going to do. And frankly, even with the comments, it’s still too difficult to figure out. We’ll return to this example later. Let’s talk briefly about classes next.

## Classes

In *Clean Code*, Bob Martin states that the first rule of classes is they should be short. The second rule is they should be shorter than that. Why should classes be short? It’s because they should be focused on doing one thing.

There’s an acronym used to describe classes: SOLID. The S stands for Single Responsibility Principle, which means that a class should have a single responsibility, with a single reason to change.

A good question to ask yourself when reviewing a class is, “Can I describe the class in 25 words or less, without using ‘if’, ‘and’, ‘or’, or ‘but’?” If the answer is no, then the class likely has too many responsibilities, and should be refactored into multiple smaller classes.

Earlier, we discussed my son’s school project in which they were attempting to validate a credit card number. A class with this responsibility might be named *CreditCardValidator*, and it might have a function named *isValid()*. Additional methods in the class might break some of the steps necessary for validating a credit card number into short, focused implementations of each step.

Such a class should not have additional responsibilities such as *getBalance()* or *makePayment()*. That’s because each of those functions would imply a slew of other considerations, such as which account to access, or how a payment is to be transmitted. These concerns take us further and further from the task at hand: validating a credit card number.

Refactoring from Sample 10 we might rename the methods as follows:

- *valid* becomes *isValid*. This makes the method name more “verb-y” and we expect some response back (True/False).
- *userInput* becomes *ccNumber* or *creditCardNumber*. Even though technically it is “user input,” let’s name the variable what it represents in the context of our system.
- *numSeparator* becomes *getDigits* as the point of this method is to return the individual digits of the credit card number in an array for further manipulation.
- *numbersDoubledSum* becomes *getOddDigitsCheckSum*. This is in keeping with one of the rules defined in the assignment.
- *numberSum* becomes *getEvenDigitsCheckSum*. This is in keeping with another of the rules defined in the assignment.

Here is the improved *isValid* method.

```
public boolean isValid()
{
    ArrayList<Integer> digits = getDigits();
    int oddCheckSum = getOddDigitsCheckSum(digits);
    int evenCheckSum = getEvenDigitsCheckSum(digits);

    int checkSum = (oddCheckSum + evenCheckSum);
    int remainder = checkSum % 10;
    return remainder == 0;
}
```

*Sample 11: Good names reveal intent (Java)*

With better naming, the implementation of the *isValid* method becomes much clearer—even though I didn’t provide you with the code for how the odd- and even- digits check sums are calculated.

## Functions

Functions should be short, and do one thing. Remember the code way back in Figure 2 and Figure 3, my VBA function that required 5 screenshots to capture in its entirety? That function had many responsibilities: collect several different inputs from the user, calculate and translate

points based on that input, and then use that data to draw something. The name of the function? *Straight*. What in the world does that mean? What intent does that convey to the next programmer who has to maintain that mess?

---

*The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.*  
--Bob Martin

---

Functions should use verb-y names, like *GetUserName*, *CheckBalance*, or *DrawLine*.

One of the things we can do to keep functions short is to separate them into two broad categories: command and query. A *command* function should do something, such as set properties or change values on an object. A *query* function should return something, such as a set of layer names or an object of some sort. Critically, functions should not do both. VB differentiates between these two types of methods by offering *Sub* and *Function* methods. In C# a *void* method is the equivalent to VB's *Sub*.

Consider the following example.

```
public bool SetUserName(string userName)
{
    UserName = userName;
    return true;
}
```

*Sample 12: Functions should do something or query for something, not both (C#)*

The function name *SetUserName* indicates that the code will do something (set a user name), but the fact that it returns a Boolean value causes confusion. The examples below make a better distinction between functions which *do* something versus functions which query for something.

```
public void SetUserName(string userName)
{
    UserName = userName;
}

public bool HasUserName()
{
    return !string.IsNullOrEmpty(UserName);
}
```

*Sample 13: Functions which clearly separate doing from querying*

## If/Else Conditionals

When using conditionals, we tend to overcomplicate things.

```
bool someCondition;  
if(someCondition == true)  
{ //do something }
```

*Sample 14: Conditionals (C#)*

The “== true” part of that statement is redundant. *If* always checks for a true condition. So leaving the “== true” out of the conditional works just fine.

```
if(someCondition)
```

You only need the check if you’re evaluating that the conditional evaluates false.

```
if(someCondition == false)
```

Or more tersely:

```
if(!someCondition)
```

Use negative condition checks sparingly. Yes, there are times where you need them, but they make your brain do extra churn. This is bad:

```
if(someCondition != false)
```

This is especially bad:

```
if(!someCondition == false)
```

In statement form, these work out to, “If *someCondition* is not false...” In other words, it’s “sort of a double negative,”<sup>2</sup> and yes, it makes my brain hurt too. If you are trying to ask “If *someCondition* is true” then just say it that way! Clean code should not make your brain hurt.

Here’s another thing we like to do:

```
public bool DoStringsMatch(string source, string target)  
{  
    if (source == target)  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

*Sample 15: Conditional evaluation--long*

The comparison *source == target* evaluates to *true* or *false*, so just return the result of the evaluation.

```
public bool DoStringsMatch(string source, string target)
```

---

<sup>2</sup> Donald Trump, Press Conference, July 17, 2018

```
{
    return source == target;
}
```

Sample 16: Conditional evaluation--terse

## Demo 1: CleanCodeLayer and SRP (Single Responsibility Principle)

One place I frequently see problems is with forms (dialog boxes) taking on too many responsibilities. Suppose you're writing an AutoCAD application in which you need to present a list of layers to the user. Upon selection of a layer, the application does something, such as setting the current layer. The sample files for this course include a C# solution found in the **CleanCodeLayer** folder. A completed, final version (after Demo 2) is located in the **CleanCodeLayer\_Final** folder.

NOTE: You will need to update the References to **AcCoreMgd**, **AcDbMgd** and **AcMgd** to those located in the Object ARX SDK, which you can download from here:

<https://www.autodesk.com/developer-network/platform-technologies/autocad/objectarx-license-download>. Be sure to change the **Copy Local** property to *false* for each of these DLLs.

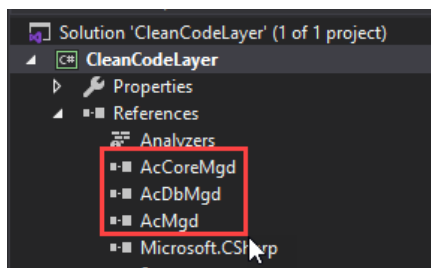


Figure 4: Remove these references and add using your installation of Object ARX SDK

You will also need to set the project's Properties > Debug > Start external program path to that of your installed version of Autocad.

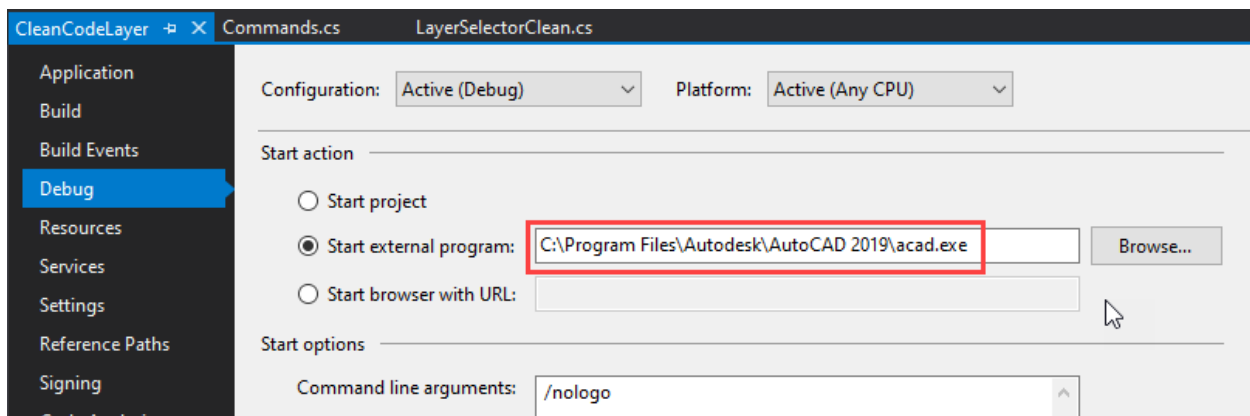


Figure 5: Setting external program path for debugging

## A form (class) with too many responsibilities

The common approach is to create a form and then embed all the code required to get the list of layers, select a layer, and do something with the selected layer, *directly in the form's code-*



*behind.* The form's **.Load()** function may contain code to extract data from a database, read from a text file, or maybe gathers information from the active AutoCAD document. The button's **.Click()** function collects the selections from the form, then uses that data to start a transaction with the AutoCAD database and draws or sets something in the drawing. This is too many responsibilities for this lowly form class.

In order to properly describe the role of this form we have to say something like, "This class gathers layer information from *somewhere*, AND collects user input, AND *does something*." All of these *ANDs* used to describe the role of the form are a giveaway that it has too many responsibilities.

Here's a sample form, and the code-behind to illustrate this concept.

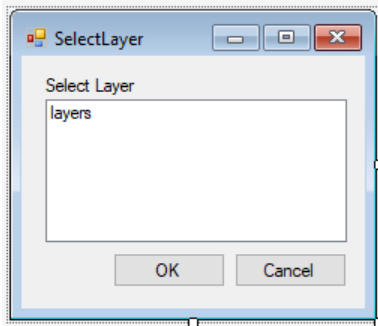


Figure 6: A form to interact with the user

```
public partial class LayerSelector : Form
{
    public LayerSelector()
    {
        InitializeComponent();
    }

    private void Ok_Click(object sender, EventArgs e)
    {
        var doc =
Autodesk.AutoCAD.ApplicationServices.Core.Application.DocumentManager.MdiActiveDocumen
t;
        var db = doc.Database;

        string layerName = (string)Layers.SelectedItem;
        if (!string.IsNullOrEmpty(layerName))
        {
            using (var transaction = doc.TransactionManager.StartTransaction())
            {
                var layerTable =
(LayerTable)transaction.GetObject(db.LayerTableId, OpenMode.ForRead);
                if (layerTable.Has(layerName))
                {
                    var layer = transaction.GetObject(layerTable[layerName],
OpenMode.ForRead);
                    db.Clayer = layer.ObjectId;
                }
            }
        }
    }
}
```

```

        transaction.Commit();
    }
}

private void LayerSelector_Load(object sender, EventArgs e)
{
    var doc =
Autodesk.AutoCAD.ApplicationServices.Core.Application.DocumentManager.MdiActiveDocumen
t;

    var db = doc.Database;

    var layers = new List<string>();
    using (var transaction = doc.TransactionManager.StartTransaction())
    {
        var layerTable = (LayerTable)transaction.GetObject(db.LayerTableId,
OpenMode.ForRead);
        foreach (var layerId in layerTable)
        {
            var layer = (LayerTableRecord)transaction.GetObject(layerId,
OpenMode.ForRead);
            layers.Add(layer.Name);
        }

        Layers.DataSource = layers;
    }
}

```

*Sample 17: Form code has too many responsibilities (C#)*

In order to make this into clean code, we need to understand the *single responsibility* of the form (the “S” in SOLID principles), which is to gather input from the user. While the form needs to display data from which the user is supposed to select, it should not be the form’s responsibility to gather the data to be displayed. Instead, the data, or the responsibility for gathering the data should be *passed into* the form. When the OK button is clicked, the form should go away, handing responsibility for processing the input data back to *the caller* of the form. The caller decides what to do with the data collected on the form.

Looking at Sample 17’s Load function, we can see that almost all of that code really boils down to getting a list of layers from somewhere—in this case from the current AutoCAD drawing. But what if we wanted to get the list of layers from a database, an Excel spreadsheet, or any other source? Do we need different forms for each source? No! If we separate out what changes (the source of the layers to display) from the true responsibility of the form (displaying *any arbitrary* list of layers for the user to pick), then we’ve taken a big step towards writing cleaner code.

### Programming to abstractions

We can do this by introducing an *abstraction* on which the form depends to retrieve the list of layers. Not all languages support abstractions, but most object-oriented languages do (such as C#, VB, and Java). An abstraction, typically an interface or abstract base class, defines a contract for the methods and properties that an implementer **MUST** provide. Classes that depend on an abstraction know they can count on those methods and properties to be present, regardless of which concrete implementation is actually passed in at run-time. If you haven’t coded using abstractions before, you’re missing out on a very powerful technique.

Knowing that our dialog box needs a list of layers, let's define the *ILayerProvider* interface, which specifies a *GetLayers()* function. Note that an interface does not contain an actual implementation of how to get layers! It merely defines the contract that implementers of this interface must supply.

```
public interface ILayerProvider
{
    IList<string> GetLayers();
}
```

*Sample 18: Interfaces define a contract*

Now, we can have the form require an *ILayerProvider* in its constructor, which will deal with retrieving layers. The form will then display the layers retrieved by the *ILayerProvider* object.

```
public partial class LayerSelectorClean : Form
{
    ILayerProvider _layerProvider;
    public LayerSelectorClean(ILayerProvider layerProvider)
    {
        InitializeComponent();
        _layerProvider = layerProvider;
    }

    private void LayerSelectorClean_Load(object sender, EventArgs e)
    {
        Layers.DataSource = _layerProvider.GetLayers();
    }

    private void Ok_Click(object sender, EventArgs e)
    { }

    public string SelectedLayer => (string)Layers.SelectedItem;
}
```

*Sample 19: Injecting dependencies into the class constructor*

Note how *clean* this is. There's hardly anything to it! This class has a very specific, limited focus. It doesn't know where the list of layers to display is coming from. It doesn't know what the caller is going to do with the selected layer. It simply presents whatever set of layers is returned by *ILayerProvider* to the user for selection.

The read-only property, *SelectedLayer*, returns the *SelectedItem* property of the *Layers* ListBox. While the caller could try and access the *form.Layers.SelectedItem* property, this pushes special knowledge of the interior workings of the form out to the caller. Namely that I used a ListBox on the form (named *Layers*), and that the selected value is accessible via the *ListBox.SelectedItem* property. What if I change from a ListBox to some other control for the user to pick from, which doesn't have a "SelectedItem" property? Should the caller have to change too? If we properly encapsulate (hide) the details of our forms, we should not have to change the calling code just because the way we implemented the form might change.

## Implementation of an interface: a “concrete” class

With this done, we need to provide an implementation of *ILayerProvider*. A class that implements an abstraction is called a *concrete class*. Here’s a simple “in memory” layer provider that generates a preset list of layers.

```
class InMemoryLayerProvider : ILayerProvider
{
    public IList<string> GetLayers()
    {
        return new List<string>
        {
            "C-ALGN-ROAD-BRNG",
            "C-ALGN-ROAD-CURV",
            "C-ALGN-SSWR-GEOM",
            "C-ALGN-SSWR-TEXT"
        };
    }
}
```

Sample 20: Implementation of *ILayerProvider*

In the sample code for this course, I’ve also included an *AutocadLayerProvider* which queries the current drawing for its list of layers, which is basically the code from Sample 17’s Load function, relocated to a new class. Now, I can pass *any* implementation of *ILayerProvider* to the form, which will happily display whatever layers the provider provides!

Sample 21 shows two commands demonstrating use of the exact same form but supplying different *ILayerProvider* implementations. The main point is that I now have a form that can work with many potential data sources! The form is also freed from the burden of needing to know what the caller wants to do with the user’s selection.

```
[CommandMethod("SETLAYER2")]
public void SetLayer2()
{
    var form = new LayerSelectorClean(new InMemoryLayerProvider());
    if(form.ShowDialog() == DialogResult.OK)
    {
        SetCurrentLayer(form.SelectedLayer);
    }
}

[CommandMethod("SETLAYER3")]
public void SetLayer3()
{
    var form = new LayerSelectorClean(new ActiveDrawingLayerProvider());
    if(form.ShowDialog() == DialogResult.OK)
    {
        SetCurrentLayer(form.SelectedLayer);
    }
}
```

Sample 21: Injecting *ILayerProvider* implementations into the *LayerSelector* form

The sample code for this class includes the *SetCurrentLayer* function which I excluded here for brevity but is essentially the same code previously found in the original form’s *OK\_Click* method. We’ll return to this project later and add some unit tests, and do some other refactoring.

## Formatting

Formatting our code in a way that makes the code easy to read is a critical but frequently overlooked activity. While many modern IDE's and text editors have powerful, automatic indentation capabilities, it's still important to spend some time ensuring your code is properly formatted.

---

*Programs must be written for people to read, and only incidentally for machines to execute.*  
*-Harold Abelson*

---

## Indentation

Let's look at an example using JavaScript. Quick, tell me something about what the code below does.

```
var rows=prompt("How many rows for your multiplication table?"); var cols=prompt("How many columns for your multiplication table?"); if(rows=="" || rows==null) rows=10; if(cols=="" || cols==null) cols=10; createTable(rows, cols); function createTable(rows, cols){var j=1; var output="<table border='1' width='500' cellspacing='0' cellpadding='5'>"; for(i=1;i<=rows;i++){output=output + "<tr>"; while(j<=cols){output=output + "<td>" + i*j + "</td>"; j=j+1;}output=output + "</tr>"; j=1;}output=output + "</table>"; document.write(output);}
```

*Sample 22: Minified, unindented code (JS)*

It's hard to make sense of that ugly mess. Full disclosure, this is minified code. Hopefully no one would actually *write* their code like this. But far too often, beginners neglect proper indenting. This not only makes it hard to find where blocks of code start and end, but it can also lead to errors such as forgetting to terminate blocks properly. Ever forgotten a closing parenthesis in LISP? Even if all code blocks are terminated correctly, unindented code is just harder to read. Let's look at the same code with improved indenting and spacing.

```
var rows = prompt("How many rows for your multiplication table?");
var cols = prompt("How many columns for your multiplication table?");

if(rows == "" || rows == null)
    rows = 10;

if(cols == "" || cols == null)
    cols = 10;

createTable(rows, cols);

function createTable(rows, cols)
{
    var j = 1;
    var output = "<table border='1' width='500' cellspacing='0' cellpadding='5'>";
    for(i = 1; i <= rows; i++)
    {
        output = output + "<tr>";
        while(j <= cols)
```

```
    {
        output = output + "<td>" + i * j + "</td>";
        j = j+1;
    }
    output = output + "</tr>";
    j = 1;
}
output = output + "</table>";
document.write(output);
}
```

*Sample 23: Indented code (JS)*

Same code, but formatted with better spacing, line breaks and indentation. Clearly this is much more readable than Sample 22.

Here's an incomplete (non-working) sample of C# code that does proper indenting for each "level" of code. Code within each successive block (i.e. within a set of braces) should be indented. Some languages, like python, require proper indentation since they lack line and block termination symbols like ; and } used in C#, JavaScript and Java.

```
namespace CleanCode
{
    class LayerService
    {
        public IEnumerable<string> GetLayers()
        {
            if(condition == true)
            {
                //do something
            }
            return layers;
        }
    }
}
```

*Sample 24: Indentations for every block (C#)*

In some languages, like javascript, you might see the standard more like the one below, with the opening brace on the same line as the statement.

```
namespace CleanCode {
    class LayerService {
        public IEnumerable<string> GetLayers() {
            if(condition == true) {
                //do something
            }
            return layers;
        }
    }
}
```

*Sample 25: Braces at end of line (C#)*

This is a little more concise, but personally I think it's harder to read because my eyes have to scan all the way to the end of each line, looking for an opening brace and then finding the corresponding closing brace is MUCH more difficult. And () and {} and [] have a way of blending together if I'm not paying attention, so the opening brace of each block just gets lost at the end.



For the sake of a few more lines in my editor, I prefer my braces to line up with each other and generally to be on otherwise empty lines.

### Horizontal and Vertical Spacing

An important aspect of readable code is making efforts to get code to *fit* within the screen. Any time you have to scroll horizontally to see the end of a line of code, you're creating a pain point. It's hard to remember what the first part of the line is doing by the time your eyes move from the line of code to watching your cursor slide over the scrollbar, then you have to click, drag and release before finally returning your eyes to the line of code. Wait, what did the first part of the line say?

Below is an example of a C# Linq query that doesn't fit on a typical screen.

```
0 references
public void AvoidLongHorizontalLines()
{
    var employees = new List<Employee>();
    var emps = employees.Where(e => e.FirstName.StartsWith("B")).OrderBy(e => e.FirstName).ThenBy(e => e.LastName).Select(e => $"{e.LastName}, {e.FirstName}");
}
```

*Sample 26: A long line of code (C#)*

In some languages, like C#, line breaks don't indicate the end of a line, so the example above can be improved by breaking the line before each DOT.

```
public void AvoidLongHorizontalLines()
{
    var employees = new List<Employee>();
    var emps = employees
        .Where(e => e.FirstName.StartsWith("B"))
        .OrderBy(e => e.FirstName)
        .ThenBy(e => e.LastName)
        .Select(e => $"{e.LastName}, {e.FirstName}");
}
```

*Sample 27: Breaking up long lines (C#)*

More important is the vertical length of functions and classes. Think back to Figure 3. A single function requiring 5 screens filled with code is simply too much for your brain to process. It's unreadable, difficult to reason about, and clearly trying to do way too much.

---

*Functions should hardly ever be 20 lines long.*  
*--Bob Martin*

---

At the same time, our code shouldn't be jammed together arbitrarily just to make it fit on one screen. Here's an example using code you might find in an AutoCAD application.

```

15 public void Draw()
16 {
17     Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;
18     Document document = Application.DocumentManager.MdiActiveDocument;
19     Database database = Application.DocumentManager.MdiActiveDocument.Database;
20     PromptPointOptions startPointOptions = new PromptPointOptions("Pick start point: ");
21     PromptPointResult pointResult = editor.GetPoint(startPointOptions);
22     if (pointResult.Status != PromptStatus.OK) return;
23     Point3d startPoint = pointResult.Value;
24     PromptPointOptions endPointOptions = new PromptPointOptions("Pick end point:");
25     endPointOptions.BasePoint = startPoint;
26     pointResult = editor.GetPoint(endPointOptions);
27     if (pointResult.Status != PromptStatus.OK) return;
28     Point3d endPoint = pointResult.Value;
29     using (Transaction transaction = document.TransactionManager.StartTransaction())
30     {
31         BlockTable blockTable = (BlockTable)transaction.GetObject(database.BlockTableId, OpenMode.ForRead);
32         BlockTableRecord modelSpace = (BlockTableRecord)transaction.GetObject(blockTable[BlockTableRecord.ModelSpace], OpenMode.ForRead);
33         Line line = new Line(startPoint, endPoint);
34         modelSpace.AppendEntity(line);
35         transaction.AddNewlyCreatedDBObject(line, true);
36         transaction.Commit();
37     }
38 }

```

*Sample 28: No white space (C#)*

Due to the lack of whitespace (extra line breaks), this code is more difficult to read than it should be. Just as paragraphs are used to break up groups of sentences in a book, empty lines should be used to break up the different ideas within our code.

Did you notice the two *if* clauses in the code (lines 22 and 27)? If you weren't looking for them, you probably missed them because I used a single line form of the *if* statement. Branching logic represents an important change in the flow of our program—here it's actually an exit point!—and we should make it as easy as possible to see these changes, just by glancing at the code. Just because we *can* do something to save a bit of vertical space, doesn't mean we should.

Below is the same sample, with some added white space in between important concepts within the function, similar to how we might use paragraphs when writing.

```

public void Draw()
{
    Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;
    Document document = Application.DocumentManager.MdiActiveDocument;
    Database database = Application.DocumentManager.MdiActiveDocument.Database;

    PromptPointOptions startPointOptions = new PromptPointOptions("Pick start point:
");
    PromptPointResult pointResult = editor.GetPoint(startPointOptions);
    if (pointResult.Status != PromptStatus.OK)
        return;
    Point3d startPoint = pointResult.Value;

    PromptPointOptions endPointOptions = new PromptPointOptions("Pick end point:");
    endPointOptions.BasePoint = startPoint;
    pointResult = editor.GetPoint(endPointOptions);
    if (pointResult.Status != PromptStatus.OK)
        return;
    Point3d endPoint = pointResult.Value;

    using (Transaction transaction = document.TransactionManager.StartTransaction())
    {

```

```
        BlockTable blockTable =
(BlockTable)transaction.GetObject(database.BlockTableId, OpenMode.ForRead);
        BlockTableRecord modelSpace =
(BlockTableRecord)transaction.GetObject(blockTable[BlockTableRecord.ModelSpace],
OpenMode.ForRead);

        Line line = new Line(startPoint, endPoint);

        modelSpace.AppendEntity(line);
        transaction.AddNewlyCreatedDBObject(line, true);
        transaction.Commit();
    }
}
```

*Sample 29: Use whitespace to enhance readability (C#)*

Breaking code up this way often leads to extract method refactoring as we recognize where chunks of code could be moved to their own functions. For example, the code for getting the start and end points is very similar and should probably be extracted out to a separate function whose only responsibility is to return a picked point.

## Comments

In the book *Clean Code*, the author states that “comments are, at best, a necessary evil.” Why would he say this? Aren’t we supposed to comment our code so that everyone else who later maintains our code will know what we’ve done? Comments certainly have a place, but they are also overused and abused.

Keep in mind that while reading code, your eyes see comments, and your brain *processes* what it sees. Comments don’t actually do anything when your program runs, but your brain still has to process them when you’re modifying the code. And then it has to spend additional time figuring out if the code actually does what the comment says it does. Or your brain assumes that the comment is correct, and doesn’t bother checking the code or the comment at all! In this case, the comments just become white noise that you train your brain to ignore.

I see a lot of sample code that relies heavily on comments to explain what is happening in the code. For a sample, *where the point of the code is to teach you how to do something*, comments are probably acceptable. For production code, many of those commented sections should probably be broken up into smaller functions, each with an **intention revealing name**, such that the need for the comment is largely eliminated. In the sections that follow, I discuss several points to consider before adding comments to your code.

## Comments need to be maintained

Comments need to be maintained every bit as much as our code does. Can you spot the problem with the comments for the Grid constructor in the C# sample below?

```
/// <summary>
/// Represents a grid with rows and columns.
/// </summary>
/// <param name="startPoint"></param>
/// <param name="numRows"></param>
/// <param name="numCols"></param>
public Grid(Point3d startPoint, int numRows, int numCols, double spacing)
{ ... }
```

#### *Sample 30: Comments must be maintained*

The comments indicate three arguments when, in fact, there are four. Presumably, the *spacing* argument was added after the initial comments were written. This kind of thing happens all the time with comments. If not meticulously maintained, comments become outdated, misleading, or just wrong.

### **Comments explaining sections of code**

We frequently add comments to break up sections of really long functions, explaining chunks of code that should probably be broken out into smaller functions. Or the comment might explain a tricky bit of logic. If these chunks of code are broken out into separate functions and given good, intention revealing names, the need for the comment disappears.

Take this JavaScript example.

```
//only do this if we're on the home page of the site
if (window.location.pathname == "/") {
    ...
}
```

#### *Sample 31: Comment explains conditional*

The comment explains what the conditional clause of the if statement does, which is arguably helpful. However, we could break that bit of logic out into a separate function with an intention revealing name.

```
function isHomePage(){
    return window.location.pathname == "/";
}
```

#### *Sample 32: Move conditional to intention revealing named function*

This eliminates the need for the comment. Our code becomes self-documenting, readable, clean.

```
if (isHomePage()) {
    ...
}
```

#### *Sample 33: Comment is no longer necessary*

### **Comments with versioning information**

Frequently we add comments to our code to supply version information, or to identify who made a particular change in the code.

```
//v1.1 2019-10-22
//Modified by Ben
//Added some comments
```

#### *Sample 34: Comment indicating who made a change*

If you're using a version control system—which you absolutely should be—these types of comments are completely unnecessary. Your version control system<sup>3</sup> tracks all that information automatically, leaving our code clean.




1119		brand	Friday, June 01, 2018 10:07:11	Bug fixes.
1118		brand	Thursday, May 31, 2018 12:54:26	Lots of enhancements to Phone and Contact per request by Darryl.
1117		brand	Tuesday, May 29, 2018 16:24:53	Added SortableEmployeePhoneList report.
1116		brand	Thursday, May 24, 2018 15:52:37	Added type selection for Action Items.
1115		brand	Tuesday, May 22, 2018 15:11:59	Budget implemented and documentation added.
1114		brand	Monday, May 21, 2018 16:37:15	Update to DevExpress 18.1.3.
1113		brand	Thursday, May 17, 2018 07:11:26	Merged branch 2018_05_03.
1099		brand	Wednesday, May 02, 2018 17:19:07	Reorganization into Onion Architecture--I hope.
1098		brand	Wednesday, May 02, 2018 16:06:04	Removed unused references from all projects.
1097		brand	Tuesday, May 01, 2018 11:09:19	Updated tag length for Instruments to 30 characters.

Figure 7: Version control information

Check out what kind of information we get from the *Blame* command in SVN. The second column reveals who last changed a particular line of code. No need to dirty up the code with comments that are hard to maintain and will likely be incorrect after a few revisions.

653	brand	32	ShapeDTO copeShape = shapesToCope.FirstOrDefault();
653	brand	33	if (copeShape != null)
653	brand	34	{
646	gboldt	35	foreach (ShapeDTO cutterShape in cutterShapes)
646	gboldt	36	{
653	brand	37	Point3dCollection pts = JIS_SteelWorx.Cope.Intersection.FindIntersectingPoints(copeShape, cutterShape);
646	gboldt	38	if (pts.Count > 0)
646	gboldt	39	{
653	brand	40	breakList = BreakShapeAtIntersection(copeShape, pts[0]);
646	gboldt	41	shapesToCope.Concat(breakList);
646	gboldt	42	pts = null;
646	gboldt	43	found = true;

Figure 8: Blame command from version control

## Zombie Code comments

Frequently we comment out chunks of code in favor of some new code. I do this a lot while I'm testing to make sure the new code works, just in case I need to immediately revert to the old code. If you're using a version control system—which you absolutely should be—this isn't necessary. Your old code is still there if you need to restore it. Proper usage of version control allows you to delete code with impunity.

Commented out code that gets committed to your repository has a dangerous tendency to turn into zombie code. It hangs in there forever, cluttering your code. By the time you circle back to that old commented out code, you have to ask yourself, "Why is this commented out? Is it still important? Will I need to add it back? Whoever wrote this must have thought it was important enough to leave in, so I better just leave it alone." And there it stays.

## TODO comments

In Visual Studio, you can add `//TODO` comments, which show up in the Tasks window. This can be an OK form of comment, as long as you regularly budget time to go in and take care of the TODOs!

<sup>3</sup> This is a good place to plug my AU course on using source control...

What usually happens to me is I never get back to them and so my task list grows and grows. If I know the code was problematic enough to warrant a TODO comment, I should have just fixed it in the first place. Below is a lovely example of my overgrown TODO list, from one of my main projects. I've added a TODO comment indicating that I may not need this abstract class. The last time I touched this file, according to version control? May 11, 2017.

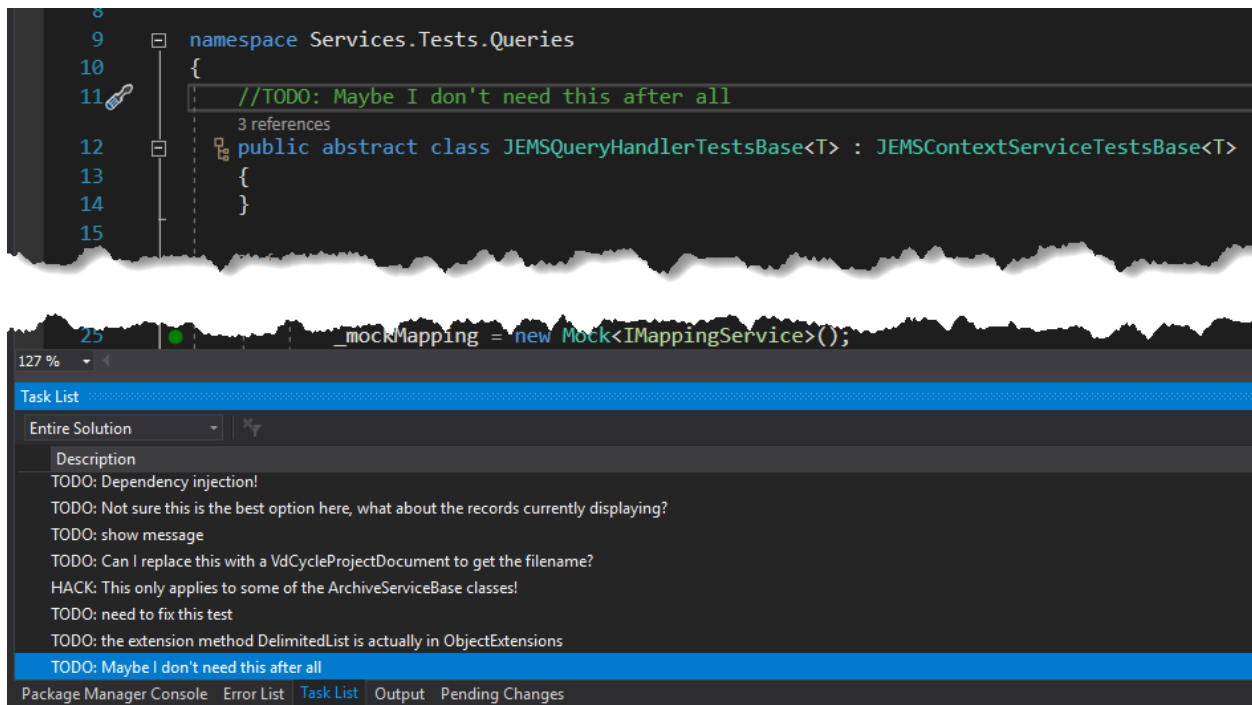


Figure 9: Overgrown TODO list

So TODOs, and the associated code that needs attention become like weeds. Let them grow untended, and they take over your garden.

## Regions

In .NET languages (VB or C#), we can declare regions which are collapsible areas in our code. We use these when our classes get lengthy and we want to collapse or hide certain sections that we may not be working on. The problem with extensive use of regions is that they frequently shield us from the fact that our class has grown overly long, probably has too many responsibilities, and needs to be refactored into multiple classes.



```
#region Fields
//private bool _ShowLoadingStatus = true;
private readonly PermissionProvider _permissionProvider;
private string _RibbonStyle;
private ObservableCollection<GalleryItemGroup> _SafetyDocumentGroup;
private bool _IsSearchHidden = true;
private ObservableCollection<GalleryItemGroup> _HRDocumentGroup;
private ObservableCollection<GalleryItemGroup> _DeliverableDocumentGroup;
private ObservableCollection<GalleryItemGroup> _EquipmentDocumentGroup;
private ObservableCollection<GalleryItemGroup> _AutomationDocumentGroup;
private ObservableCollection<GalleryItemGroup> _ProjectDocumentGroup;
private Project _ActiveProject;
private ObservableCollection<Project> _Projects;
private readonly IGalleryDocumentService _galleryDocumentService;
private readonly Dictionary<string, bool> _permissions;
private readonly IProjectRepository _projectService;
private readonly Dictionary<string, IDocument> documents = new Dictionary<string, IDocument>();
private DelegateCommand<DevExpress.Xpf.Ribbon.DropDownGalleryEventArgs> _dropDownGalleryInit;
private bool? _hasViewInvoicePermission;
private ErrorResultsViewModel _errorResults;
private DelegateCommand<string> showDocumentCommand;
/// <summary>
/// zBenTest project (default)
/// </summary>
private const int DEFAULT_PROJECT_ID = 883;
#endregion
```

Figure 10: Regions often hide problems

## Refactoring to cleaner code: Unit Testing and Applying Clean Code Principles

Unfortunately, unit testing with Autocad (or Revit or 3DS Max) objects is very difficult. Most of the time you're going to be stuck having to debug into Autocad, which takes a lot of time. In some cases, though, we can find ways to test at least some aspects of our plugins. In this section, we'll continue with our CleanCodeLayer solution, and add some unit tests.

### Adding the test project

Right-click the **CleanCodeLayer** solution at the top of the Solution Explorer window, then choose **Add > New Project**. Select the **Class Library (.NET Framework)** project template, and name the new project **CleanCodeLayerTests**.

Expand the new project node in Solution Explorer, then right-click **References** and choose **Add Reference**. Select the **Projects** node, and check **CleanCodeLayer**, then click **OK**.

Right-click the **References** node again and choose **Manage NuGet Packages**. Click **Browse**, then type **NUnit** in the Search box. Install **NUnit** and **NUnit3TestAdapter** into your project. Search again for **Moq** and install the **Moq** package into your project. Close the NuGet tab.

NOTE: I am using the popular NUnit testing framework, along with the Moq mocking framework. There are definitely alternatives to both of these choices, such as MSTest and xUnit.NET, or NSubstitute and Rhino Mocks. The principles are largely the same and which you settle on will largely come down to which syntax of the particular frameworks you prefer.

### Writing a unit test

Right-click the **CleanCodeLayerTests** node and choose **Add Class**. Name the class **LayerSelectorCleanTests**. Set the class modifier to **public**, and add the **[TestFixture]** attribute on the line above the class declaration. This attribute is part of NUnit and requires a using statement to be added to the class. With your cursor on that line in the code editor, press **CTRL+.** and then select **using NUnit.Framework** which adds the using statement to the top of the file.

All test classes in NUnit need to be public, and have the **[TestFixture]** attribute applied. All test methods in NUnit need to be public void, and have the **[Test]** attribute applied.

The first test I like to write is usually one that shows how to instantiate an object of the type we're testing. The test below is a little unusual, in that the Arrange and Act sections are basically the same. More commonly we'll new up the System Under Test (SUT for short) in the Arrange section, then call some method or property of the SUT in the Act section.

I typically name the variable representing the class I'm testing **sut**. Because I use it consistently, it makes it easy to spot.

```
[Test]
public void Instantiate()
{
    //Arrange
    //Act
    var sut = new LayerSelectorClean();
    //Assert
    Assert.IsNotNull(sut);
}
```

*Sample 35: Parts of a unit test*

Unit tests typically have three parts: arrange, act, and assert. The arrange section is where we arrange the SUT, including any initialization required to test whatever it is we're testing. The act section is typically one line of code where we call some method or property on the SUT. This call either returns some value, or sets some *state* in the SUT that we can check. The assert section is where we make one or more statements of assertion, where we check that some expected result matches the actual result.

Note that the test above will not run as is because **LayerSelectorClean** has a required argument in its constructor. We need to provide an **ILayerProvider**.

Update your code to match the following sample.

```
[Test]
public void Instantiate()
{
    //Arrange
    ILayerProvider provider = new InMemoryLayerProvider();
    //Act
    var sut = new LayerSelectorClean(provider);
    //Assert
    Assert.IsNotNull(sut);
}
```

*Sample 36: Supplying an ILayerProvider to the class constructor*

## Using Test Explorer

Click **Test > Test Explorer**, then **Run All Tests**. If all the tests pass, the top node should show a green check mark. If you want to see all the tests, select the top node in the test window, then press **CTRL+right arrow** to expand the list of tests. Note that when any of the tests fail, the nodes will automatically expand to show you which test(s) failed.

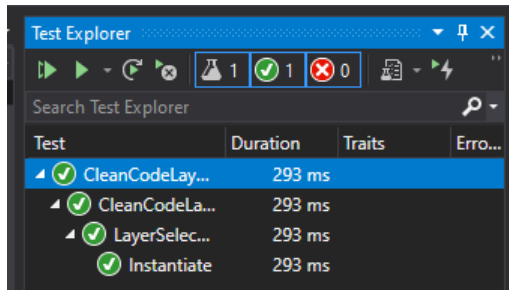


Figure 11: Test Explorer

## Adding “shims” to help with testing

We’re admittedly doing something a little bit strange for most people familiar programming with Windows Forms—we’re actually testing *form* code! Normally we have to launch a form (i.e. execute the program) in order to interact with a form and check that it works. Well, it turns out we don’t actually have to launch a form to unit test many aspects of how it works. But...we do have to provide some *shims* to help us test parts of it. In this case, we’ll create a special inherited version of the class that we can use just for testing. And in this inherited version, we’ll add some methods and properties (shims) that we only use within our tests, to mimic events that would ordinarily be triggered via the UI, while the program is running.

Add the following class to the **CleanCodeLayerTests** project.

```
class ShimLayerSelectorClean : LayerSelectorClean
{
    public ShimLayerSelectorClean(ILayerProvider provider)
        : base(provider)
    { }

    public void CallLoad()
    {
        base.OnLoad(EventArgs.Empty);
    }

    public void CallClick()
    {
        Ok_Click(Ok, EventArgs.Empty);
    }

    public ListBox LayersAccessor => base.Layers;
}
```

This class inherits from **LayerSelectorClean**. It adds a public method, **CallLoad** which we’ll use to trigger the **OnLoad** event on demand, since we aren’t going to actually *Show* the form during our unit tests. The **CallClick** method will be used to trigger the OK button’s Click event. And

finally, the **LayersAccessor** provides access to the **Layers** ListBox. We need to make a couple of small changes to **LayerSelectorClean** to allow this to work.

In the **LayerSelectorClean** code, set the **Ok\_Click** modifier to **protected**. On the form itself, select the **Layers** list box, then set its modifier to **protected** in the Properties window.

### Mocking dependencies with the Moq framework

The next test will verify that our form calls the **ILayerProvider**'s **GetLayers** method. To check this, we need to introduce the concept of *mocking*. Mocking is a way of providing a fake implementation of a class the SUT depends on. Mocking frameworks, such as Moq, give us ways to manipulate these fake implementations so they behave as though they were an actual implementation, all while writing very little additional code.

```
[Test]
public void Calls_ILayerProvider_GetLayers()
{
    //Arrange
    Mock<ILayerProvider> mockProvider = new Mock<ILayerProvider>();
    var sut = new ShimLayerSelectorClean(mockProvider.Object);

    //Act
    sut.CallLoad();

    //Assert
    mockProvider.Verify(x => x.GetLayers(), Times.Once);
}
```

Sample 37: Testing dependency interaction via mocking

The first line is a strange one. We new up a **Mock<ILayerProvider>**. Note that you'll need to add a using to the **Moq** library after typing "Mock". What is this?

Our **LayerSelectorClean** class requires an **ILayerProvider** in its constructor. Although we could spin up an **InMemoryLayerProvider**, that was a rather simplistic implementation. Most of our providers are going to be more complex, such as the **ActiveDrawingLayerProvider**, which is problematic for testing since it relies on having access to a drawing. We can't really test this class in our test environment, due to its reliance on being hosted within the Autocad executable.

One of the main things we want to accomplish in unit testing, is to *isolate* the SUT from as many external factors as possible. This is one of the reasons why programming to abstractions is so valuable, because it allows us to provide *fake* or *mock* implementations to the SUT. We can set these mock implementations to behave as though they are real, but completely controlled by our unit tests. Note that we call the mock object's **.Object** property to provide this mock "implementation" to the constructor.

We new up the "shim" version of our class, **ShimLayerSelectorClean**, as the SUT. This has all the same behavior as the real form, with a few added methods designed to help us during the tests.

In the Act section, we use one of these helper methods, **.CallLoad()** to trigger the **OnLoad** event, as though the form had just been displayed.

The Assert statement is a bit strange as well. Here, we use functionality from the Moq library to *verify* that a call to a particular method (**.GetLayers()**) actually happened. In Moq, we can check that this call did (or didn't) happen, and even check for how many times, using the static **Times**

class from NUnit. This handy class has static methods such as **Times.Never**, **Times.Once**, **Times.AtLeast**, **Times.Exactly**, and so on. Run all your tests again, and they should pass.

### Adding a Setup method to our test class

We need to add some more tests, but we're going to start seeing the same two lines in the Arrange section over and over again. Just like with our production code, we want to keep our test code as clean as possible. Because we're going to have to setup our SUT over and over again, we can lean on a method with a special NUnit attribute to do this work for us.

Add the following code at the top of your class file, just before the Instantiate test.

```
private Mock<ILayerProvider> _mockProvider;
private TestLayerSelectorClean _sut;
[SetUp]
public void Setup()
{
    _mockProvider = new Mock<ILayerProvider>();
    var data = new List<string> { "A", "B", "C" };
    _mockProvider.Setup(x => x.GetLayers()).Returns(data);

    _sut = new ShimLayerSelectorClean(_mockProvider.Object);
}
```

*Sample 38: Using the Setup method to streamline Arrangement of tests*

We mark the Setup method with the **[SetUp]** attribute. This method runs before every single **[Test]** method in the class, ensuring that we have a brand new SUT to work with. Objects created in this method are stored in the **\_mockProvider** and **\_sut** fields, which are then accessible to any of the test methods that care to use them (note that you do not have to use them, and can instead new up an SUT within a specific test however you like when the conditions for that test differ from the “default” configuration provided by the Setup method).

### Providing a return value from a mocked object

Note that we make use of another of Moq's features here.

```
var data = new List<string> { "A", "B", "C" };
_mockProvider.Setup(x => x.GetLayers()).Returns(data);
```

*Sample 39: Setting up a return value for a mocked method*

We setup a list of strings (representing layer names) and tell our **\_mockProvider** that when **GetLayers** is called, we want it to return this list of layers.

This is another important aspect of unit testing. We want our unit tests to be as controlled as possible. And while I could have provided a more realistic set of layer names, simple names suffice for the testing we're doing here.

You can now cleanup the previous test to use the fields now provided by the Setup method.

```
[Test]
public void Calls_ILayerProvider_GetLayers()
{
    //Arrange

    //Act
    _sut.CallLoad();

    //Assert
    _mockProvider.Verify(x => x.GetLayers(), Times.Once);
}
```

```
}
```

*Sample 40: Refactoring tests to utilize fields populated via SetUp*

Run your tests again, to verify that everything is still working. Our next test verifies that the list of layers from our provider appears in the Layers list box.

```
[Test]
public void GetLayers_From_ILayerProvider()
{
    //Arrange
    var expected = 3;
    //Act
    _sut.CallLoad();
    //Assert
    var actual = _sut.LayersAccessor.Items.Count;
    Assert.That(actual, Is.EqualTo(expected));
}
```

*Sample 41: Testing data retrieval from dependency into Layers listbox*

Here, I use the **expected** and **actual** variables to represent the number of items I expect to get back, versus the actual number of items we get back after loading from our mock provider. These variable names are also a convention for me. Note how readable the final assert statement is, thanks to NUnit's syntax. You can actually read that out loud: "Assert that actual is equal to expected."

UI elements on a form are typically *not* public. We hedged a bit by making the Layers list box *protected*, and then providing the shim *LayerAccessor* property to give us back-door access to it for testing. The call to **.CallLoad()** counts as the Act section for me, because this is where we force the Load event to trigger. I placed the assignment of the *actual* variable (representing the number of items in the Layers list box) under Assert, so that we can further prove interaction between our *ILayerProvider* and the form was successful.

### Testing more interaction between class (form) and dependencies

One thing that's often overlooked when working with forms is how we access the selected values. When the user clicks OK, on our form, we want the caller to easily get the name of the selected layer. But if you've looked at the properties of a *ListBox* or a *ComboBox*, it's not straightforward as to which property to choose from: *SelectedItem*, *SelectedIndex*, *GetSelected*, *GetItemText*, etc. Which one do we need? And worse, if the caller of our form has to know this, what happens when we change our mind from a *ListBox* to a *ComboBox*, or some other control? Should the caller have to keep up with all these changes?

I think it's preferable to provide a "public API" to our form, which callers can rely on. If we change the controls on our form, we may have to change the underlying form code, but our "public API" remains the same.

Back in our *CleanCodeLayer* project, add the following two properties to the **LayerSelectorClean** class.

```
public string SelectedLayer => (string)Layers.SelectedItem;
public IEnumerable<string> AllLayers => Layers.Items.Cast<string>();
```

*Sample 42: Adding a public "API" to the form*

Return to **LayerSelectorCleanTests** and add the following test.

```
[Test]
```



```
public void Can_Get_Layer_Name_From_SelectedLayer()
{
    var expected = "B";
    _sut.CallLoad();
    _sut.LayersAccessor.SelectedIndex = 1;

    var actual = _sut.SelectedLayer;

    Assert.That(actual, Is.EqualTo(expected));
}
```

*Sample 43: Testing selection of an item in the form*

I've omitted the Arrange, Act, Assert comments in favor of spacing. Once you've written enough tests, just adding spacing between each section is probably adequate. In this test, we're mimicking the user selecting the second item in the list (corresponding to layer "B" from our provider), and asserting that the new SelectedLayer property correctly returns the correct layer name.

Let's add another test, this time using a real provider, albeit our "in memory" provider.

```
[Test]
[Category("Integration")]
public void GetLayers_From_InMemoryProvider()
{
    var expected = 4;
    var provider = new InMemoryLayerProvider();
    var sut = new ShimLayerSelectorClean(provider);
    sut.CallLoad();

    var actual = sut.LayersAccessor;

    Assert.That(actual.Items.Count, Is.EqualTo(expected));
    Assert.That(actual.Items[0], Is.EqualTo("C-ALGN-ROAD-BRNG"));
}
```

*Sample 44: Integration test using a real provider*

I've added another attribute for this test, [Category("Integration")], identifying this as an "integration" test because it uses a "real" provider. Typically, integration tests connect parts of the application that might query a real database, or access the file system. Because these tests are usually slower, and there are far more reasons for them to fail, they may not be run as often, or special care may have to go into setting up the environment for these types of tests to succeed. Annotating these tests with the Category attribute can be useful as some test runners allow you to include/exclude certain tests by these attributes.

Unfortunately, we can't test the ActiveDrawingLayerProvider as a unit test, and will have to debug that through Autocad. However, we already know that our form is interacting properly with ILayerProviders, and behaving as expected, and we haven't even had to execute the code yet.

### Refactoring to add a command object dependency

I'd like to now look at refactoring our form to accept a *command* object that can be executed when the OK button is clicked. This command object will execute anything we want, such as setting the current layer. But wait, didn't we start out by removing all that code from the original form? Yes we did. What we'll be introducing here is not that specific code, but rather an

abstraction. The `Ok_Click` event will only know enough to *execute* this command, but nothing about what it actually does.

Add the following interface to the `CleanCodeLayer` project.

```
public interface ILayerSelectorCommand
{
    Func<ILayerSelector, object> GetProperty { get; set; }
    void Execute(object parameter);
}
```

Sample 45: *ILayerSelectorCommand* interface

The main thing to note here is the `Execute` method, which takes in an *object* as a parameter. We'll come back to the `GetProperty` property later.

Update the `LayerSelectorClean` code as follows.

```
ILayerProvider _provider;
ILayerSelectorCommand _command;

public LayerSelectorClean(ILayerProvider provider, ILayerSelectorCommand command)
{
    InitializeComponent();
    _provider = provider;
    _command = command;
}

...

protected void Ok_Click(object sender, EventArgs e)
{
    _command.Execute(SelectedLayer);
}
```

Sample 46: Adding *ILayerSelectorCommand* dependency to *LayerSelectorClean*

We're now requiring an `ILayerSelectorCommand` to be injected into our class. This forces some changes in our test project. In **`ShimLayerSelectorClean`**, make the following change:

```
public ShimLayerSelectorClean(ILayerProvider provider, ILayerSelectorCommand
command)
    : base(provider, command)
{ }
```

Sample 47: Refactoring "shim" class

In **`LayerSelectorCleanTests`**, update the **`Setup`** method:

```
private Mock<ILayerProvider> _mockProvider;
private Mock<ILayerSelectorCommand> _mockCommand;
private ShimLayerSelectorClean _sut;
[SetUp]
public void Setup()
{
    var data = new List<string> { "A", "B", "C" };
    _mockProvider = new Mock<ILayerProvider>();
    _mockProvider.Setup(x => x.GetLayers()).Returns(data);

    _mockCommand = new Mock<ILayerSelectorCommand>();
    _sut = new ShimLayerSelectorClean(_mockProvider.Object, _mockCommand.Object);
}
```

#### Sample 48: Refactoring Setup method in test class

Both of the tests that instantiate the `InMemoryLayerProvider` will need to be updated as follows:

```
var provider = new InMemoryLayerProvider();
var sut = new ShimLayerSelectorClean(provider, _mockCommand.Object);
```

#### Sample 49: Refactoring other tests

### Implementing `ILayerSelectorCommand`

Finally, add this test, which verifies that the `Execute` method of our mock command is called when the OK button is clicked.

```
[Test]
public void Calls ICommand_Execute_On_Click()
{
    _sut.CallClick();

    _mockCommand.Verify(x => x.Execute(It.IsAny<string>()), Times.Once);
}
```

#### Sample 50: Testing integration with `ILayerSelectorCommand` dependency

The `Execute` method has an *object* parameter. For mocking purposes, we don't care what that object actually is, we just want to know that the method is called. When we verify a mocked method, we have to account for any arguments being passed into the mocked method. Here, we don't really care what value is passed in (though conceivably in other tests we might care). The Moq framework, utilizes the `It.IsAny<T>()` convention to act as a stand-in for an expected argument.

Even though we haven't created an implementation of an `ILayerSelectorCommand` yet, we could run our tests based just on this mock implementation. However, in the **CleanCodeLayer.Commands** class, **SetLayer2** and **SetLayer3** will not compile without providing the new argument. I'd like to run the tests sooner rather than later, so comment those two methods out, then run your tests. We'll uncomment those methods and fix them up later on. If your tests pass, proceed to an implementation of `ILayerSelectorCommand`.

Add a new class, **CurrentLayerCommand** to your **CleanCodeLayer** project.

```
public class CurrentLayerCommand : ILayerSelectorCommand
{
    public CurrentLayerCommand()
    {
        GetProperty = x => x.SelectedLayer;
    }

    public Func<ILayerSelector, object> GetProperty { get; set; }

    public void Execute(object parameter)
    {
        var layerName = (string)parameter;

        Active.UsingTransaction(transaction =>
        {
            var layerTable = (LayerTable)transaction
                .GetObject(Active.Database.LayerTableId, OpenMode.ForRead);

            LayerTableRecord layer;
            if (layerTable.Has(layerName))
```

```

        {
            layer = (LayerTableRecord)transaction
                .GetObject(layerTable[layerName], OpenMode.ForRead);
        }
        else
        {
            layer = new LayerTableRecord { Name = layerName };

            layerTable.UpgradeOpen();
            layerTable.Add(layer);
            transaction.AddNewlyCreatedDBObject(layer, true);
        }

        Active.Database.Clayer = layer.ObjectId;
    });
}
}

```

Sample 51: *CurrentLayerCommand implementation*

The **Execute** code is largely the same as what was in the `Ok_Click` event when we started, other than converting the parameter from *object* to *string*. The main thing to note here is the `GetProperty` property (admittedly, this could use a better name). This property is a `Func` that can be used to get a property value from an object implementing `ILayerSelector`. We'll use this as a sneaky way of getting information from the form this command object gets injected into.

Let's define the interface.

```

public interface ILayerSelector
{
    string SelectedLayer { get; }
    IEnumerable<string> AllLayers { get; }
}

```

Sample 52: *ILayerSelector interface*

Hopefully, that looks a little familiar. It matches the “public API” we added to `LayerSelectorClean`. Let's update the form code to formally implement the new interface.

```

public partial class LayerSelectorClean : Form, ILayerSelector
{

```

Yep, that's it. Now we need to update the `Ok_Click` code, to use this `Func` property so we can extract what the command wants from the form, and supply it to the command's `execute` method.

```

    protected void Ok_Click(object sender, EventArgs e)
    {
        var parameter = _command.GetProperty(this);
        _command.Execute(parameter);
    }

```

Sample 53: *Refactoring LayerSelectorClean*

If this twists your head into a pretzel, that's OK. It's admittedly a little strange. We pass the form itself (*this*) into the `GetProperty` func. That func reads a property off the form (`SelectedLayer` in this case—see Sample 51 constructor), assigns it to the *parameter* variable, and sends that to the `Execute` method of the command.

Run your tests again. The **Calls ICommand\_Execute\_OnOkClick** should fail.

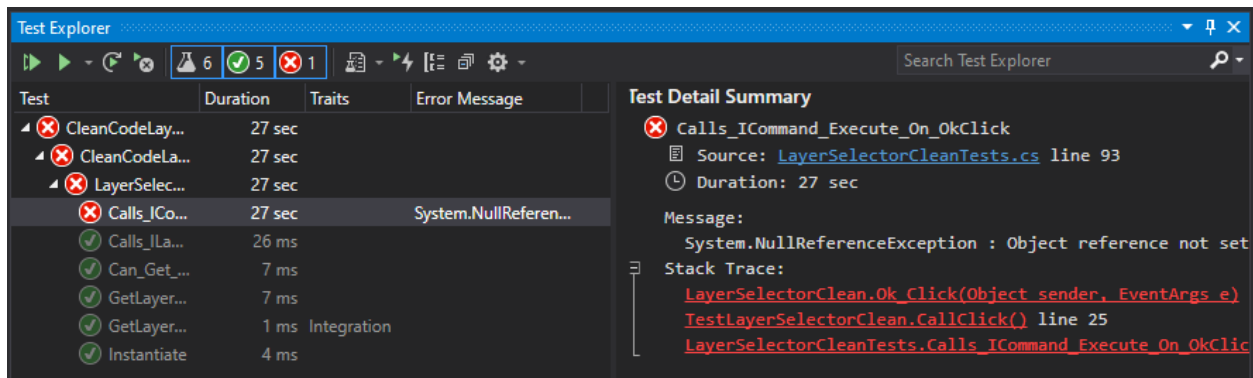


Figure 12: Failed unit test

In the stack trace, you can click the first link to jump to the point in code where the error occurs, in the `LayerSelectorClean` class.

```
protected void Ok_Click(object sender, EventArgs e)
{
    var parameter = _command.GetProperty(this);
    _command.Execute(parameter);
}
```

Sample 54: Test fails here, but why?

`GetProperty` returns null here, which you can find by setting a break point here, then right-clicking the failing test in Test Explorer and choosing Debug. The test will run to this point, and when you hover over the `GetProperty` property, you'll see that its value is null.

Understanding why is a bit tricky, until you remember that we're using a mock object, not a real implementation. Whenever we use a mock object in tests, we have to be sure that we setup the mock to mimic method or property calls that our SUT will make.

In `LayerSelectorCleanTests`, update the Setup method as follows:

```
mockCommand = new Mock<ILayerSelectorCommand>();
mockCommand.Setup(x => x.GetProperty).Returns(x => x.SelectedLayer);
```

Sample 55: Setting up our mock to return a Func

Run your tests again, and this time they should all pass.

Let's create one more implementation of `ILayerSelectorCommand`.

```
class LayersToEditorCommand : ILayerSelectorCommand
{
    public Func<ILayerSelector, object> GetProperty { get; set; }
    public LayersToEditorCommand()
    {
        GetProperty = x => x.AllLayers;
    }

    public void Execute(object parameter)
    {
        var layers = (IEnumerable<string>)parameter;
    }
}
```

```

        foreach (var layer in layers)
        {
            Active.Editor.WriteMessage($"{layer}");
        }
    }
}

```

Sample 56: LayersToEditorCommand implementation

This command grabs AllLayers from the ILayerSelector (the form), and, when executed, dumps all the layer names out to the command prompt. Now, you're probably wondering why we went to all this effort of injecting a command into the form class. And the GetProperty Func is, well, funky. The reason for this, is so that we can swap different commands, different functionality, into the form, without having to further change the form.

### Pay Day: Composable Commands

Let's uncomment those commands in our Command class, and update as follows.

```

[CommandMethod("SETLAYER2")]
public void SetLayer2()
{
    var form = new LayerSelectorClean(new InMemoryLayerProvider(),
        new CurrentLayerCommand());
    form.ShowDialog();
}

[CommandMethod("SETLAYER3")]
public void SetLayer3()
{
    var form = new LayerSelectorClean(new ActiveDrawingLayerProvider(),
        new CurrentLayerCommand());
    form.ShowDialog();
}

[CommandMethod("SETLAYER4")]
public void SetLayer4()
{
    var form = new LayerSelectorClean(new ActiveDrawingLayerProvider(),
        new LayersToEditorCommand());
    form.ShowDialog();
}

```

Sample 57: Updated CommandMethods in the Commands class

Let's see this in action! The **CleanCodeLayer** project is set as the Startup Project (right-click the project node in Solution Explorer and choose **Set as startup project**). Right-click the project and choose **Properties**, then go to the **Debug** tab. There, I've identified the acad.exe as the external program to run. You may need to change your path to the version of Autocad you're using. Click **Debug > Start Debugging** to launch Autocad.

Use the **netload** command, then navigate to the **bin\Debug** folder of wherever you downloaded the code to, then select the **CleanCodeLayer.dll** and click Open.

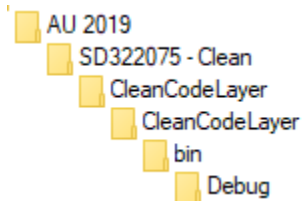


Figure 13: Locating the CleanCodeLayer.dll

When prompted, you'll need to allow the code to load. Run the following commands:

**SETLAYER1** – This is the original form with all the code in the form. Retrieves layers from the drawing and sets the current layer. Does too much and isn't flexible.

**SETLAYER2** – Uses the new form with swappable layer provider and command (here: InMemoryLayerProvider, and CurrentLayerCommand).

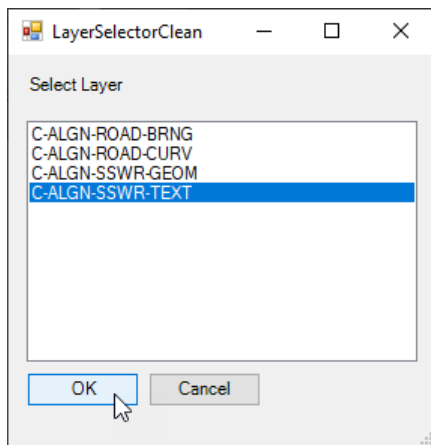


Figure 14: InMemoryLayerProvider layers displayed in the form

**SETLAYER3** -- Uses the new form with swappable layer provider and command (here: ActiveDrawingLayerProvider, and CurrentLayerCommand).

**SETLAYER4** -- Uses the new form with swappable layer provider and command (here: ActiveDrawingLayerProvider, and LayersToEditorCommand).



```

Command: SETLAYER4
0
G-Margin
G-Border
G-Border-Text
G-Border-Text2
G-Border-Text-B
G-Border-Text-L
:
Class
Supports
CADWorx1
CADWorx2
Support_Bom
C-ALGN-SSWR-TEXT
Command: |

```

Figure 15: Dumping list of layers to command prompt

Cue C+C Music Factory’s “Things that make you go hmmm...” I hope that last example really gets you thinking about what we’ve accomplished. It would be quite easy to now write an `ILayerSelectorCommand` implementation that dumped the list of layers out to a text file or even to Excel, or many other possible manipulations involving the list of layers.

The key takeaway is that we’ve made the form itself very simple, kind of dumb, actually. It depends on an `ILayerProvider` to provide it with a list of layers, and it depends on an `ILayerSelectorCommand` to actually do something. But it doesn’t know anything about how either of those dependencies works. We can now plug different behavior into this form easily.

We’ve also learned a lot about unit testing and mocking. And while it’s unfortunately difficult (impossible??) to test parts of our applications that interact with Autocad, Revit, 3DS Max etc, we can often program around those parts with (admittedly) some creative effort.

## Summary

There is much more I could and want to say about writing clean code. I hope that the things I’ve shared here will get you thinking about how you can improve your code. Keep in mind that clean code doesn’t just happen—even for the “experts.” It’s a continual process and effort to make your code better with each iteration. You get better with practice.

This class would not have been possible without me having written a LOT of bad code over the last 20 years. While I’m not necessarily proud of that fact, I have taken responsibility for it and realized that I can (and need to!) continuously learn to write better, cleaner code. This desire has led me to many important books that have helped me improve.

The most important thing I leave you with is probably the Recommended Reading List below.

### Recommended Reading List

Below are some of the more notable books which have helped me on my path. I've read ALL of these books cover to cover, some a few times. Each has helped me become a better programmer, and I'm certain they will help you as well.

*Clean Code: A Handbook of Agile Software Craftsmanship*, Robert C. Martin

*Working Effectively with Legacy Code*, Michael C. Feathers

*Dependency Injection in .NET*, Mark Seeman

*Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, et al.

*The Pragmatic Programmer: From Journeyman to Master*, Andy Hunt

*Refactoring: Improving the Design of Existing Code*, Martin Fowler

*Code Complete*, Steve McConnell

*Head First Design Patterns*, Eric Freeman

*The Art of Unit Testing: With Examples in .NET*, Roy Osherove

*AU 2015: SD12077: Being a Remarkable C# .NET AutoCAD Developer*, Scott McFarlane

<http://au.autodesk.com/au-online/classes-on-demand/class-catalog/2015/autocad/sd12077>

If you're interested in further readings, or have any questions concerning this course, please feel free to reach out to me via email: [ben@leadensky.com](mailto:ben@leadensky.com). I'm more than happy to help!