Development Principles

Writing good code is hard, defining it is even harder. What makes good code good and from what perspective? Execution time (Big O), the “ilities”: readability, extensibility, modularity, reusability, reliability etc. Then there are the functional and non functional requirements like security, performance, and management and operations. What happens when some or many of these desirable characteristics seem to conflict with one another?

Fortunately we have all had experience working with bad code either reading or in my case ahem….writing it 😃 We have all been there trying to find bugs in big balls of mud, scratching our heads wondering why seemingly simple algorithms were implemented with so much unnecessary complexity. Here in lies the nature of our task “Software's Primary Technical Imperative: Managing Complexity” - *Code Complete,* and by extension our technical imperative as software developers is to manage the complexity of the solutions we design. Code Complete goes on to divide complexity into two types: accidental and essential. Some problems are difficult by their nature and possess a certain level of essential complexity that cannot be avoided. What we can and must strive to avoid is the accidental complexity which is complexity we introduce by choosing suboptimal solutions.

Since our job is to write good code, by minimizing accidental complexity, we can derive the principles of how to do that by looking at accidentally complex code and seeing how engineers have improved it in the past. The principles are simply then the recurring themes in the patterns of these improvements.

Fortunately there are a number of great sources to see where engineers have turned bad code into good code, they are called refactorings. A quick google search gives us:

- Catalog of Refactorings
- Refactoring and Design Patterns
- Refactoring (sourcemaking.com)

You can read through all of these patterns and distill for your self the common and repeating themes or you can keep reading because I did it for you 😃

Design Themes

Low Coupling

As developers, this topic comes up often but usually without precise meaning in context. We all know tightly coupled code when we see it and we just hope it is not ours 😉 At a high level there are two types of coupling afferent and efferent (don’t worry, there is no quiz on this). Afferent coupling is when a component (class, package, or method) has many *dependents.* Components with high afferent coupling should be stable and abstract. If they are too specific in their signatures they will need to change frequently to accommodate new use cases and if they change frequently they will cause frequent breaking changes with cascading updates and test cycles required. A classic example of this is the .NET Framework itself. When is the last time Microsoft made a breaking change to the String class that required you to update recompile your application? Efferent coupling is when a component has many dependencies. Components with high efferent coupling should be located in the outer most layers of the Clean Architecture and should be very use cases specific and should not be shared, otherwise when their dependencies change, all of their dependents will need to change as well. What you want to avoid at all costs are components with both high afferent and efferent coupling also called code hubs.

Besides the in and out bound direction of dependency view, there are an additional 11 different sub-types of coupling which I will discuss in exhaustive detail…just kidding 😃 if you are interest you can read Coupling (computer programming) - Wikipedia. The cliff notes version boils down to two very simple principles.

1. Depend on a component’s contract, not its behavior. A corollary to this is do not make assumptions about the side effects of a component’s operations and make design decisions in your own component based on those side effects.
2. If you need to look at the source code to a routine in order to use it, your are too coupled to it.

Here are some examples of refactorings that reduced the coupling in code:

Replace Query with Parameter

```
targetTemperature(aPlan)

function targetTemperature(aPlan) {
  currentTemperature = thermostat.currentTemperature;
  // rest of function...
```

refactored to:

```
targetTemperature(aPlan, thermostat.currentTemperature)

function targetTemperature(aPlan, currentTemperature) {
  // rest of function...
```

In this example the targetTemprature function was coupled to the thermostat class and the caller of targetTemprature now had a transient dependency on the thermostat class and any side effects produced by calling currentTemprature. The logic of calculating the target temperature could not be unit tested absent the thermostat class and any possible I/O bound side effects or requirements.

This one is subtle. Here the calling code is Common Coupled to the totalAscent variable. Any code that reads or modifies this variable needs to make assumptions about its state which when false, will result in bugs. This also tends to lead to temporal coupling because you come to depend on routines being called in a specific order to maintain what you believe to be the correct state of totalAscent.

```
let totalAscent = 0;
calculateAscent();

function calculateAscent() {
  for (let i = 1; i < points.length; i++) {
    const verticalChange = points[i].elevation - points[i-1].elevation;
    totalAscent += (verticalChange > 0) ? verticalChange : 0;
  }
}
```

Decoupled:

```
const totalAscent = calculateAscent();

function calculateAscent() {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    const verticalChange = points[i].elevation - points[i-1].elevation;
    result += (verticalChange > 0) ? verticalChange : 0;
  }
  return result;
}
```

Finally a Separate Query from Modifier refactoring:

Anytime you use the word “and” in am method name, that is a pretty good indication that you are violating the Single Responsibility Principle.

```
function getTotalOutstandingAndSendBill() {
  const result = customer.invoices.reduce((total, each) => each.amount
+ total, 0);
  sendBill();
  return result;
}
```

Better: This refactoring is an example of Command Query Responsibility Segregation. Stated simply, code should tell you something or do something but not both.

```
function totalOutstanding() {
  return customer.invoices.reduce((total, each) => each.amount + total,
0);
}
function sendBill() {
  emailGateway.send(formatBill(customer));
}
```

High Cohesion

The notion of coupling can be a little hard to define in context which is why I actually prefer to think in terms of Cohesion. Cohesion, quite simply, is a measure of the degree to which things that are used together are scoped together. This concept is so well defined it has a mathematical measure called the Lack of Cohesion Of Methods (LCOM) which works as follows:

LCOM = 1 – (sum(MF)/M*F)

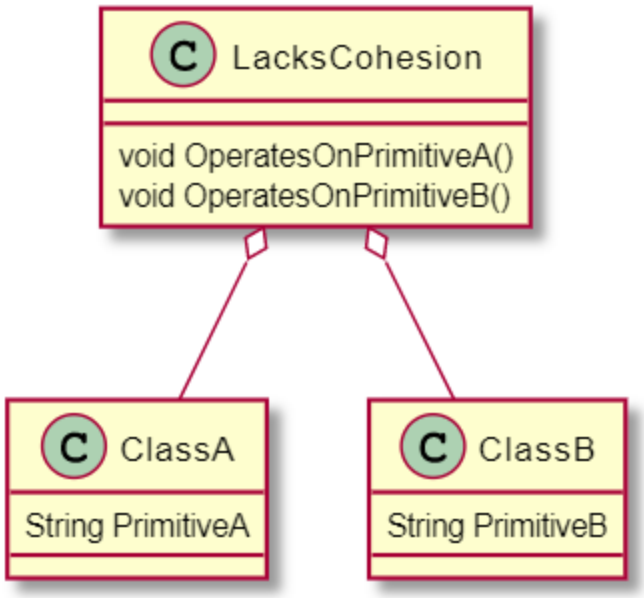
Where:

M is the number of methods in class (both static and instance methods are counted, it includes also constructors, properties getters/setters, events add/remove methods).
F is the number of instance fields in the class.
MF is the number of methods of the class accessing a particular instance field.
Sum(MF) is the sum of MF over all instance fields of the class.

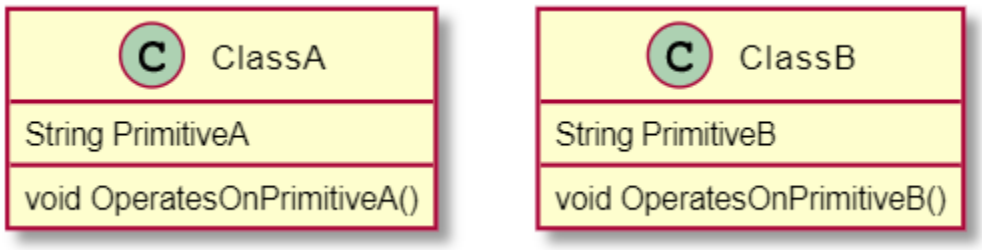
This produces a number between 0 and 1 with lower being better.

The underlying idea behind these formulas can be stated as follow: a class is utterly cohesive if all its methods use all its instance fields, which means that sum(MF)=M*F and then LCOM = 0.

The interesting consequence of focusing on Cohesion over coupling is you often get loose coupling for free! take, for instance, the following class:

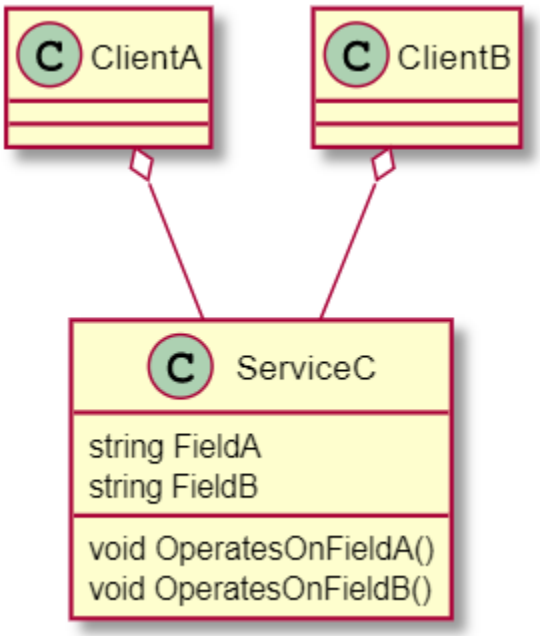


This class has two fields each operated on separately by two methods. So calculating the LCOM metric: M=2, F=2, MFa = 1, MFb=1, SUM(MF) = 2, LCOM = 1-(2/(2*2)) = .5. So clearly we can do better:



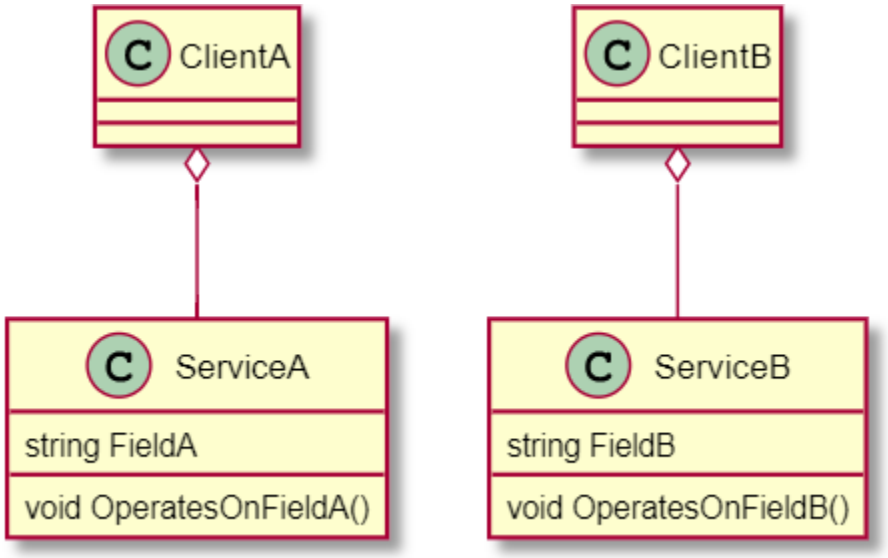
By moving the operations into the classes we were operating on we create two classes that have an LCOM of 1-(1/(1*1)) = 0, perfect cohesion! But look carefully what what else happened…**the coupling was reduced**!

Here is another example:



In this example ServiceC has high Afferent coupling with the two client classes dependent on it. Unless ServiceC is very abstract this can become a problem if it needs to change to support other use cases than those are likely to be breaking changes for the existing client classes or introduce the possibility of regression defects.

Better:



In this example you end up with two services that have high cohesion and low afferent coupling. The moral of the story: **Focus on cohesion and you often get lower coupling for free!**

Helpers & Utilities

As a corollary to the ideas of High Cohesion and Low Coupling, I would like to discourage naming classes Helper and Utility. This is based on my personal experience of looking at a lot of code over a long career and whenever I see a class named Helper or Utility I know where I need to begin refactoring! These classes are the dark corners in our code where high coupling and low cohesion tends to hide. The motivation to create them is often because you need to preform operations and cannot find a place in the Clean Architecture model where these operations fit nicely. If you find yourself in this spot I would encourage you to do two things: First, make sure there are no IO dependencies. If there are, this belongs in a controller or repository layer. Also make sure there are no business or application behavior rules because these belong in the core layers under test. The second thing I would encourage you to do is examine the nature of the operations you are preforming. Are they some type of specialized parsing or mapping? If so, these can be done as extension methods to the types the operations are being preformed on. There is nothing inherently wrong with naming a class Helper or Utility, but these classes tend to be like dust balls that collect more dust and grow over time. Also, developers often make them static and if they preform IO operations they make whatever classes that use them untestable and if it is not testable it is detestable 😃. Finally, **you cannot spell Helper without Hel** 😃

Abstraction

Ever wonder why abstract art that looks like your toddler can draw or paint it sells for millions of dollars? It is because abstraction in one of those things that looks easy but in reality is very difficult. At is heart abstraction = simplification but simplification is not really simple. Take for example this famous abstract model of matter, energy, space, and time:

$$E=MC^2$$

Many middle school students can be taught to plug in it's parameters and calculate its results, however deriving this abstract model took one of the greatest minds the world has ever known.

Perhaps we can try to understand it by understanding what it is trying to accomplish:

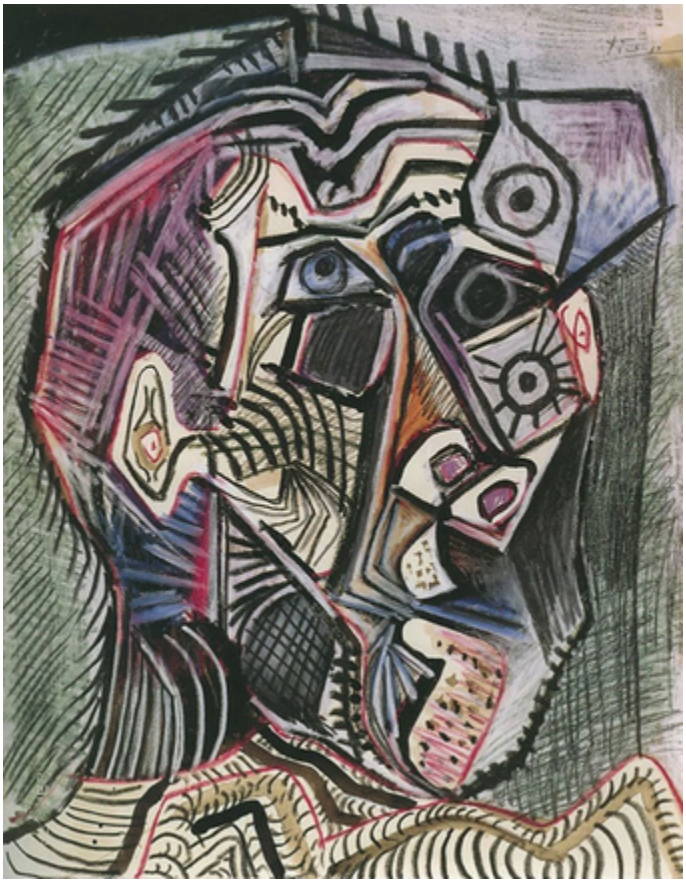
"The goal of "abstracting" data is to reduce complexity by removing unnecessary information"

or

"the process of removing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance"

Got it? Hymm…oaky…for us visual learners:

Here the artist Rembrandt is trying to portray the essence of himself but in a realistic context which shows his age and clothing and hair style which give us an insight into things like his stage of life, location and time period perhaps, and possibly his economic condition but may or may not necessarily tell us how he perceives himself. In this case we cannot easily sperate his nature as a person from the context in which he is portraying himself.



Another self portrait, this time by Pablo Picasso. Here you can see Picasso has achieved every goal of abstraction by removing all physical, spatial, and temporal details. Looking at this painting we cannot tell his age, location, economic condition, or any other details other than what he wanted us to see which is what he was trying to say about himself as a person…just don't ask me what that is 😃

So abstraction is meaning minus context or in code algorithm minus specific use case.

This is one of my favorite refactorings from both Martin Fowler and Uncle Bob Martin and apparently written by Rembrandt himself 😃

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

Here we see a lot of detail and complexity in the conditional statement. It is difficult to understand, at a glance, what the *meaning* of this constraint is.

Here is the version after Picasso refactored it:

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

Here we instantly see the meaning of the condition but the complexity of how this is determine is completely hidden from us.

This is another Fowler example called Extract Variable:

```
    return order.quantity * order.itemPrice -
      Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
      Math.min(order.quantity * order.itemPrice * 0.1, 100);
```

While terse, this statement is chalked full of details about the algorithm's parameters, operations, and steps but tells us absolutely nothing about what they mean or what it is trying to accomplish.

Better:

```
    const basePrice = order.quantity * order.itemPrice;
    const quantityDiscount = Math.max(0, order.quantity - 500) * order.
itemPrice * 0.05;
    const shipping = Math.min(basePrice * 0.1, 100);
    return basePrice - quantityDiscount + shipping;
```

This is important enough to warrant a 3rd example:

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
```

```
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

Here we see functions doing basically the same thing but highly tailored to two different use cases and therefore not very reusable. If these functions were in a shared component this lack of abstraction will cause problems when HR decides they are no longer going to give 5% and 10% raises but raises on a continuous scale between 2% and 12%.

This refactoring is an example of generalization:

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

The other abstraction related refactorings are worth studding in detail:

- Long Method
- Large Class
- Primitive Obsession
- Data Clumps
- Divergent Change
- Change Value to Reference
- Combine Functions into Transform
- Encapsulate Collection
- Hide Delegate
- Introduce Parameter Object
- Introduce Special Case
- Remove Flag Argument
- Replace Conditional with Polymorphism
- Replace Constructor with Factory Function
- Replace Function with Command
- Replace Inline Code with Function Call
- Replace Loop with Pipeline
- Split Phase
- Split Variable

Per the Stable Abstraction Principle the more dependencies a component has the more abstract it should be. Code in the core of the domain model in Clean Architecture should only represent domain concepts and contain nothing about the context or use cases the model is being used in.

The main idea behind the principle of abstraction is to "**expose simplicity and hide complexity**".

Stability

Stability, as s programming principle, is different from the concept of reliability in architecture. Stability referrers to the relationship between design components and calls from you to depend on things more stable than the component you are adding or modifying. The two principles that govern stability are the Stable Abstraction Principle and the Stable dependencies principle.

Refactoring patterns that demonstrate stability include removing the use of the new keyword by depending in injected interfaces and avoiding Shotgun Surgery.

Here is an example form Combine Functions into Transform:

```
function base(aReading) {...}
function taxableCharge(aReading) {...}
```

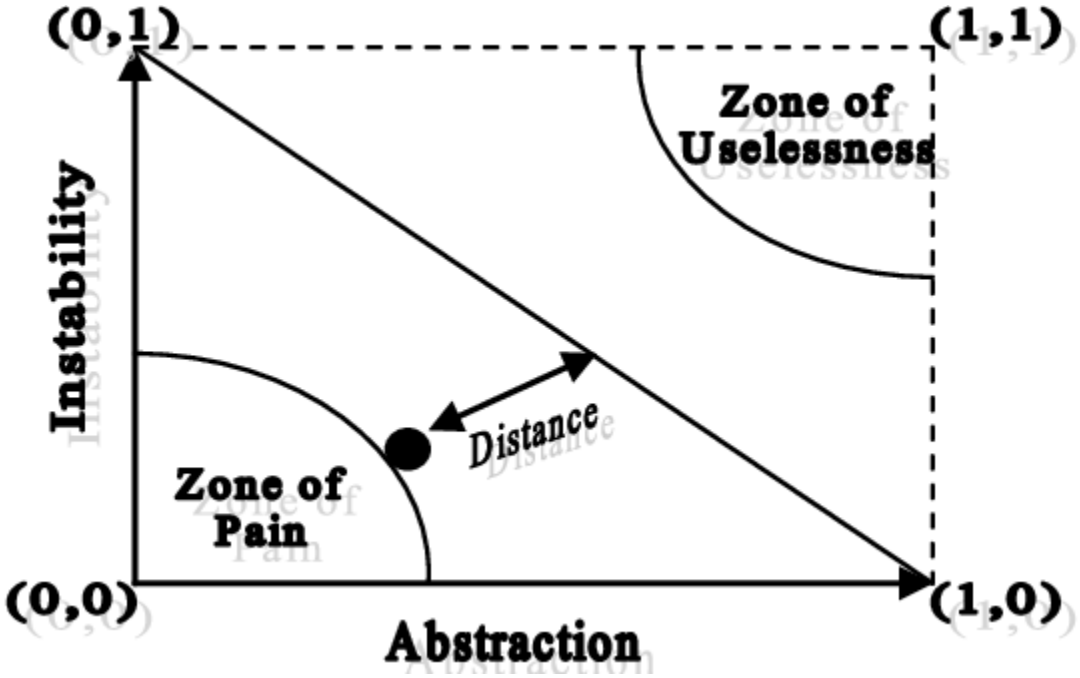
In this example a caller depends on two functions that each preform part of a task when it would be better to depend on a single function that encapsulates the entire task. This way if the details of how this task are preformed change, the caller need not necessarily change.

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge = taxableCharge(aReading);
  return aReading;
}
```

As you can see the ideas of stability, encapsulation, and abstraction are closely related and can be used together to drastically improve software design.

Distance From Main Sequence

We can algorithmically calculate the quality of a component, class, or even function relative to its balance between coupling and abstraction by measuring its Distance From Main Sequence.



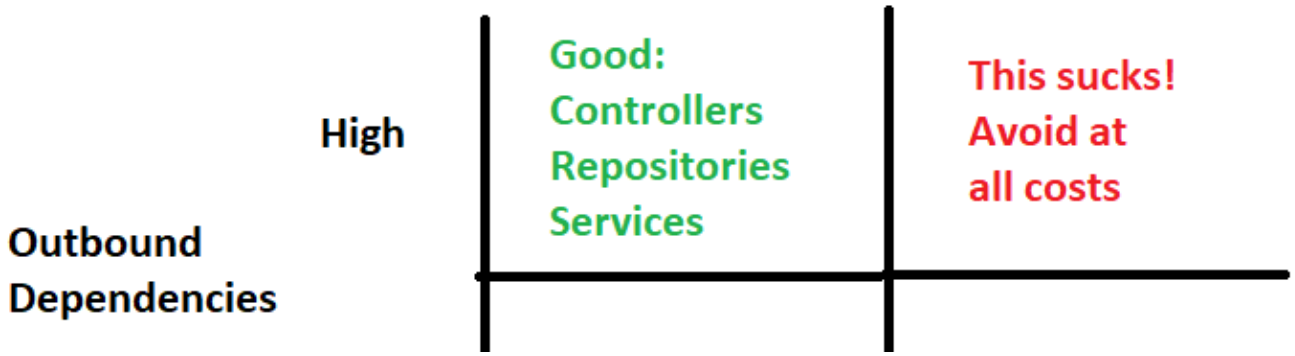
This graph illustrates the relationship between the factors of abstraction and stability but implied is the assumption* that the more dependencies a module has the more stable it must** be, otherwise we would naturally*** refactor it since it would become a constant source of breaking changes and regression defects. So in this diagram high instability means fewer dependencies and lower instability means more dependencies???? Yea, I thought it was confusing to so I came up with a better model (see below). I am presenting this so you can understand how this relationship is calculated using static code analysis software like NDepend.

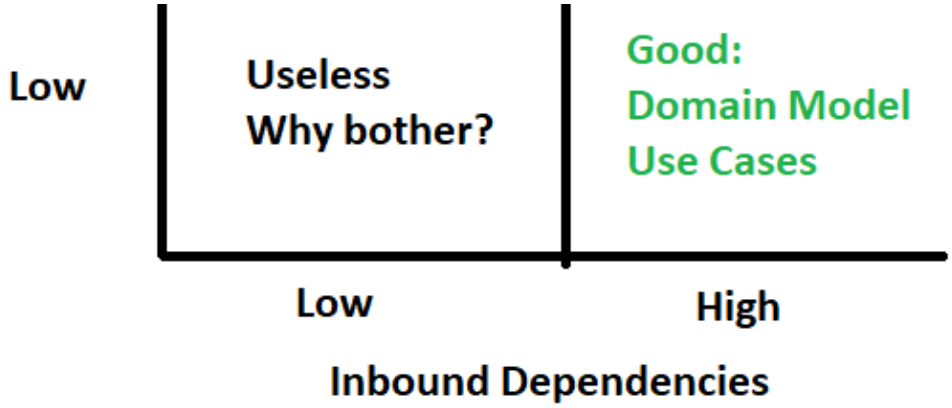
*See the Liskov Substitution Principle in the SOLID Principles document.

, * Nope, I have seen companies suffer with these for years bitterly refusing to refactor them and instead handing our Employee Assistance Program pamphlets to the developers and telling them to suck it up and deal with it.

> The perpendicular normalized distance of an assembly from the idealized line A + I = 1 (called main sequence). This metric is an indicator of the assembly's balance between abstractness and stability. An assembly squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal assemblies are either completely abstract and stable (I=0, A=1) or completely concrete and instable (I=1, A=0). The range for this metric is 0 to 1, with D=0 indicating an assembly that is coincident with the main sequence and D=1 indicating an assembly that is as far from the main sequence as possible. The picture in the report reveals if an assembly is in the zone of pain (I and A both close to 0) or in the zone of uselessness (I and A both close to 1).

Think about it this way. A class like the .net String class has a ton of code depending on it bit it only depends on Win32 APIs which it is abstracting. What if the String class had a lot of external dependencies or even made out of process calls to external APIs!!!? What a nightmare. The .NET framework would constantly be pushing mandatory updates and your application's performance would suck. This is what we want to avoid in our own code so to help, I propose the following diagram, which says the same thing as the previous diagram just in a much better way.





DRY

Dry is an acronym for Do Not Repeat yourself which states that we should strive toward abstraction and encapsulation by centralizing duplicate code. This however can get one into trouble because we as humans are good and seeing repeating patterns even when they do not accurately represent a true duplication of a concept or trend. Just ask anyone when as lost money day trading using technical indicators. Taken to an extreme removing duplicate symbols from code can make it less readable and is actually a job best left to a compiler.

So how do we know when to refactor? The rule of tree, three strikes and you refactor!

Transparency

Do not lie! Now I do not believe anyone here would ever intentionally be dishonest but in prior engagements I have seen code that appears to do one thing but in fact does another. Some call this the principle of least surprise or principle of least astonishment but 'Uncle' Bob Martin is a little more firebrand in his advocacy for these principles by pointing out that when your code says one thing and does another, it, and by extension you, are in essence lying. These lies typically take three forms:

Side effects

Put quite simply "Side effects are lies. Your function promises to do one thing, but it also does other hidden things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies." - Martin, as a result our abstract code **must** be free of side effects and where side effects are required, as part of I/O handling, this code **must** be relegated to the outer most layer in the Clean Architecture.

Changing Code

Ideally if we are following the Open Closed Principle we would only be adding new code and not modifying existing code however this is not always possible, especially when dealing with legacy code. When code is changes such that its interface signature or class, method, parameter, or variable names no longer reflect what it is actually doing or what it actually means, the code becomes deceitful. To prevent this, when you are changing code, be sure to update the names to make sure they accurately describe what is going on. It is tempting to avoid this rename refactoring when something is changing inside a highly shared component, which would not happen if we pay attention to stability, but is it something we must do to keep our code honest.

Exception Handling

This is where I have seen some of the most damaging lies. I will cover exception handling in more detail later but the most common ways I see exception handling lie is by out right swallowing the exception and not rethrowing it and by returning a "empty", null, or zero result in response to an exception rather than letting the caller know something went wrong. The best way to deal with this is to not catch exceptions in the first place until you are at a service layer where you can clean up any state changes you have made or at an IO layer where you can communicate to the caller what happened and what they can expect as a result.

Comments

"Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation." - Martin. Bob Martin has an entire chapter on comments in his book Clean Code that is worth reading in its entirety. The reason he points out as to why comments can become deceiving is because "The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong."

The general principle of code honesty is no different than interpersonal honesty: Say what you are going to do and do what you say.

Here are some examples of refactoring with the objective of increasing transparency (aka honesty).

Change Function Declaration

A rename refactoring to increase clarity and transparency

`function circum(radius) {...}`

`function circumference(radius) {...}`

Replace Derived Variable with Query

This is a subtle example but here the setter method for discount does more than just set the discount, it also modifies the discountTotal.

```
  get discountedTotal() {return this._discountedTotal;}
  set discount(aNumber) {
    const old = this._discount;
    this._discount = aNumber;
    this._discountedTotal += old - aNumber;
  }
```

Better:

```
  get discountedTotal() {return this._baseTotal - this._discount;}
  set discount(aNumber) {this._discount = aNumber;}
```

In this refactored example each function is doing exactly what it is saying it is doing.

Return Modified Value

In general we should prefer pure functions whose inputs and outputs are explicitly defined and whos outputs are a deterministic function of its inputs because it is easier for this code to be honest. This is especially true for functions in the inner layers of the Clean Architecture.

```
let totalAscent = 0;
calculateAscent();

function calculateAscent() {
  for (let i = 1; i < points.length; i++) {
    const verticalChange = points[i].elevation - points[i-1].elevation;
    totalAscent += (verticalChange > 0) ? verticalChange : 0;
  }
}
```

Refactored to:

```
const totalAscent = calculateAscent();

function calculateAscent() {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    const verticalChange = points[i].elevation - points[i-1].elevation;
    result += (verticalChange > 0) ? verticalChange : 0;
```

```
  }
  return result;
}
```

In summary: **Code should say what it does and do what it says,**

Algebraic Expressions

While not as critical as the other factors, refactoring code in more of an algebraic style can make your code more concise without sacrificing, or actually improving, its readability. Let's see some examples:

Remove Flag Argument

```
function setDimension(name, value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```

refactored to a more declarative and algebraic format:

```
function setHeight(value) {this._height = value;}
function setWidth (value) {this._width = value;}
```

Replace Derived Variable with Query

```
  get discountedTotal() {return this._discountedTotal;}
  set discount(aNumber) {
    const old = this._discount;
    this._discount = aNumber;
    this._discountedTotal += old - aNumber;
  }
```

becomes:

```
  get discountedTotal() {return this._baseTotal - this._discount;}
  set discount(aNumber) {this._discount = aNumber;}
```

Replace Inline Code with Function Call

```
  let appliesToMass = false;
  for(const s of states) {
    if (s === "MA") appliesToMass = true;
  }
```

more succulently written as:

```
appliesToMass = states.includes("MA");
```

Replace Loop with Pipeline

```
  const names = [];
  for (const i of input) {
    if (i.job === "programmer")
      names.push(i.name);
  }
```

better written as:

```
  const names = input
    .filter(i => i.job === "programmer")
    .map(i => i.name)
  ;
```

Substitute Algorithm

```
function foundPerson(people) {
  for(let i = 0; i < people.length; i++) {
    if (people[i] === "Don") {
      return "Don";
    }
    if (people[i] === "John") {
      return "John";
    }
    if (people[i] === "Kent") {
      return "Kent";
    }
  }
  return "";
}
```

More concisely written as:

```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"];
```

```
  return people.find(p => candidates.includes(p)) || '';
}
```

Runtime Considerations

Performance

Each language has its own performance characteristics and therefore its own optimization advice. Since this article is meant to be fairly language agnostic I will focus on the performance issues that tend to bite you regardless of the language you are using. And really….there are only two worth remembering. That said, they are so valuable to remember that should you become intoxicated enough to get a tattoo you later regret, make that tattoo a reminder not to do these three things when your write code 😃

Pulling too much data into memory

So you should never run into this issue because you should always be running, and watching, a memory profiler as you are writing your code….. Wait… you mean you don't?! Yeah, me either! 😃 This is an easy trap to get caught by because when you are developing on your local environment you have all the systems memory resources at your singular disposal. Let's say you have 24GB of free memory after OS and applications, this roughly means you can process a 24GB file or pull 16GB of data into .NET core from a SQL server table.

A query like this may work just fine:

```
SELECT * FROM Products
```

Assuming we are running .NET Core and Kestrel web server with default configuration that 24GB of free memory is shared by a possible 100 connections. If even a few of those connections run that query your server will crash.

There are three possible solutions:

1. Always make sure you have applied sensible constraints to your SQL queries, like selecting a single record based on a primary key, or that you are selecting from a table with a known upper bound to its size like US States.
2. Page your search results all the way to the data store. This also applies if you are not able to apply a sufficient constraint in advance to limit your result set to 500 records or fewer.
3. Process your files in a stream without necessarily reading the entire file into memory first. The latest versions of .NET and SQL server will also let you stream queries. Let's see what that would look like:

Making out-of-process calls in a loop

This one tends to take two forms. The dreaded N+1 select problem that tends to plague ORM frameworks and making REST API calls in a loop. Since ORMs abstract away the generation and rendering of SQL, this problem can be difficult to detect by simply reading the code but since we are not a true ORM shop I will not belabor this one. REST API calls in a loop are another story. No code example required for this one just simply remember the following two rules:

1. Please do not make REST API calls in a loop
2. If you ever feel you need to REST API calls in a loop, please refer to rule 1 above

This advice only apples when you are in a time sensitive situation like serving end user requests. Obviously if you are doing things like offline file processing, ETL, batch jobs, or batch queue message processing, loop away!

Calling .Wait, .Result, or GetWaiter().GetResult() in ASP.NET application

Please take the time to read this article in its entirety.

Async/Await - Best Practices in Asynchronous Programming | Microsoft Docs

The key takeaway is your call stack needs to be all async or all sync but please **NEVER** call Task.Wait or Task.Result to try and prevent needing to refactor your entire dependency graph to be async. If you cannot convert all your dependencies to be async you are better of leaving it synchronous. The small amount of scalability you loose by not taking advantage of async / await pales in comparison to what you loose by permanently deadlocking a thread until the app pool recycles or the server is rebooted. Furthermore, if you come across uses of Calling .Wait, .Result, or GetWaiter().GetResult() in the course of your development work, please take the time to refactor them out.

Additionally, when writing async code make sure to chain your **CancellationTokenSource** so that async continuations can be abandoned and cleaned up and the appropriate transient fault error message can be returned if if a timeout is exceeded.

Exception Handling

To write exceptional code we need to do an exceptional job of exception handling, no exception! 😄 Exception handling is such an important concept that Bob Martin devotes an entire chapter (7) to it in Clean Code. I see a lot of well intended attempts made to write "bullet proof" code

that not only fail to make the code more robust, they actually, in many cases, cause more defects than they were trying to prevent. This usually manifests itself in three antipatterns:

- Swallowing exceptions - you should never swallow an exception. If something was not important enough to ensure it executed correctly, it was probably not important enough to have done at all.
- Catching but not adding value - only catch an exception if plan to do something useful. See below. In general only the I/O bound and domain service components should catch exceptions. The core of the domain model in Clean Architecture should never even contain a try / catch block because all inputs should be validated prior to invoking it and it should mostly contain pure functions free of I/O bound side effects.
- Catching but rethrowing incorrect information (AKA lying)

I see this anti-pattern a lot in API controller layers:

```
try
{
    //Invoke a service layer that has an out-of-process dependency
    var response = await service.GetAsync(request);

    if (response.IsSuccessStatusCode)
        return Ok(await response.Content.ReadAsStringAsync());
    else return BadRequest("There was an error ");
}
catch (Exception ex)
{
    return BadRequest($"There was an error: {ex.Message}");
}
```

This code lies because it throwing a 400 level (client) error implying that they did something wrong when this could be an issue with a downstream dependency being down or having a bug. Better would be to catch a specific exception that could indicate if the issue was because of user input or a bug or the availability of a dependent service or down stream system then throw the appropriate HTTP status code 4XX, 500, or 503 to give the caller truthful and accurate information as to the state of the system.

```
try
{
    //Invoke a service layer that has an out-of-process dependency
    var response = await service.GetAsync(request);

    return Ok(await response);
}
catch (ValidationException valEx)
{
    return BadRequest($"Invalid request: {valEx.Message}");
}
catch (ServerTransientException transEx)
{
    return Problem("Downstream service unaviable", request.ToString(),
 503);
}
catch (ServerPersistentException ex)
{
    return Problem("We goofed!", request.ToString(), 500);
}
```

- Rethrowing a new exception and loosing the original stack trace in the process - how many times have you seen the following?

```
try
{
…do something that can fail
}
catch(Exception ex)
{
...some handling code
throw ex;
}
```

This is a subtle issue in C# where the stack trace is lost because the .NET framework creates a new Exception object when it enters the catch block so the exception will appear to have come from the catch block and not the code where it originated. Better would be to use the throw statement by itself:

```
catch(Exception ex)
{
...some handling code
throw;
}
```

When you catch an exception, you should do something useful like:

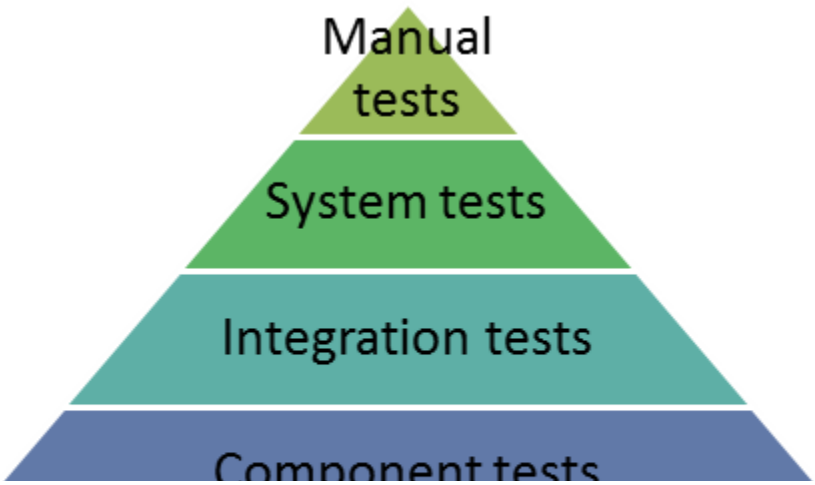
- Logging additional information that would not have been captured by APM logging or an AOP logging solution. These would include things like parameters to a SQL query or REST API call or the path to a file or blob.
- Handle a transient fault by retrying an idempotent request
- Restoring the correct state of the system if the exception occurred mid way through a multi-step process. This would include rolling back a SQL transaction or executing a compensating transaction as part of a saga pattern.
- Formatting the exception for presentation to the user. This should only be done at the controller layer in Clean Architecture. When we format and rethrow an exception for consumption by the caller there are only three types of exceptions they care about.
    - Server Transient - these are temporary anomalies that occur as a result of a dependent system or piece of hardware being down or offline. It is important to communicate to the user that this is transient in nature and set the expectation that if they try later their is a greater possibility that it will work.
    - Server Persistent - This is usually the result of a defect in the code. It is important to communicate to the user that we messed up but NOT to expect this to be fixed quickly. This will require detection, remediation, testing, and release. This is a multi hour or possibly multi day or even multi week issue, not a “go have lunch and come back and try again later” issue.
    - Client- If this was the result of errant input on the caller’s part, we need to tell them exactly what they did wrong so they can take the appropriate steps to remediate the issue on their end.

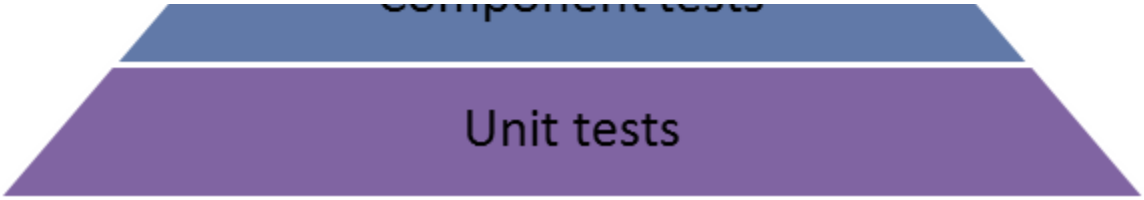
The guiding principle here is to be obsessed about the customer and providing a positive customer experience by being as transparent with the customer as possible and about giving appropriate guidance and setting appropriate expectations. One way to ensure this is done consistently is with domain level error types similar to TaxService/ApiErrorTypes.cs at main · jcbowers/TaxService · GitHub.

Unit Testing

True unit tests should be written according to F.I.R.S.T principles. Of the first principle, the most important is Isolated. If a unit test requires an out-of-process connection to a Database, REST API (or mock of a REST API), or even an internet or network connection to run, then it is not a unit test. It may be a system test but it is not a unit test.

In terms of test composition we want to follow the relative proportions described in this software quality test pyramid:





All domain core and domain service components should have unit tests that cover, at a minimum, the cyclomatic conditions in the implementation of their business rules. Because business rules can sometimes mean different thigs to different people, they are worth defining here: *Business rules describe the operations, definitions and constraints that apply to an organization regardless of the nature of the system implementing them or even if their is a digital system at all*. Because business rules describe operating constraints absent system implementation details, they are by definition abstract and should be implemented at the core of the domain model in clean architecture. Component tests should be used at the service layer and integration tests at the controller layer with the distinction between integration and system tests being that integration tests use mocks while system tests use live connections.

Further Reading

This article is a summary of the best tips I have acquired over 20+ years of doing software development. My sincere hope is regardless your level, you have found something in here that helps make your development experience more productive, fulfilling, and enjoyable. I opened this article by saying writing good code is hard, but in as much as it is challenging it is also fun. Below are the books that I have read that helped me transition from developer to lead developer to architect to senior architect and beyond. I understand your time is the most precious thing you have but I promise you the time spent reading these books will pay you back a thousand fold or more over the course of your career.

Best Wishes,

Jason

Clean Code: A Handbook of Agile Software Craftsmanship 1, C., Martin Robert, eBook - Amazon.com

Amazon.com: Domain-Driven Design: Tackling Complexity in the Heart of Software eBook: Eric, Evans: Kindle Store

Clean Coder, The: A Code of Conduct for Professional Programmers (Robert C. Martin Series) 1, Martin, Robert C., eBook - Amazon.com

Code Complete: A Practical Handbook of Software Construction, Second Edition: McConnell, Steve: 0790145196705: Amazon.com: Books

Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series) 1, C., Martin Robert, eBook - Amazon.com