AUTODESK. UNIVERSITY

SD226637

Clean Code: Tips for Writing Clear and Concise Code

Ben Rand
Job Industrial Services, Inc.

Learning Objectives

- Name classes, methods and variables meaningfully
- Write more cohesive, focused classes and methods
- Utilize unit tests to support refactoring towards clean code
- Write cleaner code

Description

Have you ever been frustrated by code that is difficult to read or understand (including your own!)? 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--I know I have! In this class, you'll learn important tips on how to elevate your coding skills to the next level. We'll explore things like 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 will 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, VB6 and 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 17 years as a CAD Manager, developer and IT Director. In 2013, he was the 2013 Top DAUG overall winner at AU, and he served a mentor for the AutoCAD Mentor All-Star team. 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 tennis team that won a national title and another team that was the finalist.

What is Clean Code? – A story

As I sat down at my desk one day this summer 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 several of the wheels had a bunch of string wrapped tightly around the inside of the wheel. 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 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 isn't that it doesn't work. The main problem with bad code is that it's so very difficult to understand it, debug it, and change it later on.

Clean Code – A Definition

What is "clean code"? Let me start with a list of 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 it suddenly 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 page, 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 = Thi
```

```
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/Rel
    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 elevat
                   If Err.Description = ERR_GETPOINT Then
                       .Prompt vbCr & "User cancelled." & vbLf
                       Exit Sub
```

```
End I
Err.(
tmpSt
varSt
 If I
```

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 elevatio
                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_GETPOIN
        .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) = varStart
        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, fourth drafts of a paper were always 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.*[1]
> *--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.

Formatting

I'm going to start with formatting, which is a critical aspect of writing clean code. 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.

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 1: 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

---

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

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 2: Indented code (JS)

Same code, but formatted with better spacing, line breaks and indentation. Clearly this is much more readable than Sample 1.

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 3: Indentations for every block (C#)

In some languages, 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 4: 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 () 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.

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 5: 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 6: 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 7: 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 5. 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.

```
(setq “osmode” (+ END_SNAP CENTER_SNAP NODE_SNAP QUADRANT_SNAP))
```

Sample 8: 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 9: Watch for repeating text (AutoLISP)

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

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

Sample 10: Replace magic strings with named variables (AutoLISP)

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 you mentally switch gears from reading code to moving your cursor over the scroll bar to looking back at the code—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 11: 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 things like this.

```
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 12: 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 long 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

Indeed, classes that take more than a screen full of code are probably trying to do too much. At the same time, our code shouldn't be jammed together arbitrarily just to make it fit. 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 13 No white space (C#)

AUTODESK. UNIVERSITY

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. Just because we *can* do something to save a bit of vertical space, doesn't mean we should. 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.

Below is the same sample, with some added white space in between important concepts within the function, kind of like paragraphs in a book. 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.

```
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 14 Use whitespace to enhance readability (C#)

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 when 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 commented 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 15 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 16 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 17 Move conditional to intention revealing named function

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

```
  if (isHomePage()) {
    ...
  }
```
Sample 18 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.

```
//Modified by Ben
```
Sample 19 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[2] 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 4 Version control information

Again, version control is really good at keeping track of all that automatically. 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.

---

[2] This is a good place to plug my AU course on using source control…

```
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 5 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. But again, 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 version control 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!

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.

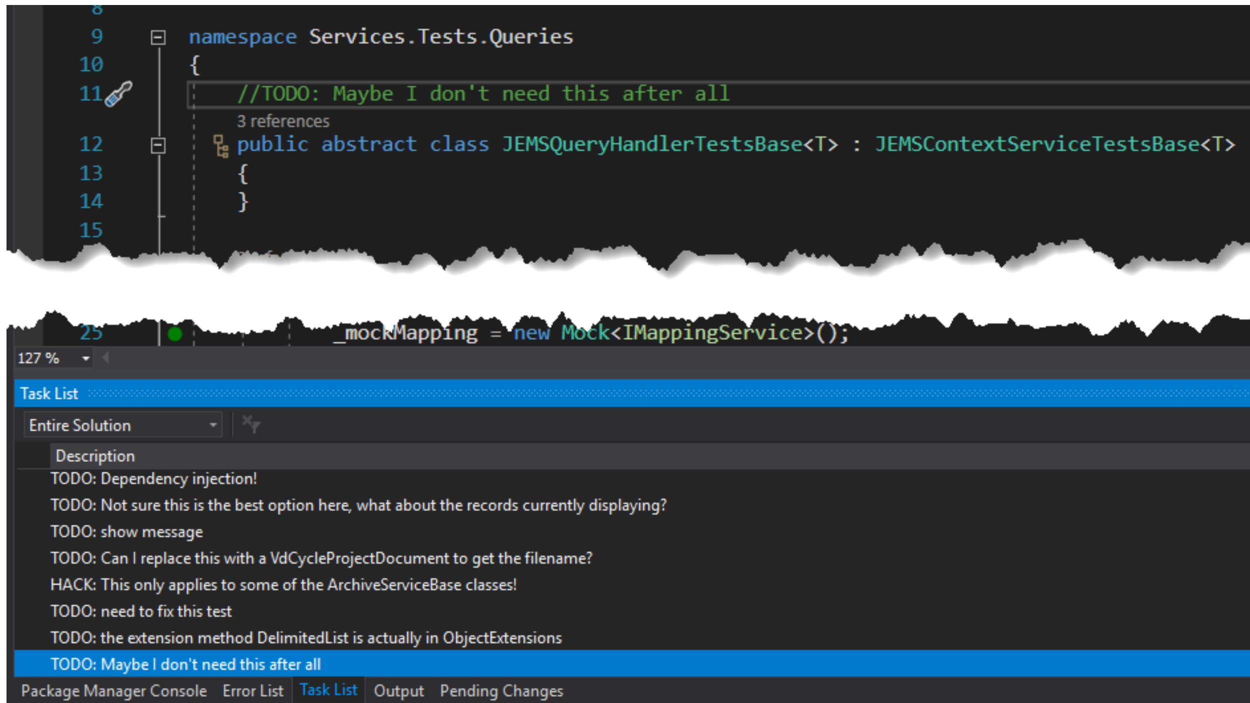

Figure 6 Overgrown TODO list

So TODOs, and the associated code that needs attention become like weeds. Let them grow too long, 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.

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 *Get*UserName, *Check*Balance, or *Draw*Line.

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 20 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 21 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 22 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,”[3] 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 23 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)
{
    return source == target;
}
```

Sample 24 Conditional evaluation--terse

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

[3] Donald Trump, Press Conference, July 17, 2018

- 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 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 25 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 26 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 27 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, name the variable *employees*.

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(numSeperator(userInput))),
    numberSum(numSeperator(userInput))) != false)
```
Sample 28 Bad names hide intent (Java)

Hopefully you're as puzzled by this as I was. Absent any other information about the problem it's trying to solve, it's just about impossible to understand what this code is supposed to do. Later, I'll demonstrate how this previously confusing code can be refactored towards a cleaner solution. Let's talk briefly about classes first.

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. A class with this responsibility might be named *CreditCardValidator*, and it might have a function named *isValid()*. 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. We'll see some of this in action in the following Refactoring sections.

Refactoring to cleaner code: Credit Card Validator

In the additional materials for this class, you'll find the **Practice2_CreditCardProblem** folder, which contains a Java project based on my son's credit card validation assignment. Look in the **Practice2_CreditCardProblem\src\edu\jatc\creditcard** folder for the following files:

- Start.java—the first iteration of the project
- Start2.java—the second iteration of the project, mostly renaming methods
- Start3.java—the third iteration of the project, moving methods to CreditCardValidator, and subsequent reworking of the main method.
- CreditCardValidator.java—final class that does the real work in this project, validating credit card numbers per a specified algorithm

There are also some unit tests found in the **Tests\CreditCardValidatorTests.java**. Even if you don't know Java, hopefully what you've learned to this point will be of value as the code is transformed. In **Start.java** we find the **main** method, complete with some comments explaining the "rules" for validating a credit card number.

```
    // 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.
    // Valid number: 4388576018410707
    // Invalid number: 4388576018402626
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Please enter your credit card number");
        long userInput = input.nextLong();
        if(valid(numbersDoubledSum(doubler(numSeperator(userInput))) ,
numberSum(numSeperator(userInput))) != false)
        {
            System.out.println("Your card is valid!");
        }
        else
        {
            System.out.println("Your card is invalid");
        }
    }
```

The variable *userInput* (line 20 in the source file) is not a good name. Instead, it should be named for what it represents—a credit card number—*creditCardNumber* or maybe *ccNumber*.

Next, the method names *valid*, *numbersDoubledSum*, *numSeperator* and *doubler* are just confusing. They don't reveal intent. We have to go look at the code within each method to have any hope of knowing what each does. Each represents one of the rules, or a necessary step in fulfilling one of those rules.

We renamed these methods to better reflect what each method does, so:

- *numSeparator()* became *getDigits()*
- *numberSum()* became *oddDigitsCheckSum()*
- *doubler()* and *numbersDoubledSum()* were merged into *evenDigitsCheckSum()*[4]
- *valid()* became *isValid()*

Ideally, method names should be some sort of verb, denoting an action that will take place. It's a lot easier to get the gist of these method names than the original *numberSum()* and *numbersDoubledSum()*.

On line 42 we found a magic number:

```
            if(numbers.size() == 16)
```

This was changed to a constant and given the name _CCLENGTH, which you can see in Start2.java, line 15.

```
    private static final int _CCLENGTH = 16;
```

The previously confusing, but all-important line of code from the original program is much more readable in Start2.java, line 23:

```
        if(isValid(oddDigitCheckSum(getDigits(ccNumber)),
evenDigitCheckSum(getDigits(ccNumber))) != false)
```

We could also lose the `!= false` check to make this less confusing.

Looking through the "Start2" class, we see improvement. But the class is still too long, it has too many responsibilities. Validation of the number itself should be moved into a separate class dedicated to just that.

With this in mind, I suggested a new class, named **CreditCardValidator**. This name is a noun, and reveals the intent of the class—as long as we don't add additional responsibilities to it such as *getAccountBalance()*. The renamed functions found in Start2, were then moved to CreditCardValidator. As I did this, I also wrote unit tests (found in **tests\CreditCardValidatorTests**) to confirm that each method behaved as expected.

Unit Tests

Let's take a look at one of the functions **getDigits()** and its associated test, **testGetDigitsReturnsDigitsInReverseOrder().** The purpose of the function was to split a long number up into an array of separate digits, and return the digits in reverse order. The solution below is certainly not the only way to do this. I was trying to preserve as much of my son's

[4] In retrospect, I would have used *calculateOddDigitsCheckSum()* and *calculateEvenDigitsCheckSum()* to make the last two method names more verb-y.

original code as possible, while still refactoring towards a better overall solution. Later, the "internals" of each method can be revisited and improved as needed.

```
    public ArrayList<Integer> getDigits()
    {
        ArrayList<Integer> digits = new ArrayList<>();
        long ccNumber = _ccNumber;
        while(ccNumber > 0)
        {
            digits.add((int) (ccNumber % 10));
            ccNumber /= 10;
        }
        while(digits.size() < _CCLENGTH)
        {
            digits.add(0);
        }
        return digits;
    }
```

And an associated test:

```
    @Test
    public void testGetDigitsReturnsDigitsInReverseOrder()
    {
        //arrange
        CreditCardValidator sut = new CreditCardValidator(4388576018410707L);
        //act
        ArrayList<Integer> actual = sut.getDigits();
        //assert
        assertEquals(16, actual.size());
        assertEquals(7, (int)actual.get(0));
        assertEquals(4, (int)actual.get(15));
    }
```

Unit tests are frequently arranged into three sections: Arrange, Act, Assert. In the first section, we "arrange" the *system under test* (i.e. the class to be tested) by setting up all the conditions necessary to test an object of that type. Here, our SUT is an instance of the CreditCardValidator class, which we instantiate with a sample credit card number.

The second section of the test, performs some action or sets some state (i.e. calls a function) that we can then test for correctness. When some value is returned by the function, I typically name the variable holding the result, *actual*. Here, we call the **getDigits** function and store the returned value.

The "assert" section is where we assert, or validate, whether the actual result from the system we're testing meets our expectations. This is typically in the form of a true/false check asserting that the actual result meets with an expected result.

Each unit test framework has slight variations on the syntax of these statements, but they usually boil down to something like the examples shown below (JUnit and NUnit respectively):

```
assertEquals(expected, actual);
```

AUTODESK. UNIVERSITY

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

There are many more tests included in the sample code for you to look at, each ensuring that the various functions of **CreditCardValidator** are working as expected.

Typically, you should avoid having too many assertions in a single test, and you also should avoid testing too many things in a single test. Each test function should test a very specific behavior of the function in question. Functions with branches should have separate tests for each branch in logic.

There's MUCH more I could say about unit testing (another class all by itself?), but alas, we need to move on. There are some great books on Unit Testing I highly recommend in the Suggested Reading List at the end of this handout.

In **Start3.java**, you can see the final progression of the **main** method.

```
public class Start3
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Please enter your credit card number");
        long ccNumber = input.nextLong();
        input.close();

        CreditCardValidator validator = new CreditCardValidator(ccNumber);
        validate(validator);
    }

    private static void validate(CreditCardValidator validator)
    {
        if(validator.isValid())
        {
            System.out.println(String.format("Your card ending with %s is valid!", 
validator.endsWith()));
        }
        else
        {
            System.out.println(String.format("Your card ending with %s is invalid", 
validator.endsWith()));
        }
    }
}
```

This is the entirety of the Start3 class—approximately 25 lines of code. It's easy to see there's an input section, and a processing/output section. *How* the processing is done is hidden from the caller. My son's previously cryptic line of code turned essentially into this:

```
CreditCardValidator validator = new CreditCardValidator(ccNumber);
if(validator.isValid())
{
       ...
}
```

Sample 29 Good names reveal intent (Java)

That's a lot more readable. Interestingly, very little of the original code within each function needed to be changed. It just needed to be better organized, with better names. So our final ***isValid()*** function in **CreditCardValidator** contains the implementation details of *how* the class determines the validity of a credit card number.

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

        int checkSum = (oddCheckSum + evenCheckSum);
        int remainder = checkSum % 10;

        return remainder == 0;
}
```
Sample 30 Hide implementation details behind well named functions

Even without knowing the details of how *oddDigitsCheckSum()* and *evenDigitsCheckSum()* are calculated, you have a fair sense in this function of what each of the parts do, and how a credit card number's validity are established (well, per the constraints of this particular assignment anyway).

Refactoring to cleaner code: SRP (Single Responsibility Principle) and Abstractions

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.

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.

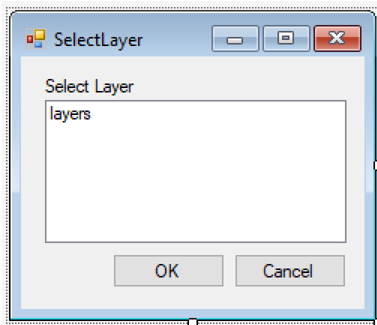

Figure 7 A form to interact with the user

```
    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 31 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, which is to gather input from the user. While your form might need to display a list of data from which the user is supposed to select, it should not typically 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 class that will be using the data. When the OK button is clicked, the dialog 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 31'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 (to display *any arbitrary* list of layers to allow the user to pick one), then we've taken a big step towards writing cleaner code.

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, along with 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 32 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();
    }

    public string SelectedLayer
    {
        get
        {
            return (string)Layers.SelectedItem;
        }
    }
}
```

Sample 33 Injecting dependencies into the form

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 items to the user for selection.

I also added a read-only property, *SelectedLayer*, which returns the *SelectedItem* property of the *Layers* ListBox. While the caller could try and access the *form.Layers.SelectedItem* property, that implies that the caller knows more about the implementation of this form than it really should—namely that I used a ListBox on the form (named Layers), and that the selected value is accessible via the ListBox's 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 necessarily have to change the calling code just because the way we implemented the form might change.

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 34 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 31'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 knows how to access!

In Sample 35, I've provided 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 any potential data source (I just need to write new *ILayerProvider* implementations). 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 35 Injecting ILayerProvider implementations into the LayerSelector form

The sample code for this class includes the SetCurrentLayer function which I excluded here for brevity.

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 can get better with practice.

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

Recommended Reading List

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 learn to write better, cleaner code. This desire has led me to many important books that have helped me improve. Below are some of the more notable books which have helped me on my path.

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

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. I'm more than happy to help!