



Khalil Stemmler

S O L I D

S O L I D

Introduction to **Software Design and Architecture** with TypeScript

Learn to write testable, flexible & maintainable code.

solidbook | The Software Design & Architecture Handbook

Learn to write testable, flexible & maintainable code

Khalil Stemmler

Contents

Intro	14
Why did you decide to write this book?	14
Why learn software design and architecture?	16
The Domain-Driven Developer	18
Software design is taking an educated guess at the future	18
Thank you	18
I. The World of Software Design and Architecture	19
Introduction	19
Software development is a young profession	19
First Principles	20
Chapter goals	21
Let's talk about you	21
The goal of software	21
Users' technical expectations vary based on their needs	22
System quality attributes (SQAs)	24
How do we design a project to be successful?	24
Aren't all system quality attributes essential?	24
What is architecture?	25
Why is software architecture important?	25
What is software design?	25
Levels of design	26
The Software Design and Architecture Stack & Roadmap	26
Resource: The Stack	27
Resource: The Map	28
Step 1: Clean code	29
Clean Code	29
Step 2: Programming paradigms	30
Step 3: Object Oriented Programming and Domain Modeling	32
Step 4: Design Principles	33
Step 5: Design patterns	34

Step 6: Architectural Principles	36
Step 7: Architectural Styles	36
Step 8: Architectural Patterns	39
Step 9: Enterprise patterns	40
Chapter conclusion	40
References	41
2. TypeScript	41
Introduction to TypeScript	41
Chapter goals	42
Primary goals of TypeScript	42
All JavaScript is valid TypeScript	43
TypeScript types	43
Convenient Implicit Types	44
Explicit Types	44
Structural Types	45
Nominal typing	45
Duck typing	46
Ambient types	48
Migrating to TypeScript	49
Typescript still emits errored JavaScript code	49
Why JavaScript doesn't scale	49
When to use TypeScript	50
Categories of hard software problems	50
1 - The performant system problem	50
2 - The embedded system problem	51
3 - Complex domain problem	51
Object-Oriented JavaScript	52
Code size	52
Production software vs. pet projects	53
Lack of Unit Tests	53
Startups	53
Working on Teams	54
Large teams	54
Communicating patterns & implementing design principles	54
Smaller teams & coding styles	56
Frameworks	57
React	57
Angular	57
Summary on when to use TypeScript	57
Getting started with TypeScript	58
Prerequisites	58
Initial Setup	58
Setup Node.js package.json	58
Add TypeScript as a dev dependency	59
Install ambient Node.js types for TypeScript	59
Create a tsconfig.json	59

Create the src/ folder and create our first TypeScript file	60
Compiling our TypeScript	60
Our compiled code	60
Useful configurations & scripts	60
Cold reloading development script	60
Creating production builds	61
Production startup script	61
View the Starter Project source	62
Scripts Recap	62
TypeScript Language Features	62
Basic types	62
Primitive Types	62
Number	63
String	63
Boolean	63
Arrays	63
Object-Oriented Programming Features	63
Classes	64
Class inheritance	64
Static properties	65
Instance variables	65
Access Modifiers	66
Readonly Modifier	67
Interfaces	67
Classes implementing interfaces	68
Interfaces extending interfaces	68
Generics	69
Convenience Generic	70
Abstract classes	71
Special types	71
Type assertions	71
The “type” keyword	72
Type Aliases	74
Union Type	74
Intersection Type	75
Enum	75
Any	76
Void	77
Inline & Literal Types	77
Type Guards	79
Typeof Guard	79
Instanceof Guard	79
In Guard	80
Chapter Summary	80
Resources	80
References	80

3. Clean Code	80
Introduction Clean code is your grip strength	80
Chapter Goals	81
Understanding clean code	82
Clean code is an overloaded term	82
What the community thinks about clean code	83
Community opinions	83
What the experts think about clean code	85
How does unclean code get written?	86
Two laws of software development maintenance	88
Why it's hard to learn clean code	88
Reason 1 — Humans are complex	88
Reason 2 — It's hard to deconstruct human psychology	88
Reason 3 — Trade skills are acquired through mentorship	89
The three pillars of clean code	89
Developer mindset	89
Coding conventions	89
Theory & knowledge	90
Notes	90
How I recommend learning clean code	90
Section One - Developer mindset	91
Summary	91
Pragmatism and Craftsmanship	91
Programming as a trade instead of an engineering practice	91
Similarities to the apprenticeship model originating from medieval Europe	92
The Agile manifesto — a code of honour for software developers	94
The Software Craftsman's Manifesto — extending and challenging the Agile Manifesto	94
Pragmatism	95
Programming is a trade	95
Other trades are normalized	96
Growth	96
Two mindsets	97
Fixed mindset	97
Growth mindset	97
You need to have a growth mindset as a software developer	97
Fixed mindset as a developer	97
Growth mindset as a developer	98
Empathy	100
Developer Experience: Give humans super powers	100
Developer experience (DX)	100
Empathy	101
Aim to build awesome developers (you and your teammates)	102
Task vs. feeling	103
Human-Centered Design: All developers should learn how to design for humans	104

Design = balancing priorities	104
You are always building an API	105
Introducing Human-Centered Design (HCD)	105
Principles of design for humans	107
Summary on Human Centered Design	107
Testing code against humans to determine code cleanliness	107
Resistance to change	109
Don't be a jerk	109
Section Two - Clean coding conventions	110
Summary	110
About this section	110
Conventions we'll cover	110
Comments	110
Code explains what and how, comments explain why	110
Comments clutter code	112
Turning comments into clear, explanatory, declarative code	113
Bad comments	114
When to write comments	115
Demonstration	115
Relationship to Human Centered Design	118
The relationship between comments & human centered design	118
More doors examples	118
Naming things	119
The seven principles of naming	119
The seven principles of naming	120
Summary	120
1 - Consistency & uniqueness	120
Consistency	120
Uniqueness	122
Best practices	123
2 - Understandability	126
Knowledge in the world	126
Representing real-world concepts	126
Best Practices	130
3 - Specificity	132
Over-specifying	133
Under-specifying	135
Best practices	136
4 - Brevity	140
Compression vs. Context	140
The law, reiterated	141
Best practices	141
5 - Searchability	145
Best practices	145
6 - Pronounceability	146
Best practices	146
7 - Austerity	147

Not everyone has the same sense of humor as you	147
Temporary concepts need to get cleaned up	147
Best practices	147
Organizing things	147
Why code organization matters	148
Practical naming	148
Principles	148
Organizing and context	148
How to settle design arguments	149
Errors and exception handling	149
Relevant links	149
Testing	149
BDD & TDD	149
Resources	150
Unit Tests	150
Refactoring	150
Formatting & style	150
Objective readability truths	151
Whitespace	151
Use obvious spacing rules	152
Keep code density low	153
Break horizontally when necessary	155
Prefer smaller files	156
Consistency	156
Capitalization	157
Whitespace rules	158
Storytelling	159
Newspaper Code and the Step-down Principle	159
Maintaining a consistent level of abstraction	163
Code should descend in abstraction towards lower-level details	163
Keeping related methods close to each other	164
Enforcing formatting rules with tooling	165
ESLint	165
Prettier	166
Husky	167
Project planning	168
Notes	168
Section Three - Skills & knowledge	169
Summary	169
Details	169
Tools in your toolbox	170
Infra	170
Know tools to deploy websites	170
Know a scripting language	170
Backend development	170
Know a general purpose language	170
Know a SQL database	170

Dependency Inversion Principle (DIP)	186
Terminology	186
Components	186
Dependency Injection	186
Dependency Inversion	189
Using a mock object	191
The primary wins of Dependency Inversion	192
Inversion of Control & IoC Containers	192
Design by Contract (DBC)	193
Separation of Concerns	194
Related blog posts	194
CQS (Command Query Separation)	194
Principle of Least Surprise	194
Law of Demeter	194
Composition over Inheritance	194
YAGNI	194
KISS (Keep It Simple, Silly)	194
DRY (Don't Repeat Yourself)	194
The Four Primary Object-Oriented Design Principles	194
Composition over inheritance	194
Aim for shallow class hierarchies	194
Encapsulate what varies	194
Program to interfaces, not to implementations	194
Relationship to Ports and Adapters architecture	194
Relationship to Dependency Inversion Principle	194
The Hollywood Principle	194
All software is composition	194
Design patterns are complexity	194
Know of them, but know when you need them	195
Separation of Concerns	195
Example: overloaded controller	195
Separation of concerns	197
Cross-cutting concerns	199
Principle of Least Surprise	199
Strive for loose coupling between objects that interact	199
Principle of Least Resistance	199
7. Design Patterns	199
Factory pattern	199
8. Architectural Principles	199
Component principles	199
Reuse-Release Equivalence Principle	199
Common closure principle (CCP)	199
The Common Reuse Principle (CRP)	199
Conway's Law	200
The Dependency Rule	200

Boundaries	200
Coupling & cohesion	200
9. Architectural Styles	200
Structural	200
Component-based architectures	200
Layered Architectures	200
Monolithic architectures	200
Message-based	200
Event-Driven architectures	200
Publish-Subscribe architectures	200
Distributed	200
Client-server architectures	200
Peer-to-peer architectures	200
10. Architectural Patterns	200
Clean architecture	200
Layers	201
Domain layer	201
Application layer	201
Infrastructure layer	201
Adapter layer	201
Similar architectures	201
Ports & Adapters	201
Vertical-slice architecture	201
Domain-Driven Design	201
Event Sourcing	201
Notes	201
Everything I've recorded about Event Sourcing so far	201
About this	201
Internal links	201
Progression to Event Sourcing	202
Sam Hotoum's Event Sourcing w/ TypeScript repo	202
Why Event Sourcing?	203
State management code can get messy	203
This is how we're ensuring that we have event handlers	203
Projections and deserializing events happens like this!	203
Copy of Best Places to Learn CQRS, Event Sourcing	204
II. Building a Real-World DDD app	205
About this chapter	205
Chapter goals	206
Domain-Driven Design	206
Ubiquitous Language	207
Implementing DDD & ensuring domain model purity	207
DDD addresses the shortcomings of MVC	208
Slim (Logic-less) Models	209
Pick your object-modeling poison	210

Concerns of the unspecified layer in MVC	213
Undesirable side-effects with a lack of a domain model	214
Model behavior and shape	214
Technical Benefits	214
Technical Drawbacks	215
Alternatives to DDD	215
DDD Building Blocks	215
Entities	216
Value Objects	217
Aggregates	217
Domain Services	218
Repositories	218
Factories	218
Domain Events	218
Architectural concepts	220
Subdomains	220
Types of subdomains	222
Benefits of using subdomains	223
Bounded Contexts	223
Deployment as a Modular Monolith	226
Deployment as Distributed Micro-services	227
How to plan a new project	228
Imperative design	229
Imperative design approaches are for small, simple CRUD applications	231
Dimensions that influence the design approach we should take	232
Use-case driven design	232
Use cases & actors	233
Applications are groupings of use cases	233
A use case is a command or a query	234
Use case artifacts	234
Functional requirements document business logic	234
Given-When-Then	234
Parallels with API-first design	236
Steps to implement use case design	236
Planning with UML Use Case Diagrams	237
1 — Identifying the actors	237
2 — Identifying the actor goals	238
3 — Identifying the systems we need to create	238
4 — Identifying the use cases for each role	240
Roles, boundaries, and Conway's Law in Use Case Design	243
Role dictates responsibility	244
Boundaries	247
Using subdomains to define logical boundaries in DDDForum	247
Conway's Law	251
Summary on use case diagrams	252
Event Storming	252
Why we need event storming	254

Handling the upvote post request	320
Inside the Upvote Post use case	321
Aggregate design principles	324
Rule #1 - All transactions happen against Aggregates	324
Rule #2 - Design Aggregates to be as small as possible	325
Rule #3 - You may not alter entities within the aggregate's transaction boundary without going through the aggregate	325
Using a Domain Service	325
Implementing the Upvote Post logic in a Domain Service	325
Persisting the upvote post operation	328
Signaling relationship changes	328
Persisting complex aggregates using database transactions	334
Feature 3: Get Popular Posts	335
Read models	335
Modeling read models as domain concepts	335
Modeling read models as raw data	338
Handling an API request to Get Popular Posts	339
Using a repository to fetch the read models	340
Implementing pagination	341
Where to go from here?	341
Resources	342
References	342

Intro

Why did you decide to write this book?

It all started with “How would you design your business-logic layer?”

Right after I graduated from University, I started the process of interviewing for jobs.

I read and practiced all of the basic interview questions, studied Cracking The Coding Interview, and made sure I knew my stuff on algorithms, data structures, JavaScript, and its language quirks. I went over tons of things I haven’t had to think about for a long time, like closures, IIFEs, passing by reference vs. value, etc.

Most of the interviews went pretty well. I did interviews at several startups and a couple larger companies. Yet, it was when an interview for a Full-Stack JavaScript job at a popular AI startup that a cog in my mind irreversibly activated.

The recruiter for that role told me to really know my stuff on AI and Python (which, as a Node.js developer, I admit I didn’t know enough about), so I spent the majority of my time cramming knowledge about AI and popular Python libraries, attempting to appear as if I knew something about that space.

After getting through the first round of interviews and getting invited for a second technical interview, I felt like I had it in the bag.

The interview was going smooth right up until my interviewer asked, “how do you design your business-logic” layer?

That’s the question that started everything for me.

It was as if he was asking me to speak another language. I hadn’t the slightest idea of how to answer it.

I just recited what I knew about MVC (model-view-controller), but as the words were leaving my mouth, I was realizing more and more that nothing about MVC really screamed “business-logic”.

Safe to say, I didn’t get the job. My recruiter even dropped me. But it was that moment I realized there was an entire world of software design and architecture that I **needed** to teach myself. That failure and inadequacy made me really interested to learn the answers.

At the end of that hiring cycle, I ended up in a job as a Frontend Consultant. I was mostly looking for a low-stress job I could perform to hack my career and expedite my learning by:

- purchasing and studying as many books on software design, patterns, and architecture as possible
- and applying everything I learned from those books to improve my cumbersome ~300k-line Node.js startup app

Every evening after work for 8 months, I read books and wrote code.

I started from the basics. Reviewing everything I knew about Object-Oriented Programming, making sure I really understood the concepts I learned in school, but probably never internalized (abstract classes, the 4 principles of object-oriented programming).

I browsed the internet, writing down all of the terms and concepts that I’ve **heard of but never really cared to try to understand** (POJOs, dependency injection, dependency inversion, inversion of control, concrete classes, design patterns, etc.)

Then I moved over to Architecture.

I didn’t get too far without studying basic UML relationships again. This is what most of the literature about architecture and software design utilizes to express ideas. After getting familiar with that stuff again, I went started looking into things like:

- Coupling
- Cohesion
- Managing dependencies
- Separation of concerns
- Layered / onion / clean architectures / ports & adapters
- Conway’s law
- Use-case driven development
- Packaging large applications in modules
- TDD
- and the SOLID principles.

The point where it all really felt like it paid off was when I discovered Domain-Driven Design. My learning approach was to learn by doing in addition to teaching others (which is why my blog exists today).

Over those 8 months, a lot happened. I quit my job, refactored Univjobs' codebase to Domain-Driven Design, made considerable improvements to the codebase's **maintainability**, **flexibility**, and **reliability** and have been passionately attempting to share what I've learned with my peers online @ khalilstemmler.com.

Those books cost me a lot of money though. Among the purchased were "Clean Architecture", "Clean Code", "Domain-Driven Design", "Implementing Domain-Driven Design", "The Clean Coder", "Refactoring", "Patterns of Enterprise Application Architecture", etc. I easily spent over \$1000 on books.

But now, if you were to ask me "how would you design your business-logic layer", I'd have a lot to say...  WORTH IT.

Why learn software design and architecture?

Design problems occur more frequently than algorithmic ones

In our daily programming jobs, we're more often required to:

- name things well so that they're understood and can be found
- structure things well so that they can be understood and changed
- change things quickly

If you went to school for Computer Science, you'd probably remember focusing a lot on mathematics and algorithms.

For many (myself included), these were some of the **hardest courses** during my undergrad.

The first week of my second year at Brock University, taking Advanced Data Structure and Algorithms, the "welcome back" assignment was to build a LISP interpreter using Java.

The assignment had a prerequisite knowledge of linked-lists, queues, stacks, and other ADTs. I remember sitting down to solve that programming assignment on a Friday night in my basement apartment. My parents had just come to drop off some food, and my roommate left to visit his family for the weekend.

The Sunday night, my roommate returned, and I asked me, "have you left that spot since Friday?". I realized that I actually hadn't. I'd been sitting in the same spot for 3 days working on the solution to the problem. Eventually, I solved the challenge, handed it in, and got a decent grade.

I attribute my success to solving this problem to pure **brute force**.

While there is a lot of value in studying mathematics and practicing algorithms, the magnitude of algorithms that we typically encounter in our daily programming lives looks more like this:

```
export class JobTemplateUtils {  
    public static getJobTemplatesSorted (): IJobTemplate[] {  
        return Object.keys(JobTemplatesTypes).map((key) => {  
            return JobTemplatesTypes[key]  
        })  
        .sort((a, b) =>
```

```

        (a.templateName < b.templateName)
        ? -1 :
        (a.templateName > b.templateName)
        ? 1
        : 0)
    }
}

```

That code simply sorts a hashtable of job templates alphabetically.

Underwhelmed?

That covers 95% of the complex code you'll end up writing as a web developer.

Although the type of industry you're working in and the domain you're writing code in does have an influence on the just how often you'll be crafting algorithms, in most web applications, they're relatively simple.

If only there was a course in university that taught you how to **consistently provide value to a business without writing unmaintainable code**.

Another reason you should study software design and architecture is:

Over ~\$85 billion was spent fixing bad code in 2018

I wrote an article all about this remarkable phenomenon and where I believe it originated. The executive summary is:

- Developers aren't being taught the **essential software design skills**
- Most companies practice agile
- Practicing agile means changing and refactoring code
- To refactor code, we need tests
- To write tests, we need to know how to write testable code
- **Most developers can't write testable code**, which results in productivity plummeting

The truth is, *not a lot has changed about the fundamentals of software design* over the past 20 years, but there's a **huge lack of training** on it.

I tend to agree with Eric Elliot, who writes excellent content about software composition.

He says, "99% of working developers lack solid training in software design and architecture fundamentals. 3/4 of developers are self-trained, and 1/4 of devs are poorly trained by dysfunctional CS curriculum. And almost zero companies make up for those deficiencies with in-house training and mentorship. In other words, if you simply accept the status quo and refuse to offer training in-house, your team will be the blind leading the blind.
- [^1](https://medium.com/@_ericelliott/if-training-is-not-realistic-youre-in-the-wrong-industry-32e488b864ad)"

It's Santayana's curse,

"and those who know not of history are also doomed to repeat their mistakes".

The Domain-Driven Developer

Have you seen this backend developer roadmap? Where we're going in this book is what comes **right after this**.

Fall in love with the problem, not the technology.

Software design is taking an educated guess at the future

Sometimes, I equate software design to playing midfield in soccer. As a midfielder, you have to be aware of what's going on around you at all times. A good midfielder should, at all times, be attempting to predict what's going to happen 3 seconds in the future.

A great midfielder is very perceptive and alert to their surroundings. They will often be positioned on the field at a location that their teammates need them to be, even before they know they're going to need them to be there.

They're able to identify **if and when** their teammates are going to get blocked and pressured to pass the ball, so they position themselves to be available for that pass.

Software design & architecture is similar. We're making best guesses (through abstractions and interfaces) at what we predict is going to need to happen in the future, without investing all of the upfront energy of implementing concretions of things we don't need (YAGNI).

The only way for us to make those informed and educated design decisions?

Understand the domain we're working in

If we don't understand the domain we're writing code in, we're doomed to make expensive messes, because software requirements are **sure** to change over time.

Thank you

It's hard to foresee that you'd ever want to write a book, but I have the people whose emails I've read in the mornings, describing their gratefulness and enthusiasm towards the Enterprise Node.js + TypeScript blog to thank.

Specifically, I'd like to thank my girlfriend, Annick, for your seemingly infinite amount of heart, understanding, and care in between the long days of me working on this book and the startup. You must *really* like me ☺. Thanks to Peter Levels, for the inspiration behind the deployment of this site and to Eric Rafat from FoundersBeta for pushing me "to just launch the damn thing".

To my sharp reviewers Fahad Ahmad, Mario Tacke, Sophia Brandt, Ameur Khaldi, Benji Speer, Muhammad Umair, and *many more*, for your ideas and help in shaping this book.

To Patrick Roza for introducing me to the *scary* functional TypeScript stuff and contributing to the blog.

To Tania Rascia, whose in-depth articles inspired me to try blogging again, and whose site design I *more, or less* copied the best parts of.

And to the greats: Eric Evans, Alistair Cockburn, Uncle Bob, Vaughn Vernon, Martin Fowler and James Coplein, for distilling years of knowledge and experience into artifacts that have shaped our industry forever.

I. The World of Software Design and Architecture

Introduction

The United States Secretary of Defense, Donald Rumsfeld, once said that there are:

- **known knowns** (the things that we *do* know)
- **known unknowns** (things we know that we don't know)
- and **unknown unknowns** (things we don't know that we don't know).

Learning a new field of study can hold a *lot* of **unknown unknowns**.

If for some reason, you had to learn a new trade tomorrow- like welding; for you and me, there are probably more *unknown unknowns* than *known unknowns* and even *known knowns*.

Software design and architecture, while being related to computing, is a *field of study* in and of itself, just like DevOps or UX Design.

Just because you know how to code, it doesn't mean you know how to write well-designed software.

Fortunately, it's something that can be taught.

Unfortunately, it's something that most of us were not (*formally*) taught (and I'm the greatest example of that).

Some of us went to school and got CS degrees.

Some of us took the self-taught path to learn enough to get paid for the work we do.

But just because we know what *classes* are, that doesn't mean we necessarily know how to write good *object-oriented* code.

Just because we put all our code into *services*, that doesn't mean we're implementing a *service-oriented* architecture.

And just because we use the *Model-View-Controller* architectural pattern, it doesn't mean that's it's the *best* for your project.

Software development is a young profession

Software development (and computing in general, *really*) is a **relatively young profession** when compared to other trades.

For example, the seminal book on Design Patterns was released in 1994, Domain-Driven Design was only observed and documented in 2003, and the best source for architecture was compiled and released in 2017.

Meanwhile, other trades- like plumbing and welding, go all the way back to the 18 and 1500s.

Even though the profession is young, brilliant developers paved the way for us by discovering patterns, principles, paradigms, artifacts, and approaches involved in developing quality software.

In the scientific community, the results of research projects are published in peer-reviewed journals so that other researchers can repeat experiments or build upon it to discover new findings. That's not something that only smart people in lab coats do. Building upon already proven truths is the best way forward.

Over the past 40 years, the technologies we use daily have changed a lot- but the fundamentals have stayed the same.

In Uncle Bob's "Clean Architecture", he recollects the fact that only 3 dominant programming paradigms have been discovered (each of which uniquely constrains how we write code). We're very unlikely to realize another.

For years, these established truths about **software design and architecture** discovered by our code-forefathers have been scattered across books and Wikipedia pages. This book you're reading now aims to remedy that problem. It seeks to present the foundational ideas that I believe all people getting paid to write code should *probably* know about.

And honestly, there are a lot of things I wish I knew way before I discovered JavaScript and went gung-ho brute-forcing code until it worked.

"Those who forget the past are condemned to repeat it." - George Santayana

"Those who forget the past are condemned to repeat it." - George Santayana

First Principles

We're going back to **first principles** on this one.

First-principles is the most effective way to break down problems. It works by deconstructing a problem all the way down to the atomic level where we can't deconstruct it anymore, and then reconstructing a solution from the parts that we're absolutely sure are true.

It's what Elon Musk did when he was trying to figure out how to build a more cost-efficient rocket.

It's what Karl Benz did when he invented the car. He challenged the status quo towards transportation, as riding horses was the bee's knees at the time, and made the initial discoveries against what would go on to become the automobile.

In fact, it's what you did in high school when you had to "solve for A" in simple algebra problems like:

$$5a = 2b + c$$

You had to remember the laws of addition, rules of subtraction, division, and multiplication to solve simple problems and decompose bigger ones.

But we always started with those **first, undeniable truths**, didn't we?

That's what we're going to do with this book.

We are going to apply a relaxed version of Methodological reductionism (felt cool to say that) in order to make sense of the industry.

Methodological reductionism: The scientific attempt to provide explanation in terms of ever smaller entities. A lot of the tools and principles that we use to build software at the high-level today, can be constructed from discoveries we have built up over time at the low-level.

Software design and architecture is a huge thing to get acquainted with. But if we deconstruct it, going all the way down to those purely undeniable truths about what constitutes well-designed software, and then reconstruct it all the way up to architecture, I think we'll be in good shape to have minimized those unknown unknowns.

Chapter goals

In this chapter, my goal is to help you **minimize the unknown unknowns about software design and architecture** by first discovering the **goal of software**, then identify the fundamental, undeniable truths about software design and architecture that enable us to meet those goals.

Let's talk about you

I don't think I actually got the chance to welcome you to the **world of software architecture and design**.

Welcome!

If you're here and you're reading this, it's because someone has entrusted you to take part in writing code on something that matters, or you're deeply invested in your own growth as a developer and would like to learn how to write code that does more *good* than bad.

You're the kind of person that when it comes to writing code, getting the job *done* isn't enough for you. Getting it done *well* is equally essential.

You're also the kind of person that's well aware that the actual act of *coding* itself is **easy**.

You could say it's just typing.

Moreover, anything is *easy* if it's not subject to a certain standard of quality.

What's **hard** is producing software that's:

- Simple
- Clean
- Satisfies the needs of its users *today*
- Can be changed to satisfy the needs of its users *tomorrow*

That's **hard**. That's also the **primary goal** of software.

The goal of software

To dive into the depths of what makes software good, let's understand the goal of software.

The goal of software is to **continually produce something that satisfies the needs of its users** while minimizing the effort it takes to do so

Whether it be a clock, a note-taking app, or even the code that runs on Java in your washing machine, goal #1 is to **satisfy the users' needs**.

Goal #1 of software: Satisfy the users' needs.

Does the software meet the needs of its users? Yes? Awesome!

Now, if we can accomplish that impressive feat *at least once*, goal #2 is to figure out how we can consistently achieve goal #1- over and over, with minimal development effort.

Goal #2 of software: Consistently accomplish Goal #1, with as minimal development effort as possible.

That's usually the tricky part. This is where code quality and design start to matter. Cleaner, simpler, and generally better-designed code is a lot easier to take from point A to point B than brute-forced code that serves the *initial* needs of the users.

It's not as black and white as I'm making it sound, though.

Satisfying the users' needs is incredibly subjective.

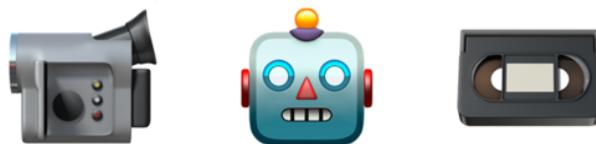
Here's what I mean...

Users' technical expectations vary based on their needs

Depending on the application, what it does, how users intend to use it, and how badly users actually need it, their technical expectations will vary.

Let's look at a few examples.

First one. Let's consider the technical expectations of a machine-learning application that, with AI, was able to take your old VHS home videos and (drastically) improve the picture quality.



AI-Improved Home Movies: Users might be OK with the rendering process taking 2 hours to complete (- **speed**), as long as it always works (- **reliability**).

However, users might *not be* OK with it taking long (- **speed**) AND falling victim to render jobs occasionally failing around the 1 hour, 15 min mark (- **reliability**).

Another example, a common one, is being able to tap your debit card to pay instead of having to insert or swipe (I believe the actual name for this is called Contactless Payment).



Tap-enabled Debit Cards: Buyers are able to pay for things by tapping their debit card on the merchant's card-reader, removing the need for the buyer to type their pin every time they make a purchase (- **speed**, - **efficiency**).

Older models readers don't have tap built-in, so a software upgrade won't work. Replacing the reader is the only option (- **adoptability**). Even still, for buyers, there are a fair amount of merchants with readers out there that *do* accept tap, so it's worthwhile to have a tap-enabled debit card.

If cards were easy to hack, as in, if it were trivial for someone to create a fake payment reader and bump your pocket with it (- **safety**), that may drastically affect the technology's adoption rate, and the opinion of it being **good** software.

One more example. It's actually based on a scenario I'm intimately acquainted with.



Real-time online job fair: Consider you've been asked to build a real-time online job-fair system.

Your client currently runs a popular forum (hacked together with Wordpress) where people can find co-founders and learn how to build a startup.

Once every month, your client wants to run an online job-fair featuring one or more startups. Job seekers should be able to join the chatroom, ask questions, and make connections with recruiters from the startup companies that attend the event.

They've been able to run a couple of these events without any problems on some very cheap shared hosting.

The reason why you're being asked to work on this project today is that the last event had over 500 people in a chatroom at one time. During that time, several users complained about lag and messages taking a long time to send.

Your client has no plan to stop growing. Eventually, he wants to host online events containing over 50,000 concurrent users.

How will the *user (and business) needs* affect the **resulting code**?

It appears that **scalability** is of primary concern to your *client* (and the continued **success of the business, overall**). Meanwhile, for the users, responsiveness is of primary concern. They want it to feel snappy (- **responsiveness**).

If you take the gig, you're tasked with figuring out the best way you can organize code to fulfill those needs.

System quality attributes (SQAs)

In each project example, there was a set of metrics that needed to be satisfied to make the users happy.

Those metrics, like **speed**, **reliability**, **availability**, and **scalability**, are what we call *system quality* attributes (SQAs).

Here's an entire list of software quality attributes for you to check out if you're interested.

How do we design a project to be successful?

When starting a new project (like any of the ones listed in "Users' technical expectations vary based on their needs"), I listen very carefully to the problem statement.

But most importantly, I keep my ear peeled to pluck out those **essential SQAs** in between the lines.

Those SQAs are going to signal to you **what is most important** for the system to succeed, **what has the most significant potential to make the system fail**, and what *architectural choices* will stack the odds of success in your favor.

Aren't all system quality attributes essential?

If we designed a system that possessed **all** of the positive SQAs to a very high degree of effectiveness, and none of the negative ones, it just might be asymptotically *perfect*.

But achieving that is both *very hard*, and also probably *not necessary* to the business.

Realistically understanding that we can't be good at everything right off the bat, depending on what the system is supposed to do, only *some* of these are critically system quality attributes essential.

Identifying those will help us make some of those big up-front decisions.

That's what architecture is about.

What is architecture?

Architecture is about the important stuff. Whatever that is. — Martin Fowler

Architecture is about identifying the software system quality attributes that are **most related to the success or failure of a system**, then stacking the odds of success by choosing the right technologies, tools, frameworks, and overall design of the entire system, around that.

Architecture is about continually ensuring that those essential make-or-break attributes stay healthy throughout the system's lifespan.

It's also easy to say that **architecture** is the stuff that we wished we got right from the start because changing the overall architecture of a system in the future can be very challenging and time-consuming to do efficiently.

Why is software architecture important?

In each of the software project examples from the previous section, we were given a problem to solve — some real-life business opportunity, that could potentially make someone a lot of money.

At this stage, if we know the SQAs that we need to protect, what we need next is a **foundation** that best protects the SQAs, tells us **how to wire the large pieces together**, and **how to write code within that skeleton**.

It's "the foundation that has a profound effect on the quality of what is built on top of it".

There are several architectural patterns (such as Domain-Driven Design, Model-View-Controller, and Microservices) that each originate from a specific architectural style (structural, message-based, or distributed). Each architectural pattern is *uniquely equipped* to protect certain SQAs.

This is a massive part of the early decision-making process when it comes down to making the big up-front decisions about architecture.

"Plans are worthless, but planning is everything" — Eisenhower, 1950

What is software design?

There's a lot of confusion about the difference between **software design** and **architecture**.

Ultimately, *they mean the same thing*.

It's all design, really.

Software design and architecture are two sides of the same coin.

Software *architecture* & *design* is the structure of a system, the elements it contains, and the relationship between those elements.

I love this definition. We'll come back to this in a few because there's something within it that I think is really exciting.

For a second, think about the word *architecture*. The word itself and the connotations it carries makes it easy to see why it seems like architecture and software design are two separate things.

They're not.

Levels of design

The word *architecture* carries with it a connotation of being related to things that are "large" and "high-level".

The word also often draws the mind to parallels among the construction of buildings, cars, or other vital and expensive-to-change things.

However, professionals responsible for the **high-level** designs of these creations, like houses, planes, or Teslas, also understand the implementation details (the **low-level** things) that will need to be accomplished to fulfill the design.

The point I'm trying to make is that you **can't be a high-level designer** (software architect), without knowledge of the **low-level** details (writing clean code, using a programming paradigm effectively, adhering to design principles). The devil is in the details.

You can, however- be a **low-level** coder, proceed to dump code into a product to make the next feature work, all without respect for the **high-level** design, and possibly get away with it maybe a couple of times.

But remember goal #2? Being able to *consistently satisfy the users' needs*?

When the **low-level details** go against the grain of the **high-level policy**, it's only a matter of time until we have a legacy system that's no longer easy (or worthwhile) to maintain on our hands.

Both levels of software design (high and low) are essential. They form a symbiotic relationship with each other that when in sync, can lead to high-quality software that is easy to maintain and change.

That means **everyone on the team** holds the shared responsibility of understanding the high-level architecture and how the low-level details support it.

The Software Design and Architecture Stack & Roadmap

Let's take a second to recap what we just discovered. We'll regularly be doing this throughout the book:

- The **goals of software** are to:
 - Goal #1: Satisfy the users' needs while minimizing the effort it takes to do so.
 - Goal #2: Consistently accomplish Goal #1 as the requirements change.
- **Architecture** is about identifying the **system quality attributes** that will stack our odds of successfully performing Goal #1, and our choice of **architectural pattern** that will best accommodate the project (we could dive much deeper here though).
- Lastly, **software design** isn't much different from architecture besides the fact that they have different **levels of design** that they appear at.

Great, that's a good start. We've defined what we're fighting for, and we understand at a high-level how to get there.

Let's continue the descent to get to the bottom of what's involved in designing good software.

I'd like to introduce to you **two artifacts** that I spent a really long time thinking about to visualize the scope and breadth of software design and architecture.

They are the **stack** and the **roadmap**.

Resource: The Stack

The software design and architecture stack

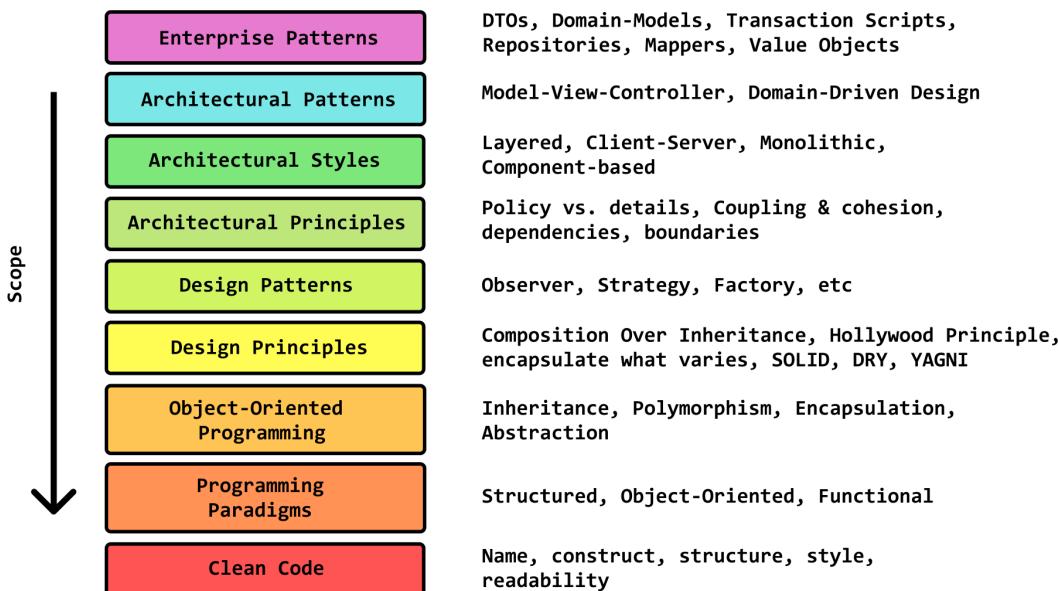
The *stack*. It depicts the scope of learning from the most intimate details of the *enterprise pattern* you've chosen to the way you write *clean code*.

The knowledge required to get to the top of the stack is layered. Similarly to the OSI Model in networking, each layer of the stack builds on top of the foundation of the previous one.



The Software Design & Architecture Stack

Khalil Stemmller
@stemmlerjs



The software design and architecture stack depicts the layers of software design and architecture.

In the graphic stack, I've included examples to *some* of the most important concepts at each respective layer. Because there are just too many concepts at each layer, I didn't include all of 'em.

Resource: The Map

The Software Design and Architecture Roadmap

Check out the **map**. While I think the stack is good to see the bigger picture of what we have to cover at a glance, the *map* is a little bit more detailed, and as a result, I think it's more useful.

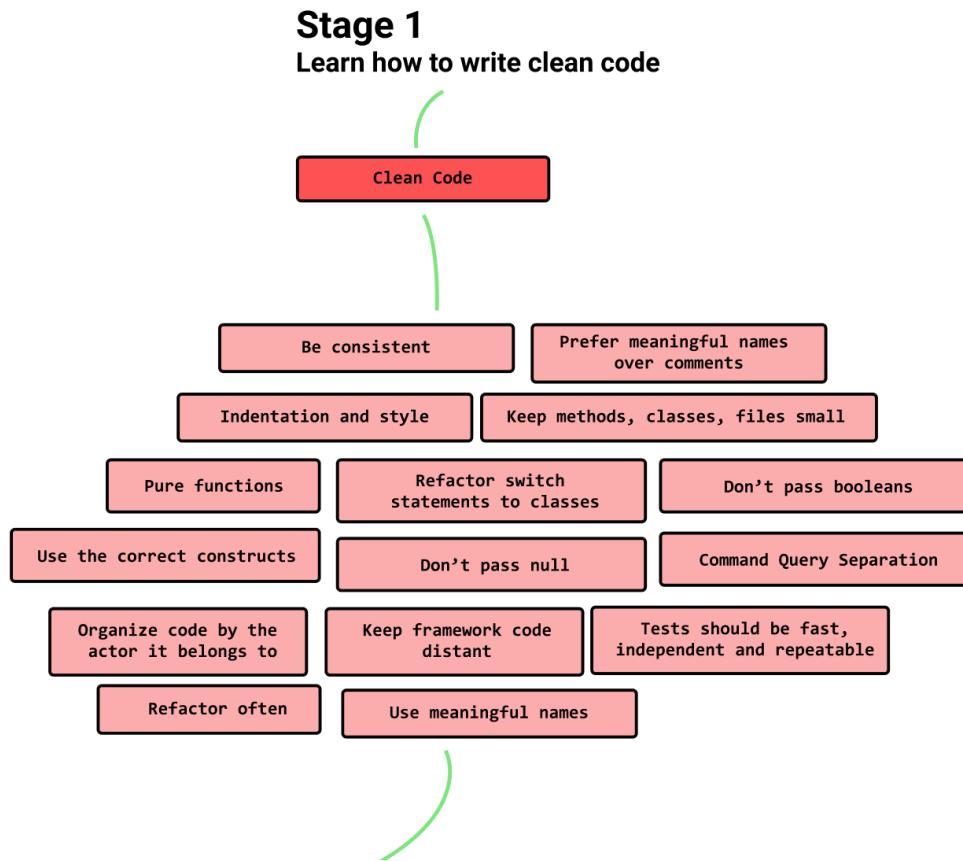
From clean code to Domain-Driven Design concepts, the map is indicative of the path that we're going to take in this book to get you ramped up in the world of software design and architecture.

You can view the entire map (it's quite large and not included here, just for EPUB and PDF readers) here via this link.

Step 1: Clean code

Clean Code

Goal: Learn how to write clean code.



The very first step towards creating long-lasting software is figuring out how to write clean code.

If you ask anyone what they think constitutes *clean code*, you'll probably get a different answer every time. A lot of times, you'll hear that *clean code* is code that is easy to understand and change. At the low-level, this manifests in a few design choices like:

- being consistent
- preferring meaningful variable, method and class names over writing comments
- ensuring code is indented and spaced properly
- ensuring all of the tests can run
- writing pure functions with no side effects
- not passing null

These may seem like small things, but think of it like a game of Jenga. In order to keep the structure of our project stable over time, things like indentation, small classes and methods, and meaningful names, pay off a lot in the long run.

If you ask me, this aspect of *clean code* is about having good coding conventions and following

them.

I believe that's only *one* aspect of writing *clean code*.

My definitive explanation of clean code consists of:

- Your developer mindset (empathy, craftsmanship, growth mindset, design thinking)
- Your coding conventions (naming things, refactoring, testing, etc)
- Your skills & knowledge (of patterns, principles, and how to avoid code smells and anti-patterns)

So much of what makes software great happens before we even touch the keyboard.

One requirement is that you should care enough to learn about the business you're writing code within. If we don't care about the domain enough to understand it, then how can we be sure we're using good names to represent domain concepts? How can we be sure that we've accurately captured the functional requirements?

If we don't care about the code that we're writing, it's a lot less likely that we're going to implement essential coding conventions, have meaningful discussions, and ask for feedback on our solutions.

We often think that code is solely written to serve the needs of the *end user*, but we forget the other people we write code for: us, our teammates, and the project's future maintainers. Having an understanding of the principles of *design* and how **human psychology decides what is good and bad design**, will help us write better code.

So essentially, the best word that describes this step of your journey? Empathy.

Once we've got that down, learn the *tricks of the trade* and continue to improve them over time by improving your knowledge of the essential software development patterns and principles.

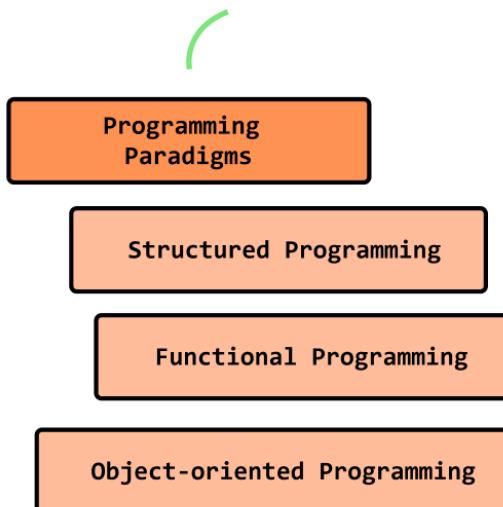
In Chapter 3, we discuss what clean code is and how to write code that is clean.

Step 2: Programming paradigms

Goal: Understand the differences between each mainstream programming paradigm, what each uniquely brings to the table, and when to use them.

Stage 2

Understand the differences between each mainstream programming paradigm, what each uniquely brings to the table, and when to use them.



Now that we're writing readable code that's easy to maintain, it would be a good idea to really understand the 3 dominant programming paradigms and the way they influence how we write code.

In Uncle Bob's book, *Clean Architecture*, he brings attention to the fact that:

- **Object-Oriented Programming** is the tool best suited for defining how we cross architectural boundaries with polymorphism and plugins
- **Functional programming** is the tool we use to push data to the edges of our applications and elegantly handle program flow
- and **Structured programming** is the tool we use to compose algorithms

This implies that robust software uses a hybrid all 3 programming paradigms styles at different times.

While you *could* take a strictly functional or strictly object-oriented approach to write code in a project, understanding where each excels will improve the quality of your designs.

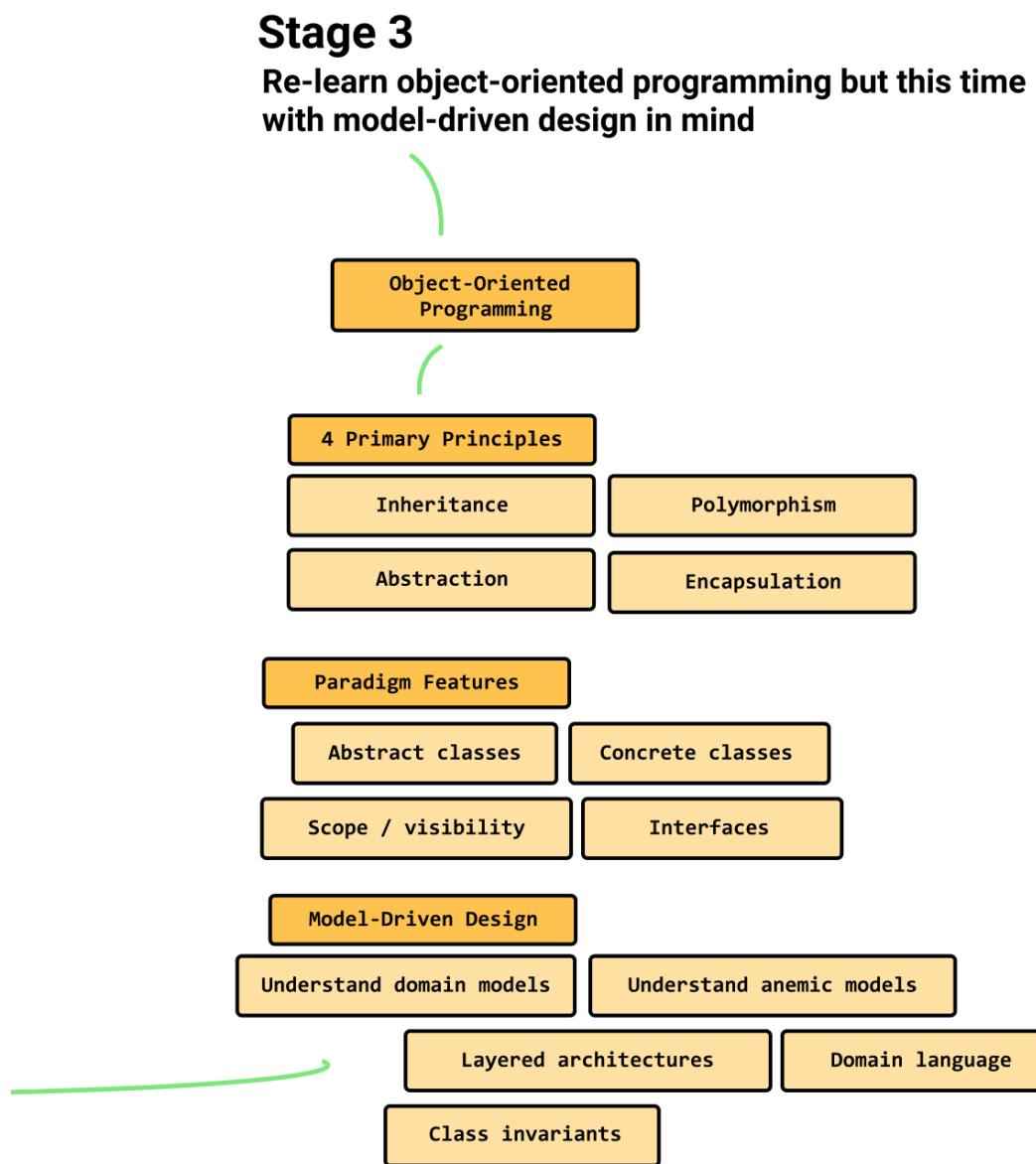
It's one of those scenarios where:

...if all you have is a hammer, everything seems like a nail.

In Chapter 4, we discuss these pretty bold statements about programming paradigms.

Step 3: Object Oriented Programming and Domain Modeling

Goal: Re-learn object-oriented programming but this time, with model-driven design in mind.



In a book about software design and architecture, Object-Oriented Programming is going to get a lot of love because it's the **clear tool for architecture**.

Not only does Object-Oriented programming enable us to create a **plugin architecture** and build flexibility into our projects, OOP comes with the **4 principles of OOP** (encapsulation, inheritance, polymorphism, and abstraction) that helps us create **rich domain models**.

Most developers learning Object-Oriented Programming never get to this part: learning how to create a *software implementation of the problem domain*, and enabling it to live in the center of a **layered web app**.

Functional programming seems to be growing in popularity recently, and I expect that it has a lot to do with React and the JavaScript ecosystem, but don't be so quick to dismiss OOP, model-driven design and Domain-Driven Design.

In Chapter 5, we spend some time towards understanding the big picture on how object-modelers encapsulate an *entire business and their processes* within a zero-dependency domain model.

Why is that a huge deal?

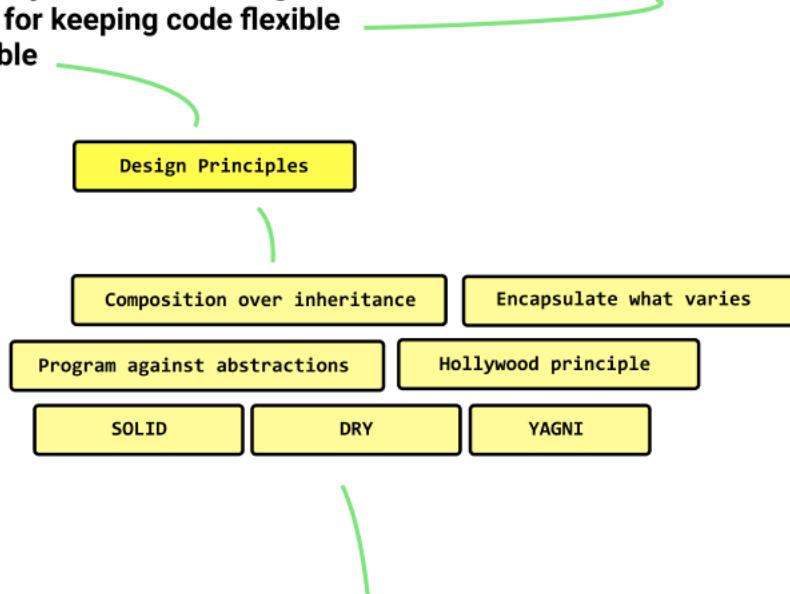
Because if we can create a mental model of a business, we can create the software implementation of the business.

Step 4: Design Principles

Goal: Learn the object-oriented design principles for keeping code flexible, testable, and maintainable.

Stage 4

Learn the object-oriented design principles for keeping code flexible and testable



Object-Oriented Programming is beneficial for encapsulating rich domain models and solving the 3rd type of “Hard Software Problems” - Complex Domains, but it can introduce some design challenges.

When should I use extends and inheritance?

When should I use an interface?

When should I use an abstract class?

Design Principles are well-established and battle-tested object-oriented best practices that we can use as guardrails.

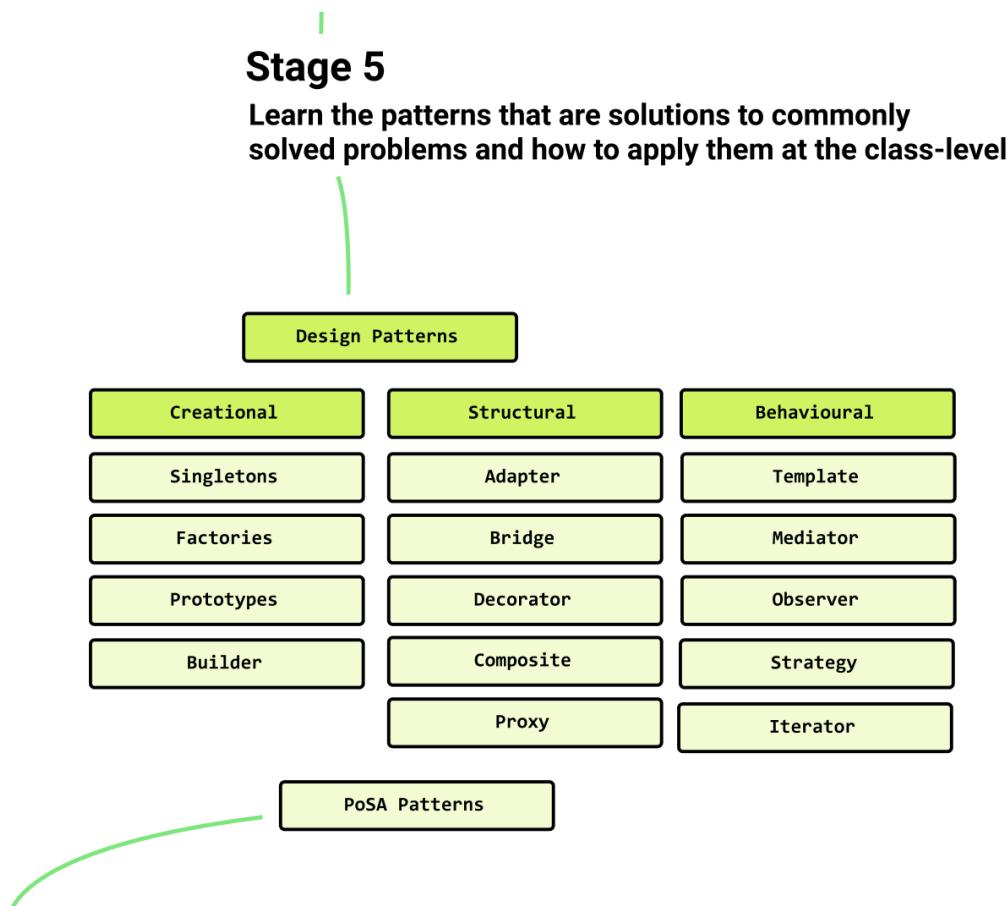
Examples of common design principles we will familiarize ourselves with are:

- Composition over inheritance
- Encapsulate what varies
- Program against abstractions, not concretions
- The Hollywood principle: “Don’t call us, we’ll call you.”
- The SOLID principles, especially the Single responsibility principle
- DRY (Do Not Repeat Yourself)
- YAGNI (You Aren’t Gonna Need It)

These are just a few of many OO design principles that can help us improve our designs. We discuss them in detail in Chapter 6.

Step 5: Design patterns

Goal: Learn the patterns that are solutions to commonly solved problems and how to apply them at the class level.



Generic versions of the most commonly occurring problems in software development have already been categorized and solved. We call these patterns. *Design patterns*, actually.

There are 3 categories of design patterns: **creational**, **structural**, and **behavioral**.

Creational Design Patterns

Creational patterns are patterns that control how objects are created.

Examples of creational patterns include:

- The **Singleton pattern*** for ensuring only a single instance of a class can exist
- The **Abstract Factory pattern**, for creating an instance of several families of classes
- The **Prototype pattern**, for starting out with an instance that is cloned from an existing one

Structural Design Patterns

Structural patterns are patterns that simplify how we define relationships between components.

Examples of structural design patterns include:

- The **Adapter pattern**, for creating an interface to enable classes that generally can't work together, to work together.
- The **Bridge pattern**, for splitting a class that should actually be one or more, into a set of classes that belong to a hierarchy, enabling the implementations to be developed independently of each other.
- The **Decorator pattern**, for adding responsibilities to objects dynamically.

Behavioral Design Patterns

Behavioral patterns are common patterns for facilitating elegant communication between objects.

Examples of behavioral patterns are:

- The **Template pattern**, for deferring the exact steps of an algorithm to a subclass.
- The **Mediator pattern**, for defining the exact communication channels allowed between classes.
- The **Observer pattern**, for enabling classes to subscribe to something of interest and to be notified when a change occurred.

Design pattern criticisms

Design patterns are great and all, but sometimes they can add additional complexity to our designs. It's essential to remember YAGNI and attempt to keep our designs as simple as possible. Only use design patterns when you're really sure you need them. You'll know when you will.

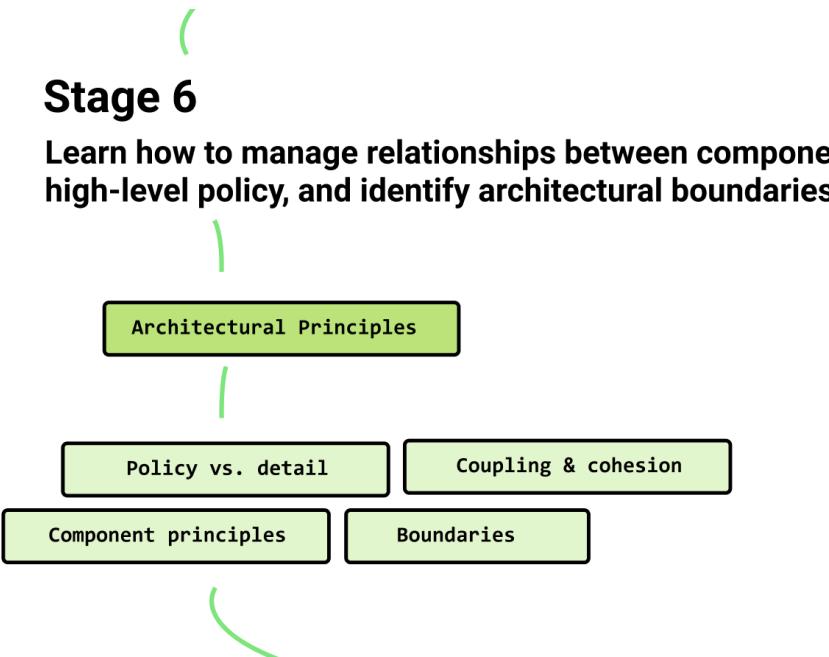
If we know what each of these patterns is, when to use them, and when to *not even bother* using them, we're in good shape to begin to understand how to architect larger systems.

The reason behind that is because **architectural patterns** (Chapter 10) are just **design patterns blown-up in scale to the high-level**, where design patterns are low-level implementations (closer to classes and functions).

We discuss design patterns in Chapter 7.

Step 6: Architectural Principles

Goal: Learn how to manage relationships between components, express high-level policy, and identify architectural boundaries.



Now we're at a higher level of thinking just above the class level.

At this point in our journey, we understand that the relationships between components will have a significant impact on the maintainability, flexibility, and testability of our project.

In Chapter 8, we'll cover the guiding principles that help us:

- Improve flexibility within our codebase to be able to react to new features and requirements
- Separate concerns
- Improve readability and scan-ability by organizing our code into cohesive modules dictated by the use cases of our application

Here's a glimpse of what we're interested in learning:

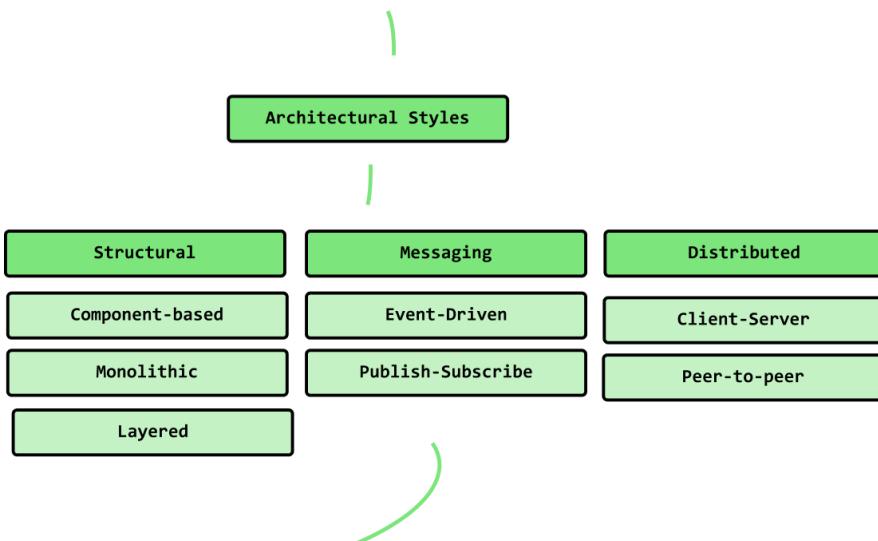
- **Component design principles:** The Stable Abstraction Principle, The Stable Dependency Principle, and The Acyclic Dependency Principle, for how to organize components, their dependencies, when to couple them, and the implications of accidentally creating dependency cycles and relying on unstable components.
- **Policy vs. Detail**, for understanding how to separate the rules of your application from the implementation details.
- **Boundaries**, and how to identify the subdomains that the features of your application belongs within.

Step 7: Architectural Styles

Goal: Learn the different approaches to organizing our code into high-level modules and defining the relationships between them.

Stage 7

Learn the different approaches to organizing our code into high-level modules and defining the relationships between them.



We discovered that System Quality Attributes (SQAs) are the metrics we need to protect to *stack the odds of success* of our application.

Architectural styles are groupings of all the different types of architectures that you can employ. Each of these styles has uniquely positive effects on maintaining the health of one or more SQAs.

For example, a system that has a lot of **business logic complexity** would benefit from using a **layered architecture** to encapsulate that complexity.

A system like Uber needs to be able to handle a lot of **real time-events** at once and update drivers' locations, so **publish-subscribe** or **event-driven** style architecture might be most effective.

I'll repeat myself here because it's important to note that the **3** categories of architectural styles are similar to the **3** groups of design patterns because **architectural styles are just design patterns at the high-level**.

Structural

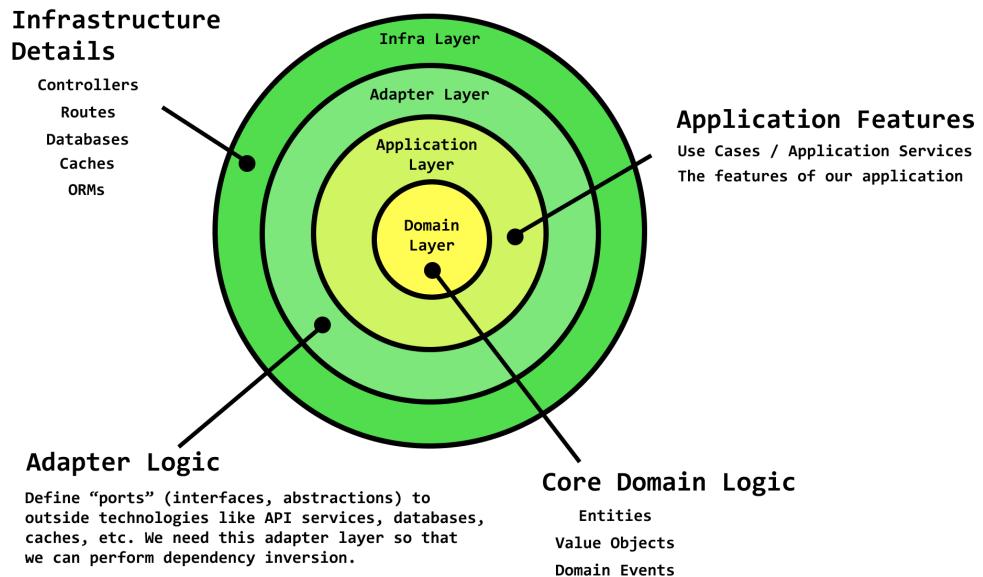
Projects with *varying levels* of components, and wide-ranging functionality are usually looking for - **flexibility** as an SQA. Structural architectural styles make it easier to extend and separate the concerns of complex systems.

Here are a few examples:

- **Component-based** architectures emphasize *separation of concerns* between the *individual components* within a system. Think **Google** for a sec. Consider how many applications they have within their enterprise (Google Docs, Google Drive, Google Maps, etc). For platforms with lots of functionality, component-based architectures divide the concerns into loosely coupled independent components. This is a *horizontal separation* of concerns.

ration.

- **Monolithic** means that the application is combined into a single platform or program, deployed all together. *Note: You can have a component-based AND monolithic architecture if you separate your applications properly, yet deploy it all as one piece.*
- **Layered** architectures separate the concerns by cutting software into infrastructure, application, and domain layers. This is a *vertical* separation.



An example of cutting the concerns of an application *vertically* by using a layered architecture.

Message-based

Messaging might be a crucial component to the success of the system. Message-based architectures build on top of functional programming principles and behavioral design patterns like the observer pattern.

Here are a few examples of message-based architectural styles:

- **Event-Driven** architectures view all significant changes to the state as events. For instance, within a vinyl-trading app, an Offer's state might change from "*pending*" to "*accepted*" when both parties agree on the trade. Commands and Events become the primary mechanisms to invoke and react to changes within the system.
- **Publish-subscribe** architectures make heavy use of the Observer design pattern by enabling subscribers to listen in on something of interest (a chatroom, or event stream) and publish events to all appropriate subscribers. A subscriber can be something within the system itself, end-users / clients, and other systems and components.

Distributed

A distributed architecture simply means that the components of the system are deployed

separately and operate by communicating over a network protocol. Distributed systems can be handy for scaling throughput, scaling teams, and delegating (potentially expensive tasks or) responsibility to other components.

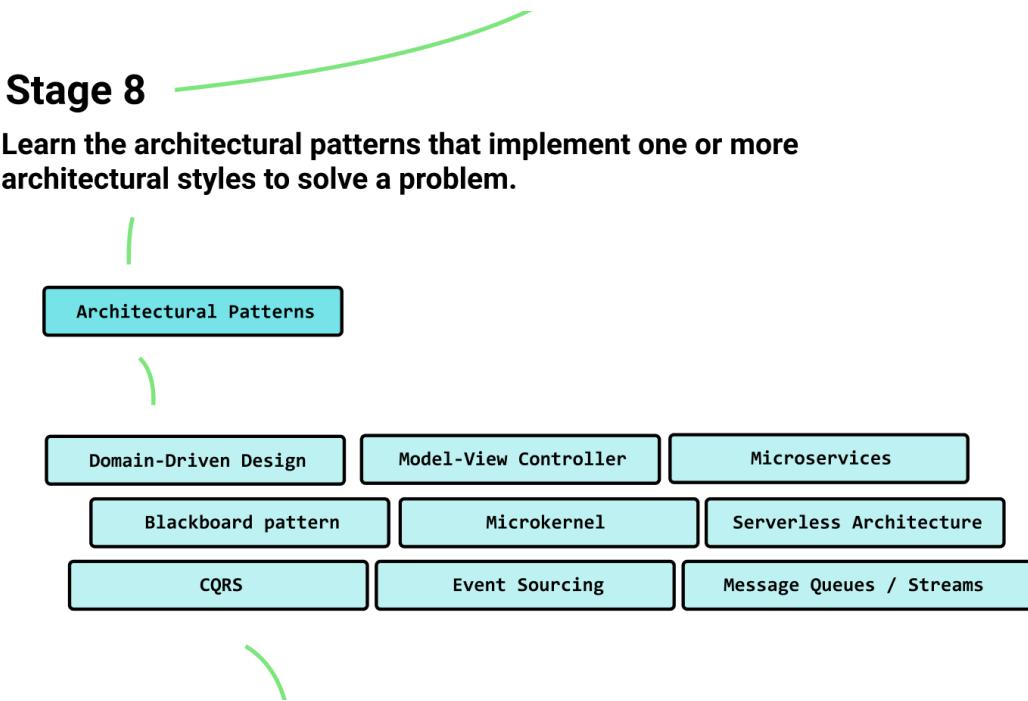
A few examples of distributed architectural styles are:

- **Client-server architecture**. One of the most common architectures, where we divide the work to be done between the client (presentation) and the server (business logic).
- **Peer-to-peer** architectures distribute application-layer tasks between equally-privileged participants, forming a peer-to-peer network.

We discuss architectural styles in more detail in Chapter 9.

Step 8: Architectural Patterns

Goal: Learn the architectural patterns that implement one or more architectural styles to solve a problem.



Architectural *patterns* are tactical implementations of one or more architectural *styles*.

And when I say “tactical” implementation, I really mean that. These are *exact* patterns you can use to create the architecture that protects your SQAs.

Here are a couple of examples of architectural patterns in addition to the styles that they inherit from:

- Domain-Driven Design is an approach to software development against really complex problem domains. For DDD to be most successful, we need to implement a **layered (structural style) architecture** to separate the concerns of a domain model from the infrastructural details that make the application actually run, like databases, web servers, caches, etc.

- Model-View-Controller is probably the *most well-known* architectural pattern for developing user interface-based applications. Stylistically, it's a **distributed architecture**. It works by dividing the app into 3 components: model, view, and controller. MVC is incredibly useful when you're first starting out, and it helps you piggyback towards other architectures, but there hits a point when we realize MVC isn't enough for problems with lots of business logic.
- **Event sourcing** is a functional approach where we store only the transactions, and never the state. If we ever need the state, we can apply all the transactions from the beginning of time. You probably guessed, but this is an **event-driven** approach to architecture.

We discuss these in Chapter 10.

Step 9: Enterprise patterns

Goal: Learn the ins and outs of the concepts involved in implementing your chosen architectural pattern.

Depending on the architectural pattern you chose best suits your needs, there's going to be plenty of new constructs and *technical jargon* to make sense of.

For example, when you decide that Domain-Driven Design is the architectural pattern makes the most sense for your project, you need to learn about:

- Entities: they describe models that have an identity.
- Value Objects: these are models that have no identity, and can be used to encapsulate validation logic.
- Domain Events: these are events that signify some relevant business event occurring, and can be subscribed to from other components.

And if you decide that Event Sourcing makes sense, you'll have an entirely new set of concepts to learn like:

- Retroactive Events: Automatically correct the consequences of an incorrect event that's already been processed.
- Eventual Consistency: a way to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. - via Wiki

Note: It's very common to combine **several** of the *architectural patterns* together into an architecture that meets your SQAs. Consider the challenges of DDD + Event Sourcing, or MVC + Message Queues / Streams.

Depending on the architectural style you've chosen, there are going to be a ton of concepts for you to learn to implement that pattern to its fullest potential.

In Chapter 11, we build a real-world app with Domain-Driven Design.

Chapter conclusion

We just went from 0 to 100 on software design and architecture.

The ultimate goal of this chapter was to take you for a walk through what there is to know. Hopefully, we were able to knock down your *unknown unknowns* a notch.

The biggest takeaways from this chapter:

The **goals of software** are to:

- **Goal #1:** Satisfy the users' needs while minimizing the effort it takes to do so.
- **Goal #2:** Consistently accomplish Goal #1 as the requirements change.

Architecture is about

- Identifying the **system quality attributes (SQAs)** that will stack our odds of successfully accomplishing Goal #1, and
- Choosing the correct **architectural pattern** to satisfy the critical SQAs.

Software design isn't much different from architecture besides the fact that:

- They have different **levels of design** that they appear at. The best example of this is **design patterns**, which are essential at the class level and appear as (high-level) architectural patterns that help us meet our SQAs.

References

- Wikipedia: List of architectural styles and patterns
- Architectural styles vs. architectural patterns vs. design patterns
- The Clean Architecture
- How & why do scientists share results
- What is clean code and why should you care?
- Software Architects Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts
- Reductionism

2. TypeScript

Introduction to TypeScript

Over the past few years, I've witnessed JavaScript development take off. It's come a long way. JavaScript was once something that my manager would laugh at and call a *toy language*. Now, it has become a first-class citizen for serverless applications, AR / VR experiences, event-driven architectures at fortune 500 companies, and pretty much anything else you can think about- it's not going anywhere.

For a consistent number of years now, JavaScript has been the most popular programming language.

Languages come and go, but some things **don't change**.

Software design principles and the fact that types improve code quality and readability has been something that we've known for a long time. Yet, it's been left out of modern JavaScript for a long time.

TypeScript, developed and appropriated labeled by Microsoft as “*JavaScript that scales*”, is a **superset of JavaScript**, meaning that everything JavaScript can do, TypeScript can do (and more better).

In this chapter, we’ll get acquainted with TypeScript:

- A language more appropriate for learning software design and architecture than JavaScript and,
- A language that we’re going to be primarily focused on throughout the book

For developers skilled with TypeScript already, I’d encourage you to skip this chapter and go straight to the next one.

Chapter goals

The for this chapter are to:

- Understand the design decisions of TypeScript and the shortcomings TypeScript was meant to solve improve in JavaScript development
- Know which types of JavaScript projects TypeScript is best suited for
- Learn how to install and setup a TypeScript project
- Explore TypeScript language basics and features that might be new to developers familiar with either Java or C#

Primary goals of TypeScript

TypeScript was meant to solve two problems:

1 -Provide JavaScript developers with an **optional type system**.

Because JavaScript projects have gotten increasingly complex in recent years, developers have realized the utility of having a type system to manage complexity.

“How do we convert a legacy JavaScript project to TypeScript”? It would surely take months to do it in one go. Is there a way that we can **gradually** convert the codebase to use types, without leaving the app in an unstable state?

Yep. TypeScript was designed to compile both TypeScript *and* JavaScript. This enables developers to **opt-in** to using types and take as long as they need to convert a codebase to TypeScript over time.

2 - Provide JavaScript developers with the ability to utilize planned features from **future JavaScript editions** against current JavaScript engines.

While being used for several other things now, JavaScript is still the language of the browsers. Even Node.js, the server-side runtime that allows us to create backend applications with JavaScript, is built on top of the Chrome V8 engine.

However, it can be tough for modern web browsers to keep up with supporting the latest version. This is mostly due to the fact that the language is continuously undergoing changes, and features are added continually to future ECMAScript proposals and finding their way into new ECMAScript standards.

As of writing this, the current JavaScript version fully supported by modern browsers is ECMAScript v5.

To write code using these new features and have them work on modern browsers, we need a tool that can compile newer ECMAScript language features down to ES5.

That's another thing TypeScript can do.

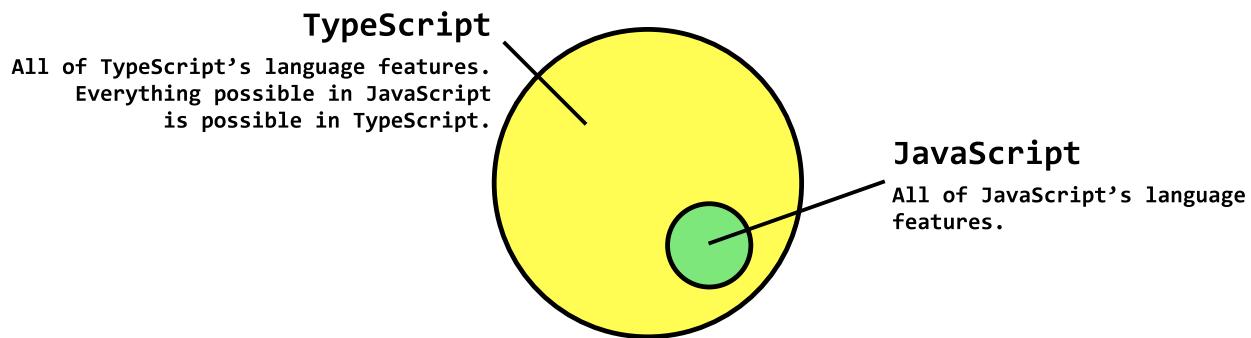
ECMAScript: ECMAScript is a *language specification standard* that was created in order to standardize JavaScript.

All JavaScript is valid TypeScript

TypeScript allows us to use language features not available in current JavaScript versions (such as decorators, access modifiers, type annotations). Many of these features will seem familiar to C# and Java developers but may be new to developers that only know JavaScript.

Because TypeScript is a **superset** of JavaScript, everything in JavaScript is also valid in TypeScript. However, the inverse is not true (*not* everything in TypeScript is valid in JavaScript).

TypeScript is a superset of JavaScript



TypeScript is a superset of JavaScript

Superset — a set which includes another set of sets. In this case, the entire set of things possible in JavaScript is included as a small part of the larger set of things possible with TypeScript.

For example, usage of the **type annotation** is valid TypeScript but not valid JavaScript.

```
const age: number = 12; // Valid TypeScript, invalid JavaScript
```

Yet, the not including a type annotation for a variable **is both valid** JavaScript and TypeScript.

```
const age: number = 12; // Valid TypeScript, Valid JavaScript
```

Type annotations are one of many features supported only by TypeScript.

TypeScript types

There are four primary *types* of types in TypeScript you'll be dealing with: implicit, explicit, structural, and ambient.

Convenient Implicit Types

Explicit Types

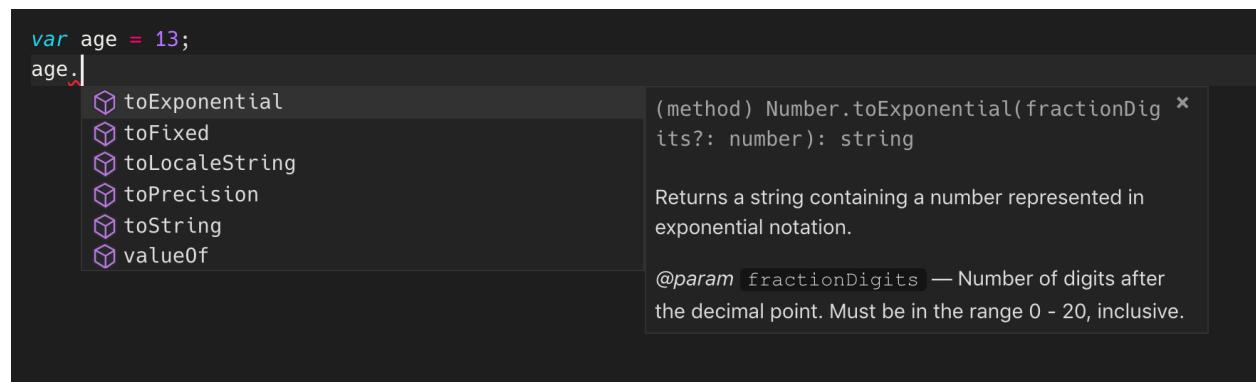
Structural Types

Ambient types

Convenient Implicit Types

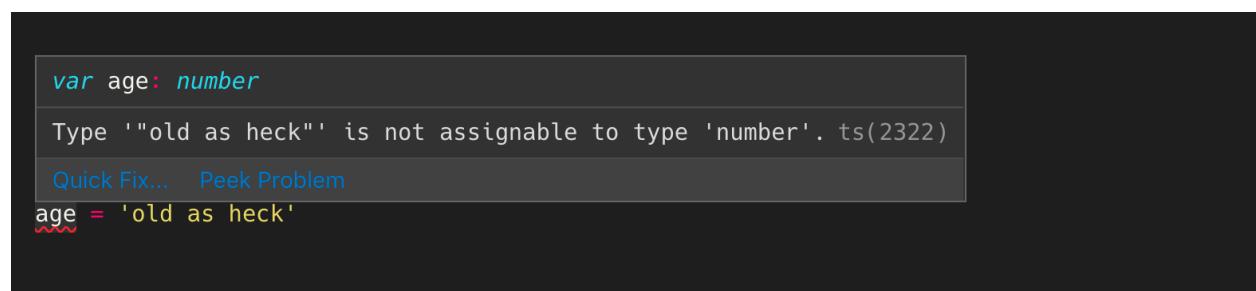
Inheriting from the carefree nature of JavaScript, TypeScript will do its very best to try to figure out the types of your variables if you don't **explicitly** define them.

For example, using Visual Studio Code, a variable `age` declared with the value `13` will assume that the type is `number` and provide all of the primitive `number` methods as a result of that type inference



This *convenience* enables JavaScript developers to continue to code without having to define the types of their variables and enables old JavaScript code to be migrated to TypeScript.

And in the following example, because TypeScript infers the types of variables not explicitly expressed, changing the value of `age` to a type that isn't `number` will present an error.



Depending on who you ask, those implicit type checks are either **amazing** or a **nuisance** (I lean strongly on the amazing side).

Explicit Types

Implicit checks are pretty handy for catching silly errors. And while the compiler can make a lot of assumptions about what we're likely trying to do, we should prefer *explicit type annotations* to validate the compiler's assumptions.

```
*const* artist: string = "Nick Cave and the Bad Seeds";
```

In TypeScript, the type annotation is applied to the variable in a Polish postfix notation style (where the *operator/type* follows the *operand/variable*). This might feel a little bit strange for developers coming from a Java or C# background, but you'll get used to it.

Similar to Java and other strictly typed languages, functions and methods can also **specify a return type**. Un-similarly to C# and Java, the return type is also denoted using Polish postfix notation.

```
function sayHello (): string {
    return 'Hello'
}

class Greeter {
    constructor () {}

    sayHello (name: string): string {
        return `Hello ${name}`;
    }
}
```

If we attempt to assign the return value to an improperly typed variable, we'll also get the errors one would expect from strict type checking.

```
const num: number
'num' is declared but its value is never read. ts(6133)
Type 'string' is not assignable to type 'number'. ts(2322)
Quick Fix... Peek Problem
const num: number = sayHello();
```

Structural Types

We have a lot of options for explicitly defining types. Let's talk a little bit about **Nominal** and **Duck Typing**.

Nominal typing

Duck typing

Nominal typing

In **Nominal** type systems, a particular type is deemed valid based on:

1. The **explicit declaration** of the name of the type and/or,
2. If the type is a subtype.

Nominal type systems primarily use abstractions like *interfaces* and *classes* to determine type compatibility.

In the following example, we create an `Instrument` class which is subclassed by a `Guitar` class. The `Guitar` class is then subclassed by a `Fender` class.

```
class Instrument {}

class Guitar extends Instrument {}

class Fender extends Guitar {}

// Valid, through subtypes
const thingThatMakesNoise: Instrument = new Fender();
// Valid, through subtypes
const pluckedThingThatMakesNoise: Fender = new Guitar();
// Valid, explicitly defined as a Fender
const fender: Fender = new Fender();
```

This creates a hierarchy scheme of: `Instrument` => `Guitar` => `Fender`.

Therefore, a `Fender` is a valid type for both a `Guitar` and an `Instrument`.

Note: This is a demonstration. You typically don't want to create elaborate class hierarchies like this. See "Composition over Inheritance" from Chapter 6 — Design Principles.

Languages like Java use nominal typing. They require you to explicitly *declare* and *cast* the types of your variables. Even in scenarios where you *know* that a particular type (structurally) satisfies the required members of abstraction, you still need to explicitly define the type, because that's how nominal type systems can work. This can sometimes result in a lot of redundant code.

In TypeScript, the rules are much more relaxed.

Because one of TypeScript's design goals was to provide an optional-type system without disrupting productivity for JavaScript developers, TypeScript comes with the concept of **Duck Typing** built-in.

Duck typing

Duck Typing is an example of a *structural type system* in which type compatibility and equivalence are determined by the type's actual structure.

The name comes from the saying that "if it looks like a Duck and it quacks like a Duck... it must be a Duck".

In the following example, the `postComment(comment: Comment)` function needs something that *looks* like a comment. The **attributes** that exist on the `Comment` interface needs to be present in the type of the argument passed into the function.

```
interface Comment {
  id: number;
  name: string;
  content: string;
}
```

```

interface Reply {
  id: number;
  name: string;
  content: string;
  parentCommentId: number;
}

const comment: Comment = {
  id: 1,
  name: 'Khalil',
  content: "Is anyone here?"
};

const reply: Reply = {
  id: 2,
  name: 'Don Draper',
  content: "Yes, I'm right here.",
  parentCommentId: 1
};

function postComment (comment: Comment) {
  // Do something with the comment
}

// Perfect - exact match
postComment(comment);

// OK - extra information still alright
postComment(reply);

// Missing info not OK.
// Type '{ id: number; }' is missing the following properties from
// type 'Comment': name, content
postComment({ id: number });

```

Notice that an object satisfying the *shape* of a Reply is OK to be passed in because it has all the *structural attributes* that would deem a Reply to be a Comment?

Yet, not satisfying the minimum requirements of the Comment interface can be caught at compile time.

That's pretty powerful. Compare that with the previous JavaScript convention of checking for types at runtime. Before TypeScript, Duck Typing in JavaScript looked a lot like this:

```

/**
 * This method posts a comment. A comment needs an
 * 'id' attribute of type number, a 'name' of type string,
 * and a 'content' of type string.

```

```

*/
function postComment (comment) {
  const isIdPresentAndValid = comment.hasOwnProperty('id')
    && !isNaN(comment.id);

  const isNamePresentAndValid = comment.hasOwnProperty('name')
    && typeof comment.name === "string";

  const isContentPresentAndValid = comment.hasOwnProperty('content')
    && typeof comment.content === "string";

  if (!isIdPresentAndValid) throw new Error('Must provide an integer id');
  if (!isNamePresentAndValid) throw new Error('Must provide a string name');
  if (!isContentPresentAndValid) throw new Error('Must provide a string content');

  // Do things
  ...
}

```

Joi Validation: While TypeScript can do compile time structural type checking, Joi, a popular JavaScript validation library, can do **runtime** structural type checking.

Ambient types

A primary design goal of TypeScript was to make it possible for you to safely and efficiently use existing JavaScript libraries in TypeScript.

TypeScript does this through declaration. TypeScript provides you with a sliding scale of how much or how little effort you want to put in your declarations, the more effort you put, the more type safety + code intelligence you get. Note that definitions for most of the popular JavaScript libraries have already been written for you by the DefinitelyTyped community so for most purposes either:

1. The definition file already exists.
2. Or at the very least, you have a vast list of well-reviewed TypeScript declaration templates already available

As a quick example of how you would author your own declaration file, consider the trivial case of using jQuery. By default, TypeScript expects you to declare (i.e., use var, let or const somewhere) before you use a variable. If you're loading jQuery through a <script> tag and expecting it to be loaded into the global scope, TypeScript will complain that it doesn't know anything about that.

```
$('.awesome').show(); // Error: cannot find name '$'
```

As a quick fix, you can tell TypeScript that there is indeed something called \$:

```
declare var $: any;
$('.awesome').show(); // Okay!
```

If you want, you can build on this basic definition and provide more information to help protect you from errors:

```
declare var $: {  
  (selector:string): any;  
};  
$('.awesome').show(); // Okay!  
$(123).show(); // Error: selector needs to be a string
```

Migrating to TypeScript

By merely adding `allowJs: true` to the `tsconfig.json` compiler options, you can take your time migrating even the hugest projects from JavaScript to TypeScript by changing the file extension from `.js` to `.ts`.

TypeScript still emits errored JavaScript code

TypeScript still emits errored JavaScript code

TypeScript's tolerance for JavaScript is genuinely unlike any other programming language.

When TypeScript comes across *JavaScript* code with errors in it, like the age example shown before, TypeScript will still compile and *emit* the JavaScript code to the resulting compiled build, regardless of if there are type errors in it.

This code compiles and is emitted:

```
// index.js  
var age = 13;  
age = 'old as heck'; // TypeError: number not assignable to string
```

TypeScript's tolerance of totally un-typed vanilla JavaScript makes it *very easy* for someone to **gradually refactor a legacy JavaScript project to TypeScript**.

If you tried to do this in a `.ts` file, the code **wouldn't compile**. The idea is that `.js` files are the Wild West, able to get away with whatever. Who really knows how things were back then? Whereas `.ts` files embody the new civilized era of enterprise craftsmanship, held to the TypeScript compiler's scrutiny. No cowboy-code here.

Why JavaScript doesn't scale

Microsoft called TypeScript "*JavaScript that scales*"... what's so *unscalable* about JavaScript?

Concerning software development, there are **two ways** to think about scalability.

1. Performance scalability
2. Productivity scalability

TypeScript is meant to address **2. Productivity scalability**.

Like most dynamically-typed languages, the **lack of types** in JavaScript can drastically improve **initial productivity levels** on certain projects. Still, some factors exist in other

projects (team size, code size, intended code lifespan, domain complexity), in which the lack of types can be detrimental to **code quality** and **understandability**.

It's been agreed upon that:

- It's better to enable the compiler to catch silly bugs, typos, and other errors at compile-time, rather than in production at runtime.
- Tests are the best documentation possible for your code. Types are no substitute for writing tests, but they can do an excellent job of reducing the surface area of bugs.
- Tests also enable faster and safer refactoring. Similarly, if no tests exist, types can (at the very least) catch syntactic inaccuracies.

When to use TypeScript

Even though TypeScript has made a significant effort to be as uncumbersome as possible, some developers still find it preferable to use vanilla JavaScript.

When I first discovered TypeScript, I certainly didn't see the benefit... up until I started experiencing some really annoying stuff like builds, not failing when they should. Eventually, the buggy code and typos that continuously found their way into production code started to get to me.

To make matters worse, as my project demands started to get more complex, I found it increasingly challenging to express domain concepts in a clean and clear object-oriented way.

9 months into adopting TypeScript, I've built new features in Angular apps for clients, changed a massive React/Redux front-end to TypeScript, and ported several backend services to TypeScript from vanilla Node.js, refactoring and deleting mass amounts of code along the way.

In this section, we'll take a look at some of the most common scenarios and identify when it might be vital to use TypeScript, and when we could probably do without it and stick to vanilla JS.

Categories of hard software problems

Firstly, I'd like to present to you a framework for how I categorize all of the hard software development problems that currently exist. They fit into one of three categories of **hard software problems**.

1. The Performant system problem
2. The Embedded system problem
3. The Complex domain problem

I - The performant system problem

To best explain this, let's take Twitter as an example.

Twitter is a straightforward concept in actuality. You sign up, you make tweets, you like other people's tweets, and that's pretty much it. If Twitter is that simple, why couldn't someone else do it?

Because the real challenge for Twitter is not actually so much as “what it does”, but it’s “how it’s **able** to do what it does”.

Twitter has the unique challenge of serving requests from approximately 500 million users every single day.

The hard problem that Twitter solves is actually a **performance problem**.

When the challenge is *performance*, the discussion about the language and paradigm you choose to solve that problem is a lot less critical.

2 - The embedded system problem

An embedded system is a combination of computer hardware and software to enable control over the mechanical or electrical aspects of a system.

Most systems we use today are built on a very complex layer of code that, if not initially written in, compiles down to C or C++ usually.

Coding in these languages is not for the faint of heart.

In C, there is **no such thing as objects**, and we as humans like objects because **we can easily understand them**. C is also procedural, which makes the code in this language a challenge to keep clean. Embedded systems problems also require knowledge of the lower-level details.

C++ does make life a whole lot better because it has object-orientation, but the challenge is still fundamentally interacting with lower-level hardware details.

Because we don’t really have that much of choice on the languages we use and because the *fundamental complexity* here is primarily interaction with low-level details, it’s less useful to discuss language and paradigm here.

3 - Complex domain problem

Number 3 is my favorite challenge in software development, which is the *complex domain problem*.

With these problems, the challenge is less about scaling in terms of handling more requests, but instead scaling the codebase to keep it **maintainable** and **flexible** while pushing out new features for users, without breaking existing ones (**reliability**).

To me, this is the exciting stuff that enterprise companies like Gitlab, Salesforce, Google, and Airbnb have invested countless hours in managing **problem domain complexity**. The most significant engineering challenges are usually:

- Modeling the domain concepts and actually solving the problems of the domain, using terminology everyone can understand.
- Being able to logically separate parts of that monolith into smaller apps.
- Physically split those parts of the app into teams to be assigned to maintain those apps (microservices).
- Integrating business logic and synchronizing data between these microservices.
- Not getting lost in the mass amounts of code written.

- Not degrading productivity and slowing down to the point where it becomes impossible to add new features without breaking existing ones.
- Writing code that accurately expresses the domain model and language, enabling new developers to learn the domain by simply scanning the code (i.e., embedding the business in the software).

I've mainly described the types of problems that Domain Driven Design solves. For these types of projects, you wouldn't even think about not using a strictly-typed language like TypeScript.

Object-Oriented JavaScript

For **complex domain** problems, if you don't choose TypeScript and instead, choose JavaScript, it will require some extra effort to be successful. Not only will you have to be **extra comfortable** with your object modeling abilities in vanilla JavaScript, but you'll also have to know how to utilize the 4 principles of object-oriented programming (encapsulation, abstraction, inheritance, and polymorphism).

This can be hard to do. JavaScript doesn't naturally come with concepts of interfaces and abstract classes.

"Interface Segregation" from the **SOLID design principles** isn't easily achievable with vanilla JavaScript.

Using JavaScript alone would also require a certain level of discipline as a developer to keep the code clean, and this is vital once the codebase is sufficiently large. You're also left to ensure that your team shares the same discipline, experience, and knowledge level on how to implement universal design patterns in JavaScript. If not, you'll need to guide them.

In Domain-Driven projects like this, the strong benefit from using a strictly typed language is *less* about expressing what **can be done**, but more about using encapsulation and information hiding to *reduce the surface area of bugs* by limiting what domain objects are **actually allowed to do**.

We can live without this on the front-end, but it's a **hard language requirement for the backend** in my books. It's also the reason why I moved my Node.js backend services to TypeScript.

Out of all three categories of hard software problems, the Complex Domain Problem is the only one where I would recommend TypeScript as an absolute necessity.

Besides this, other factors might determine when it's best to use TypeScript for your JavaScript project.

Code size

Code size often ties back to the *complex domain problem*, where a large codebase means a complex domain, but that's not always the case.

When the amount of code in a project gets to a certain size, it becomes **harder** to keep track of everything that exists and becomes **easier** to end up re-implementing something already coded.

Duplication is the enemy to well-designed and stable software.

This is especially heightened when new developers start coding on an already large code-base.

Visual Studio Code's autocomplete and Intellisense helps to navigate through huge projects. It works really well with TypeScript, but it's somewhat limited with JavaScript.

For projects that I know will stay simple and small, or if I know that it will be thrown away eventually, I would be less pressed to recommend TypeScript as a necessity.

Production software vs. pet projects

Production software is code that you care about or code that you'll get in trouble for if it doesn't work. It's also code that you've written tests for. The general rule of thumb is that if you care about the code, you need to have unit tests for it.

If you don't care and won't get in trouble for not having tests, don't worry about tests.

Pet projects are self-explanatory. Do whatever you like. You have no professional commitment to upholding any standards of craftsmanship whatsoever.

Go on and make things! Make small things, make big things.

Maybe someday you'll experience the pain when your pet project turns into your main project, which **turns into production software**, which is buggy because it **didn't have tests or types** — not like I've been there or anything...

Lack of Unit Tests

It's not always possible to have tests for everything, because, well... **life**.

In that case, I'd say that if you don't have unit tests, the next best thing you could have is compile-time checking with TypeScript.

However, compile-time checking is **not a substitute** for having unit tests. The good thing is that unit tests can be written in any language- so the argument for TypeScript here is irrelevant. What's important is that tests are written, and we are confident about our code.

Startups

Definitely use whatever helps you become productive.

At this time, the language you choose matters a lot less.

The most important thing for you to do is to validate your product.

Choosing a language (Java, for example) or a tool (like Kubernetes) that you heard would help you scale in the future, while being totally unfamiliar with it and needing to spend time learning, may or may not be the best option in the case of a startup.

Depending on how early you are, the most important thing for you to do is to be productive.

In Paul Graham's famous article, The Python Paradox, his main point is that startup engineers should just use the technology that maximizes their productivity.

Overall, in this case, use whatever you're most comfortable with: types or no types. You can always refactor towards a better design once you know you've built something people actually want.

Working on Teams

Depending on the size of your team and the frameworks you're using, using TypeScript might be a make or break kind of thing.

Large teams

When teams are sufficiently large (because the problems are sufficiently large), it's a good reason to use an opinionated framework, like Angular for the front-end, and TypeScript for the backend.

The reason why using an opinionated framework is beneficial is because you limit the number of possible ways for people to accomplish something. In Angular, there's pretty much one main way to add a Route Guard, use Dependency Injection, hook up Routing, Lazy-Loading, and Reactive Forms.

The huge benefit here is that the API is well specified.

With TypeScript, we save massive amounts of time and make communication efficient.

The ability to quickly determine the required arguments and its return type for any method, or the ability to explicitly describe program intent through public, private, and protected variables alone are incredibly useful.

Yes, some of this is possible with JavaScript, but it's hacky.

Communicating patterns & implementing design principles

Not only is it hard to express program intent without explicit types, but design patterns, the solutions to commonly occurring problems in software are more easily communicated through explicit strictly-typed languages.

Here's a JavaScript example of a common pattern. See if you can identify what it is.

```
// audioDevice.js

class AudioDevice {
  constructor () {
    this.isPlaying = false;
    this.currentTrack = null;
  }

  play (track) {
    this.currentTrack = track;
    this.isPlaying = true;
    this.handlePlayCurrentAudioTrack();
  }
}
```

```

handlePlayCurrentAudioTrack () {
    throw new Error(`Subclasss responsibility error`)
}
}

class Boombox extends AudioDevice {
constructor () {
    super()
}

handlePlayCurrentAudioTrack () {
    // Play through the boombox speakers
}
}

class IPod extends AudioDevice {
constructor () {
    super()
}

handlePlayCurrentAudioTrack () {
    // Ensure headphones are plugged in
    // Play through the ipod
}
}

const AudioDeviceType = {
    Boombox: 'Boombox',
    IPod: 'IPod'
}

const AudioDeviceFactory = {
    create: (deviceType) => {
        switch (deviceType) {
            case AudioDeviceType.Boombox:
                return new Boombox();
            case AudioDeviceType.IPod:
                return new IPod();
            default:
                return null;
        }
    }
}

const boombox = AudioDeviceFactory
    .create(AudioDeviceType.Boombox);

```

```
const ipod = AudioDeviceFactory
    .create(AudioDeviceType.IPod);
```

If you guessed **Abstract Factory Pattern**, you're right. Depending on your familiarity with the pattern, it might not have been that obvious to you.

Let's look at it in TypeScript now. Look at how much more intent we can signify about `AudioDevice` in TypeScript.

```
// audioDevice.ts

abstract class AudioDevice {
    protected.isPlaying: boolean = false;
    protected.currentTrack: ITrack = null;

    constructor () {}

    play (track: ITrack) : void {
        this.currentTrack = track;
        this.isPlaying = true;
        this.handlePlayCurrentAudioTrack();
    }

    abstract handlePlayCurrentAudioTrack () : void;
}
```

Immediate improvements

- We know the class is abstract **right away**. We needed to sniff around in the JavaScript example.
- `AudioDevice` can be instantiated in the JavaScript example. This is bad, we intended for `AudioDevice` to be an abstract class. And abstract classes shouldn't be able to be instantiated, they're only meant to be subclassed and implemented by concrete classes. This limitation is set in place correctly in the TypeScript example.
- We've signaled the scope of the variables.
- `CurrentTrack` refers to an interface. As per the Dependency Inversion design principle, we should prefer depending on abstractions instead of concretions. This isn't possible in the JavaScript implementation.
- We've also indicated that any subclasses of `AudioDevice` needs to implement the `handlePlayCurrentAudioTrack`. In the JavaScript example, we exposed the possibility for someone to introduce runtime errors trying to execute the method from either the illegal abstract class or the non-complete concrete class implementation.

Takeaway: If you work on a large team and you need to minimize the potential ways someone could misuse your code, TypeScript is a good way to help fix that.

Smaller teams & coding styles

Smaller teams are a lot easier to manage coding styles and communication. Paired with linting tools, frequent discussions about how things will get done and pre-commit hooks, I think small teams can be really successful without TypeScript.

I think that success is an equation involving the size of the codebase and the size of the team.

As the codebase grows, the team might find that they need to rely on some help from the language itself to remember where things are and how they should be.

As the team grows, they might find they need more rules and restrictions to keep the style consistent and prevent duplicate code.

Frameworks

React

Much of what draws me and other developers to React is the ability to write code however you want and in an elegant/clever way.

It's true that React makes you a better JavaScript developer because it forces you to approach problems differently, it forces you to be aware of how *this binding* in JavaScript works and enables you to compose large components out of small ones.

React also allows you to have a bit of your own style. And because of the **number of ways I can implement any given task**, I will most often write vanilla React.js apps when:

- The codebase is small
- It's just me coding it

And I will **compile it with TypeScript** when:

- More than 3 people are coding it, or
- The codebase is expected to be very large

Angular

I will also optionally use Angular for the same reasons I will compile React with TypeScript.

Summary on when to use TypeScript

In summary, **you should consider TypeScript when**:

- Out of the 3 types of Hard Software Problems, yours is #3, the Complex Domain problem (meaning that the real challenge comes from you being able to manage the complexity of a really complex domain).
- Complex domains call for implementing Domain-Driven Design. TypeScript is very well suited for that.
- When the codebase is expected to be exceptionally large or live a long life.

TypeScript makes it easier to:

- Catch errors and express expected types
- Implement design patterns and classical object-oriented programming principles (JavaScript doesn't even have interfaces)

- Write SOLID code (much easier with TypeScript)
- When you're working on large teams.
- Larger teams means **considerably more effort required** towards communicating program intent and restricting illegal program behavior. If we use TypeScript, the compiler can enforce a lot of that. If we don't, we have to rely on linters and a lot of verbal agreements within our team.
- This is also why Angular is so popular in the enterprise community. Because it's opinionated and there's a single specific way to accomplish any one task whereas with React, you could implement the same feature in one hundred different ways.

Using JavaScript is probably a good choice when:

- The problem you're solving is not a Complex Domain problem.
- And if it is, only a good idea when you and your team have an exceptionally good grasp on domain modeling using JavaScript's prototypal OOP style and how to write SOLID code without the presence of interfaces or abstract classes.
- The code size is relatively small.
- It's just you working on it.
- You're in a startup and finding product-market fit and traction is way more important than engineering something that might not stick.

Getting started with TypeScript

Let's walk through the process of creating a basic TypeScript application and compiling it. You'd be surprised at how trivial it is.

Afterwards, we'll setup a few scripts for cold-reloading in development, building for production, and running in production.

Fork the TypeScript starter code: You can view and fork the source code the starter TypeScript project right here.

Prerequisites

- You should have Node and npm installed ([install here](#))
- You should be familiar with Node and the npm ecosystem
- You have a code editor installed (preferably VS Code, it's the champ for TypeScript, [install here](#))

Initial Setup

Let's create a folder for us to work in.

```
mkdir typescript-starter
cd typescript-starter
```

Next, we'll setup the project package.json and add the dependencies.

Setup Node.js package.json

Using the -y flag when creating a package.json will approve all the defaults.

```
npm init -y
```

Add TypeScript as a dev dependency

This probably doesn't come as a surprise ;)

```
npm install typescript --save-dev
```

After we install typescript, we get access to the command line TypeScript compiler through the tsc command. More on that below.

Install ambient Node.js types for TypeScript

Recall that TypeScript has Implicit, Explicit, and **Ambient** types? Ambient types are types that get added to the global execution scope. Since we're using Node, it would be good if we could get type safety and auto-completion on the Node APIs like file, path, process, etc. That's what installing the DefinitelyTyped type definition for Node will do.

```
npm install @types/node --save-dev
```

Create a tsconfig.json

The tsconfig.json is where we define the TypeScript compiler options. We can create a tsconfig with several options set.

```
npx tsc --init --rootDir src --outDir build \
--esModuleInterop --resolveJsonModule --lib es6 \
--module commonjs --allowJs true --noImplicitAny true
```

- **rootDir**: This is where TypeScript looks for our code. We've configured it to look in the src/ folder. That's where we'll write our TypeScript.
- **outDir**: Where TypeScript puts our compiled code. We want it to go to a build/ folder.
- **esModuleInterop**: If you were in the JavaScript space over the past couple of years, you might have recognized that modules systems had gotten a little bit out of control (AMD, SystemJS, ES Modules, etc). For a topic that requires a much longer discussion, if we're using commonjs as our module system (and for Node.js apps, you should be), then we need this to be set to true.
- **resolveJsonModule**: If we use JSON in this project, this option allows TypeScript to use it.
- **lib**: This option adds *ambient* types to our project, allowing us to rely on features from different ECMAScript versions, testing libraries, and even the browser DOM api. We'd like to utilize some es6 language features. This all gets compiled down to es5.
- **module**: commonjs is the standard Node module system in 2019. Let's use that.
- **allowJs**: If you're converting an old JavaScript project to TypeScript, this option will allow you to include .js files among .ts ones.
- **noImplicitAny**: In TypeScript files, require every type to have been explicitly, never implicitly — typed. Every type needs to either have a specific type or be explicitly declared any. No implicit any.

At this point, after you should have a tsconfig.json that looks like this:

```
{  
  "compilerOptions": {  
    /* Basic Options */  
    "target": "es5",  
    /* Specify ECMAScript target version: 'ES3' (def  
    "module": "commonjs",  
    /* Specify module code generation: 'none', 'comm  
    "lib": ["es6"],  
    /* Specify library files to be included in the c  
    "allowJs": true,  
    /* Allow javascript files to be compiled. */  
    "outDir": "build",  
    /* Redirect output structure to the directory. */  
    "rootDir": "src",  
    /* Specify the root directory of input files. Us  
  
    /* Strict Type-Checking Options */  
    "strict": true,  
    /* Enable all strict type-checking options. */  
    "noImplicitAny": true,  
    /* Raise error on expressions and declarations w  
  
    /* Advanced Options */  
    "resolveJsonModule": true  
  }  
}
```

For legibility purposes, I've cleaned up all the other commented-out options.

Now we're set to run our first TypeScript file.

Create the src/ folder and create our first TypeScript file

```
mkdir src  
touch src/index.ts
```

And let's write some code.

```
console.log('Hello world!')
```

Compiling our TypeScript

To compile our code, we'll need to run the `tsc` command using `npx`, the Node package executer. `tsc` will read the `tsconfig.json` in the current directory, and apply the configuration against the TypeScript compiler to generate the compiled JavaScript code.

```
npx tsc
```

Our compiled code

Check out `build/index.js`, we've compiled our first TypeScript file.

```
"use strict";  
console.log('Hello world!');
```

Useful configurations & scripts

Cold reloading development script

Cold reloading is nice for local development. In order to do this, we'll need to rely on a couple more packages: `ts-node` for running TypeScript code directly without having to wait for it to be compiled, and `nodemon`, to watch for changes to our code and automatically restart when a file is changed.

Cold vs. Hot-reloading: *Cold* reloading is when the entire application must **restart** in order to update. *Hot* reloading is when source code changes can be applied against the **currently running process** without requiring a restart.

Let's install 'em with this command.

```
npm install --save-dev ts-node nodemon
```

Add a `nodemon.json` config.

```
{  
  "watch": ["src"],  
  "ext": ".ts,.js",  
  "ignore": [],  
  "exec": "ts-node ./src/index.ts"  
}
```

And then to run the project, all we have to do is run `nodemon`. Let's add a script for that.

```
"start:dev": "nodemon",
```

By running `npm run start:dev`, `nodemon` will start our app using `ts-node ./src/index.ts`, watching for changes to `.ts` and `.js` files from within `/src`.

Creating production builds

In order to *clean* and compile the project for production, we can add a build script.

Install `rimraf`, a cross-platform tool that acts like the `rm -rf` command (just obliterates whatever you tell it to).

```
npm install --save-dev rimraf
```

And then, add this to your `package.json`.

```
"build": "rimraf ./build && tsc",
```

Now, when we run `npm run build`, `rimraf` will remove our old `build` folder before the TypeScript compiler emits new code to build.

Production startup script

In order to start the app in production, all we need to do is run the `build` command first, and then execute the compiled JavaScript at `build/index.js`.

The startup script looks like this.

```
"start": "npm run build && node build/index.js"
```

I told you it was simple!

[View the Starter Project source](#)

A reminder that you can view the entire source code for this here.

Scripts Recap

```
npm run start:dev
```

Starts the application in development using nodemon and ts-node to do cold reloading.

```
npm run build
```

Builds the app at build, cleaning the folder first.

```
npm run start
```

Starts the app in production by first building the project with `npm run build`, and then executing the compiled JavaScript at `build/index.js`.

TypeScript Language Features

Since TypeScript has a lot in common with other strictly-typed classical object-oriented programming languages like C# or Java, you might be familiar with the majority of what TypeScript has to offer.

The novel aspect of the language is how **expressive** the structural type system can be.

In this section, we'll gloss over the most common language features you'll use in your development efforts with TypeScript.

TypeScript Playground: If you're curious to see what the resulting JavaScript code your TypeScript compiles to, check out TypeScript Playground.

Basic types

Object-Oriented Programming Features

Special types

Basic types

We mentioned earlier that types are annotated using the *polish postfix* notation (though, that probably doesn't mean much to you). Just know that anytime you want to specify the type of something, it'll appear in the form of a `:TypeAnnotation`.

To demonstrate, let's see usage with some of the basic primitive TypeScript types.

Primitive Types

The primitive types of JavaScript are also primitive types of TypeScript. That's `number`, `string`, and `boolean`.

Number

```
let num: number = 12;
let binary: number = 0b110;

num = 55;
binary = '222' // Error - Type "222" is not assignable to type 'number'.
```

String

Either single quotes or double quotes are ok.

```
let firstName: string = 'Khalil';
let lastName: string = "Stemmler";
```

You can also use the backtick (`) and string embed expression (\${ }) in order to embed other primitives.

```
let firstName: string = 'Khalil';
let lastName: string = "Stemmler";

let message: string = `This book was written by ${firstName} ${lastName}`;
```

Boolean

It's pretty much what you would expect.

```
let isLoading: boolean = false;
isLoading = true;
isLoading = 'false'; // Error
```

Arrays

There are two ways to declare an array in TypeScript. You can use the type [] format or the *Generic* format.

```
// Common usage.
let firstFivePrimes: number[] = [2, 3, 5, 7, 11];
// Using Generics (more later). Not as common.
let firstFivePrimes2: Array<number> = [2, 3, 5, 7, 11];
```

Object-Oriented Programming Features

TypeScript lets you write code using the traditional object-oriented style (opposed to JavaScript's prototypal OOP style) and enables us to utilize familiar object-modeling constructs like abstract classes and interfaces.

If you're rusty on your OOP, that's totally cool. We'll get you back up to speed in Chapter 5 — Object-Oriented Programming & Domain Modeling.

Classes

Object-Oriented developers who've started with C# or Java will be pleased to find that it feels pretty similar in TypeScript.

```
class Point {  
    x: number; // instance variables  
    y: number;  
  
    constructor (x: number, y: number) { // constructor  
        this.x = x;  
        this.y = y;  
    }  
  
    add (point: Point) { // method  
        return new Point(this.x + point.x, this.y + point.y);  
    }  
}  
  
var p1 = new Point(0, 10);  
var p2 = new Point(10, 20);  
var p3 = p1.add(p2); // {x:10,y:30}
```

Class inheritance

Similar to other languages, classes in TypeScript support **single inheritance**. This means that we can use the `extends` keyword to create a class hierarchy, but only **once** per class:

```
import { Point } from './point'  
  
// Point3D extends the Point class  
class Point3D extends Point {  
    z: number;  
  
    constructor(x: number, y: number, z: number) {  
        super(x, y); // Required  
        this.z = z;  
    }  
  
    add(point: Point3D) {  
        var point2D = super.add(point);  
        return new Point3D(point2D.x, point2D.y, this.z + point.z);  
    }  
}
```

Any time we extend a class, the subclass **must invoke the parent's constructor using `super()`**. This is a mandatory thing, and TypeScript will yell at you if you don't do it.

In some languages, the **first statement in a constructor** from a child class needs to be one

that calls `super()`. ****In TypeScript, it doesn't matter *when* you call `super` so long as you do.

```
constructor(x: number, y: number, z: number) {
  this.z = z; // this also works!
  super(x, y);
}
```

Try it out yourself: See what the resulting JavaScript looks like on [TypeScript Playground](#).

Static properties

TypeScript also supports the ability to label properties as `static`. `static` properties (this could be members/attributes or methods) are different in the sense that they belong to the *class* themselves, **not** to *instances* of the class — objects.

```
class Player {
  static instancesCreated = 0; // class variable

  constructor () {
    Player.instancesCreated++;
  }

  // Static (class) method (only accessible through the class itself)
  public static createPlayer (type: PlayerType): Player {
    ...
  }

  // Instance method (only accessible through an instance of the class)
  public shoot (): void {
    ...
  }
}

var p1 = new Player();
var p2 = new Player();
console.log(Player.instancesCreated); // 2

p1.shoot();

Player.shoot(); /* ErrorProperty 'shoot' does not
exist on type 'typeof Player'. */

console.log(p1.instancesCreated); /* Property 'instancesCreated' is a
static member of type 'Player' */
```

Instance variables

There's a lot of confusion around what we call some of these things. An instance variable is

a non-static class member/attribute. They are accessible only through instances of the class. From *inside the class*, using the `this` keyword gives us access to the instance variables.

```
class Point {  
    x: number; // instance variables  
    y: number;  
  
    ...  
  
    public printCoordinates (): void {  
        // Accessed through `this`  
        console.log(`x: ${this.x}, y: ${this.y}`)  
    }  
}
```

From *outside* the class, when working with an **object** created from that class, we can access instance variables using dot-notation.

```
const point = new Point(12, 14);  
console.log(point.x);  
console.log(point.y);
```

Of course, your ability to access these variables depends entirely on the *access modifiers* that describe the scope of them.

Access Modifiers

TypeScript supports **public**, **private**, **protected** modifiers, which determine the accessibility of a **class property**.

Access modifier scopes in TypeScript

Access modifier	Access from other classes?	Access from subclasses?
public	yes	yes
protected	no	yes
private	no	no

A `public` modifier is the most permissive. When declaring a property on a class, if we don't include an access modifier, by default, TypeScript assumes that the property is public.

```
class Person {  
    name: string; // public, by default.  
    ...  
}
```

A method or member/attribute with a `public` modifier can be accessed through:

- an instance of the class (object)
- inside the containing class (`this`)

A property with a `private` modifier can only be accessed from *inside* the class where it's defined. This means that instances of the class (objects created using the `new` keyword) don't have the ability to access these properties.

A `protected` modifier dictates that **only** the class that defines a `protected` modifier **and subclasses of that class** can access it.

Readonly Modifier

`Readonly` properties are properties that can't be changed once they've been set. A `read-only` property must be initialized at their declaration or in the constructor.

```
class Spider {  
    readonly name: string;  
    readonly number0fLegs: number = 8;  
    constructor (theName: string) {  
        this.name = theName;  
    }  
}
```

Interfaces

Interfaces allow us to declare the structure of classes and variables. For a class or a variable to be deemed valid to the type specified by the interface, it needs to include all of the properties and methods included in the interface definition.

```
interface Coordinate {  
    latitude: number;  
    longitude: number;  
    dateCreated?: Date; // Properties can also be optional  
}  
  
const coordinate: Coordinate = { latitude: 42.122, longitude: -28.241 }
```

One really interesting thing to note about interfaces in TypeScript is that they don't compile to anything in JS.

So if we took the following TypeScript code:

```
console.log('No coordinate interface exists');

interface Coordinate {
    latitude: number;
    longitude: number;
}

console.log("See, it's not there.")
```

... and ran it through a TypeScript compiler, the resulting JavaScript code would look like this:

```
console.log('No coordinate interface exists');
console.log("See, it's not there.")
```

Classes implementing interfaces

When a class implements an interface, for the class to be complete, it needs to include all of the members defined in the interface (whether that be properties or methods).

```
interface ILogEvents {
    logger: Logger;
    logEvent: (event: Event) => void;
}

class DomainEvents implements ILogEvents {
    logger: Logger; // logger property

    constructor (logger: Logger) {
        this.logger = logger;
    }

    logEvent (event: Event) : void { // same method signature as
        this.logger.log(event);           // logEvent: (event: Event) => void */
    }
}
```

Interfaces extending interfaces

Unique from other languages with interfaces, interfaces can actually **extend one or more interfaces**.

```
interface ICircle {
    readonly id: string;
    center: {
        x: number;
        y: number;
    },
}
```

```

    radius: number;
    color?: string; // Optional property
}

interface ICircleWithArea extends ICircle {
    getArea: () => number;
}

const circle3: ICircleWithArea = {
    id: '003',
    center: { x: 0, y: 0 },
    radius: 6,
    color: '#fff',
    getArea: function () {
        return (this.radius ** 2) * Math.PI;
    },
};

```

Generics

Generics are incredibly useful.

The key motivation for generics is to document meaningful type dependencies between members. The members can be:

- class instance members
- class methods
- function arguments
- function return value

Let's look at an example of a queue that throws me back to my Java days of creating Abstract Data Types.

```

interface Queue<T> {
    data: T[];
    push: (t: T) => void
    pop: () => T | undefined;
}

```

By this declaration, we can create a Queue of numbers, strings, Monkeys, Dealies, and any other type we want.

```

interface Monkey {
    name: string;
    color: string;
}

class MonkeyQueue implements Queue<Monkey> {
    data: Monkey[];

```

```

constructor () {
    this.data = [];
}

push (t: Monkey) : void {
    this.data.push(t);
}

pop () : Monkey | undefined {
    return this.data.shift();
}

```

Convenience Generic

Here's a common use case. Consider trying to apply types to values coming into from the internet and handled by a web controller. Usually, when we pass complex objects to backend applications, they come back as raw strings.

```

type CreateUserRequestDTO = {
    userId: string;
    email: string;
    password: string;
}

class CreateUserController {
    ...
    public handleRequest (
        req: express.Request, res: express.Response
    ): Promise<void> {

        const createUserDTO: CreateUserRequestDTO = req.body; /* Error,
            not assignable to 'CreateUserRequestDTO' */
    }
}

```

One thing I like to do is handle marshall the raw string into a JSON object and then apply the type to it using a ParseUtils class with a `parseObject<T>` method.

```

export class ParseUtils {

    public static parseObject<T> (raw: any): T {
        let returnData: T;

        try {
            returnData = JSON.parse(raw);
        } catch (err) {
            throw new Error(err);
        }
    }
}

```

```

    }

    return returnData;
}

}

```

We can use this by:

```

type CreateUserRequestDTO = {
  userId: string;
  email: string;
  password: string;
}

class CreateUserController {
  ...
  public handleRequest (
    req: express.Request, res: express.Response
  ): Promise<void> {

    const createUserDTO: CreateUserRequestDTO = ParseUtils
      .parseObject<CreateUserRequestDTO>(req.body);
  }
}

```

Abstract classes

abstract classes can be thought of as an access modifier. We present it separately because opposed to the previously mentioned modifiers, it can be on a class as well as any member of the class. Having an abstract modifier primarily means that such functionality cannot be directly invoked, and a child class must provide the functionality.

- abstract classes cannot be directly instantiated. Instead, the user must create some class that inherits from the abstract class.
- abstract members cannot be directly accessed, and a child class must provide the functionality.

Special types

Let's get into the fun ones. Here's a collection of some of the most common useful features of TypeScript.

Type assertions

Similar to *type casting*, when we have a better understanding of what a particular type might be than the compiler does, we can assert the type. This helps the compiler figure out how to deal with that type.

```

const friend = {};
friend.name = 'John'; /* Error! Property 'name' does
    not exist on type '{}'
*/

interface Person {
    name: string;
    age: number;
}

const person = {} as Person;
person.name = 'John'; // Okay

```

The “type” keyword

In TypeScript, there are several different ways to specify the *type* of something. You can type a variable, parameter, or return value using a class, interface, or the type keyword.

Let's say we were working on a feature to create a user. We can define a type that contains everything we need in order to do that.

```

type CreateUserRequestDTO = {
    userId: string;
    email: string;
    password: string;
}

function createUser (request: CreateUserRequestDTO): User {
    // Do things to create and return a user
}

```

Looks good. Watch this, though. We can achieve **exact same thing** using an interface.

```

interface CreateUserRequestDTO {
    userId: string;
    email: string;
    password: string;
}

function createUser (request: CreateUserRequestDTO): User {
    // Do things to create and return a user
}

```

Not only that, but because TypeScript is *structurally typed*, we could also use a class. This works as long as the *class properties structurally equivalent* to the members of the interface or type. Check it out.

```

class CreateUserRequestDTO {
    public userId: string;
    public email: string;
}

```

```

public password: string;

constructor (userId: string, email: string, password: string) {
    this.userId = userId;
    this.email = email;
    this.password = password;
}
}

<�断点>
/** 
 * The class members for CreateUserRequestDTO looks like:
 *
 * {
// ===== Public properties

userId: string;
email: string;
password: string;

// ===== Class constructor
// Remember, a type is valid
// even if it has more than the required attributes.

new: () => User;
}
*/


function createUser (request: CreateUserRequestDTO): User {
    // Do things to create and return a user
}

```

Class properties: The entirety of members (attributes) and methods of a class are the *class properties*.

This does beg the question of when we might consider the use of type over interfaces or classes? If they can all do the same thing, why bother using type at all?

Unlike an interface, **type aliases** can be used to create more complex types.

```

// Primitive
type Name = string;

// Tuple
type Data = [number, string];

// Object
type PointX = { x: number; };
type PointY = { y: number; };

```

```
// Union (Or - At least one required)
type IncompletePoint = PointX | PointY;

// Extends (And - All required)
type Point = PointX & PointY

const pX: PointX = { x: 1 };
const incompletePoint: IncompletePoint = { x: 1 };

const point: Point = { x: 1 } // Error Property 'y' is missing
                           // in type '{ x: number; }' but
                           // required in type 'PointY'.
```

Type Aliases

Types can refer to primitive data types. Sometimes this makes sense to do in order to make your code more expressive and intention-revealing.

```
type BandName = string;
```

We can alias just about any existing type. For example, check out how we can create an alias for an array of Jobs.

```
class Job {
    public title: string;

    constructor (title: string) {
        this.title = title;
    }
}

type JobCollection = Job[]; // Alias for an array of jobs

const jobs: JobCollection = [];
jobs.push(new Job("Software Developer"));
jobs.push(12) // Error
```

Union Type

TypeScript allows us to create a type from one or more types. This is called a union type.

```
type Password = string | number;
```

The union type works like a **conditional OR**. In order to pass type checking, a variable must conform to **at least one** of the types defined in the union.

```
type Password = string | number;

let password = "secretpassword";
```

```
password = 1234354

password = true // error - Password isn't assignable to type 'boolean'.
```

Intersection Type

The intersection type is a type that combines all of the properties of one or more types.

```
interface PointX {
  x: number;
}

interface PointY {
  y: number;
}

type Point = PointX & PointY;

const initialPoint: Point = { x: 0, y: 0 }
const incompletePoint: Point = { x: 0 } // Error Property 'y' is missing
                                         // in type '{ x: number; }' but
                                         // required in type 'PointY'.
```

Enum

An enum is a way to organize a collection of related values. This is a language feature that's common in other programming languages but was never added to JavaScript. TypeScript, however, has this feature.

Usage looks like this, similar to other languages:

```
enum Instrument {
  Guitar,
  Bass,
  Keyboard,
  Drums
}

let instrument = Instrument.Guitar;

instrument = "screwdriver"; /* Error! Type '"screwdriver"'
is not assignable to type 'Instrument'.
*/
```

It's really interesting how enums work under the hood. Since everything we write in TypeScript has to be compiled to valid JavaScript, look what the resulting JavaScript looks like for the `Instrument` enum after it goes through the compiler.

```

var Instrument;
(function (Instrument) {
    Instrument[Instrument["Guitar"] = 0] = "Guitar";
    Instrument[Instrument["Bass"] = 1] = "Bass";
    Instrument[Instrument["Keyboard"] = 2] = "Keyboard";
    Instrument[Instrument["Drums"] = 3] = "Drums";
})(Instrument || (Instrument = {}));

```

The first line in the function block says `Instrument[Instrument["Guitar"] = 0] = "Guitar";`.

It's said that:

- The value of "Guitar" is 0 AND
- The value of 0 is "Guitar"

This results in the following object:

```

/**
 * {
 *   0: "Guitar",
 *   1: "Bass",
 *   2: "Keyboard",
 *   3: "Drums",
 *   Guitar: 0,
 *   Bass: 1,
 *   Keyboard: 2,
 *   Drums: 3
 * }
 */

```

That means that enums are, by default, **number-based**. We access that first item in the list with either `Instrument[0]` or `Instrument.Guitar`.

If you don't like using numbers, alternatively, we can initialize enums with strings.

```

enum Instrument {
    Guitar = 'GUITAR',
    Bass = 'BASS',
    Keyboard = 'KEYBOARD',
    Drums = 'DRUMS'
}

```

Any

`any` is a type that we can be used with all types. Anything can *be assigned* to `any`, and we can *assign anything* with `any`. We often use `any` when we want to opt-out of type checking for the moment.

```

let anything: any = 'any is now a string';
anything = 5;

```

```
anything = false;
anything.aMethodThatMightNotExist(); /* If this
doesn't exist at runtime and we try to call on it,
it will throw an error. */
```

In legacy projects migrating to TypeScript, it's not uncommon to temporarily type things as `any` before adding more specific types over time during refactoring.

Void

`void` is the absence of having any return type. For methods that return no value, it's a good practice to type them as `void` explicitly.

```
function executeCommand (name: string): void {
  console.log(`Executing ${name}`);
}
executeCommand('Say hello');
```

There's an object-oriented design principle titled Command/Query Segregation that specifies that a method should be **either** a command that changes the system in some way but returns no value, OR a query that returns a value but *causes no side-effects*. We talk more about this principle in **Section 4 - Design Principles**.

Inline & Literal Types

Sometimes, instead of defining an entire interface for a type, you might feel inclined just to define the type *inline*. Take the following example where we might receive an update to a field name of a (**string primitive**) **Literal Type** of either "email", "password," or "phonenumber".

```
function onUpdate (
  props: { fieldName: 'email' | 'password' | 'phonenumber', value: any }
): void {

  // Possibly => { fieldName: 'email', value: 'me@khalil.com' };
  // Or maybe => { fieldName: 'password', value: 'secretpassword' };

  this.setState({
    ...this.state,
    [fieldName]: value
  })
}
```

Literal Types like in the example shown above are most commonly used with the union type to create quick abstractions.

Here's another common one. What about this **hashtable** of `number`'s to `string`'s?

```
const GenreType = {
  1: "Metal",
```

```
2: "Rap",
3: "Pop"
}
```

How do we properly define the type for this? Easy. Here's an inline type that says that every *key* of this object is a *number*, and the value is a *string*.

```
const GenreType: { [index: number]: string } = {
  1: "Metal",
  2: "Rap",
  3: "Pop"
}
```

And now we get the type safety to reach inside of the hash-table properly. This pattern comes if we want to use the factory pattern to create domain models from a pre-determined list of possible variations.

```
const GenreType: { [index: number]: string } = {
  1: "Metal",
  2: "Rap",
  3: "Pop"
}

interface GenreProps {
  id: number;
  description: string
}

class Genre {
  private props: GenreProps;

  get id (): number {
    return this.props.id;
  }

  get description (): string {
    return this.props.description
  }

  constructor (props: GenreProps) {
    this.props = props;
  }
}

function createGenreFromGenreId (id: number): Genre | null {
  if (id < 1 || id > 3) {
    // It's not great to return null like this, but we'll keep it
    // simple for now.
  }
}
```

```

        return null;
    }
    return new Genre({ id, description: GenreType[id] })
}

```

Type Guards

Type Guards allow you to narrow down the type of an object within a conditional block.

Typeof Guard

Using `typeof` in a conditional block, the compiler will know the type of a variable to be different. In the following example, TypeScript understands that outside the conditional block, `x` might be a boolean, and the function `toFixed` cannot be called on it.

```

function example(x: number | boolean) {
    if (typeof x === 'number') {
        return x.toFixed(2);
    }
    return x.toFixed(2); // Error! Property 'toFixed' does not exist on type 'boolean'.
}

```

Instanceof Guard

Similar conditional checking is possible using the `instanceof` guard. We can conditionally rule out type possibilities by asserting if a class is or is not an instance of a particular class.

```

class MyResponse {
    header = 'header example';
    result = 'result example';
    // ...
}

class MyError {
    header = 'header example';
    message = 'message example';
    // ...
}

function example(x: MyResponse | MyError) {
    if (x instanceof MyResponse) {
        console.log(x.message); // Error! Property 'message' does not exist on type 'MyResponse'.
        console.log(x.result); // Okay
    } else {
        // TypeScript knows this must be MyError

        console.log(x.message); // Okay
        console.log(x.result); // Error! Property 'result' does not exist on type 'MyError'.
    }
}

```

In Guard

The `in` operator checks for the existence of a property on an object.

```
interface Person {  
    name: string;  
    age: number;  
}  
  
const person: Person = {  
    name: 'John',  
    age: 28,  
};  
  
const checkForName = 'name' in person; // true
```

Chapter Summary

You made it! That concludes our discussion about the TypeScript language in particular. We covered a lot so far. You learned:

- The design decisions of TypeScript and the shortcomings TypeScript was meant to solve improve in JavaScript development
- Which types of JavaScript projects TypeScript is best suited for
- How to install and setup a TypeScript project
- TypeScript language features and the basics in order to be deadly

Resources

- The Definitive TypeScript Handbook
- TypeScript Deep Dive
- Javascript, the Good Parts

References

- Why is JavaScript so popular?
- The Tragedy of Craftsmanship
- TypeScript Official Site
- Why TypeScript?

3. Clean Code

Introduction | Clean code is your grip strength

If you're into lifting weights, one of the first things we should do is work on our grip strength, though not many people *do*. Without proper grip, we're not going to be able to lift the heavy weights, and we could end up hurting ourselves.



The equivalent of hurting yourself in programming before you get to the *heavy weights* is writing *unclean code*. Unclean code is a lot of things. Ask any developer, and they'll have a lot to say.

Off the top of the dome, unclean code is:

- code we can't understand
- code that makes it hard to find what we're looking for
- code that's poorly formatted, untestable, and inflexible
- code that you and your team are afraid to change
- expensive

Unclean code kills companies. Unclean code kills companies because it results in re-work, wasted time and money, and failed projects.

Clean code is your grip strength. It's your *form*. We need to get it right before we take on the *heavy weights*. Writing clean code is not *everything*, but it's worth investing in learning how to do right—building good habits, following well-known best practices, and being responsible. It's worth your dedication *today* because unclean code compounds and becomes unmanageable *later*.

We're going to arm you with the **correct mindset**, a set of **coding conventions**, and the ability to understand the consequences your designs may have in the future.

With these out of the way, when we finally get to the heavy weights (those challenging yet *fulfilling* problems to solve), writing clean code will feel as easy and natural as *breathing*.

So let's get that grip strength up for those critical situations; lifters, we all know you don't want a one-way trip to *snap city*.

Chapter Goals

This chapter is broken up into four parts:

Understanding clean code

- Fully understand what it means for code to be clean.
- Learn a gradual and principled approach to endlessly improving your ability to write cleaner and more structured code using the *Pillars of Clean Code*.

Developer mindset

- Understand what makes programming a trade, the goals behind the software craftsmanship *code of honor*, and why we need a standardized set of ways to build software.
- Use pragmatism to reject unrealistic perfection, make intelligent compromise, provide value constantly, and complete projects confidently.
- Write better code and produce better designs **cultivating a growth mindset**, utilizing positive and negative feedback effectively, and acknowledging imperfections as growth opportunities.
- Learn the human-centered design principles to increase discoverability, and make your code easy to understand and work with.

The goals for this chapter are as follows:

- Fully understand the topic of *clean code* and what we need to master in order to write it.
- Learn how to foster a developer mindset and the prerequisite design skills you need as a developer writing code for humans.
- Learn a collection of the most critical language-agnostic *tricks of the trade*. Discover industry standards and coding conventions passed down and widely accepted by the industry.
- Learn the best practices behind formatting, naming things, utilizing comments, error handling, TDD, and refactoring.
- Understand the importance of continuing to learn design principles, patterns, architectural principles to continue to improve your practical skills.

Understanding clean code

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand”. - Martin Fowler (1999)

Everyone seems to have their own opinion about what they deem to be clean code.

I've heard clean code described as plainly as when “everything is organized nicely”. I've heard it take a more opinionated form as “code with tests”. And I've heard it referred to as everything else in between.

Clean code *is*, in part, about code cleanliness. It *is* about following best practices, making code readable, and writing it so that every positive *structural* and *stylistic* trait you can assign to a codebase is evident.

Even as exhaustive as that sounds, it still barely scrapes the surface.

The term, *clean code*, is incredibly overloaded.

Clean code is an overloaded term

We expect clean code to be readable, understandable, flexible, testable, and maintainable.

That's a lot to ask right off the bat (especially if it's the first step in our software design and architecture roadmap).

As a **new or junior developer** now being paid to write code in a professional setting, the process of learning how to write clean code is both **daunting** and somewhat nebulous.

It's daunting because your professional reputation rides on doing a good job, and it's nebulous because telling someone to write clean code is like telling someone to *just run faster* — it's not easy to *just start* doing either of those things.

In this chapter, I'll introduce you to The Pillars of Clean Code. It's a guided approach for learning how to write clean code.

1. 🚧 Developer mindset
2. 🚧 Coding conventions
3. 🚧 Skills & knowledge

Ultimately, you get your head right, develop a set of principled coding conventions, and improve them over time by learning the essential software design principles and patterns.

Sounds straightforward enough, right?

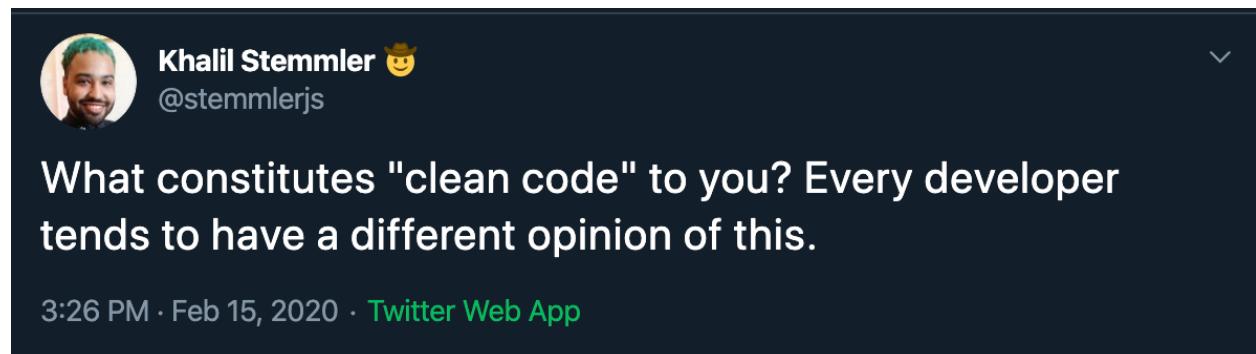
Before we dive into the framework, I want to show you something.

I want to show you **just how vast** this topic of *clean code* really is. During my first pass at taming this material, I really under-appreciated it. Let's start with some of the opinions about what other developers and experts from the community think about clean code.

What the community thinks about clean code

I opened the flood gates.

By that, I mean I went on Twitter and asked the general public for their opinions on what they think constitutes clean code.



Khalil Stemmler 🎉
@stemmlerjs

What constitutes "clean code" to you? Every developer tends to have a different opinion of this.

3:26 PM · Feb 15, 2020 · [Twitter Web App](#)

- See thread here.

What follows are a bunch of really good opinions. I can't say I disagree with any one of them. But as someone who wants to educate themselves about clean code, without a framework, you may have a hard time **narrowing down actionable next steps**.

Community opinions

“Clean code is easy to read and modify, reveals the intended purpose without any obfuscation, and speaks a clear and consistent domain language.”

My thoughts and comments:

- *Easy to read* could be a comment on naming things. It could also be a comment about how the code is physically styled and formatted to be easily read by humans.
- *Easy to modify* could be a comment on code being able to refactor code without breaking it.
- *Reveals the intended purpose* could mean that the names of the classes, methods, functions, and variables are so understandable that they couldn't be named any better.

"[Clean code is] easy to read. It also has no coupling between libraries."

My thoughts and comments:

- Human readability is undoubtedly one of the most important considerations. In the coding conventions section of this chapter, we discuss human-centered-design and explore what makes code readable and what doesn't.
- The second comment is fascinating. It points to something more *architectural*. Admittedly, it's a bit of a rabbit hole - but they're right. Coupling, packaging, dependency inversion, separation of concerns, and decomposition. These are all about enforcing architectural boundaries and keeping dependencies at a distance. It's funny. Coupling isn't one of the first things you think about in the conversation of *clean code*, but it's not to be excluded. The section about Skill & knowledge argues that being principled is what fosters better coding conventions. To me, this never ends. If you've got your Developer Mindset right, you're always investing in your knowledge, picking up best practices, and learning ways to handle common problems in software development. For example, to keep libraries or modules decoupled, we, as an industry, have established well-known approaches to deal with this exact scenario.

"It tells a good story about the domain, and it is evolvable."

- Evolvable software. Yes. We previously established this as one of the primary goals of software : The goal of software . Change is a constant. We're always going to need to change code. So let's design it in a way to accommodate change.
- Telling the *story of the domain* means that new developers can learn how the business works by reading the code. Code that encapsulates the **essence of the business** is desirable. In Chapter 5 - Object-Oriented Programming & Domain Modeling, we learn how to separate the concerns of our app and carve out a domain model to build a codified version of the business. Regardless of if we do that or not, naming things by their real-life counterparts makes for a good naming convention.

"I can read it without it being painful. I can change it without it breaking everything."

- Readability! Again! But also, how do we write code that enables others to change it without breaking things? Tests, that is. Speaking of tests...

"Generally speaking, clean code is testable code!"

Easier said than done! Inherently testable writing code isn't obvious. At least, it wasn't to me when I first started. I certainly do agree with this statement, though.

"To me, it's kinda like art. As a kid, I used to look at Picasso and think: I can do this stuff. Now, I look at it, and it's genius. I really can't tell why... sure I learned

more about shapes, colors, etc. But it's more than just technique. Coding is the same."

This is a charming way of looking at it. Making something appear simple often takes a lot more time, effort, and experience than it does to make something appear complex. Following along with the art analogy, it takes an appreciation of constructs, whether it be a square, circle, brushstroke, or a class, method, or variable.

"No magic. Simplicity."

This echoes simplicity over complexity. I could say a lot more here as well:

- Sometimes the solution is complicated because the problem is complicated. Accidental and Essential Complexity is a way to determine if the problem is difficult, or we make it difficult.

"KISS, DRY, YAGNI, POLA, DIP, ideally facilitated by TDD"

Keep it simple, silly, Do Not Repeat Yourself, You Aren't Gonna Need It, Principle of Least Astonishment, Dependency Inversion Principle, and good ol' Test-Driven Development. Yeah, this is definitely some design principle-soup. But honestly, if you know what each of these are, even if you choose not to follow 'em, having them in the back of your mind while coding **can** improve the structural quality of your designs.

"Easily replaceable."

How do you design code to be replaceable? By making it simple, readable, and providing tests. Using the SOLID principles also helps.

And the last one,

"Code that I don't curse its creator"

Yeah- we've all been there...

What the experts think about clean code

Alright, now let's look at how a few of the experts in our industry eloquently describe clean code.

"The cost of ownership for a program includes the time humans spend to understand it. Humans are costly, so optimize for understandability". — Mathias Verraes

Mathias has a fantastic point here. Have you ever started on a new project in a new domain and had to get ramped up on an existing codebase? The amount of time it takes for you to contribute code is directly related to how understandable it is. Understandability, as we'll learn — is another way to say *discoverability*, in *designer-*terminology.

"Code is like humor. When you have to explain it, it's bad". — Cory House

Perhaps Cory is taking a stance in the topic of leaving *comments* in code. Is code that requires comments clean? Is it dirty? I have an principled argument for this, and you'll hear about it soon enough.

“Clean code always looks like it was written by someone who cares. There is nothing obvious you can do to make it better.” — Michael Feathers

Perhaps the first thing to discuss isn’t actually *how* to write clean code. Instead, maybe we should start with determining if you’re in the right *headspace* to advocate for code cleanliness, craftsmanship, and your fellow human beings. If you’re apathetic about the quality of the code and future maintainers’ ability to maintain your code, we might want to get to the bottom of that first. Section One - Developer Mindset talks about this in more detail.

“Programming is the art of telling another human what one wants the computer to do.” — Donald Knuth

Consider where you were when you first started coding. Remember *wading* through bad code (written by you and maybe others)? Remember all the struggle you went through in order to *just make it work*? Remember how good it felt to *finally* get something to work after flipping lines back and forth for hours? I always think back to my Java days in university... I sat on that couch for hours.

If you feel like you’re flipping lines back and forth, just throwing more code onto something until it works, you’re probably writing code for the computer, in which case, we can assume it will not be something a human being finds pleasure in reading.

The more I learn about what makes a project successful, the more I believe we should be **writing code for humans over computers**. Once you’re proficient with a programming language and no longer fighting with the syntax or how to accomplish common things, figuring out how to make your designs optimized for *humans* comes next.

“If you want your code to be easy to write, make it easy to read” — Robert C. Martin

Isn’t it fascinating that most of the opinions about clean code from the **experts** have more to do with humans than they do about the code? Call it *chance*, but these folks have spent decades doing this stuff.

Consider the fact that you and I spend around ~60% of our time *reading* code vs. *writing* it (I made that number up). No one just lets loose and hammers at their keyboard like a God (I wish). That’s simply not the way it is. We are reading (and writing) the book of our company’s solution space, so take your time and write it well.

How does unclean code get written?

Unclean code doesn’t just sneak up on you like seasonal affective disorder. Here are ways it can manifest.

When we try to go fast with arbitrary deadlines and really need more time. Have you ever done *anything* well being rushed? When we skip out on planning and estimation or find ourselves needing more time for whatever reason, instead of pushing back, we rush to tie up loose ends. This usually results in suboptimal, and unclean, code.

When we don’t write tests. You might curse me out for this one, but I do believe that code without tests is unclean. Code without tests is code that cannot be refactored or changed

safely. Since more time is spent on maintenance and improvement than on initial development, and change is always a constant factor, we need to feel like we can change code safely. Code unable to be changed safely induces unnecessary stress and anxiety. It also introduces risk. Code without tests is bad for your sanity, and it's certainly not clean.

When we don't care about the domain. I can admit to this one. Sometimes you just really *don't give a s**t* about the project you're working on. It's less likely the next person will be able to maintain the code if we don't care about the domain, and the names don't reflect the business. Language is a huge part of software development. Not only might names be bad, but code ends up in the wrong place, abstractions get missed, architectural boundaries are illegally crossed, and things inevitably become a mess.

When we don't care about the longevity of a project. I get it. To some, programming is just something you do for a paycheck. Maybe you got pinched throwing together a website for your aunt. Let's get it done, and get the hell out, right? That's a very irresponsible way to look at your job.

When we don't care about craftsmanship. When we don't care about craftsmanship, any solution will do. If the code works, it's good enough. When we don't care about craftsmanship, a *mess that works* is acceptable. When we don't care about craftsmanship, we won't challenge our peers' code with constructive criticism. Potential conflict *is* scary, and people that don't care about craftsmanship won't figure out ways to have productive discussions. Apathetic developers will tolerate bad code to stay comfortable.

When we're too scared to have design discussions with other developers. Social anxiety is all too common in the programming industry. It's really unfortunate, because some of the best design decisions I've ever made came from more than one developer's point of view. Through tools like Slack, GitHub, and Jira, we can build collaboration into the process of developing software- but it takes will-power.

When we try to get clever with our code: I used to use the `~` (tilde) operator in JavaScript because it made me feel really cool and clever. What *didn't* make me feel cool and clever was watching my teammates avoid the code I wrote because they couldn't understand it. Writing clean code often means avoiding *finesse*, even when it's fun to do so.

When our design skills are poor: There are two groups of people that we're writing software for: our users, and ourselves (you, your teammates, future maintainers). As we've said several times throughout the duration of this book so far (and we're only in Chapter 3), making code work *once* isn't enough— the initial completion of a project is only the starting point. Design dictates if the project can endure or not: and design relies on a basic understanding of human psychology, empathy, and how to make things discoverable and understandable. They ain't teachin' that in school, bub.

When systemic issues in the way we educate developers exists. We live in a society that prefers to take a *reactive* approach to dealing with problems rather than *preventative* one. Like most politics, a lasting change to this problem is much hard and not straightforward to fix in the real-world. If developers are making messes in production codebases, potential ways to fix it are to: a) invest in better onboarding and training, b) educate developers with practical coding skills in post-secondary education and bootcamps, c) standardize practical coding skills instead of new technologies.

These are just *some* of the ways that unclean code gets written. Long story short, it's about people.

Two laws of software development maintenance

Sun Microsystems, the company behind Java, has the following to say about maintenance of a software project:

- 40%–80% of the lifetime cost of a piece of software **goes to maintenance**.
- **Hardly any software is maintained for its whole life by the original author.**

Wow.

If that's true, then the importance of writing clean code is *paramount*.

Why it's hard to learn clean code

Reason 1 — Humans are complex

Clean code has been hard to describe, learn, and teach because it aims to bring structure around **writing code for humans**, ****and humans are complex.

There's a lot that we still don't understand about how we — human beings, work.

The human brain is still very much a big mystery to us. For example, we don't understand jack about how the brain processes **visual information** (AllenInstitute.org).

"Imagine you have to infer who loves whom, who backstabs whom, and what's going on from just a 1200x1200 pixel image". Humans are so complex that we *still* do not fully understand the common principles at play in which our neurons come up with these answers.

Reason 2 — It's hard to deconstruct human psychology

While we can apply *reductionism* to computing and engineering, trying to do the same with humans is hard (and typically *wrong*). This is likely why formal education has chosen to avoid attempting to teach us the following topics:

- How to name things
- How to organize code
- How to decide on and enforce coding conventions
- How to detect code smells
- How to avoid anti-patterns

The answer to each of these questions is deeply rooted in our understanding of human psychology, not something that can easily be tacked into a college curriculum.

Designers learn the basics of human psychology, but traditionally, *software developers* don't. I believe we should. I believe software developers are also software *designers*.

Reason 3 — Trade skills are acquired through mentorship

Formal education isn't bad.

They *excel* at teaching us scientific and theoretical aspects of computer science; but that's because they consider programming an **engineering discipline**, not a **craft**.

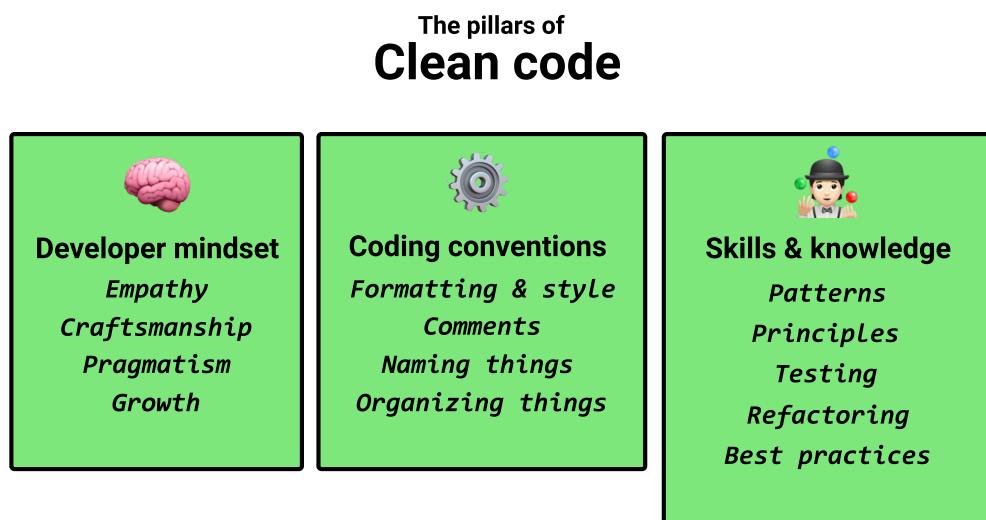
The practical *tricks of the trade*, things you'll spend 90% of your time doing in the real-world, like testing, refactoring, and *design* are passed down to us from mentors and more experienced developers, usually in an apprenticeship, co-op, or as a junior developer.

The three pillars of clean code

Developer mindset, coding conventions, and skills & knowledge

The Pillars of Clean Code is a framework I developed to illuminate each component involved in learning to write clean code.

It breaks the entire topic of *clean code* into three parts:



■ Developer mindset

Design thinking, empowering humans, adding value, rejecting perfection, and growth from failure.

■ Coding conventions

Developing your own set of principled coding conventions to produce the highest quality software possible using industry standards, tools, approaches, and methods.

II Theory & knowledge

Theoretical knowledge about software design and architecture which influences our preferred coding conventions.

Historically, conversations about clean code were either exclusive to **one** of these three pillars or **too vague** to express the intricacies of how they intertwine.

Notes

Todo: Refactor this bit

How I recommend learning clean code

- It starts with your mindset
 - If we want to optimize our ability to do a good job, I think it starts from inside your head, and before you write a single line of code.
 - How do you feel about writing code? Is it strictly for money? Do you *care* about the quality of your work? Are you *open* to improvement?
 - We ask some questions, do a little bit of soul searching, and I give you some strategies to ensure the quality of your work.
 - Pragmatism
- Then, we work towards building our own set of coding conventions.
 - Learn all the important stuff:
 - * Testing, refactoring, naming things, organizing a project, designing for change, etc.
 - Question everything
 - * I want you to question every single one of these principles, because if you're just doing it because I told you to, you're doing it wrong.
 - I want you to be **principled** about the way you work, and **pragmatic enough** to know when to break your principles.
 - This is what's different. With every coding convention we take a look at, I'll show you my principled way of arriving at it.
 - It will be 90% a combination of other essential software design principles and patterns, and 10% my own opinion.
 - If you don't know what to do in a situation and you have to reinvent the wheel, there's a chance you just don't know enough yet.
 - Learn new skills and acquire knowledge about **design principles, patterns, approaches** that will **refine and reconstruct** your coding conventions.

How do we learn this?

- How developers used to learn clean code
 - Reading Uncle Bob's book
 - * Problems with this?
 - His book is a collection of coding conventions, but I don't want you to just follow coding conventions. I want you to build your own set of conventions guided by principles, question everything, and continue to either **reconstruct** or **refine** what it is you do everyday.

- * How are we going to do this?
 - A framework for improvement.
- The Pillars of Clean Code
 - In summary, I understand how vast this topic is.
 - We'll get to it.

Section One - Developer mindset

Summary

Design thinking, empowering humans, adding value, rejecting perfection, and growth from failure.

Pragmatism and Craftsmanship

Growth

Empathy

Pragmatism and Craftsmanship

The yin and yang of software development — perfection and reality.

As software developer, we're challenged in really unique ways.

We have to deal with gathering requirements, scoping projects, keeping them in scope, meeting deadlines, pushing back on said deadlines, and — it's a lot. To be honest, it seems to have a lot more in common with trades like construction or plumbing than it does with scientific practices such as plant biology.

Getting into this industry, they never told me so much of our work is dealing with people.

I had this image that software developers spent the majority of their time sitting on their computer cranking out code by themselves without ever talking to anyone.

I was way off.

I thought programming was like, some Norman Osborne stuff, super science-y and what-not. It's not. It's a trade. And a community of developers who have been doing it way longer than you and I seem to agree.

Programming as a trade instead of an engineering practice

If programming was solely an engineering practice, we'd be way more concerned with *science* and *experimentation* than people.

If engineering were at the foreground, there would be a lot more “precision, predictability, measurements, risk mitigation, well-defined statistical analysis, and mathematics” in our daily professional vocabulary (wiki).

Funny enough, for those of us who attended University, sometimes that's really the picture that first year computer science paints. In a conversation with Sam Julien from Autho, he explained that he initially dropped out of computer science because he was led to believe that the industry was all about calculus, statistics, and mathematical proofs.

Of course, if you work for a company that builds 3D-rendering engines, that may be pretty accurate. But for the rest of us, as hired developers being paid to do web development, it's not uncommon to feel under-equipped with the actual tools of the *trade*.

How many of us had to learn *testing* on the job? Learn *refactoring* on the job? Learn *how to make estimates* on the job?

Most of us.

Similarities to the apprenticeship model originating from medieval Europe

The first time I heard the word, *journeyman* was when I picked up a copy of the 1999 book, "The Pragmatic Programmer: From Journeyman to Master", by Andy Hunt & Dave Thomas.

The word *journeyman* originates from medieval Europe. It means "a worker, skilled in a given building trade or craft, who has successfully completed an official apprenticeship qualification" (wiki). A *journeyman* is the same thing as an **apprentice**.

In an apprenticeship, there's a **master** and an **apprentice**.

The master demonstrates the correct way of completing a task, then the apprentice attempts to imitate the master's skills, and gets corrected for any mistakes. For trades like carpentry, steel-smithing, and baking, this is how new workers were taught the skills of the trade.



Le charpentier.

Der Zimmermann.

The carpenter.

The carpenter - By Anonymous artist - <http://www.digibib.tu-bs.de/?docid=ooooo286>,
Public Domain, <https://commons.wikimedia.org/w/index.php?curid=981584>

That apprenticeship model is still very similar to how developers advance skill levels today: from junior to mid-level, senior, to architect.

That book, *The Pragmatic Programmer* — it was the first to make this distinction. It described the “so-called tricks of the trade” that are picked up throughout the career of a software developer in the industry.

It distilled much of the practical knowledge and skills you’d learn from a senior developer at your first job: things like refactoring, testing, design, and keeping code clean.

The book was the first of its kind to lay this distinction, and it was also the first to suggest that these are the essential tools for a *journeyman* to pick up their quest to becoming a **master**. It described that, as a developer, there’s a level of *professionalism* to be followed.

Here are four examples from the book:

- 4. Don’t Live with Broken Windows. Fix bad designs, wrong decisions, and poor code when you see them.

- 5. Be a Catalyst for Change. You can't force change on people. Instead, show them how the future might be and help them participate in creating it.
- 6. Remember the Big Picture Don't get so engrossed in the details that you forget to check what's happening around you.
- 7. Make Quality a Requirements Issue Involve your users in determining the project's real quality requirements.

— The Pragmatic Programmer: Quick Reference Guide from The Pragmatic Programmer Book, from: *Hunt, Andrew; Thomas, David. "The Pragmatic Programmer: From Journeyman to Master."*

The Pragmatic Programmer is **still a very relevant book** even though it was released around 20 years ago.

Like knights in medieval Europe had Honour Codes, software developers needed one too. Two years later, it was developed.

The Agile manifesto — a code of honour for software developers

In 2001, seventeen professional and influential developers (including Kent Beck, Ward Cunningham, Uncle Bob, and Martin Fowler) came together to discuss the various software development methodologies they were working on at the time.

At the end of this meeting, they founded the “Manifesto for Agile Software Development”.

Based on each of these seventeen developers' experiences of developing software and helping companies, they signed and declared the **Agile Manifesto**. It is as follows:

- **Individuals and interactions over processes and tools**
- **Working software over comprehensive documentation**
- **Customer collaboration over contract negotiation**
- **Responding to change over following a plan**

The Software Craftsman's Manifesto — extending and challenging the Agile Manifesto

In 2008, Uncle Bob gave a keynote speech at Agile 2008. In his talk, he proposed an amendment to the Agile Manifesto called “*Software Craftsmanship over Crap*”.

While it was never added to the original Agile Manifesto as a *fifth* item, a new set of values were developed. A new set that would *extend* and *challenge* the original manifesto.

It was assembled by 24 developers including Uncle Bob, and it was titled the **Software Craftsman's Manifesto**, with the sole purpose of fighting *crap code*.

It is as follows:

“As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

- **Not only working software**, but also well-crafted software

- **Not only responding to change**, but also steadily adding value
- Not only individuals and interactions, but also a community of professionals
- Not only customer collaboration, but also productive partnerships

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

© 2009, the undersigned.

This statement may be freely copied in any form, but only in its entirety through this notice"

"As of today, there are over [30,000] signatures on the Manifesto. [30,000] people are fighting against crap code. Those who have been fighting crap code now know that they are not alone in their fight. Those who write crap code now know that there are [30,000] people fighting against them."

—Micah Martin

You can actually sign it here if you like.

Pragmatism

The first two,

- **Not only working software**, but also well-crafted software
- **Not only responding to change**, but also steadily adding value

...can be really hard. But it's a fight worth fighting for. The fight against crap code is unforgiving. Even though we *intend* to do well, *reality* gets in the way (deadlines, bad legacy code, life's mishaps, etc).

Sometimes the **best practice** or **correct solution** is nearly impossible to accomplish given the constraints.

In fact, a lot of people don't like to listen to Uncle Bob because he has very strong opinions about craftsmanship.

Uncle Bob says that you should strive for 100% test coverage. He also says that not writing unit tests is *unprofessional*.

No matter your stance on the topic, you have to respect the opinion. It comes from a craftsman: someone who is tired of crap code.

To address reality though, we need to be **pragmatic**.

Pragmatism is philosophy for approaching problems and their solutions. It means thinking beyond the immediate problem, understanding the larger context, then making *intelligent compromise* and *informed decisions*."

Pragmatic programmers understand the trade-offs involved, seek to write *Good-Enough Software*, and reject *unrealistic perfection*.

Programming is a trade

In essence, yes — programming is a trade.

We have a code of honor.

We learn by listening to and working with developers more skilled than us. If you're a senior developer, do not underestimate the impact you're making on the development of a junior developer.

If you're a junior developer, ask questions, be open-minded, and work on handling constructive criticism.

To move up a tier, you need to have a **growth mindset**.

Other trades are normalized

Growth

Not only are people with a growth mindset not discouraged by failure, but they don't actually see themselves as failing in those situations — they see themselves as learning.

— Carol S. Dweck, Ph. D, author of Mindset: The New Psychology of Success

While working as a Junior Software Consultant, I remember feeling visibly annoyed this one occasion I submitted a PR for a feature on an Angular project. The team lead/senior developer left two or three comments and suggestions for revisions on just about every file I submitted to be merged.

I muttered, “Man... Isn’t it good enough? I spent so long on this. What the hell...”

Looking back on it, I understand where he was coming from.

In that project, the entire architecture, designed by both him and another senior developer, was constructed to solve the *hard problems* using Angular Observables.

Their approach to architecture was elegant. Elegant, but hard to contribute to. You could say the developer experience was poor.

I wasn’t proficient with Observables, and after two hours trying to learn them, I gave up and used Promises instead.

In the PR, most of the comments from the team lead were detailed examples of how to replace my Promises with Observable. Today, I applaud his effort to mentor me. At the time, I didn’t care- I just wanted to use Promises.

Looking back, and after a conversation with him and the other developer, I understood why I was wrong.

As the team lead, he had to act as something of a code groundskeeper. He explained the importance of a consistent architecture and using the same coding conventions throughout the entirety of the project. Even though the approach had a learning curve, he introduced me to a variety of challenges that his functional approach was well-equipped to solve, and how to push side-effects to the edges.

This was my introduction to functional programming.

Since then, I decided I'd really listen and try to understand others' points of view. Even if you don't agree, you'd be surprised how much there is to gain by hearing someone out.

Two mindsets

Our behavior and relationship with success and failure (both professionally and personally), has to do with our mindset.

Studies have shown that in the face of challenge and potential failure, people have either a *fixed mindset* or a *growth mindset*.

Fixed mindset

People with a fixed mindset assume they can't improve. They believe that their personality, character, and intelligence, are *static* and *circumstantial*.

The fixed mindset assumes a deterministic view of the world: it's just the way it is.

Growth mindset

People with a growth mindset believe that they can improve by learning. Who they are today, is not who they have to be tomorrow. They believe that their personality, character, and intelligence are all *fluid* and capable of being *changed* and *enhanced*.

The growth mindset assumes that nothing dictates who we are except us — it's free will: and we can accomplish spectacular things through learning.

You need to have a growth mindset as a software developer

It really does go without saying, but as a software developer — you need to have a **growth mindset**.

Fixed mindset as a developer

People with a fixed mindset tend to be scared of challenge. Perhaps they only work on the front-end because to them, the back-end seems challenging. They work on the same projects, use the same technologies, and attempt to find ways to do the exact same work. They also dread having their code reviewed.

Fixed mindset folk are **afraid of failure**.

They're afraid of failure because failure means people could figure out who they *really are*. And since who they are can never be improved (in their minds), having their faults on public display can be a very heavy thing to deal with. Fixed mindset developers make careful efforts to only place themselves in situations where they can look smart. These are the developers you see leaving mean and pedantic comments on StackOverflow.

Fixed mindset folk:

- Avoid challenging situations
- Enter discussions to express themselves, not to potentially change their minds

- Develop code in a vacuum, away from others, in an attempting to avoid receiving feedback
- Aren't great listeners
- View negative feedback as a personal attack on their character
- Give up easily
- Lack emotional intelligence and empathy skills
- Don't put in more effort — since their abilities are static, more effort is a waste of time

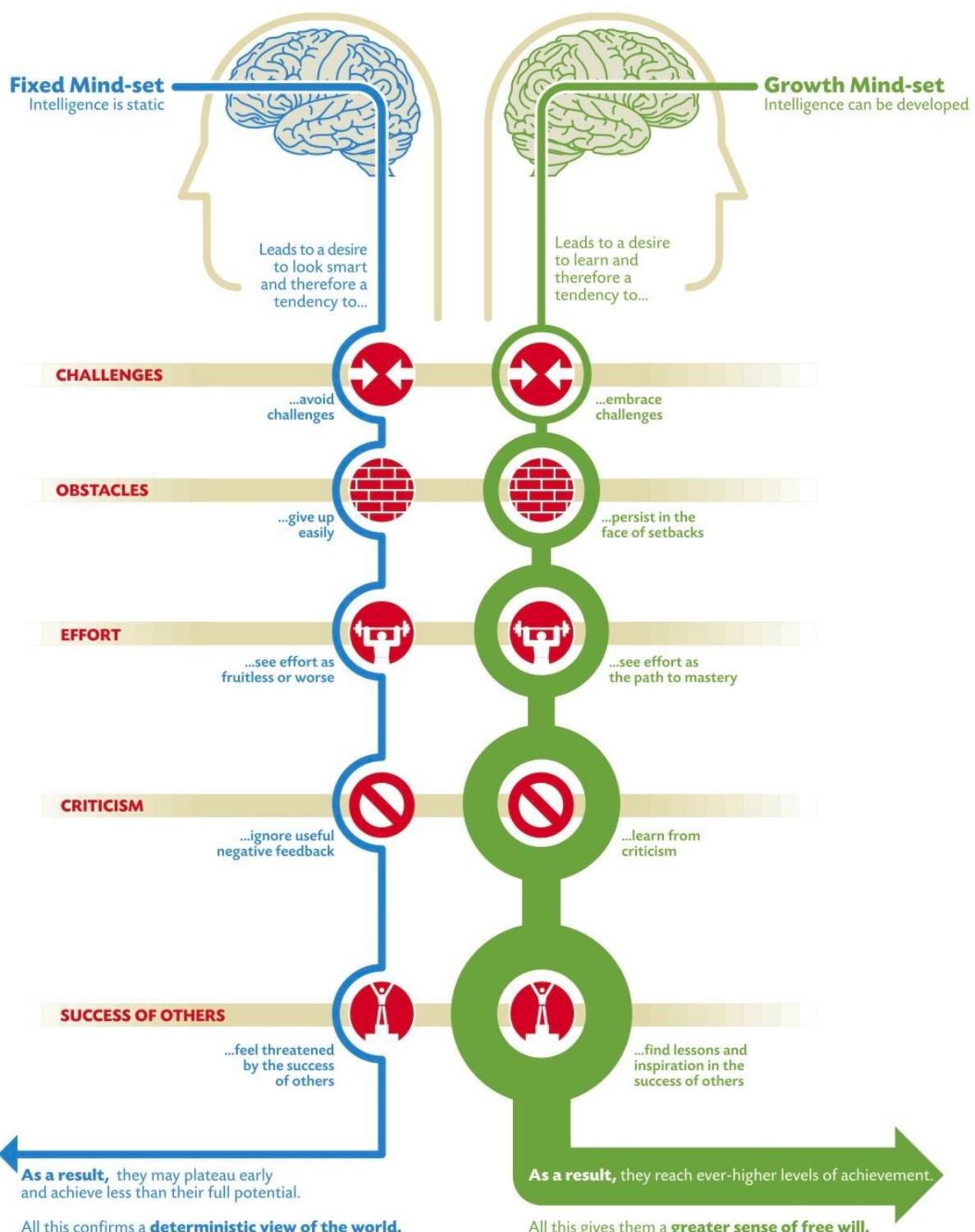
Growth mindset as a developer

Since people with a growth mindset thrive on failure, they constantly challenge themselves.

For growth mindset folk, **failure is a “springboard” for growth** and stretching our capabilities. It's an opportunity to discover what they don't know, and then expand to fill in those blanks.

Growth mindset folk:

- Put themselves in situations where they have room to grow
- Involve others when they develop code
- Aren't afraid of looking silly or not knowing the answers up-front (they believe they can *find* the answers)
- Are great listeners — and always challenge their own way of solving a problem using someone else's POV
- Treat negative criticism as a way to learn how to grow
- Don't give up easily
- See putting in effort as the way to accomplish more



GRAPHIC BY NIGEL HOLMES

Growth Mindset & Fixed Mindset by Carol S. Dweck, Ph.D <https://fs.blog/wp-content/uploads/2015/02/Carol-Dweck-Two-Mindsets.jpg>

The capacity for happiness and fulfillment you can have with a growth mindset is the biggest reason why I advocate for trying to adopt one. Honestly - life becomes more fun. It's fun to look at the world as a set of systems that we get the opportunity to master, starting from nothing. Maybe that explains why I've played Fallout 3 all the way through at least five times.

I've personally found that we learn the most from experiences where we try things out ourselves, allow ourselves to be wrong or fail, **and then seek out the answers from those more experienced than us.**

In Pragmatism & Craftsmanship, we learned that the apprenticeship model is exactly how we move towards mastery. For that to work, we need a growth mindset.

There are experienced developers out there. They're at your workplace. They're on Twitter. They've written books. They've documented the Design Patterns they've seen. They've helped discover Design Principles. They've uncovered the unwavering fundamentals truths of software development over the last 40 years.

The question: are you ready to listen to what they have to say?

Empathy

You can't design without empathy" — Jeff Weber, co-designer of the legendary Herman Miller "Aeron" chair

Developer Experience: Give humans super powers

Human-Centered Design: All developers should learn how to design for humans

Testing code against humans to determine code cleanliness

Resistance to change

Don't be a jerk

Developer Experience: Give humans super powers

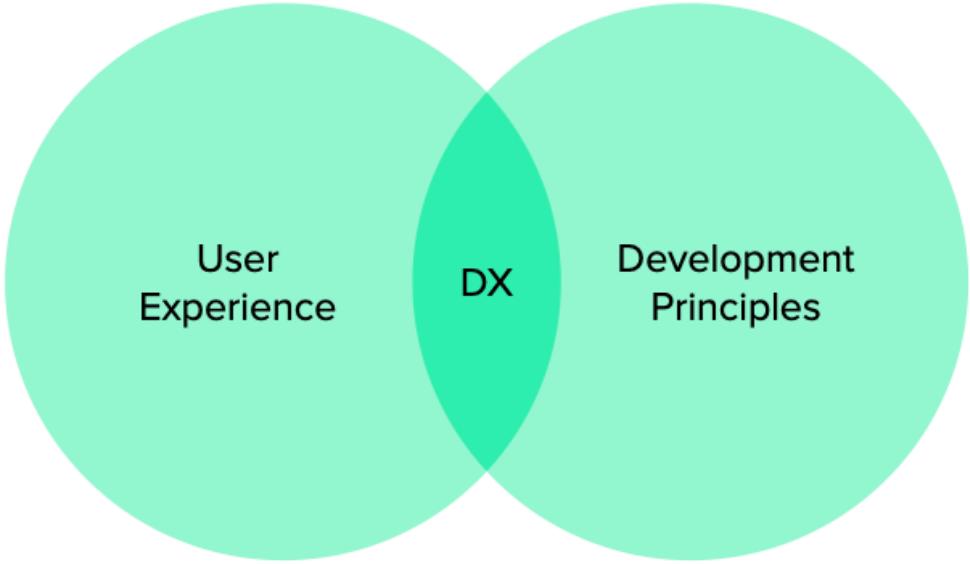
What if I told you it was possible to write code in a way that it gave others super powers?

Being empathetic **as a developer** means **striving to empower your team, yourself, and future maintainers** to accomplish tasks.

Optimize the **developer experience** by making it frictionless to understand, add, and change features.

Developer experience (DX)

Where UX (user experience) design is about designing applications for end users, **DX (developer experience) design is about designing APIs for developers.**



— via “What is Developer Experience?”

Experience is everything.

Experience is the reason why we choose one library or framework that solves the *exact same problems* over another.

It's also what makes us say, “this is such a nice codebase, I love working in it” or “this legacy code is terrible — we should just rewrite it”.

“Experience is critical, for it determines how fondly people remember their interactions. Was the overall experience positive, or was it frustrating and confusing? When our home technology behaves in an uninterpretable fashion we can become confused, frustrated, and even angry—all strong negative emotions.

When there is *understanding* it can lead to a feeling of control, of mastery, and of satisfaction or even pride—all strong positive emotions.

Cognition and emotion are tightly intertwined, which means that the designers must design with both in mind.”

— Norman, Don. “The Design of Everyday Things: Revised and Expanded Edition.”

It's important to lean on knowledge, coding conventions, and best practices to produce structurally high quality code, but it's also important to do that in a way that keeps the developer experience high.

Empathy

Empathy is the ability to understand the feelings of another person. It's also the ability to **experience those same feelings**

Empathy towards your teammates & future maintainers

An empathetic developer will keep in mind how their design decisions affect the other developers on the team, and how it may affect potential maintainers. If they've implemented a solution that requires a lot of specific knowledge, they write documentation. If they've chosen a unconventional algorithm or architecture for a particular problem, they leave a comment to **explain why**.

Empathetic developers are able to tell when something may reduce the developer experience for a future maintainer or teammate. If a teammate is struggling to change some code, or something has been discovered to be confusing, empathetic developers will look for ways to make the code more intentional and understandable.

Empathetic about the project

When we're empathetic about the project, we do what's right to ensure that the project lives a long and healthy life. We learn the business, understand the requirements, and push back and ask for more time when necessary (since we know how expensive it is to add new code on top of bad code). We face social anxieties and conduct constructive conversations about code before it gets merged into production.

Aim to build awesome developers (you and your teammates)

How do you know when your code is empathetic?

There's this book I love by Kathy Sierra. It's called "Badass: Making Users Awesome". The main idea is that when you're building products, it's great to build products that *you* think are awesome, but a better goal is to build an awesome *user*.

... I didn't get it right away either.

Here's what she means.

People don't buy products because the *product* is awesome. People buy products because *they* want to be awesome. *People* want to be better. For example, people buy bigger and better houses because they have an intrinsic drive to feel important or successful. Some buy designer clothes because it makes them feel cool or sexy.

Perhaps you bought this book because you are fed up with writing untestable, inflexible, and generally *not great* code. Maybe you bought it because you felt like you were completely overwhelmed about the scope of software development.

The goal of this book is to **make you feel like a professional software developer**. I want you to feel comfortable with what there is to know, I want you to understand the importance of tests, be able to participate in conversations about design, and make good decisions. I want you to have some strong ideas of professionalism, know when you're writing bad code and how to fix it, and also I want you to have a good handle on the basics of Domain-Driven Design.

I'm investing in making **you** better.

In the same vein, you should write code that will make the **maintainers feel like they're the best coders on Earth**.

That's what writing clean code is all about.

Invest in **making your teammates and future maintainers feel like great coders** with how quickly they're able to understand, locate, and change code without breaking anything. If you're not an empathetic thinker, that's OK — it's a skill that can be learned.

To get a better understanding, consider the following scenarios. In each one, we decouple **the task that needs to get done** from **how we want to feel doing those tasks**.

Task vs. feeling

Scenario	Task	What we want to feel
Using Mailchimp for email newsletters	Send emails to our audience	Like we're connecting with our users, building relationships, and we're good at email marketing.
Landscaping our front yard and planting trees	Buy tools, equipment, plants and get to work	We want to feel like we live in a very nice home, and we want our neighbors to think that too.
Working on a software project	Produce working code and meet deadlines	<i>Like we're competent problem solvers that can implement features quickly, prevent bugs and satisfy users.</i>

What's the difference between Mailchimp and Mailjet? Both of them can accomplish what we need to do, but one of them (arguably) does a better job at making us feel like we're connecting with our audiences.

Remember, you want whoever touches the code next to feel like a God. You want them to be impressed at themselves for how fast they were able to ship that feature, change those two lines of code, or contribute an additional condition in here.

“Paint, not the thing, but the effect which it produces.” — Stephane Mallarme

So here's the rule to print out and put in your workstation:

Code that *empowers humans* is best.

Boring code can be very exhilarating in a strange way. Boring code:

- is consistent
- is not clever
- is easily understandable
- makes locating features easy
- adheres to Principle of Least Surprise (things do what I assume they would)
- takes a minimal amount of effort to change
- uses the right tool (paradigm, language) for the job

We call it boring because it's simple, not exciting, and doesn't surprise us in the fun way a surprise party might. This is great. It leaves room for us to get excited at **other stuff**, like:

Being able to consistently change code, add new features, spend less time fixing bugs, and completing projects.

That's **peak developer experience**.

Though writing *boring code* isn't easy.

Existential question: Could what appears to be *boring code* at the surface, truly actually be the **real clever code**?

Human-Centered Design: All developers should learn how to design for humans

If clean code is, in part, about optimizing code for humans, then software developers are also software **designers**.

That means all developers are in need of the same practical (human-centered) design skills as **conventional** designers.

Design is hard.

You know that.

It's hard for a lot of reasons. Here's one.

Design is a push and pull between **reality** and **what the user wants**.

Design = balancing priorities

Design is a push/pull of priorities.

Most of the time, when we write code, we're writing it for both **a) the end user**, and **b) future developers/maintainers**. Creating something that meets the standards of *both* can be a challenge.

This extends beyond design in programming.

In the real world, office furniture manufacturers need to produce items in a cost efficient way. From the consumer side, good quality is a must. The battling attributes are: cost vs. quality. They contend with each other directly.

Other examples:

- Note-taking = context/compression.
- Office furniture = cost/quality.
- Security system = security/ease of configuration.

We face similar challenges in software design.

In software design, we want to produce something that checks off as much as we can from the Software Quality diagram. Adding more structure, constraints, robustness, and code to accommodate for those capabilities can **increase the reliability, security, efficiency, and size of the software** while applying the *inverse effect* of decreasing the **maintainability**

& developer experience for humans. Too much code, complexity, and structure has the potential to be hard to work with.

The push/pull for software design is software quality vs. developer experience.

Generally speaking, we know the answers for how to make code more **reliable, secure, efficient, and maintainable**. We learn testing and refactoring. We study algorithms, design patterns, and principles. The tricks of the trade.

But how do we keep the *developer experience* high? How do we make this a codebase that people love to work in? This takes another set of skills.

You are always building an API

Perhaps you've clued into this, but **you're always writing an API**.

An API is more than what gets developed and published to the public for consumption like SoundCloud, Facebook, Twitter, or Google APIs.

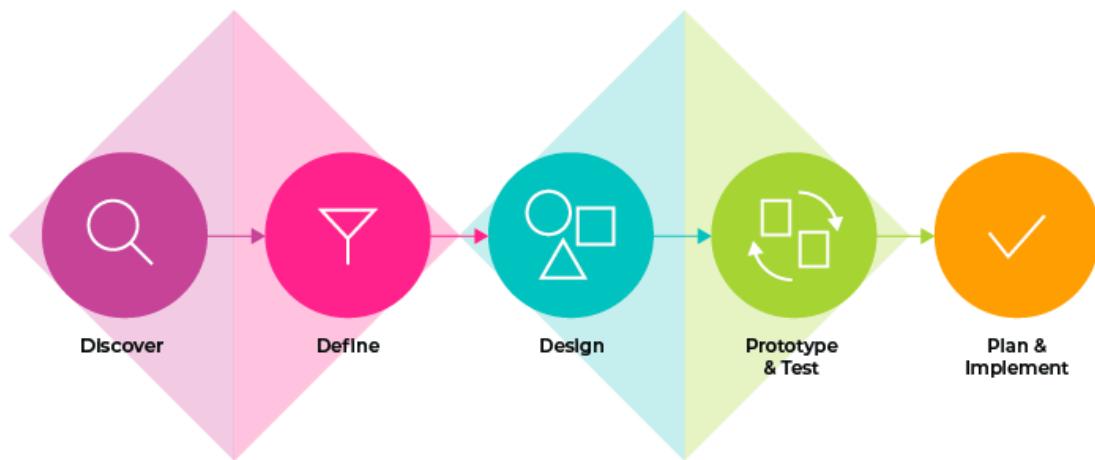
Each and every line of code we write is contributing towards the **internal API for maintainers to add, remove, or change features**.

We think about how to make things easy to be used by the public, why not do the same at home.

Introducing Human-Centered Design (HCD)

Human-centered design (HCD) is a design philosophy.

It places "humans"—the people who use a product or service (end users), **and those who take part in the experience being designed** (developers)—at the center of all activities.



Copyright 2019 | Outwifly Inc. | HCD Process

— Human Centered Design approach (from Outwifly Inc)

Designers that study HCD approach design in this manner:

First, discover the users' needs, behavior, characteristics, pain points and motivations. Use that knowledge to brainstorm, design, and test implementations.

The path isn't always linear — sometimes you go back and forth, but this is how to include **humans** at the forefront of design.

HCD can create more empathetic developers

HCD is formally studied by UX designers — those who actually design the products that we've been asked to write the code for.

I think that studying HCD can help software developers write better code.

Functional programming is a very powerful paradigm, but it has the capacity to get so *in the weeds* that no one wants to use it. Often, code written using it can become unintelligible.

HCD gives developers the framework to think about alternative solutions to solving problems in a usable way.

HCD matters for better developer tools

We talked about when **developers** are the **secondary users** for a product (where end-users are the primary), but what about when **developers** are the **primary users**? *Developer tools* companies put the experience of developers at the forefront, because their customers *are* the developers.

What do Apollo, React, Twilio, and AWS have in common? They build tools for developers, and they have to be great.

Common developer needs

We said that HCD was about understanding the users' needs and behaviors first, right?

Well, we're in a unique design situation here. Normally we'd have to get out there and actually interview the users — luckily for us, *we are* the users. This makes the process a lot easier.

Here are a few needs that **all developers share in common when writing code on a project**:

- I need to understand how code is organized
- I need to figure out how to add a new feature and where it belongs
- I need to locate a specific feature within the code so that I can change it based on a new specification
- I need to see if the code I just wrote works or not
- I need to learn how to run the tests
- I need this code to be **discoverable**

Think about how grateful you'd feel if the original author of a codebase you've been hired to work on knew you had these needs, and made decisions to enable you to get them met.

Discoverability

Discoverability is arguably the most important attribute of good design.

- Discovering **what** this code does — and **why** it's necessary
- Discovering **how** it does it — and how fast, slow, complex, simple is it
- Discovering **what operations are possible** — and what operations are possible
- Discovering **where** something is — and how that fits into the larger context

Principles of design for humans

There are six major principles of design.

The first five aim to optimize **discoverability**.

The last aims to provide **true understanding**

- 1 - Affordances
- 2 - Signifiers
- 3 - Constraints
- 4 - Mappings
- 5 - Feedback
- 6 - Conceptual models

When we encounter things that we've never seen before, we utilize a combination of knowledge that we've acquired from the world, and knowledge that we've saved in our heads

Summary on Human Centered Design

Use the principles of design when designing your software modules

The principles of design aren't just for UX designers. They were recorded to help communicate how human beings **discover the capabilities of things they've yet to come across**.

Since designing code for maintainers is a secondary task, study these principles and keep them in the back of your mind when writing code.

Learn more about Human-Centered Design

Human-Centered Design is a topic worthy of more research. If you're intrigued, I suggest you spend some time learning more about design. I recommend the book, "The Design of Everyday Things, by Donald Norman".

Testing code against humans to determine code cleanliness

How to tell if your code is clean

Ask a friend.

Really. When I studied computer networking, my professor would tell me about some of the past projects he worked on. When he'd open the networking closet and look through the router configs, he often told me of times he was exposed to *barbarian-ism*. At least, that's how he described the people who wrote the code- he would menacingly say, "*damn those barbarians*".

Being able to determine if your code is clean is more or less something that **comes from experience working in both very clean codebases and unclean codebases of substantial**

size. Project size is relevant, because if you only get a chance to build small applications or proof-of-concepts, and don't have support any long-time projects, you may not get a chance to experience the difference between truly *clean* and cripplingly *unclean* code. It's either you stay around long enough and get the opportunity to get to see the codebase thrive or you stay around long enough to realize that it has problems. It's important to experience both of these. Work on a project long enough to get to the *maintenance* phase. You'll know if it's good or not based on how easily you can understand, maneuver, and change things.

If you're working on a personal and you really want to know if your code is clean, *ask another developer to interact with your code.*

Prototype & test is the second-last step in the HCD (Human-Centered Design) philosophy: we can determine our code cleanliness by observing just how well we **empower our fellow developer**. Try these questions.

Ask: What does my code do?

Test against **how quickly they're able to understand what it does**. Don't even tell them the problem domain. Allow them to look at the files, folders, classes, and variable names. If someone isn't able to tell what the domain you're working in, that could be a signal that that either your domain is incredibly abstract, or you've failed to understand and codify domain concepts. With this question, we're testing against:

- Readability
- Clarity
- Brevity & succinctness

We will discuss how to improve this by using:

- Good names (classes, variables, methods, folders, files)
- Encapsulation (simplify APIs)
- Intention revealing interfaces

Ask: Find the code that needs to be changed

Let's say they now understand the problem domain. How about changing a feature? Test against **how quickly they can find the code that needs to change** for a feature. We're testing against:

- Locatability, scannability, structure

Things that massively influence this are:

- Good names (classes, variables, methods, folders, files)
- Smaller files
- Good packaging (coupling of constructs involved in a feature)

Ask: Change this code without introducing bugs

Ramping it right up there, ask them to change the business logic for feature. **How safely can they change the code without breaking other features?** We're testing:

- Stability
- Flexibility

At this point, the concepts we care most deeply about are:

- Tests
- Coupling
- Dependency Inversion or decomposition of code
- Boundaries & separation of concerns
- *Type safety*

Try those three questions without providing guidance and observe how successful they are. This is like a time-machine test. It's more responsible to watch your friend get disgruntled in person today and fix the problems than it is to put it on someone else 4 years after you are long gone from the project.

Resistance to change

Don't be a jerk

With great power comes great responsibility. The more you learn about software design, architecture, and how to write testable and flexible code, the more you'll be able to contribute to meaningful discussions with your teammates.

The golden rule is: **don't be a jerk.**

People don't care how brilliant you are—if you're not a pleasure to work with, people won't want to listen to your ideas.

Watch your tone

Here's a short list of phrases and words to try to stay away from in technical conversations.

- Avoid saying "just" — what may appear to "just" be easy to you, may actually be a challenging task to someone else. It's possible that someone may feel resentful towards you if a) the recommendation you made came across patronizing, b) they weren't able to accomplish the recommendation you gave them. In the case of b), your making the task seem trivial while being something they couldn't accomplish, could hurt their self-esteem. People don't like others that hurt their self-esteem. No one likes being told they're inadequate.
- Avoid saying "simple" — refrain from this for the same reasons listed above.
- Avoid saying "you should" too much — try saying "we could" instead. Y'all are on the same team.

Empathy is hard.

Honestly, humans are harder than anything discussed in this book.

My advice? Read "How to Win Friends and Influence People" semi-quarterly, ask your friends what you can work on, and stay humble.

Section Two - Clean coding conventions

Summary

Developing your own set of principled coding conventions to produce the highest quality software possible using industry standards, tools, approaches, and methods.

About this section

There are hundreds, maybe even thousands of coding conventions to learn. In this section, we'll cover a curated set of conventions. I've chosen conventions uncovered by formal education, yet, sorely in need of guidance for our daily programming work.

My aim is that these conventions are useful regardless of the team, language, paradigm, or architecture of your project.

While the conventions do apply to just about any language, we'll call out the TypeScript/JavaScript specific conventions as they occur.

Conventions we'll cover

Comments

Naming things

Organizing things

Errors and exception handling

Testing

Refactoring

Formatting & style

Comments

I used to love comments. I thought that code was incomplete if each method didn't have a comment. I thought that commenting my code made it more readable and improved the overall quality of the code.

The discussion about when to comment your code can get pretty heated. Some swear by comments. Some say it's unprofessional not to comment. Some think it clutters your code.

Here's my methodology.

Code explains what and how, comments explain why

When we use English to write declarative code that uses good names, we establish the *what*.

What is this code doing? Ah, this `createMusicRecommendations` method uses my `listeningHistory`, the artists I follow, and my playlists to create `musicRecommendations`.

```
export class SpotifyService {  
  
    public createMusicRecommendations (   
        history: ListeningHistory,  
        artists: Artists,  
        playLists: Playlists  
    ): Promise<MusicRecommendations> {  
        ...  
    }  
}
```

At a very high-level, we should be able to deduce from the method name, parameters, and as much declarative code as possible inside of a method or function body- *what* the code does.

Without descending a layer deeper into subsequent method calls and lower-level details, the high-level code **excels at explaining the what**.

Lower-level code describes the *how*. With each descending layer of abstraction, the code should continue to explain that *how*.

If you can get all the way to the bottom without feeling like comments are necessary because the code is that clear and readable, applaud yourself.

However, it's possible (and likely) that you'll:

- Encounter something that can't easily be refactored to be simpler.
- Require the use of an algorithm or implementation that is **fundamentally more complex** (absolute complexity), usually for performance or optimization reasons.

At this point, it's a good idea to write a comment; not to describe the *what* or *how*, but to explain *why*.

```
/* You might be wondering why we're using a Splay Tree. We've  
discovered that a binary search tree gets very slow once we  
have over 5000 entries. We're using Splay Tree because  
everytime an entry is accessed, it pushes it closer towards the  
top of the tree, making subsequent retrievals more efficient. */
```

Comments shouldn't explain what the code is doing. Make the code explain that.

Assume you have a one-liner that is necessary, but complex.

In JavaScript, this is how you *pad a two-digit number with zeros*.

```
num = ('0'+num).substr(-2);
```

This line converts a number, such as 1, into a zero-padded string like "01".

At the moment, this code answers **none of the following questions**.

- What: What does this do?
- How: How does it do it? (**relies on first knowing what**)
- Why: Why is it done this way?

Our first refactoring solves the question of **what** and **how**.

```
function padZeros (num: number | string): string {
    return ('0'+num).substr(-2)
}
...
num = padZeros(num);
```

That's great, but we still don't really know *why* we'd need to use this.

This is where we could use a well-placed comment.

```
/** 
 * We've found that when conditionally displaying numbers
 * with leading zeros, like seconds in the format of hh:mm:ss,
 * we can't rely on JavaScript's default string formatting abilities.
 *
 * Use this when the minutes or seconds are less than ten, and
 * you want to see :00 and not just :0.
 */

function padZeros (num: number | string): string {
    return ('0'+num).substr(-2)
}
...
num = padZeros(num);
```

Furthermore, since the *why* is clear, we now also know that this function probably belongs with other Text or Date utilities (see also, context).

```
export class TextUtils {

    /**
     * We've found that when conditionally display numbers
     * with leading zeros, like seconds in the format of hh:mm:ss,
     * we can't rely on JavaScript's default string formatting abilities.
     *
     * Use this when the minutes or seconds are less than ten, and
     * you want to see :00 and not just :0.
     */

    public static padZeros (num: number | string): string {
        return ('0'+num).substr(-2)
    }
}
...
num = TextUtils.padZeros(num);
```

Comments clutter code

Comments that answer *why* are the only time I can advocate for them being necessary. To make my argument in another illustration, look at the following *unclean* code with comments in it.

```
const _x: number = abs(x - deviceInfo.position.x) / scale;
let directionCode;
if (0 < _x & x != deviceInfo.position.x) {
  if (0 > x - deviceInfo.position.x) {
    directionCode = 0x04 /*left*/;
  } else if (0 < x - deviceInfo.position.x) {
    directionCode = 0x02 /*right*/;
  }
}
```

Comments don't necessarily mean readable code. In fact, those comments wedged in there don't help. It's lipstick on a pig. It doesn't make the unclean code any cleaner and the code is just as hard to understand.

Now watch how much more readable the code gets when we refactor it by stripping out the comments, declaring variables at the top and using the constant-variable naming convention.

```
const DIRECTIONCODE_RIGHT: number = 0x02;
const DIRECTIONCODE_LEFT: number = 0x04;
const DIRECTIONCODE_NONE: number = 0x00;
const oldX = deviceInfo.position.x;
const directionCode = (x > oldX) ? DIRECTIONCODE_RIGHT : (x < oldX) ? DIRECTIONCODE_LEFT
```

Comments often hurt more than it helps readability.

Turning comments into clear, explanatory, declarative code

Usually, it's possible for us to refactor comments into code.

```
// Check to see if buyer eligible for loan for property
  // if the buyers credit score is greater than the minimal approval
    // and the last job they were at, they were there for longer than the
      // minimum employment length
    // AND, their downpayment preferred value is the minimum downpayment
    // based on the type of property it is,
    // THEN, we will approve their loan
```

The names of the variables and methods can use the same words from our comments, and this is what makes it *declarative*.

```
// Check to see if buyer eligible for loan for property
if (
  buyer.creditScore >= MIN_APPROVAL_SCORE) &&
  (buyer.jobHistory
    .getLast()
    .getEmploymentLength() >= MIN_EMPLOYMENT_LENGTH) &&
```

```
(downpayment.value) >= getMinimumDownpayment(  
    property, downpaymentPercentage)  
)
```

This is the heart of what we do in the domain layer in a clean architecture using Domain-Driven Design, but it's possible in any code.

This code is a bit verbose, so we can now encapsulate that complexity within the correct objects, maintaining the language we initially used when we wrote the comments.

```
if (buyer.isEligibleForLoan(property, downpaymentPercentage))
```

Which would you rather read?

When possible, use a variable or a method/function to express **what** you would normally try to express as a comment.

Bad comments

Redundancy - Using comments to say something that is already adequately expressed with code.

```
export class UserService {  
  
    /**  
     * This method gets the user by user id.  
     */  
  
    getUserByUserId (userId: string): Promise<User> {  
        ...  
    }  
}
```

Log/Journal entries - Comments that describe when and what was changed, and who issued the change. This information should be tracked in source control instead of the source code itself.

```
/**  
 * 01-03-2008 - Tony Soprano - Added the ability to also work on strings.  
 * 01-02-2008 - D. Draper - Added this method to act as a utility that  
 * can be reused.  
 */
```

Commented out code - Code that is commented out should be deleted.

```
// function parseHandle (url, type) {  
//     switch (type) {  
//         case "twitter":  
//             url = url.replace("@", "");  
//             url = removeUpTo(url, 'twitter.com/');  
//             url = stripQueryParams(url);
```

```
//      return url;
//  default:
//      return url;
// }
// }
```

Closing brace comments

```
if (this.exists(userId)) {
    if (this.wasEmailNotificationSent(userId)) {
        ...
    } // end of inner if
} // end of if
```

When to write comments

To summarize, here's the way that comments should naturally occur in your code.

- Notice that code seems complex or that another human being may not understand it.
- First attempt to refactor the code.
- If it can't be easily refactored or further refactoring would make it even more complex, **leave a comment describing why it's implemented as such.**

Ultimately, as a rule of thumb...

Prefer refactoring code instead of writing comments

Another way for me to say this and maintain the same sentiment is to say:

Prefer refactoring (imperative) code (to declarative code) instead of writing comments

Demonstration

My mate, Swizec, has a different methodology to commenting than I do. Here's a tweet from him suggesting that senior developers leave comments "to explain why things work".



Swizec Teller @Swizec · May 29

Junior:

Your code must be self-documenting. If it needs a comment you're bad and you should feel bad 😞

Senior:

I'll forget why this works, better add a comment 🤦

```
        label: field.name
    })
    // roll them up
    .reduce((fields, field) => {
        if (!Array.isArray(fields)) {
            // initial case where accumulator is first value of array
            fields = [fields];
        }

        // currently last field in section
        const last = fields.slice(-1)[0],
              lastValue = this.values[last.id];

        if (this.values[field.id]) {
            // answered fields always show
            return [...fields, field]
        } else if (this.values[last.id] && lastValue.next_field_id === field.id) {
            // this is the next field depending on previous last value
            return [...fields, field]
        } else {
            // this field shouldn't be visible yet
            return fields
        }
    })
},
```

9

8

82

↑

As you've heard me say in this section already, if you're curious about **how the code works**, it's the responsibility of the code to explain that to you, not the comments.

I thought it would be fun to demonstrate a refactoring of this code.

Here's Swizec's code below, with the comments.

```
class Stuff {
    // the datamodel supports recursion but we haven't used that since 2017
    // for simplicity. The UI skips it for now. You can add it back here.
    get formConfig () {
        ...
        // Roll them up
        .reduce((fields, field) => {
            if (!Array.isArray(fields)) {
                field = [field]
            }

            // Currently last field in section
            const last = fields.slice(-1)[0],
                  lastValue = this.values[last.id];
```

```

        if (this.values[field.id]) {
            // answered fields always show
            return [...fields, field]
        } else if (this.values[last.id] && lastValue.next_field_id === field.id) {
            // this is the next field depending on the previous value
            return [...fields, field]
        } else {
            // this field shouldn't be visible yet
            return fields;
        }
    })
})
}
}

```

And here's the new version, with refactorings to make the code more declarative, un-reliant on comments.

```

class Form {

    private isSectionFirstInArray (fields): boolean {
        return !Array.isArray(fields);
    }

    private makeArray (field) {
        return [field]
    }

    private getLastValue (fields) {
        const lastField = fields.slice(-1)[0];
        return this.values[lastField.id]
    }

    ... // and so on

    get formConfig () {
        ...
        .reduce((fields, field) => {

            if (this.isSectionFirstInArray(fields)) {
                fields = this.makeArray(fields);
            }

            if (this.isAnsweredField(fields, field) || this.isNextField(fields, field)) {
                return [...fields, field]
            }
        })
    }
}

```

```

    // Shouldn't be visible yet - I left this, as it explains what cannot
    // reasonably be refactored to be said in words
    return fields;
}
}
}
}

```

Relationship to Human Centered Design

The relationship between comments & human centered design

The relationship between comments & human centered design

- **The argument I'm trying to make**

Comments should only explain the **why**. If you see a comment, it should never explain **what**. It should never need to explain **how**. If you need to explain these two things with a comment, it probably means that the design needs work.

In our industry's research into design, we've found that two of the most important characteristics of design are **discoverability** and **understanding**.

Discoverability = What are the possible actions? Where are they? How do I perform them?

Understanding = What does it mean? How is this supposed to be used? What do the controls mean?

In the book, *The Design of Everyday things*, the author tells a story of when his friend got trapped in the doorway of a post office in a European city.

More doors examples

- Two of the most important
 - A door with a *curved grippable* handle on it - what would you assume it can do?
 - A door with a *flat metallic surface* - what would you assume it can do?
 - A door with no handle, no flat, metallic surface, and no labels
 - A door with a *handle* that says **PUSH** - what would you assume it could do?



Naming things

Among the hardest problems in computer science.

It's remarkable that when we start out as developers, we learn about all different kinds of things like variables, methods, classes, and object-oriented programming, yet we almost never formally talk about **naming things**: one of the first and most challenging problems we face as programmers.

Naming is at the center of everything we do at work. From the words we type, to the conversations we hold about the *things* we've typed, good names have a lasting impact and it's an investment to name things well.

When names are *good*, it tells readers **what** our code does and **how**.

When names are *poor*, it subtracts from the understandability and overall quality of our code. This means that it takes new developers longer to ramp up to a new codebase, learn the glossary of new terms from the domain, and contribute in meaningful ways.

The seven principles of naming

This section teaches you how to approach the task of naming *programming concepts* using psychological science, engineering, and structure, rather than approaching it from a place of art, creativity, or novelty that we typically use to name companies, bands, and products for commercial reasons.

These principles were originally enumerated by Tom Benner from NamingThings.co.

From everything I've learned throughout my personal experiences working on good and bad codebases, reading Eric Evans' "Domain-Driven Design", Vaughn Vernon's "Implementing Domain-Driven Design", and Robert C. Martin's "Clean Code", the takeaways all appear to be congruent across these principles.

Of course, like most things in science, there's no such thing as **perfect**- and with naming things, you'll certainly find that to be true. However, using these principles, I believe there's a way to get pretty dang close to *good enough*.

The seven principles of naming

1 - **Consistency & uniqueness**

2 - Understandability

3 - **Specificity**

4 - **Brevity**

5 - **Searchability**

6 - **Pronounceability**

7 - **Austerity**

Summary

Names are important but it's important to remember that they can always be refactored during development. Don't get too hung up. Use these principles, have 'em in the back of your mind, pick a name, ask for feedback and improve 'em over time.

I - Consistency & uniqueness

Each concept should be represented by a single, unique name - via NamingThings.co

Consistency

The human brain, in order to learn, relies on consistency.

Skilled conference speakers take advantage of this fact by strategically using repetition in order to really drive home the main takeaways.

As humans, we rely a lot of patterns, past experiences, and conceptual models in order to equip ourselves with how we should act in future situations (remember what we discussed about design for humans).

For example, if you told me I had to hit a home run with a baseball bat, well... that's just not happening. Since I have pretty much zero previous baseball experience, no understanding of proper form, and no prediction on how things might go, it's a potentially uncomfortable situation to be in. To be asked to demonstrate something that takes a lot of skill and knowledge, and possessing none of it upfront is uncomfortable.

Jumping into a new codebase can feel *very* similar.

It can feel as if it might take a long time to begin to feel productive.

And while we do work in an industry containing widely accepted patterns around paradigms, styling, formatting, packaging, architecture, and coding conventions, it's very likely that you'll come across projects that throw all of that out the window.

Not only that, it's also likely that you'll start work on project that operates in a domain that you know nothing about. Frankly, it's impossible to assume that you're going to know everything about **marine magnetometers**, **call centers and telephony**, or **technical recruiting**. It is, however, possible (and *expected*) that you will learn.

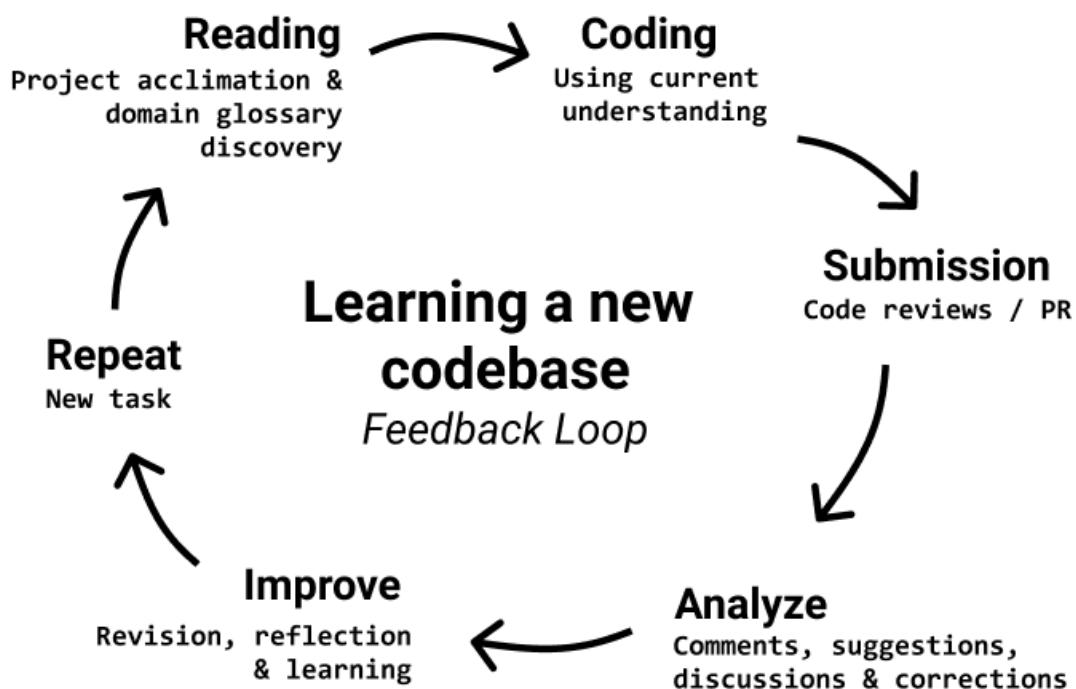
In a new codebase,

- We read code
- We identify the choices that have been made
- We ask questions
- We look for patterns

Using the patterns we've identified, we attempt to extend those patterns in future scenarios.

If we fail or need to be corrected, we adjust and improve the next time around.

This process takes a variable amount of time, but we should be consistently improving- like a *feedback loop*.



We gain confidence when we know what to expect, and then we *get* what we expect.

Full understanding is when we can correctly address future scenarios on our own.

Consistency (helps reinforce) → Expectations (when met, turn into) → Confidence (which implies) → Understanding

Consistency in naming

Failing to be consistent with names has the potential to degrade each of the other principles of naming things. For example:

- (Name) **understandability** is negatively affected when consistency is poor because not only do we have to understand what a term means *once*, but we also have to determine if another *slightly similar* yet different name also refers to the same concept.
- (Name) **searchability** is negatively affected when consistency is poor. When a concept needs to be renamed or changed, what would normally be easy to change using the *find and replace* feature in a text editor becomes challenging.

Consistency with everything

It's not just important to *naming*.

If architectural decisions are consistent, **we understand the architecture faster**.

If packaging decisions are consistent, **we understand how packaging is handled on this project, faster**.

If *certain paradigms* are being used in *certain situations* on a consistent basis, that **paves the way for us to assume what paradigm to lean towards in those same situations in the future**.

Therefore, be consistent because:

- Consistency helps build momentum.
- It improves each iteration of that feedback loop, increasing our ability to become productive, and
- Seeing things done a particular way consistently prepares us for how to handle similar situations in the future. By presenting knowledge in a consistent way, people can build patterns against it.

Uniqueness

Name uniqueness is important for several of the same reasons that consistency is important, but I think the most important is that it **ensures a singular understanding of a new concept**.

Starting work on a new project means that we first have to spend a lot of time reading. When we come across words or domain concepts that we're not familiar with, we have to ask questions to figure out what they mean in order to move on.

If we master the understanding of that class, method, or function once, then we understand its responsibility and subsequent usage throughout the entirety of the codebase.

Confusion arises when **the exact same concept** or one **extremely similar** appears and is represented using a different word.

Here's a confusing situation:

- Q: What's the difference between a Job and JobObject?
- Q: When should I use Job and when should I use JobObject?

- Q: Let's say I have some new functionality to add. Should I add it to Job or JobObject?

The good thing is that we have lots of best practices against naming things to avoid these types of scenarios entirely.

For example, **Prefer Domain-Specific Names over Tech-y Sounding Names** from **3 - Specificity** introduces one possible way to mitigate introducing confusion like this.

Consistency & uniqueness are two of the most important principles because they affect all of the others.

Here are a few best practices to follow to reinforce 'em.

Best practices

Avoid using similar words to express the same concept (thesaurus names)

What's the difference between get, show, display , and present? If they're all supposed to represent the same behaviour, it's not easy to determine that. If they're all supposed to represent different behaviour, it's not easy to determine that either.

```
export class PostAPI {
  getAuthor (): Promise<Author> { ... }
  fetchPosts (): Promise<Posts> { ... }
  showTags (): Promise<Tags> { ... }
  presentCategories (): Promise<Categories> { ... }
}
```

My intention for each of these was to merely fetch data from a backend service and return it to the caller.

Because we're using different words, the reader will constantly need to check inside the function or method block to see what it actually does - that's not very Intention Revealing.

Instead of using different words, choose one word to express the concept that we're trying to represent, and stick to that throughout the entirety of the project.

Follow programming language and project (naming) coding conventions

TypeScript and JavaScript have a set of coding and naming conventions, like using pascal case to signify that an identifier is a class, type, enum, or interface.

```
// This is conventional! Use PascalCase for types.
type User = {
  id: string;
  name: string
}

// Also conventional usage!
interface Serializable {

  // But notice that property names/attributes are NOT PascalCase, but
```

```

    // camelCase instead
    toJSON (): string;
}

// PascalCase on classes are conventional
class UserModel implements Serializable {
    ...
}

```

But non-constants, variables, and functions are supposed to be represented using camel case in TypeScript and JavaScript.

```

const shouldListenToJohnMaus = likesWeirdMusic()
    && isGenerallyAHappyPerson();

```

Of course, the language conventions are secondary to whatever coding conventions that you and your team decide upon enforcing (hopefully using tooling).

- Example (naming) coding conventions you could enforce (via Formik):
 - Use PascalCase for type names
 - Do not use “I” as a prefix for interface names
 - Do not use “_” as a prefix for private properties
 - Use `isXXXXing` or `hasXXXXed` for variables representing states of things (e.g. `isLoading`, `hasCompletedOnboarding`)

The most important thing is that you’re consistent with your choices.

Avoid very similar variable names by mis-spelling (or using correct, alternate spellings)

Sometimes by misspelling a variable name, we end up with two or more copies of the same. There’s also the case of different spellings occurring for the same word. Man, English can be weird.

- Example of different spelling: `color`, `colour`
- Example of misspelling: `PaymentProcessor`, `PaymentProccessor`

Don’t use the same name to express different concepts from within the same namespace

It’s important to really:

- Imagine importing `formatEmail` from `shared/utils` and importing `formatEmail` from `users/domain/email`.
- This is really just another way to say write **DRY** code.

Then again:

- AThng in a shipping subdomain, and AThng from a bidding subdomain have entirely different meanings depending on the context.
 - Make sure that if you’re going to use similar names to enforce boundaries between those domains to allow the same names to be used in a way that doesn’t clash syntactically (errors) and doesn’t clash semantically (our understanding).

- It does have the potential to introduce confusion if a new subdomain C relied on a concept from A that also has something with the same name from B.
 - If all we know is the name, how does C know which one is the correct one to use?

Don't recycle variable names

You have a keyboard with all 26 numbers of the alphabet. When writing functions with temporary variables, it's advised to create as many temporary variables as you need rather than re-initialize and overwrite *one* used for several different purposes.

If a variable is being used for several purposes, it's actually rather challenging to distinguish the **current purpose** of the variable at any point in time, and it's rather easy to forget to reinitialize it when purposes change.

Here's an example of reusing the temporary `sum` variable for two different purposes.

Bad

```
let sum = 0;
categories.map((c) => {
  if (c.name === selectedCategory) {
    sum += 1;
  }
})

console.log("Category match total score is", sum);

sum = 0;
tags.map((t) => {
  if (t.name === selectedTag) {
    sum += 1;
  }
})

console.log("Tag match total score is", sum);
```

A better demonstration would be to get more *specific* with the names to make sure that each variable serves a **single responsibility** and is **unique** in the process.

Good

```
let categorySum = 0;
categories.map((c) => {
  if (c.name === selectedCategory) {
    categorySum += 1;
  }
})

console.log("Category match total score is", categorySum);

let tagSum = 0;
```

```

tags.map((t) => {
  if (t.name === selectedTag) {
    tagSum += 1;
  }
})

console.log("Tag match total score is", tagSum);

```

Names should be unique, regardless of the case

`empName`, `EmpName`, and `Empname` all refer to the same concept. They should not all exist in the same program as different things. That's a sure-fire way to foster confusion.

2 - Understandability

A name should describe the concept it represents. - via NamingThings.co

When we understand something, it means we understand what something is, and how we can use it. In a previous discussion about the psychology of design, we learned that discovery involves combining:

- Knowledge in the head (logic, conceptual models, semantic, memory), and
- Knowledge in the world (culture, experiences, physical things)

Knowledge in the world

As a software designer, when naming things, lean more on **naming things to rely on knowledge in the world**.

Knowledge in the *head* is memory — things we have to think about. Things that could take us a moment to recall how they work. It's often logical or requires some upfront processing to summon.

Ie: What's 12×9 ?

Knowledge in the world is **easily recollected**. It's so deeply ingrained in us that we barely have to think about it. We know what it is, and we know what to do right away.

Ie: If you saw a snake or a large spider on the ground near your foot, how long would it take you to discover that you should run? Not long at all, I hope.

Knowledge in the world is much deeper and can be used in more dexterous ways with less effort.

If we can give things names that tap into **knowledge in the world**, they can potentially be really good.

Therefore, to be successful with a name, **represent the real world concept**.

Representing real-world concepts

Being clever enough to name things after real-world concepts assumes that we're a developer dedicated to the craftsmanship of the product and empathetic enough to *care* about learning the domain if we're not familiar with it.

The way this works is:

- If you're familiar with the domain, then every domain concept that appears in code is going to have meaning for you
- If you're not familiar with the domain, then as you learn the business, the code starts to make more sense

I think this is the best methodology for naming things. That way, when we get caught up to speed on the way the business works, not only do we know what's being referred to in the code, but we've got a better understanding of what these objects can semantically *do* in the real-world.

They help us work towards a **conceptual model of the entire business as a system**.

Domain-specific names are a long term investment

Using names that describe the concept they represent is a long term investment.

It's much easier to just choose a name that works for now, but if the original author leaves the company and is inaccessible, the meaning could be lost forever and it may be very hard to interpret in the future.

"Why did Khalil name this class, Dealie? I have no idea what the hell that is.
Does anyone have his email? Can you check to see if you still have his number?
Do you know where he ended up?"

Personally, I had a teacher that used to call just about everything a dealie. So to *me*, a dealie is something temporary or ephemeral. But unless you were there, you wouldn't know that.

Names that refer to their real world counterparts are more likely to live longer lives and continue to maintain their understandability.

This is true. How likely is it that the core aspect of a business would change. Jim Bob's *Furniture Warehouse* isn't going to shift their business model to decide to do kitten grooming instead, overnight. If we can count on the business to fundamentally be the same, we can be sure that names will maintain their usefulness as soon as the business is understood by new developers.

The domain layer describes core business rules

To write names that describe the business, that kinda implies that there's a place in our code we can use to do this.

That's called the *domain layer*.

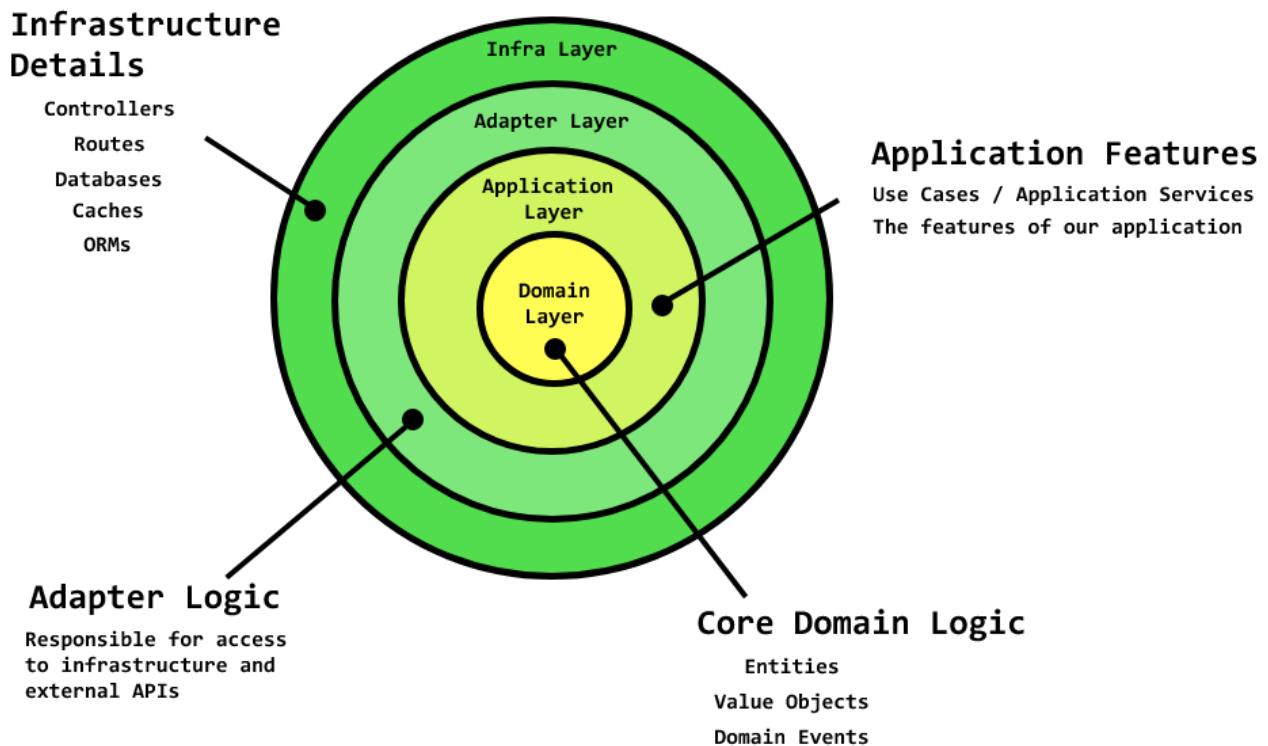
There's a software design and architecture principle called **Separation of Concerns**. It says that we should divide code into sections that each address specific concerns.

For example, one of the most popular ways to think about building web applications is to apply the **Model-View-Controller** architectural pattern. In MVC, the model, view, and

controller take on a unique set of responsibilities that, when combined, give us the basic architecture needed to power a web app.

As we've discussed in "Knowing When CRUD & MVC Isn't Enough", the M in MVC isn't very specific about the structure and responsibility it holds.

To remedy this problem, we can look to a more robust Separation of Concerns through the use of the Clean Architecture (also known as Hexagonal Architecture).



The *domain layer*, which is at the center, best describes the way the business looks in the real world.

Check out the domain-specific names of the classes within the domain layer for ddd-forum here on GitHub.

Here, you'll find classes, interfaces, methods, and variables named exactly as they appear in conversation. These classes contain methods and interfaces which describe the business rules and should be written with as much clarity and expressiveness to the domain as possible.

```
// UserName is from a separate subdomain, `Users`
import { UserName } from '../users/domain/types'

type MemberId = string;
type Member = {
  id: MemberId,
  name: UserName
}

type Text = string;
```

```

type Link = string;

type PostTitle = string;
type PostId = string;

type Post = {
  postId: PostId
  postedBy: Member,
  title: PostTitle,
  content: Text | Link,
}

type Upvote = {
  postId: PostId,
  memberId: MemberId
}

type Downvote = {
  postId: PostId,
  memberId: MemberId
}

```

You probably have a good idea about how the domain works simply by looking at the names of the types and their relationships.

The idea is to make the code here so clear and understandable that even non-technical domain experts could understand it.

Naming things in more technical layers

The domain layer is the most *declarative* layer of code that refers to things as closely as their real-world counterparts.

The application layer contains use cases. It's extremely straightforward to name use cases by name: `CreatePost`, `UpvoteComment`, `DeleteComment`, `GetPostBySlug`, etc.

Infrastructure layer, however concerns itself with things like controllers, routers, caches, repositories, etc. These concepts are pretty much purely technical and they can only be understood by programmers that are familiar with those types of objects in computer science.

Using architecture & frameworks to dictate how to name things

Frameworks like Angular and Nest.js introduce a convention where the name of the *construct* is included in the name of the file.

Examples:

- `cats.service.ts` — it's a *service* that operates against **cats**
- `cars.module.ts` — it's a *module* that aggregates everything to do with **cars**

This convention can be helpful to increase discoverability of a file.

There's a construct for everything

Don't know what to name that class or file?

There's a chance that you **might not yet know the best construct for it.**

One of the best practices is to *Avoid tech-y sounding names*. That includes things like Processor, Manager, and Helper. They're fluffy. They don't really add to helping understand what the class does. We're writing *code* — isn't all code processing, managing or helping in some way, shape, or form?

I was calling classes that perform *validation logic* Validators for a long time. That works, but I later discovered the correct construct was a Value Objec.

Examples of different valid constructs for the same resource type:

- User, UserMapper, UserRepository, Username, UserCreated.

Best Practices

Don't randomly capitalize syllables within words

It's remarkable how much meaning a variable loses when we capitalize a syllable in the middle of a word. Example: PaidJobDeTails.

Avoiding names with digits

Since the lower case l looks a lot like the digit 1, we should avoid names with digits in 'em.

For example, 101 is more likely to be mistaken for 101 than 10L is.

If you have to refer to a number, prefer the English word to represent the number rather than the numerical version:

- ThirdCard > 3rdCard
- ThreeSixtyNoScope > 360NoScope
- OneIota > 1Iota

Use pronounceable names

When names are pronounceable, readers can communicate them with domain experts and developers in conversation. Try meeting up with your customers and asking them what XEN_94_PHXX_ is supposed to mean.

Use the CQS (Command Query Separation) principle for naming methods

The CQS principle helps to simplify code paths. It states that there are two types of operations: commands and queries. To mitigate confusion and unexpected side-effects, a command results in side-effects and returns no data, while a query returns data and performs no side-effects.

Be very clear with this in your method signatures.

```
export class UserRepository implements IUserRepo {  
    createUser (user: User): Promise<any> { ... }
```

```
getUserById (userId: UserId): Promise<User> { ... }  
}
```

Document side effects in methods with several notable side effects

If methods have unexpected side-effects, be clear with the side effects in the name of the method so that users don't need to read into the entire method to figure out **what** it does.

Examples:

- `createUser` vs. `createUserAndSendAccountVerificationEmail`

Doing this is implementing the Principle of Least Surprise.

Use domain concepts to things from the business

The **main** best practice. Name things as they occur from the business within the *domain* and *application* layers.

Use technical concepts to express technical things

When you're naming instances of *caches*, *databases*, *factories*, *adapters*, *mappers*, *repositories*, etc — use the correct name of the construct. Not something you made up.

Use simple (grammatically correct) English (without spelling errors)

This Wikipedia article lists all the programming languages that **do not** use keywords taken from or inspired by English vocabulary. They're few and far, so let's agree that in order to write code to be understood by humans, use as formal English as possible.

Use simple nouns and adjectives. For example,

- bad > amateurish
- student > pupil
- fast > expeditious

Avoid spelling errors and use grammatically correct versions of words.

Don't omit vowels or unnecessarily abbreviate words

Most programming languages let you create very long identifier names. There's no need to say `Cntrllr` instead of `Controller`.

Avoid misleading names

Say what you mean and mean what you say. If the class or method says it does something, make it do that, and that only.

For example**, you don't want to see an `isValid(x)` that doesn't return a boolean, but instead returns a string. Make the obvious thing happen.

Avoid using negatives in methods that return boolean

Consider a function that validates an email. Perhaps the name is `isEmailNotValid(email: string)`. Try to refrain from using *negatives* in the names of the variables. It's less obvious and takes a small amount of *logical processing* to figure out if they should negate the response or not.

```
if (isEmailNotValid(email)) {  
    return;  
}
```

Instead

```
if (!isValidEmail(email)) {  
}
```

Avoid irrelevant names

```
marypoppins = (superman + starship) / god;
```

Make meaningful distinctions

Distinguish names in such a way that reader knows what the differences offer.

For example,

moneyAmount is indistinguishable from **money**,

moneyInRupees is clearly distinguishable from **moneyInDollars**.

3 - Specificity

A name shouldn't be overly vague or overly specific. — NamingThings.co

I can't remember where I heard this piece of advice, but it was that it's "better to over-specify than under-specify". I can see the logic in that. If we're not really sure what to name something yet, a *longer* and more *specific* name might prove easier to refactor than a short and nebulous one.

Nowadays, with the find-and-replace tools we have, it can be much easier to write a very specific name and move on, leaving room to come back to that name later.

Let's say we were working on writing some *auth* code and we ended up writing this very specific method name:

```
class AuthenticationService {  
    public userHasAuthenticatedAndVerifiedEmailShouldTheyBeNonAdmins (  
        user: User  
    ): boolean { ... }  
}
```

OK, that's a lot — yes. But let's try to find some *good* in it.

Good things

- It's well intentioned — the author is trying to make it well known what this is for.
- There is more information provided about what it's for, and how we're determining it.

And now, the *bad* things.

Bad things

- Fatiguing — I have a suspicion that you dislike the name as much as I do. There might be more information encoded in there, but it's hard to read.
- Misinformation — Each word in here needs to contribute to us understanding how to use it. There are several words in this name that do nothing for us. This takes us back to English class in a way. In this name, here are the useless words that don't help us: and, should, they, be, non.

It's also useful to point out that some of these words signify *conditionals* — which tells us that they could be split into their own methods.

```
class AuthenticationService {  
    public userIsAuthenticated (user: User): boolean {  
        ...  
    }  
  
    public userVerifiedEmail (user: User): boolean {  
        ...  
    }  
  
    public userIsAdmin (user: User): boolean {  
        ...  
    }  
}
```

Interesting, right? Suddenly, without the over-specification, this code appears to be much less abrasive and more useful. To decide whether a user should be allowed to access a resource might be a whole lot clearer

```
function canUserViewResource (): boolean {  
    if (authService.userIsAdmin(user)) {  
        return true;  
    }  
  
    if (authService.userVerifiedEmail(user) &&  
        authService.userIsAuthenticated(user)) {  
        return true;  
    }  
  
    return false;  
}
```

Over-specifying

What leads us to needing to *over-specify* anyways?

I think it's two **main things**.

- When more than one variant of a concept is known to exist
- When we feel we have not provided adequate context

Multiple variants

A classic example of multiple variants is an application that lets you create a bunch of different ducks.

Bad

```
class Duck {}
class DuckThatCanFly extends Duck {}
class DuckThatCanFlyAndCook extends Duck {}
```

Naming *variables* this way is totally cool, but naming more concrete things like *methods*, *functions*, and especially *classes* — not only is this over-specifying, but **it's a really stinky code smell. We can fix this by refactoring to design patterns, namely the Factory Pattern here.

Good

```
const duckThatCanCook = DuckFactory.create({ capabilities: ['cook'] });
const duckThatCanFlyAndCook = DuckFactory.create({ capabilities: ['cook', 'fly'] })
```

Lacking necessary context

In the following example

Bad — Example of a variable providing context as to how it is about to be used

```
async function exists (userId: userId): Promise<boolean> {
  // This variable explains how it's going to be used
  const tempUserToDetermineIfExistsOrNot = await this.userRepo.getUser()
  return !!tempUserToDetermineIfExistsOrNot;
}
```

It's improved by a sense of name-spacing. This is why I prefer classes over stray functions, because the methods within class exist to perform something that semantically makes sense for it to do within the class.

This removes the need for us to describe **why the variables exist** and allows us to shift to naming them based on what is stored within them.

Good — the variable simply describes what is stored in it, not any additional context as to why it was created.

```
export class User implements IUserRepo {

  public async exists (userId: userId): Promise<boolean> {
    const user = await this.userRepo.getUser()
    const userExists = !!user;
    return userExists;
  }

  ...
}
```

Here's a rule:

In as *minimal words necessary*,

A name should describe **what is inside the variable — nothing more, nothing less.**

Under-specifying

Let's say we've got this Option class, but we want it to be used in a very specific way by a specific class. Actually, let's also say that all usage of this class *outside* of that very specific way is completely wrong.

Check the following code snippet, what's wrong with it?

```
class Option {  
  
    private food: Food;  
    private selected: boolean;  
  
    public get displayValue (): string {  
        return this.food.text;  
    }  
  
    constructor (food: Food) {  
        this.food = food;  
        this.selected = false;  
    }  
  
    public isSelected (): boolean {  
        return this.selected;  
    }  
}
```

There's a lot of reference to things about *food*, it appears this Option class isn't as **generic** as we thought it might initially be. I mean, the name Option doesn't really signify that it's *only* going to be used as a way to display Food options. If that's a constraint that we're imposing

Optional type annotations

In TypeScript, we have the option of annotating a variable with the type like so:

```
// The optional type explicitly tells us that this is a `User` type.  
let u: User = this.getUser();
```

Since this is an *optional* feature, it makes it that much more important to choose variable names that **describe what is inside the variable**.

React hooks API

I want to talk about the React Hooks API for a second. I found it kind of *funny* when I first encountered it.

React Hooks is a way for you to enable view-layer reactivity using the React library. It acts

as a place for you to locate **component state** and interaction logic. When you call useState, you can pass in an initial value, and in return, get an *array* that can be decomposed into two variables.

The first variable contains the **current value** of state.

The second variable contains a **function** that— when invoked, sets the current value of the state.

It looks like this:

```
const [replyText, setReplyText] = useState('');
```

Take a moment to first of all understand how *unintuitive* this API is.

Now, take a moment to appreciate how *good the names are*.

replyText is a string value — the name says no more than what is stored within it.

setReplyText is a function capable of changing the replyText. It's a *variable*, but because it's named **like a method or a function**, appreciate our ability to discover how to use it.

```
setReplyText('This is interesting, indeed!')
console.log(replyText); // "This is interesting, indeed"
```

Consider if the names were different. Here's are several examples of under-specifying **what is stored in each variable** and even in some cases, *incorrectly*.

```
const [text, text2] = useState('');
const [t, t1] = useState('');
const [a, b] = useState('');
const [_, __] = useState('');
const [setVal, val] = useState('');
```

Hopefully we're seeing just how much this API relies on us:

1. Remembering the order of the variables to be decomposed
2. Remembering that the first variable is the type of the value that was passed in initially
3. Remembering that the second is a function to change state
4. **Choosing names that adequately specify both variables' capabilities and how they may be used.**

Best practices

Singular / plural naming

Main takeaways

1. (optional) Use a to signal that a singular item is stored within the variable

```
// Instead of this
const car = new Car();

// Try this
const aCar = new Car();
```

-
2. Use collection, list, array or *plural* names to signal that a group/array of items is stored within the variable

```
// Good
const todos: Todo[] = [
  { id: 0, text: "First todo", completed: false },
  { id: 1, text: "Second todo", completed: false },
]

// Good
const todosList: Todo[] = [
  { id: 0, text: "First todo", completed: false },
  { id: 1, text: "Second todo", completed: false },
]

// Bad
const todo: Todo[] = [
  { id: 0, text: "First todo", completed: false },
  { id: 1, text: "Second todo", completed: false },
]
```

With abstracted collections, the fact that a variable is actually a list can be sometimes be hidden.

Abstracted collections

```
type Todos = Todo[]
type TodosList = Todos;

class TodosCollection {
  private todo: Todo[];
  constructor (todos: Todos[]) {
    this.todos = todos;
  }
}

class Collection<T> {
  private items: T[];
  constructor (items: T[]) {
    this.items = T;
  }
}
```

Abstracted collections can be useful. It's important to apply this rule especially when working with them.

Avoid referring to things as variables, classes or methods in the name

Unless you're building the next CodeAcademy, writing your own compiler, or doing some

very meta-level programming, it's not necessary to include the name of the identifier type in the name of a variable.

Instead of something like this:

```
const userVariable = new User();
```

Prefer omitting the identifier type name.

```
const user = new User();
```

Name people by their roles

It's unlikely that the best thing to call a *user* in every single application is actually a *User*. This is very much a domain-driven concern, but understanding the *role* of a particular type of *user* in your domain leads

For example, a *User* could actually be an *Admin*, *Editor*, *Employee*, *Cashier*, *Visitor*, etc.

Directly related concepts: Single Responsibility Principle & Conway's Law.

Specificity against unions

If it's possible for a variable to hold one or more types, it's a good practice to address that possibility in the name.

```
export class UserRepo {  
    // The name is inclusive to describe what may be stored in the variable  
    // using a union type.  
    public getUser (userOrUserId: User | UserId): Promise<User> {  
        ...  
    }  
}
```

Avoid blob parameters

Blob parameters are parameters that say nothing about what needs to be passed in.

Here's some JavaScript code of a *User* factory method.

```
export class User {  
    public static create (args) {  
        ...  
    }  
}
```

I want to say that this is mostly a JavaScript issue, and not a TypeScript one, but using the *any* keyword with TypeScript can create the same conundrum.

Always strictly type *object* parameters.

```
interface UserProps {  
    email: Email;  
    password: Password;  
    firstName: FirstName;
```

```

    lastName: LastName;
}

export class User {
  public static create (props: UserProps) {
    ...
  }
}

```

Avoid number series parameters

The number of cases where it's actually preferable to name parameters based on their order, rather than an intention revealing name, is *rare*. Take the following example.

```

function cloneArray (arr1, arr2) {
  arr1.forEach((item, key) => {
    arr2[key] = item;
  });
  return arr2;
}

```

This could be made much clearer using the words `source` and `destination`.

```

function cloneArray (source, destination) {
  source.forEach((item, key) => {
    destination[key] = item;
  });
  return destination;
}

```

Utilize namespaces

It's possible for a name to exist within several *domains* of your application. The concept of a Job in the MediaProcessing domain of your app might be a lot different from the meaning of a Job in the HR portion.

Ways to enforce this:

- The namespace language construct
- Name-spacing by packaging things into modules
- And in worse case scenarios, name-spacing using domain-specific prefixes.
 - Ie: MediaProcessingJob vs. Job.
 - We should make every effort to avoid this situation. Imagine someone from an entirely different city told you that you weren't allowed to have the same name as them because they already have that name.

Use abbreviations sparingly

AFI, what is that? The American Film Association? The punk band, A Fire Inside? An Absolute File Instance? Context can scope things things down, but understand that abbreviations are a form of *under-specification*. Common abbreviations are cultural, and can take time to catch on. **

Example:

- Do you know what `btoa` and `atob` are in programming?

References

For further reading, check out this conversation from Ward Cunningham's wiki ironically titled Very Long Descriptive Names That Programming Pairs Think Provide Good Descriptions.

4 - Brevity

A name should be neither overly short nor overly long.

Code that's too verbose (long) is hard to read. Weirdly, code that's too succinct (short) is **also hard to read**.

This phenomenon is the fight—the push and pull of **compression vs. context**.

Compression vs. Context

Compression

To communicate anything, we have to compress it. Examples of this exist everywhere. Book summaries, movie trailers, blog post subtitles, etc. They convey the *main idea but sacrifice the details in the process*.

By compressing ideas, we say a lot by saying little. This is why people like life quotes.

“Life is what happens when you’re busy making other plans.” — John Lennon

Compressed **too much**, ideas lose all context and meaning.

Context

To explain something, we provide context. Context is provided by reading the book, watching the movie, reading the article, and digesting the material.

For example, the Naming Things chapter stated that *naming* comes down to seven principles. To provide context to that statement, we're spending some time making an argument for each one.

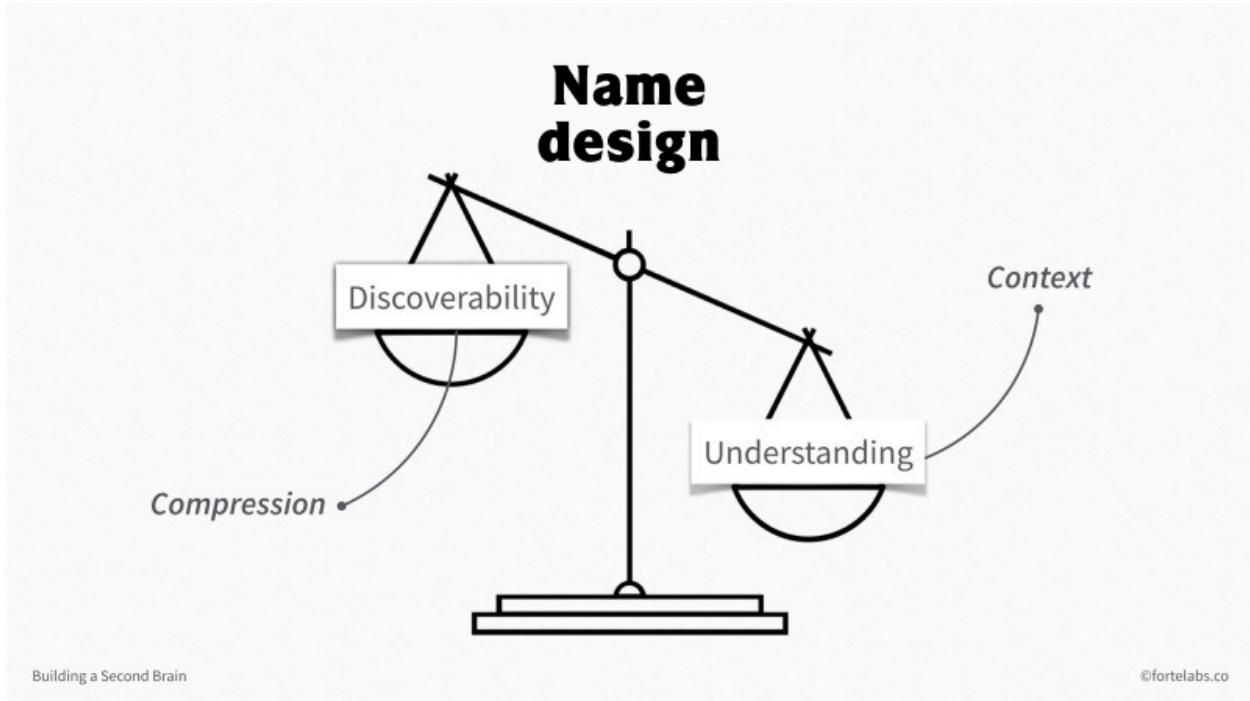
This takes time to do.

Additionally, going too far **contextually by providing too much information, we can lose the main takeaway**.

With more compression comes more *discoverability ****but less *context*.

With more context comes more *understanding* but less *compression*.

It's like balancing a **scale**.



The law, reiterated

Communicate *what* with names, explain *why* with context.

I believe this is the answer. It's the same statement from the last section, but I believe in it.

Ask yourself the questions:

- Are we adequately describing what is in this name?
- And then, does the context help to reinforce why it makes sense for this name to exist?

Do you see the pattern here with respect to comments? Comments can be used to help provide additional **context** by explaining **why**: things not communicated with code.

This principle shares a lot in common with the specificity because the important attributes, context and compression are at the forefront of what makes them successful.

Best practices

Use concise English

Some names contain words that, when removed, do nothing to change the meaning of the name. There's a book called *The Elements of Style*. It lists several rules for good writing. One of the names is to "omit needless words".

Here are practical examples:

- in order to = to
- be able to = to
- in spite of ⇒ despite
- a number of ⇒ some
- in the event that → if

- has the opportunity to → can
- despite the fact that → Although

Consider that this can be applied to creating names as well. Ask yourself if removing a word from the name would change its meaning or not.

Refrain from non-conventional single-letter variables

When we first start programming, we encounter conventions that eventually feel as natural as breathing.

One of those conventions is **using short local variables inside of short blocks.**

Conventional (local variables inside of short methods):

- `sum(a, b)` — common math utilities like this are culturally known by programmers.
- `for (let i; i = 0; i++)` — short variables in *for-loops* are also conventional. Though they're convention, I'd almost always recommend refactoring them to array methods like `map`, `filter`, `forEach`, etc.

These few conventions, widely accepted in programmer culture, are OK.

Outside of convention, short variables used in unconventional functions and methods are harmful.

Since there's no convention to go based on, we could improve the developer experience by choosing names that help users discover how they are to be used.

- `createUser(a, b)` — this says nothing to us
- `createUser(arg1, arg2)` — this *still* says nothing
- `createUser(email, password)` — there we go, that's much better.

Grouping indicates context

How can we provide context without encoding it into a name?

Classes, namespaces, folder names, and proximity to other similar functions or constructs—these all communicate context.

Take the following code example. It provides absolutely no context as to what we're operating against.

```
function create(props) {}

function edit(id, props) {}

function del (id) {}
```

How do we improve the understandability of this? We can do any of the things we described above. Introduce context. One way is to encapsulate the operations within a class or an object.

Here's the class version.

```
export class StudentRepo {
```

```
public create(props) {}

public edit(id, props) {}

public del (id) {}
}
```

And here's the object version.

```
export const StudentRepo = {

  create: (props) {}

  edit: (id, props) {}

  del: (id) {}
}
```

Much better, right? Before the refactor, it was impossible to have known that these were operations against *students*.

This best practice leads into the next one nicely.

The operation and resource name must be in context

In the previous example, we provided a bunch of functions that indicated they were capable of performing the actions: `create`, `edit`, and `delete`. The **operations were in context**.

```
function create(props) {}

function edit(id, props) {}

function del (id) {}
```

But the *resource* that they operated against — these were **not** in context.

If we don't have both the **operation** and the **resource** that the operation is executed against in context, then it's going to be hard for us to **understand** the name.

```
// Poor
function create (props) {}

// Good
function createUser (props) {}

// Good
class User {
  function createUser (props) {}
}

// Good
```

```
const user = {
  create: (props) => {}
}
```

Don't use unnecessary member prefixes

Sometimes, developers prepend their private member variables with an underscore to signify that it's private.

Example: `_firstName`.

In JavaScript, there's no concept of private access modifiers, so the convention is to signal that it's private this way.

In TypeScript, we *do* have private access modifiers.

Refactor member prefixes out of objects

Another thing sometimes promoted is to prefix some members with a specific set of characters to denote that they're special.

In AngularJS, it was recommended to put a `vm` in front of your members to signal that they're *view model* variables.

Example: `vmFirstName`

In my opinion, we should group these variables into an object, maybe created from a class or object called `ViewModel`, to remove the need for us to include signaling in the member names.

Instead of this

```
// Instead of this
class UserController {
  get firstName () {
    return this.vmFirstName;
  }

  get lastName () {
    return this.vmLastName;
  }

  get avatar () {
    return this.vmAvatar;
  }

  constructor () {
    this.vmFirstName = "";
    this.vmLastName = "";
    this.vmAvatar = null;
  }
}
```

Try this

```
// Try this
class UserController {
    get firstName () {
        return this.model.firstName;
    }

    get lastName () {
        return this.model.lastName;
    }

    get avatar () {
        return this.model.avatar;
    }

    constructor () {
        this.model = new UserViewModel();
    }
}
```

5 - Searchability

A name should be easily found across code, documentation, and other resources.
— NamingThings.co

Searchable names is good because it:

- Makes refactoring easier — we can change all occurrences of something that can be found using a text editor
- Can be located in documentation
- Can be understood where and how it works throughout the entirety of a codebase.

There are certain things can makes names **unsearchable**, however.

- Names being too short — ex: a
- Names being too generic — ex: user
- Different names referring to the same concept (outdated documentation).

Search-ability is mostly influenced by the previous principles: consistency and uniqueness, specificity, and brevity.

To do this well, follow *those* practices and consider the following ones here as well.

Best practices

Avoid using numeric constants

Consider we wanted to figure out where the default movie rating setting was.

```
// Default rating of 8
function addMovieRating (rating = 8) {}
```

That would be hard using a numeric constant.

Numeric constants can be replaced with a constant variable.

```
const DEFAULT_MOVIE_RATING = 8;

function addMovieRating (rating = DEFAULT_MOVIE_RATING) {}
```

Keep documentation up to date

Ideally, make updating documentation a part of the release process for new code and features.

6 - Pronounceability

A name should be easy to use in common speech. — NamingThings.co

Names that aren't pronounceable aren't easily communicated. And being able to discuss code with our peers is a great way to improve the quality of our designs.

Not only that, but it's a lot harder to remember names that we have trouble pronouncing (this is just as true at social gatherings with new friends).

Best practices

Very standard abbreviations don't have to be pronounceable

Most people understand that ssn stands for **Social Security Number**.

Use camel case to signal word breaks in variables

Imagine you have the name preparearead.

It's saying "prepare a read", but it takes an unnecessary amount of mental processing to discover that.

We can fix this with camel casing: prepareARead.

Booleans should ask a question or make an assertion

Booleans and methods that return booleans should either ask a question or make an assertion that is either truthy or falsy.

Good

- Assertion — potIsEmpty()
- Question — isPostEmpty()

Bad

- makeVar — this actually signals that the variable and perform an operation
- hello — does not signal that a boolean will be stored here

Don't omit vowels

For example. This...

```
type timeStamp = Date;
```

is better than...

```
type tmStamp = Date;
```

7 - Austerity

A name should not be clever or rely on temporary concepts. — NamingThings.co

I love a good laugh as much as anyone else, but we should remember what we're working towards. Encoding cute and funny things in our code is hilarious in the short-term, but dreadful to work with in the long-term.

And 90% of what we do is in the long-term (I made that number up).

Not everyone has the same sense of humor as you

There's already enough challenge in writing a name to be understood. So why write a name with secondary meanings?

Temporary concepts need to get cleaned up

Historical events, popular culture, memes, jokes, and anything that is generally unrelated to the business is all stuff that will eventually need cleaning up.

Some projects survive for 5 years. Some survive for 10, 15, and 20 years.

Imagine having to explain a joke over and over for a decade. I don't think the maintainer working on those core modules would find it very funny.

Best practices

Avoid being cute, funny, clever

I know, I'm a stickler — but really, you want your peers to be successful working with your code. Save the quips and the humor for happy hour after we successfully wrap up the sprint.

Don't include include popular culture references

Not everyone has seen Star Wars. And while that may be a shame, someone shouldn't need to know Han Solo's best mate to understand what a particular block of code is responsible for.

Organizing things

Coming soon!

Why code organization matters

- The way you organize code has a profound impact on:
 - How long it takes to develop new features
 - How long it takes to locate a feature
 - How much mental energy it takes to change code
 - * Flipping back and forth between files and folders
 - Knowing where new code should go
- Colocation
- Different types of packaging and the ergonomics of 'em.

Practical naming

Principles

- Use conventions
 - Your source code goes in the src/ folder
 - docs go in the docs/ folder
 - config
 - examples
 - dist/build/ for compiled code
- Keep your code DRY / don't repeat yourself with files
- Group files related to a feature close to each other (colocate files with high cohesion).
 - Experiment and notice how often you're flipping around and getting fatigued.
 - Examples:
 - * DDDForum use cases
- As flat as possible
- Screaming architecture
- Shared folder for anything that doesn't
- Use package by infrastructure on small projects
- Package by feature on larger projects

Great examples:

- DDDForum
- Apollo Client open source library

Organizing and context

Here we have all of these files adhering to the same naming convention. They all have the name of the use case at the front.

```
useCases/  
  createUser/  
  editUser/  
  deleteUser/  
    DeleteUserUseCase.ts  
    DeleteUserErrors.ts
```

```
DeleteUserController.ts  
DeleteUserResolver.ts
```

But what if

How to settle design arguments

- In subjective conversations about design, there's only really one way to decide on which approach is better — observe users. This is what we do in Human-Centered-Design.
 - GOMS — this process exists so you can measure the efficiency it takes a user to accomplish a task
 - If you're uncertain about some approach to designing your code or making some API easy to use, this is a way to handle that.

Errors and exception handling

Coming soon!

Makes me think we should address nullability and errors. In DDDForum, we throw an error.

Relevant links

- 200 OK! Error Handling in GraphQL

Testing

Coming soon!

This is also very important to clean code

Things I should talk about in the **testing chapter**

- BDD
- TDD
- Unit testing
 - What makes a unit test a unit test?
 - Is it worth it?
- Integration Testing
- End-to-end Testing
- Kent C. Dodds' testing pyramid
- Nuances of testing on the backend
- Nuances of testing on the frontend

BDD & TDD

- BDD tends to lend itself more towards integration testing
- TDD tends to lend itself more towards unit testing

Unit Tests

Resources

<https://kentcdodds.com/blog/how-to-know-what-to-test>

<https://kentcdodds.com/blog/write-tests>

Honestly, with the amount that there is to know about testing and testing in React apps, we might just be better off getting Kent's course and getting work to pay for it
— <https://testingjavascript.com/>

Unit Tests

What is a Unit Test?

Also, **Michael Feather's definition of a Unit Test**

A test is not a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Related

Refactoring

Coming soon!

Formatting & style

Each of the following suggestions are tactical ways for us to improve the design of our code to promote better **readability** and **discoverability**.

Formatting is the *visual appearance* of the source code. By applying *whitespace*, *consistency*, and *storytelling*, code can become more pleasant to read.

It's important to address that **formatting is incredibly subjective**. Some developers prefer to use tabs over spaces; some prefer the using max line lengths of 80 characters, while others prefer it to be much longer. These are subtle choices that will always be debatable.

If you're working alone, you're free to enforce your own style.

If you're working on a team or an open-source project, you have two tasks:

- Establish the formatting rules that you would like every to adhere to.
- Enforce those rules.

After we shed some light on the style choices you need to agree with your team, I'll show you how to enforce those decisions using modern tooling for TypeScript & JavaScript projects.

Objective readability truths

A moment ago, I said that formatting is subjective. This is true of anything visual. Code looking *good* is **extremely subjective**.

Who are you to say that de Kooning is *bad* and van Gogh is *good*?

Java developers may prefer to write their conditional statements like this:

```
if (isAccountOverdue) {  
    ...  
}
```

Where C developers may prefer the braces to be on the next line.

```
if (isAccountOverdue) {  
    ...  
}
```

Is one approach better than the other? Who knows. That's an age-old debate.

However, since you and I are both human beings, **there are certain physiological limitations we all share in common**.

For instance, if you hold your breath underwater, you will eventually need to come up for air.

If something is very far away, it may be very hard to read.

Bringing it back into context, because we are human:

- Without proper use of *whitespace*, code becomes virtually impossible to read.
- Without *consistency*, we can't build turn on the pattern matching algorithm in our brains. This increases the amount of time it takes to grow accustomed to a new codebase.
- Without *storytelling* and presenting the most important details upfront, readers can lose interest and get fatigued.

Whitespace

Whitespace an atomic component of fostering readable code. It's used to separate thoughts, tokens, algorithms, and so on.

On a small scale, look at this code. While it is syntactically correct, the lack whitespace makes it hard to read.

```
const artists=this  
.artistRepo.getArtists();const artistNames:string=artists.map(  
(a)=>a.name);
```

Doesn't make you feel good, right? Reading code this way is hard. Consider how unhappy you would feel working on a codebase where all of the code was formatted this poorly. On a slightly larger scale, try to understand the following code block, what it does, and how it does it.

users/services/usersService.ts

```
export class UsersService extends BaseAPI implements IUsersService { private referralCode: string;
  const state: any = store.getState(); const userState: UsersState = state.users;
  if (userState.isAuthenticated) {
    return (userState.user as User).profilePicture; } else {
    return ""; }
  } public updateProfilePicture (picture: File): Promise<any> {
  const data = new FormData();
  data.append('profilepicture', picture);
  return this.post('users/picture/new', data, null,
    { authorization: getAccessToken() })
  }
} private static decodeToken (token: string) : JWTProps {
  return JSON.parse(atob(token.split('.')[1]));
} private saveAuthToken (authToken: string) { window.localStorage.setItem("univjobs-access-token",
  JSON.stringify({ token: authToken }));
CookieUtil.setCookie('univjobs-access-token', authToken);
} private removeAuthTokens (): void { localStorage.removeItem("univjobs-access-token");
CookieUtil.eraseCookie('univjobs-access-token');localStorage.removeItem('last-email-seen');
} private saveLastEmailSeen (email: string) : void {
window.localStorage.setItem('last-email-seen', email);
} public isASStudent (user: User): boolean {
  return !!user.roles.find((r) => r === 'Student');}
}
```

Not that easy, right? Whitespace, sometimes referred to as a separator, is one of the five different types of tokens that exists in any programming language. When good whitespaceing is used, reading code becomes significantly easier.

Five different types of programming tokens:

1. Keywords are reserved words: function, class, const.
2. Identifiers are what we are allowed to use as class, variable, and method names: user, userOne, userModel.
3. Operators are tokens that enable us to do logical operations: assignment (=), addition (+), and subtraction (-). **4. Separators create whitespace in our code: tabs, spaces, and newlines.**
4. Literals are integers, decimals, and strings.

Here are a few ways to use whitespace well.

Use obvious spacing rules

How to use spacing is almost never explicitly taught, but it's one of the first things we figure out when we start programming.

You're likely aware of the obvious spacing rules. For most languages, when we start learning, we're shown examples with spaces in between keywords, identifiers, operators, and literals.

We're also shown to use *horizontal indentation* when we step inside of a class, method, function or block (in general).

These are the basic spacing rules.

Try to stick to 'em.

Bad horizontal spacing

```
const x=12;

const user={name:"khalil"}

class Employer {
    public update(details:CompanyDetails):Result<UpdateResult>{
        ...
    }
}
```

Good horizontal spacing

```
const x = 12;

const user = { name: "khalil" }

class Employer {
    public update (details: CompanyDetails): Result<UpdateResult> {
        ...
    }
}
```

Using horizontal spacing helps to delineate token types. Each line of code is slightly easier to read. These small acts of care for the code compound over an entire codebase.

Horizontal indentation also makes it easier to visualize the *scope* of a method, class, or function. Scope is easy to see by comparing how statements sit up and down the Y-axis.

Keep code density low

Code density is a measurement of how many lines of code go without a line break.

Line breaks are like commas in English. Both signal a resting point, another step, or a separate thought being expressed. Line breaks help to make code easier to digest.

Here's an example from Apollo GraphQL's open-source RESTDataSource API.

```
export class HTTPCache {
    ...
    async fetch(
        request: Request,
        options: FetchOptions = {},
    ): Promise<Response> {
```

```

/**
 * 1. Create the cache key. You can either supply a cache key or leave it blank
 * and Apollo will use the URL of the request as the key.
 */

const cacheKey = options.cacheKey ? options.cacheKey : request.url;

/**
 * 2. Using that key, see if the cache has the value already.
 */

const entry = await this.keyValueCache.get(cacheKey);

/**
 * 3. If it doesn't already have the response, we'll need to
 * get it, store the response in the cache, and return the
 * response.
 */
if (!entry) {
  const response = await this.httpFetch(request);

  const policy = new CachePolicy(
    policyRequestFrom(request),
    policyResponseFrom(response),
  );

  return this.storeResponseAndReturnClone(
    response,
    request,
    policy,
    cacheKey,
    options.cacheOptions,
  );
}

/**
 * 4. Returns the object from the cache (respecting any
 * cache invalidation policies).
*/
...
}
}

```

You don't really have to fully understand what this code is doing, but line breaks help make it easier to digest and comb through potentially challenging logic.

It's also good when functions or class methods are kept small. Line breaks between each method are not only conventional, but help readability.

chatEvents.ts

```
export class ChatEvents {

    public subscriptions: any = {};

    public registerSubscriberForEvent (
        eventName: ChatEvent,
        subscriptionName: string,
        cb: SubscriptionCallback
    ): void {
        this.createEventKeyIfNotExists(eventName);
        this.addEventSubscription(eventName, subscriptionName, cb);
    }

    private createEventKeyIfNotExists (eventName: ChatEvent): void {
        const exists = this.subscriptions.hasOwnProperty(eventName);
        if (!exists) {
            this.subscriptions[eventName] = {};
        }
    }

    private addEventSubscription (
        eventName: ChatEvent,
        subscriptionName: string,
        cb: SubscriptionCallback
    ): void {
        this.subscriptions[eventName][subscriptionName] = cb;
    }
}
```

Break horizontally when necessary

Sometimes lines get a little too long to read. I advise to strategically break your code horizontally when it surpasses an appropriate line length (most developers use line lengths from 80 to 120 characters long).

Bad horizontal breaking

```
export class UpvotePost implements UseCase<UpvotePostDTO, Promise<UpvotePostResponse>> {
    ...
    constructor (memberRepo: IMemberRepo, postRepo: IPostRepo, postVotesRepo: IPostVotesRe...
```

```
}
```

Good horizontal breaking

```
export class UpvotePost implements UseCase<
    UpvotePostDTO,
    Promise<UpvotePostResponse>
> {
    ...
    constructor (
        memberRepo: IMemberRepo,
        postRepo: IPostRepo,
        postVotesRepo: IPostVotesRepo,
        postService: PostService
    ) {
        this.memberRepo = memberRepo;
        this.postRepo = postRepo;
        this.postVotesRepo = postVotesRepo
        this.postService = postService;
    }
}
```

Breaking your code this way prevents others with smaller monitors from needing to stop to scroll horizontally in order to read your code.

Prefer smaller files

In Uncle Bob's research on Clean Code, he discovered that the average file size across several enterprise Java projects were **200 to 500 lines long**.

Smaller files are generally easier to read and maintain. Less code in a file means less to read.

If there's less to read, there's less to understand, and the surface area of getting confused about what the file **does** (separation of concerns), and what it's **responsible for** (single responsibility) is smaller.

That is, smaller files are an indication that good Separation of concerns and Singular Responsibility were implemented.

Consistency

Readability truth #2 - Consistency helps readers build comprehension momentum

As you grow acclimated to a new project, your ability to read code and understand how things work should increase exponentially.

Humans pick up on patterns. It's how we make sense of the world. It's how we learn and build momentum. Without consistency, we can't identify patterns, and we certainly can't build momentum.

Capitalization

The way we use capitalization in programming is strategic.

There are three primary capitalization conventions:

- Pascal case, which LooksLikeThis
- Camel case, which looksLikeThis, and
- Underscores, which looks_like_this

Capitalization provides additional information about identifiers. For example, if we use camel casing for variables and methods, and pascal case for classes and namespaces, we can quickly figure out when what type of construct we're looking at just by the casing alone.

That's your internal pattern matching at work.

In JavaScript and TypeScript, camelCase is preferred for variables, functions, and class members, while PascalCase is preferred for everything else like class names, namespaces, types, and interfaces.

```
// Type names are pascal-cased
type User = {
    // Type members are camel-cased
    id: string;
    name: string;
}

// Class names are pascal-cased
class UserModel {

    // Accessors are camel-cased
    get id (): string {
        return this.props.id;
    }

    get name (): string {
        return this.props.name;
    }

    ...
}

// Variables and functions are camel-cased
const shouldListenToJohnMaus = likesWeirdMusic()
    && isGenerallyAHappyPerson();

type CanPlaySynth = {
    // Class, type, or interface members are camel-cased.
    favouriteSynth: Synth;
}
```

In other languages, like Python, the rule for everything is that “function and variable names should be lowercase, with words separated by underscores”.

This means that Python code often looks much different than TypeScript or JavaScript code, where underscore casing is rarely used.

```
user_name = "Khalil";
```

In C#, methods are pascal-cased.

```
user.ResetPassword(newPassword);
```

In the end, it doesn’t matter *what* you choose as your capitalization rules, so long as you and your team stick to it.

Bad (TypeScript):

```
const DAYS_IN_WEEK = 7;
const daysInMonth = 30; // Not consistent

const songs = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];
const Artists = ['ACDC', 'Led Zeppelin', 'The Beatles']; // Not consistent

function eraseDatabase() {}
function restore_database() {} // Not consistent

type animal = { /* ... */ }
type Container = { /* ... */ } // Not consistent
```

Good (TypeScript):

```
const DAYS_IN_WEEK = 7;
const DAYS_IN_MONTH = 30;

const SONGS = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];
const ARTISTS = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {}
function restoreDatabase() {}

type Animal = { /* ... */ }
type Container = { /* ... */ }
```

Whitespace rules

Ever worked on a project where you were able to tell who wrote the code you’re looking at **just by the way it was formatted?** That’s not normally a good thing.

If one developer’s code editor adds 4 spaces when they hit the TAB, and another adds 2 spaces when *they* hit TAB, you’ll start to resent any developer writing code in files you’ve previously written code, with them messing up your beautiful formatting.

Who wants to spend time and effort cleaning up the living room if someone is just going to track mud everytime they walk through the house?

As a code writer, this is annoying- and if you care about the code, you'll work to fix it. That's time away from doing meaningful work.

As a code reader, it's distracting.

You want to get this fixed and out of the way as soon as possible on any project. Luckily, this is something that can be addressed with the proper tooling. See #3-2-1-2 - Tooling to enforce conventions for how to remedy this.

Storytelling

Readability truth #3 - People lose interest and get confused if the most important details aren't provided up front

Newspaper Code and the Step-down Principle

In a traditional story, we take the reader through a journey by setting up the scene, introducing them to the characters, then posing the conflict.

We don't want to do that with code. Instead, let's get to the interesting stuff **right away**.

The *Newspaper Code Principle* says to **front-load** a file with the **most important things**. By putting the most essential things that we want the reader to know about *first*, and moving the less critical (yet likely still important) details towards the bottom, readers can learn the primary reason why the class exists in the first place, much quicker.

Here's an example of a RecordingStudio class.

```
export class RecordingStudio {  
  
    private bandroom: Bandroom;  
    private metronome: Metronome;  
    private controls: Controls;  
  
    constructor (...) {  
        ...  
    }  
  
    // OK, this is the first thing I'm seeing... I'm not sure how this  
    // is useful to me as a reader. Also, it's private. It's most likely  
    // a detail that is used somewhere else. Onwards...  
    private getDemoFromLibrary (demoNameQuery: string, artist: Artist): Demo {  
        return this.demoLibrary.find(demoNameQuery, artist);  
    }  
  
    // Another private method. Seems like another internal detail.  
    private recordVocals (demo: Demo): void {  
        ...  
    }  
}
```

```

}

// Yet another internal detail.
private prepareInstrument (instrument: Instrument): void {
    switch (instrument.class) {
        case Guitar.class:
        case Bass.class:
            instrument.tune();
            instrument.setTone();
        case Drum.class:
            instrument.replaceDrumHeads();
            instrument.tune();
        default:
    }
}

private assembleMusiciansForInstruments (instruments: Instrument[]): void {
    instruments.forEach(
        (instrument) => this.bandroom.callMusicianFor(instrument)
    );
}

private masterDemo (demo: Demo): void {
    ...
}

private mixLevels (demo: Demo): void {
    ...
}

// Ah, finally - something that clients can call. This is probably
// what this class is for... I just wish I didn't have to get to the
// bottom of the file and read everything else in order to find it.
public async recordSong (demoNameQuery: string, artist: Artist): void {
    const demo = this.getDemoFromLibrary(demoNameQuery, artist);

    const instruments = demo.getInstruments();
    for (let instrument of instruments) {
        this.prepareInstrument();
    }
    this.metronome.setBpm(demo.bpm);
    this.assembleMusiciansForInstruments(instruments);
    this.controls.startRecording();
    await this.bandroom.performSong(demo);
    this.controls.stopRecording();
    await this.recordVocals(demo);
}

```

```
        await this.mixLevels(demo);
        await this.masterDemo(demo);
    }
}
```

By *convention*, when we use classes, the first two things that appear at the top of the class are the class member variables and the constructor. That's convention, so we usually don't mess with that.

Now, after that (the constructor and member variables, that is), in the example provided, we're looking at private methods that appear to be responsible for low-level details. To speed up the reader's ability to understand the usefulness of this class and why it exists, we should promote the most important method, the public recordSong method, to the top.

recordSong should be higher up not only because it is more important, but because it is publically exposed to the client. For someone reading this class, within the first couple of seconds, we should aim to let them know what the most important methods are, and recordSong is important because it is what the client will call in order to kick off the process.

Let's improve this by moving recordSong closer to the top of the file.

```
export class RecordingStudio {

    private bandroom: Bandroom;
    private metronome: Metronome;
    private controls: Controls;

    constructor (...) {
        ...
    }

    public async recordSong (demoNameQuery: string, artist: Artist): void {
        const demo = this.getDemoFromLibrary(demoNameQuery, artist);

        const instruments = demo.getInstruments();
        for (let instrument of instruments) {
            this.prepareInstrument();
        }
        this.metronome.setBpm(demo.bpm);
        this.assembleMusiciansForInstruments(instruments);
        this.controls.startRecording();
        await this.bandroom.performSong(demo);
        this.controls.stopRecording();
        await this.recordVocals(demo);
        await this.mixLevels(demo);
        await this.masterDemo(demo);
    }
}
```

```

private getDemoFromLibrary (demoNameQuery: string, artist: Artist): Demo {
    return this.demoLibrary.find(demoNameQuery, artist);
}

private recordVocals (demo: Demo): void {
    ...
}

private prepareInstrument (instrument: Instrument): void {
    switch (instrument.class) {
        case Guitar.class:
        case Bass.class:
            instrument.tune();
            instrument.setTone();
        case Drum.class:
            instrument.replaceDrumHeads();
            instrument.tune();
        default:
    }
}

private assembleMusiciansForInstruments (instruments: Instrument[]): void {
    instruments.forEach(
        (instrument) => this.bandroom.callMusicianFor(instrument)
    );
}

private masterDemo (demo: Demo): void {
    ...
}

private mixLevels (demo: Demo): void {
    ...
}
}

```

That's an improvement, but what do we do with the rest of the methods? Do we just leave them where they are?

The *Newspaper principle* suggests to re-order these so that the least important details are towards the bottom, so we may be able to leave this if we choose to follow that principle.

Alternatively, the *Stepdown Principle*, which pairs nicely with the *Newspaper principle*, says that we should **organize code so that we can read it from top-to-bottom**. Easier said than done, my friends.

To accomplish this, we have to make method callers and callees close together. If a method calls another method, the callee should be directly below it. This would emulate the effect

of reading a book.

I have found this to be more challenge than it's worth since it forces you to refactor your methods in an unnatural way.

You could give it a try, but if you *are* going to implement it, consider the CQS principle and try to stick to it as you refactor.

Maintaining a consistent level of abstraction

Sometimes you'll notice that there's a mismatch with the level of *abstraction* and *details* within a method.

Check this out. In a refactored version of the previous example, after we get the `demo` object, the set of instructions that follows seems a lot more detail-oriented than the set of instructions that happens towards the end of the method.

```
export class RecordingStudio {  
  ...  
  public async recordSong (demoNameQuery: string, artist: Artist): void {  
    const demo = this.getDemoFromLibrary(demoNameQuery, artist);  
  
    // The level of abstraction here...  
    const instruments = demo.getInstruments();  
    for (let instrument of instruments) {  
      this.prepareInstrument();  
    }  
    this.metronome.setBpm(demo.bpm);  
    this.assembleMusiciansForInstruments(instruments);  
    this.controls.startRecording();  
    await this.bandroom.performSong(demo);  
    this.controls.stopRecording();  
  
    // ... is different from the level of abstraction here  
    await this.recordVocals(demo);  
    await this.mixLevels(demo);  
    await this.masterDemo(demo);  
  }  
}
```

This leads us into another convention you could implement.

Code should descend in abstraction towards lower-level details

We can group those first few operations as `recordMusicFromDemo`, maintaining a similar level of abstraction from within `recordSong` method and leaving the details to live within each respective method for further decomposition.

```
export class RecordingStudio {  
  ...
```

```

public async recordSong (demoNameQuery: string, artist: Artist): void {
    const demo = this.getDemoFromLibrary(demoNameQuery, artist);
    await this.recordMusicFromDemo(demo);
    await this.recordVocals(demo);
    await this.mixLevels(demo);
    await this.masterDemo(demo);
}
}

```

It will almost never be perfect, but just attempting to optimize for readability is empathetic and small efforts add up over time.

Keeping related methods close to each other

Sometimes there are methods or functions that just belong together. For example, take this getter/setter pair broken up by a logout method.

Bad

```

export class Member extends Aggregate<MemberProps> {

    ...

    get username (): Username {
        return this.props.username;
    }

    // Breaks relationship between getter/setter above and below
    public logout (): void {
        this.addDomainEvent(new UserLoggedOut(this));
    }

    set username (username: string): void {
        const newUserNameResult: Result<Username> = Username.create(username);
        if (newUserNameResult.isSuccess()) {
            this.addDomainEvent(new UsernameChanged(this));
            this.props.username = newUserNameResult.getValue();
        }
    }
}

```

When there is an inherent grouping between related methods, strive to keep them close to each other.

Good

```

export class Member extends Aggregate<MemberProps> {

    ...

```

```

get username (): Username {
    return this.props.username;
}

set username (username: string): void {
    const newUserNameResult: Result<Username> = Username.create(username);
    if (newUserNameResult.isSuccess()) {
        this.addDomainEvent(new UsernameChanged(this));
        this.props.username = newUserNameResult.getValue();
    }
}

public logout (): void {
    this.addDomainEvent(new UserLoggedOut(this));
}
}

```

In summary, be empathetic that we're writing code for humans and that we all have constraints. When you format code,

- Use whitespace appropriately
- Be consistent with formatting
- Use storytelling to put the most important things first and logically group code that belongs together

Enforcing formatting rules with tooling

Now. How the heck do we enforce these guidelines? What do we do when we're on a team with 20+ developers? How do we get everyone to be on the same page?

We use tooling.

The current trifecta of tooling for formatting in the JavaScript/TypeScript community is:

- ESLint (over TSLint)
- Prettier
- Husky

ESLint

ESLint is a JavaScript linter that enables you to enforce a set of style, formatting, and coding standards for your codebase. It looks at your code, and tells you when you're not following the standard that you set in place.

For example, if I wanted to check my code to make sure that there were no `console.log` statements, I could use an ESLint rule to enforce that.

`eslint.rc`

```
{  
  "root": true,  
  "parser": "@typescript-eslint/parser",  
  "plugins": [  
    "@typescript-eslint",  
    "no-loops"  
,  
  ],  
  "extends": [  
    "eslint:recommended",  
    "plugin:@typescript-eslint/eslint-recommended",  
    "plugin:@typescript-eslint/recommended"  
,  
  ],  
  "rules": {  
    "no-console": "error" // you can also use the int value "2"  
  }  
}
```

And with an npm script in my package.json for the project, I could add a `lint` command:

package.json

```
{  
  "scripts": {  
    ...  
    "lint": "eslint . --ext .ts",  
  }  
}
```

And run it with:

```
npm run lint
```

Depending on how I've configured my rules, I will either see nothing, a *warning*, or an *error* in my console.

In ESLint, you can set your rules to be either `off`, `warn`, or `error`.

- “`off`” means `0` (turns the rule off completely)
- “`warn`” means `1` (turns the rule on but won’t return a non-zero exit code)
- “`error`” means `2` (turns the rule on and will return a non-zero exit code)

ESLint becomes more useful when we combine it with Prettier and Husky.

Prettier

What if we don’t want to have to remember to run the linter everytime we write new code? What if there was a way that we could, while coding, have it *automatically* format things based on our conventions?

Prettier is an opinionated (yet fully configurable) code formatter. ESLint can *kind of* format code too, but ESLint is mostly intended to simply sniff out when we’re not following the

mandated coding conventions.

Prettier can be configured to format your code (ie: make it look *prettier* 😊) after you save a file or manually tell it to. By default, it comes configured with a set of common code cleanliness rules.

With ESLint and Prettier,

- ESLint **defines the code conventions**
- Prettier **performs the auto-formatting** based on the ESLint rules in the config.

Prettier can either be installed as VSCode plugin, or configured to format your code via the command line.

You can set and enforce rules like setting a `max printWidth` and deciding on if `trailingCommas` are allowed by not by installing `prettier` as a dev dependency, then writing a `.prettierrc` config file.

```
npm install --save-dev prettier && touch .prettierrc
```

`.prettierrc`

```
{  
  "semi": true,  
  "trailingComma": "none",  
  "singleQuote": true,  
  "printWidth": 80  
}
```

There's a little bit of configuration involved to get Prettier to look to ESLint for the rules. You can learn about how to do that in this short guide I put together.

The last piece in the puzzle is Husky.

Husky

Husky is an npm package that “makes Git hooks easy”.

When you initialize Git (the version control tool that you’re probably familiar with) on a project, it automatically comes with a feature called hooks.

If you go to the root of a project initialized with Git and type:

```
ls .git/hooks
```

You’ll see a list of sample hooks like `pre-push`, `pre-rebase`, `pre-commit`, and so on. This is a way for us to write plugin code to execute some logic before we perform the action.

If we wanted to ensure before someone creates a commit using the `git commit` command, that their code was properly linted and formatted, we could write a `pre-commit` Git hook.

Writing that manually isn’t a task for the faint of heart. It would also be a challenge to distribute and ensure that hooks were installed on other developers’ machines.

These are some of the challenges that Husky aims to address.

With Husky, we can ensure that for a new developer working in our codebase (using at least Node version 10):

- Hooks get created locally
 - Hooks are run when the Git command is called
 - Policy that defines how someone can contribute to a project is enforced.
-

Therefore,

- ESLint **defines the code conventions**
- Prettier **performs the auto-formatting** based on the ESLint rules in the config.
- Husky **ensures that the formatting scripts get applied** before any code makes its way into source control.

In summary, don't spend time fumbling around formatting rules in PRs when you can use tooling to enforce 'em.

Read the **Clean Code Tooling series** here: 1. How to use ESLint with TypeScript 2. How to use Prettier with ESLint and TypeScript in VSCode 3. Enforcing Coding Conventions with Husky Pre-commit Hooks

Project planning

Notes

This is a section that has a lot of the content already written about in How to plan a new project.

How do people plan projects?

- A lot of times, developers don't *really* plan — they just start building out the API
 - API first
 - * This is when we start building out the API endpoints and then start querying them.
 - Benefits:
 - We're starting **imperatively**. We're actually starting out by building out something.
 - Drawbacks
 - It's easier to miss small nuances about the
 - Works well for:
 - CRUD apps
 - Doesn't work well for
 - Apps with domain logic complexity
 - Database first

Section Three - Skills & knowledge

Summary

Theoretical knowledge about software design and architecture which influences our preferred coding conventions.

Skills & knowledge

- Coding conventions can be implemented blindly. But when we hold a deep understanding of the implications of what we're doing positively impacts the *structural* quality of the code, we can adapt to new situations, scenarios, code in new programming languages, understand the implications of using new technologies, detect anti-patterns & code-smells, etc.
- This means learning the patterns, principles,

Write a summary so that I can put it at the start of the chapter

- At this point, it's about the skills and knowledge.
 - Understandability, flexibility, maintainability.
- When code is understandable, flexible, maintainable, projects get completed and code can live a long and healthy life.
 - Craftspeople are interested in arming themselves with knowing how to keep each of these pillars strong.
 - They want to know best practices, but they also know when to break.
 - * You have to know the rules first to know when to break 'em.
 - * They know the impact it's going to have when they break 'em.
- Where to go from here?
 - The software design and architecture roadmap. Clean code is the start. You want to invest in the right mindset (empathy, growth mindset), invest in setting up your codebase conventions (rules, create a sense of consistency with your team), and then continue to acquire as much knowledge about software design as possible.
 - There is a pattern, architectural style, tool, approach or methodology for most problems right now.
 - It's remarkable how much has stayed the same over the past 40 years of software design.
 - And those who know not of history are also doomed to repeat their mistakes.

Details

Tools in your toolbox

Best practices, principles, and patterns

Software Quality

Code Smells

Anti Patterns

Optimizing code

Absolute and relative complexity

Tools in your toolbox

Infra

Know tools to deploy websites

Know a scripting language

Backend development

Know a general purpose language

Know a SQL database

Know an ORM

Know a noSQL database

Frontend development

Know a state management library

Know a view-layer library or framework

Know a CSS pre-processor

Software development approaches

Know how to gather requirements and plan a project

Know how to write tests

Know how to consistently provide value

Best practices, principles, and patterns

Software Quality

Todo: Explain that software quality can be understood in two strategic ways.

When we talk about the quality of software, we're talking about either **structural quality** or **functional quality**.

- **Structural quality** - can be formally measured
- **Functional quality** - can only be measured through testing

Structural quality

Structural quality describes the attributes about software that can be formally measured.

Application Architecture Standards

- Multilayer design compliance (UI vs App Domain vs Infrastructure/Data)
- Data access performance
- Coupling Ratios
- Component (or pattern) reuse ratios

Coding Practices

- Error/exception handling (all layers UI/Logic/data)
- If applicable - compliance with OO and structured programming practices
- Secure controls (access to system functions, access controls to programs)

Complexity

- Transaction
- Algorithms
- Programming practices (eg use of polymorphism, dynamic instantiation)
- Dirty programming (dead code, empty code...)

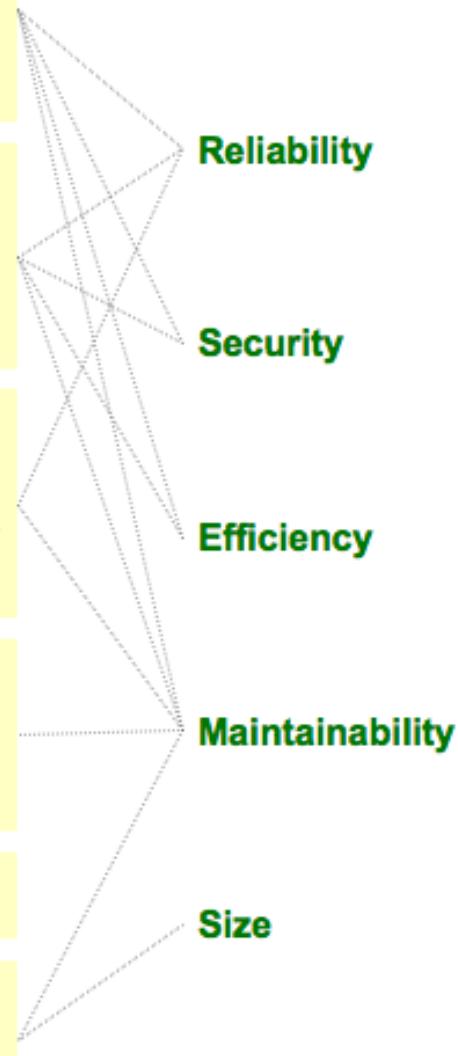
Documentation

- Code readability and structuredness
- Architecture -, program, - and code-level documentation ratios
- Source code file organization

Portability: Hardware, OS and Software component and DB dependency levels

Technical and Functional Volumes

- # LOC per technology, # of artifacts, files
- Function points - Adherence to specifications (IFPUG, Cosmic references..)



Relationship between software desirable characteristics (right) and measurable attributes (left).

CC BY-SA 3.0, <https://en.wikipedia.org/w/index.php?curid=32370588>

Software quality

Report on programming errors

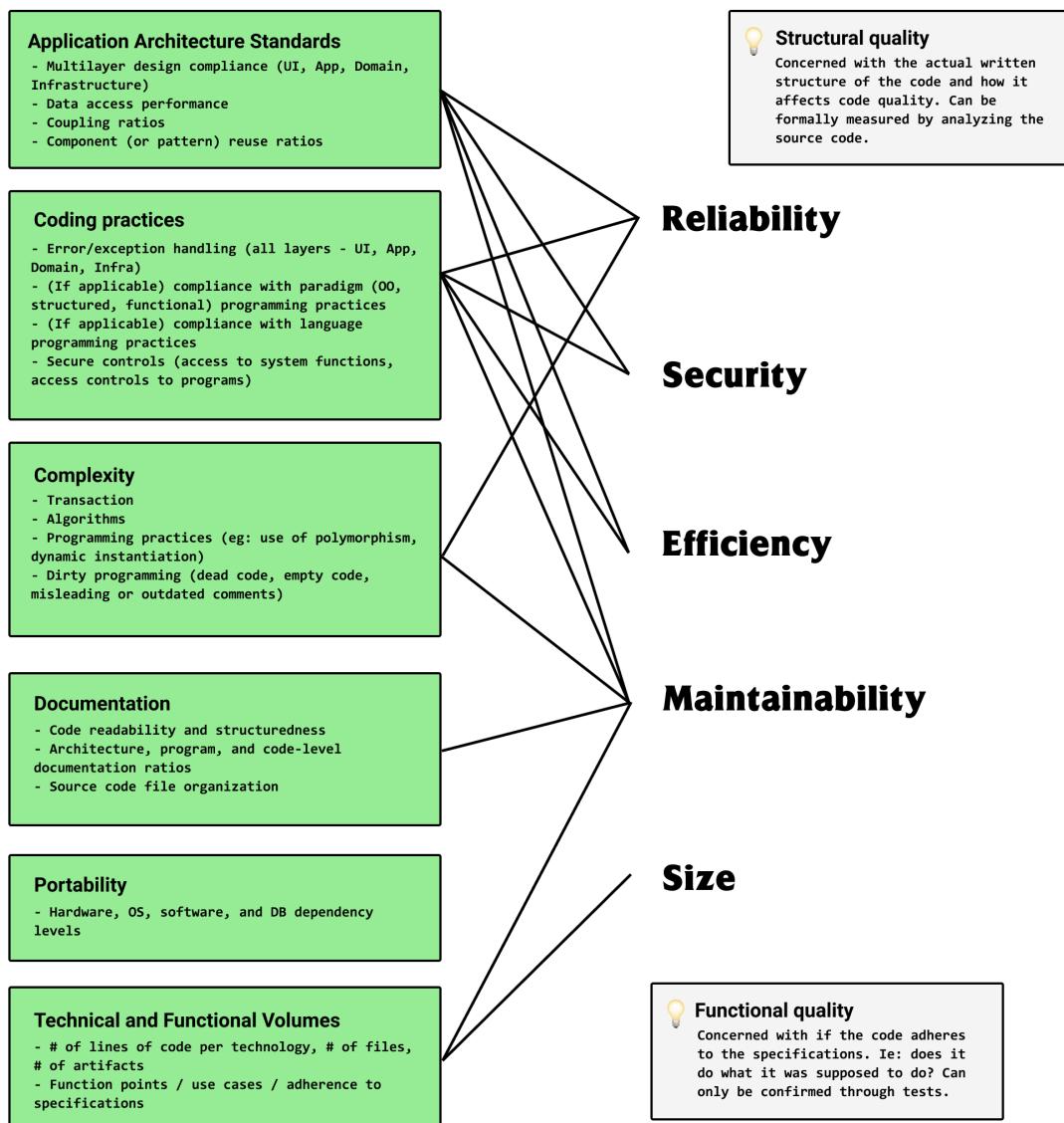
“Correlations between programming errors and production defects unveil that basic code errors account for 92% of the total errors in the source code. These numerous code-level issues eventually count for only 10% of the defects in production. Bad software engineering

practices at the architecture levels account for only 8% of total defects, but consume over half the effort spent on fixing problems, and lead to 90% of the serious reliability, security, and efficiency issues in production.”

How to measure (structural) Software quality

@stemmlerjs

CC BY-SA 3.0, <https://en.wikipedia.org/w/index.php?curid=32370588>



Software quality thread

<https://twitter.com/stemmlerjs/status/1266380004293783553>

Points:

- Most of the time, the way that junior developers interpret the quality of software has

to do with coding conventions.

```
// An example of the discussion that junior developers often have.

// This is weird - don't do it
if (!!id === false) { }

// Do this instead
if (!id) { }
```

- That's fine, we can have these discussions too - but there are much more meaningful things to discuss as well, such as *Architecture* - which is very often missing almost entirely.

Code Smells

No, a *code smell* is not the smell of your laptop overheating while waiting for your Java code to compile.

Our industry describes a *code smell* as **some characteristic of our code that is not ideal** and typically signifies that there may be a larger problem at hand.

For example, consider this method with the *Too many parameters* code smell.

```
export class PaymentCalculator {

    public calculatePayment (
        userType: UserType,
        user: User,
        currency: Currency,
        hours: Hours,
        salary: Salary,
        isContractor: boolean,
        isEmployee: boolean,
        isAdmin: boolean
    ): PaymentDetails {
        ...
    }
}
```

Not only is this method awkward to read, and testing is probably a nightmare, but the amount of parameters it has provides a really good signal that there's probably an issue with our design.

A common refactoring for something like this is to refactor the list of parameters into an object like this:

```
type CalculatePaymentConfig = {
    userType: UserType,
    user: User,
    currency: Currency,
```

```

hours: Hours,
salary: Salary,
isContractor: boolean,
isEmployee: boolean,
isAdmin: boolean
}

export class PaymentCalculator {

    public calculatePayment (
        config: CalculatePaymentConfig
    ): PaymentDetails {
        ...
    }
}

```

This works, though someone looking at this might also sniff out the Data clump code smell.

A better refactoring would be to refactor the method into smaller ones so that each method is responsible for a specific part of the entire algorithm. By the end, we may end up realizing that the parameters of the method (like UserType, User, and Currency) need to be remodeled in a way that makes more sense to the domain.

The funny thing about *code smells* is that they **aren't bugs**. It's not blaringly obvious that they're hurting the design, and they don't stop the code from compiling either. It's not like the house is burning down (yet)- you can just smell something funny coming from the kitchen.

If not addressed, *code smells* can tally up, and turn an entire codebase into an unmaintainable pile of stinky laundry. It's at this point that developers **are stopped** from being able to make future progress on a project.

Catalog of code smells: There are tons of code smells. Familiarize yourself with some of these here. Code smells are organized into three groups: Application-level smells, Class-level smells, and Method-level smells.

A code smell is an unideal (though not terminal) characteristic of our code that signifies there may be a larger design problem at hand

Code smells depend on the language, context, and developer

Each language comes with its own set of conventions to follow. For example, the Java programming language has it's own documented set of conventions.

Coding Conventions: Conventions are guidelines for a language which makes recommendations against things like style, file organization, naming conventions, best practices, and handling whitespace. The second part of this chapter presents the most common, language-agnostic coding conventions.

In untyped languages like JavaScript and Python, underscores are conventionally used to signal to other developers that a variable should be used internally only.

user.js

```
function User (username, firstname, lastname) {  
    this.userId = uuid();  
    this.firstname = firstname;  
    this.lastname = lastname;  
    this._hash = createUniqueObjectHash(); // internal use only, please  
}
```

In typed languages like Java and TypeScript, usage of the *underscore* convention is discouraged because we have access to *scope modifiers* like private and protected. We should use the language constructs, given their availability, and look to conventions when a common task cannot be done with the language.

In some languages and frameworks, **the convention may actually be a requirement**. For example, some React/JavaScript developers follow the convention that you should have *one component per file*. This isn't necessary at all, and aggressive usage may actually lead to the *Lazy Class* code smell (a class that does too little). Meanwhile, in Java, the compiler actually throws an error if it finds *more than one public class per file*.

Some languages also have language constructs that while we *can* make use of them, **it is recommended to limit (or abstain from) using**. In JavaScript, the eval() function enables you to evaluate JavaScript code as a string. Most developers will advocate against using eval() just because of the huge security risk it poses (it's quite easy for a malicious actor to get bad JS code to run).

Similarly, JavaScript has the ability to perform *Labeled Breaks* within control blocks. This is essentially the same thing as the goto statement, shunned for making it too hard to debug and trace.

```
loop: for (let y = 0; y < 3; y++) {  
    switch (y) {  
        case 0:  
            console.log(0)  
            break  
        case 1:  
            console.log(1)  
            break loop  
        case 2:  
            console.log(2)  
            break  
    }  
}
```

Labeled breaks with JavaScript. This prints out "0 1". Not recommended because of its reminiscence to GOTO statements, abolished from most programming languages.

Depending on the context or domain, some code smells may actually be ideal. For example, in microcontroller development, it's often preferred to have all of the code in a single file or a maintain a few large files. This is the God object code smell (read about it [here](#)), and normally considered bad practice, but in microcontroller development, you may be able to get

a performance increase this way, and performance is more important than maintainability and programming elegance in the context of microcontrollers.

Anti Patterns

An *anti-pattern* is an approach to some commonly occurring problem in software that we deem to be *bad*. The chosen approach is in some way ineffective, messy, or problematic to the design.

Anti-patterns and *code smells* are similar in definition, though some believe a *code smell* is something done unwillingly or without much thought behind how it impairs the design, where some believe an *anti-pattern* to be much more **deliberate** in its intent to (unfortunately, poorly) solve a problem.

Anti-patterns are usually a full-blown approach to solving a problem that has very negative consequences. Here are a few examples.

Soft-code (anti-pattern): Storing business logic in configuration files rather than source code. You know about hard-coding right? That's when you manually write the values in for commonly changed variables. Not ideal. However, abstracting too much turns into soft code. With the advent of companies trying to make coding obsolete with no code, this sounds like something that definitely happened in the real-world somewhere. Someone, somewhere, is trying to put their entire application in a YAML file. When the configuration files become the source code, and the source code becomes an abstraction, we run into problems. Too much abstraction makes understanding the code and maintaining it considerably harder.

Interface bloat (anti-pattern): Making an interface so robust that it is challenging to implement. This is a funny one. In typed languages, we use interfaces to create a contract. If we wrote an interface for a IWarrior, we get the opportunity to specify all the attributes and methods that deem an object to be an IWarrior.

Typically, we try to keep interfaces relatively small. If we have stuff related to how the warrior eats, hunts, or even his apparel, it's good practice to extract that responsibility into separate interfaces. It's preferable to have several interfaces with single responsibility, instead of one big one. Now think about how you'd feel if you knew you needed to create a IWarrior implementation, and there were 100 attributes and methods for you to implement in order to create it... Mad. You'd feel mad. "Why do I need to define what this warrior's favourite tv show is? Why the hell is this a part of the IWarrior interface?"

Anti-patterns are not determinate

What *is* and what *is not* an *anti-pattern* is not determinate.

Depending on the *programmer, the project*, and the software development methodology / approach, there are scenarios when a pattern is a vice in one context, yet a strategic saving grace in another.

Anemic Domain Models vs. Entity Component System

One of my favourite anti-patterns to bring attention to is the Anemic Domain Model. An *Anemic Domain Model* is something that develops when all of the business logic for our models *doesn't actually live in the model*, but instead lives in things like *services*, *controllers*, or *helper* classes.

Consider the example where you have a User model and a UserService. The UserService has methods called createUser and updateUser that each do validation and encapsulates any domain logic that dictates how someone may create or update a user.

user.ts

```
export class User {  
    public id: string;  
    public name: string;  
    public email: string;  
  
    constructor (id: string, name: string, email: string) {  
        this.id = id;  
        this.name = name;  
        this.email = email;  
    }  
}
```

userService.ts

```
export class UserService {  
    ...  
  
    public createUser (userFields: UserFields): User {  
        // Validation logic here  
        if (  
            !this.isValidUserName(userFields.name) ||  
            !this.isValidEmail(userFields.email)  
        ) {  
            throw new Error("Invalid fields for user");  
        }  
  
        return new User(this.createId(), userFields.name, userFields.email);  
    }  
  
    public updateUser (userFields: UserFields, user: User): void {  
        // Same validation logic here  
        if (userFields.hasOwnProperty('name') && this.isValidUserName(userFields.name)) {  
            user.name = userFields.name;  
        }  
  
        if (userFields.hasOwnProperty('email') && this.isValidEmail(userFields.email)) {  
            user.email = userFields.email;  
        }  
    }  
}
```

```
    }  
}
```

What's stopping someone from circumventing the `UserService` entirely and just creating a `User` using `new User({})`? Nothing, really. That's because the **operations against the model** are separate from the **model itself**. Therefore, the `User` model is *anemic*. It doesn't really have any real purpose.

You don't want an *Anemic Domain Model* because it leads to writing a lot of duplicate code. I want the code that does the validation logic against my `User` to live *in the* `User`, not in `UserService`. How am I supposed to remember to write the validation logic for `User` in every single method that changes the `User` in `UserService`?

This bad design is remedied by using a *Domain Model*, which we'll explore in detail in Chapter 5.

It turns out that there's a context where it's actually *advantageous* to have an *Anemic Domain Model*. That context is game programming.

With web development, we don't want an *Anemic Domain Model*. However, in game programming, having the **model** separate from the **operations that you can perform against a model** is ideal. That principle is formalized as Entity Component System which favors composition over inheritance.

Have you ever played a game called Garry's Mod? In it, you can *be* just about anything. There's a popular game mode called *Prop Hunt* where the *hiders* need to find any prop on the map (car, flower, plant, box, shoe, you name it), select it to *become* it, and hide so that the *seekers* don't find them. The goal is to look so inconspicuous that no one finds you.

When you start out, you're a *human*, and you're able to *move*. When you find your special prop and become it, you can still *move*, though you move at the same speed as when you were a *human*.

In this case, the *behaviour* of *moving* is abstracted away from the *human*, *box*, *shoe*, etc model. That logic is not part of the model. Isn't that bad?

Not in this case! Because new items are constantly getting added, this architecture enables game developers to satisfy new maps with new props by simply creating the models, without having to worry about *how* each model works (that can be decided upon later).

Anti-patterns are usually full-blown approaches to solving a problem. They're usually the wrong approach. It's an anti-pattern when choosing the wrong approach has negative long-term consequences.

DRY vs Overengineering

Dan Abramov, creator of Redux and engineer at Facebook, released a blog post called "Goodbye, Clean Code". In it, he tells the story about how his engineer checked in some code that had a lot of repetition in it. Here's the code.

```

let Rectangle = {
    resizeTopLeft(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeTopRight(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeBottomLeft(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeBottomRight(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
};

let Oval = {
    resizeLeft(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeRight(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeTop(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeBottom(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
};

let Header = {
    resizeLeft(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeRight(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
};

let TextBlock = {
    resizeTopLeft(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeTopRight(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
};

```

```

    resizeBottomLeft(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
    resizeBottomRight(position, size, preserveAspect, dx, dy) {
        // 10 repetitive lines of math
    },
};

```

To remove the repetition, Dan grouped the code together and abstracted its parts like the following.

```

let {top, bottom, left, right} = Directions;

function createHandle(directions) {
    // 20 lines of code
}

let fourCorners = [
    createHandle([top, left]),
    createHandle([top, right]),
    createHandle([bottom, left]),
    createHandle([bottom, right]),
];

let fourSides = [
    createHandle([top]),
    createHandle([left]),
    createHandle([right]),
    createHandle([bottom]),
];

let twoSides = [
    createHandle([left]),
    createHandle([right]),
];

function createBox(shape, handles) {
    // 20 lines of code
}

let Rectangle = createBox(Shapes.Rectangle, fourCorners);
let Oval = createBox(Shapes.Oval, fourSides);
let Header = createBox(Shapes.Rectangle, twoSides);
let TextBox = createBox(Shapes.Rectangle, fourCorners);

```

Alright. Over to you, reader.

Which version is cleaner?

The first version, while full of repetition, was much easier to understand and change than the version that abstracted everything away.

Implementing the DRY (do not repeat yourself) principle pulls us closer towards *Overengineering* (a terrible anti-pattern), while *Repeating yourself* (also an anti-pattern) may pull us closer towards more human readable code in some contexts.

Computer science is a game of trade-offs, and where *Overengineering* trades readability for abstraction, it's almost always a bad trade. If we can't understand and change the code, it is no longer adhering to what we have deemed one of the **primary goals of software**.

Always put humans first! Make your fellow developers awesome!

Catalog of anti-patterns: Read the anti-patterns that pertain to software engineering here. I highly recommend you spend some time on this page and peak through them. We all know software is hard. Take in the fact that each of these patterns exists because somewhere, somehow, a project failed- and it was the emergence of one or more of these patterns that played a part in its failure. Anti-patterns are decidedly organized into Software design, Object-oriented programming, Methodological, Programming, and Configuration management related anti-patterns.

[Overengineering](#)

[Cyclomatic Complexity](#)

[**Overengineering**](#)

[**Cyclomatic Complexity**](#)

[**Optimizing code**](#)

Optimizing code for humans means making it easier to read and understand.

Optimizing code for computers means making it more efficient.

Often, optimizing code for computers means making it less readable and understandable for humans.

This is a tough balance to accomplish, but usually, I recommend optimizing for **humans > computers**.

[**Absolute and relative complexity**](#)

[**Guiding principles for clean coders**](#)

[**Principle #1 - Care deeply about the project and the domain**](#)

You need to care deeply. None of this works if you don't care.

Reasons you might not care about writing clean code:

- You hate your job
- You're going to leave the company

- It's just a hacky project
- You're the only one working on it
- The project isn't going so well and it feels like a death march

Without this first principle, nothing else matters and you may never write clean code.

Get yourself into a job you enjoy with people you don't mind working with, or into a domain that has the potential to seem interesting, and become passionate about the success of the project, and become empathetic about the lives of other developers that will need to interact with your code tomorrow and 5 years from now.

Principle #2 - Aim to empower teammates and future maintainers

A goal of yours should be to make other developers feel like they are good at their job. They should be able to understand, locate, and change code quickly and safely.

Put real code you wrote in front of a fellow developer and ask them to comprehend, locate, and change a feature. Do this to determine if your empathetic coding ability needs tuning.

Principle #3 - Humans > computers

Principle #4 - Conventions and patterns are helpful but they are secondary to the needs of your users and future maintainers

When it is not a detriment to the empowerment of other developers, enforce conventions, design patterns, and principles.

DRY is secondary.

Abstraction is secondary.

Even the popular coding conventions that I'm about to show you in the next section are secondary if they don't empower your team and future maintainers.

Where to go from here

The rest of what you're going to learn in this book are principles, practices, theory and approaches to designing software.

Everything from here on out is food for thought to better inform your preferred coding conventions.

The theory you learn in Programming Paradigms should help you understand the benefits of Object-Oriented Programming and when it might make sense to take a more functional approach in your work.

Learning how to

Resources

Curate this section almost like how David Perrell curates his stuff on his email list to send to others. Give each of these books and links a little blurb about the takeaways. So for each book, explain why — what will the reader get from it.

- The Design of Everyday Things
 - For empathy & learning how to design for humans
 - HCD & empathy
 - * Empathy in Human Centered Design
- Clean code
- The clean coder
- The Pragmatic Programmer
- The Software Craftsman: Professionalism, Pragmatism, Pride
- On the topic of coding and empathy
 - <https://www.benjaminjohnson.me/blog/empathetic-code/#:~:text=Having> a consistent code style is empathetic because it allows, they are trying to solve.
 - <https://compassionatecoding.com/> — April has a site called compassionate coding, really cool!
- Kathy Sierra's "Badass: Making Users Awesome".

Clean Code by Uncle Bob

My introduction to *Clean Code* was the book of the same name by Robert C. Martin (Uncle Bob). I believe this is the norm for many other developers as well.

In *Clean Code*, we're taken through a series of coding conventions and opinionated best practices against everything from naming variables to handling errors.

This is an exceptionally good book.

It's a great book, even despite the fact that many developers disagree with the often asymptotically unreasonable opinions from Uncle Bob. For example, Uncle Bob declares 100% test coverage a **must**. He also states that we **must** be using TDD to design our software. These, and even the more reasonable opinions in his book are subject to debate.

Clean Code, the book, has had a lasting impact on our industry: for good and for *not-so-good*. Where some have taken Uncle Bob's teachings and soared, others follow his teachings **dogmatically** and don't give contesting ideas room to breathe.

I think that's incredibly dangerous for a number of reasons, and I wanted to do something different with this book.

Resources

https://en.wikipedia.org/wiki/Agile_software_development#The_Agile_Manifesto

https://en.wikipedia.org/wiki/Software_craftsmanship

<https://8thlight.com/blog/paul-pagel/2009/03/11/history-of-the-software-craftsmanship-manifesto.html>

<https://en.wikipedia.org/wiki/Journeyman>

<https://www.brainpickings.org/2014/01/29/carol-dweck-mindset/>

- On knowledge in the head vs. the world
 - [https://medium.com/@matthewraychiang/doet-knowledge-in-the-head-and-in-the-world-64f901627eb3#:~:text=Knowledge in the head is, that

needs to be remembered.&text=Memory for arbitrary things is straight memorization](<https://medium.com/@matthewraychiang/do-it-knowledge-in-the-head-and-in-the-world-64f901627eb3#:~:text=Knowledge%20in%20the%20head%20is>)

References

This is just a collection of items and things that I used to develop this section, personally.

<https://alleninstitute.org/what-we-do/brain-science/news-press/articles/5-unsolved-mysteries-about-brain>

https://en.wikipedia.org/wiki/Code_smell

<https://en.wikipedia.org/wiki/Softcoding>

<https://en.wikipedia.org/wiki/Anti-pattern>

https://en.wikipedia.org/wiki/No_Silver_Bullet

https://en.wikipedia.org/wiki/God_object

[https://en.wikipedia.org/wiki/Poltergeist_\(computer_programming\)](https://en.wikipedia.org/wiki/Poltergeist_(computer_programming))

https://en.wikipedia.org/wiki/Call_super

https://en.wikipedia.org/wiki/Yo-yo_problem

https://en.wikipedia.org/wiki/Principle_of_least_astonishment

https://en.wikipedia.org/wiki/Coding_conventions

<https://medium.com/mindorks/how-to-write-clean-code-lessons-learnt-from-the-clean-code-robert-c-martin-9ffc7aef870c>

<https://overreacted.io/goodbye-clean-code/>

<https://cvuorinen.net/2014/04/what-is-clean-code-and-why-should-you-care/>

https://dev.to/d_ir/clean-code-dirty-code-human-code-6nm

<https://dev.to/carlillo/clean-code-applied-to-javascript-part-i-before-your-start-16ic>

<https://www.python.org/dev/peps/pep-0008/>

https://medium.com/@p_arithmetic/a-collection-of-my-6-favorite-javascript-one-liners-7c80a4b731f8

<https://blog.codinghorror.com/code-tells-you-how-comments-tell-you-why/>

https://www.reddit.com/r/askscience/comments/24z4qv/why_is_it_so_difficult_to_remember_a_short/

“Code Conventions for the Java Programming Language”

<https://www.oracle.com/technetwork/java/index-135089.html>

<https://en.wikipedia.org/wiki/Softcoding>

https://en.wikipedia.org/wiki/Code_smell

<https://en.wikipedia.org/wiki/Anti-pattern>

https://en.wikipedia.org/wiki/God_object

<https://medium.com/better-programming/how-to-create-meaningful-names-in-code-20d7476537d4>

4. Programming Paradigms

The three organizational archetypes

As I said earlier, there are really only three organizational archetypes we need to master: behavioral objects, state objects, and namespaces. Once you're there, all the particular tribal schools like DDD don't matter.

Apparently, these make up the majority of software design

- behavioral objects,
- state objects,
- and namespaces

To me, this sounds like the nature of DDD

(Theory) vs. (Implementation) Behaviors = domain events
State objects = aggregates
Namespaces = subdomains/bounded contexts

5. Object-Oriented Programming & Domain Modeling

6. Design Principles

SOLID

Single Responsibility Principle

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle

Dependency Inversion Principle (DIP)

Single Responsibility Principle

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle

Dependency Inversion Principle (DIP)

One of the first things we learn in programming is to decompose large problems into smaller parts. That divide-and-conquer approach can help us to assign tasks to others, reduce anxiety by focusing on one thing at a time, and improve modularity of our designs.

But there comes a time when things are ready to be hooked up.

That's where most developers go about things the wrong way.

Most developers that haven't yet learned about the solid principles or software composition, and proceed to write tightly couple modules and classes that shouldn't be coupled, resulting in code that's **hard to change** and **hard to test**.

In this section, we're going to learn about:

- Components & software composition
- How NOT to hook up components
- How and why to inject dependencies using Dependency Injection
- How to apply Dependency Inversion and write testable code
- Considerations using Inversion of Control containers

Terminology

Let's make sure that we understand the terminology on wiring up dependencies before we continue.

Components

I'm going to use the term **component** a lot. That term might strike a chord with React.js or Angular developers, but it can be used beyond the scope of web, Angular, or React.

A component is simply a **part of an application**. It's any group of software that's intended to be a part of a larger system.

The idea is to break a **large application** up into several modular components that can be independently developed and assembled.

The more you learn about software, the more you realize that good software design is **all about composition** of components.

Failure to get this right leads to *clumpy* code that can't be tested.

Dependency Injection

Eventually, we'll need to hook components up somehow. Let's look at a trivial (and non-ideal) way that we might hook two components up together.

In the following example, we want to hook up a UserController so that it can retrieve all the User[]s from a UserRepo (repository) when someone makes an HTTP GET request to /api/users.

```
// repos/userRepo.ts

/**
 * @class UserRepo
 * @desc Responsible for pulling users from persistence.
 */

export class UserRepo {
    constructor () {}

    getUsers (): Promise<User[]> {
        // Use Sequelize or TypeORM to retrieve the users from
        // a database.
    }
}
```

And the controller...

```
// controllers/userController.ts

import { UserRepo } from '../repos' // Bad

/**
 * @class UserController
 * @desc Responsible for handling API requests for the
 * /user route.
 */

class UserController {
    private userRepo: UserRepo;

    constructor () {
        this.userRepo = new UserRepo(); // Also bad, read on for why
    }

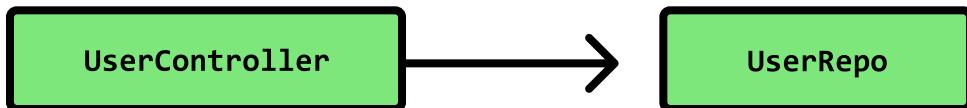
    async handleGetUsers (req, res): Promise<void> {
        const users = await this.userRepo.getUsers();
        return res.status(200).json({ users });
    }
}
```

In the example, we connected a UserRepo directly to a UserController by **referencing the name** of the UserRepo class from within the UserController class.

This isn't ideal. When we do that, we create a **source code dependency**.

Source code dependency: When the current component (class, module, etc) relies on at least one other component **in order to be compiled**. Source code dependencies should be limited.

The problem is that every time that we want to spin up a UserController, we need to make sure that the UserRepo is also **within reach** so that the code can compile.



When might you want to spin up an isolated UserController?

During testing.

It's a common practice during testing to *mock* or *fake* dependencies of the **current module under test** in order to isolate and test different behaviors.

Notice how we're a) importing the concrete UserRepo class into the file and b) creating an instance of it from within the UserController constructor?

That renders this code **untestable**. Or at least, if UserRepo was connected to a real live running database, we'd have to **bring the entire database connection** with us to run our tests, making them very slow...

Dependency Injection is a technique that can improve the testability of our code.

It works by passing in (usually via constructor) the dependencies that your module needs to operate.

If we change the way we inject the UserRepo from UserController, we can improve it slightly.

```
// controllers/userController.ts
import { UserRepo } from '../repos' // Still bad

/**
 * @class UserController
 * @desc Responsible for handling API requests for the
 * /user route.
 */

class UserController {
    private userRepo: UserRepo;

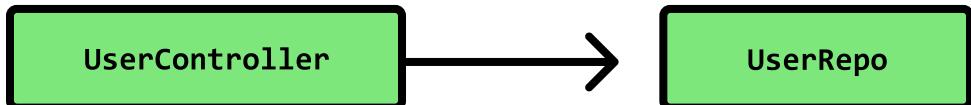
    constructor (userRepo: UserRepo) { // Better, inject via constructor
        this.userRepo = userRepo;
    }

    async handleGetUsers (req, res): Promise<void> {
        const users = await this.userRepo.getUsers();
        return res.status(200).json({ users });
    }
}
```

```
    }  
}
```

Even though we're using dependency injection, there's still a problem.

UserController still relies on UserRepo *directly*.



This dependency relationship still holds true.

Even still, if we wanted to mock out our UserRepo that connects to a real SQL database for a mock **in-memory repository**, it's not currently possible.

UserController needs a UserRepo, specifically.

```
// controllers/userRepo.spec.ts  
let userController: UserController;  
  
beforeEach(() => {  
  userController = new UserController(  
    new UserRepo() // Slows down tests, needs a db running  
  )  
});
```

So.. what do we do?

Introducing the **Dependency Inversion Principle!**

Dependency Inversion

Dependency Inversion is a technique that allows us to **decouple** components from one another. Check this out.

What direction does the **flow of dependencies** go in right now?



From left to right. The UserController relies on the UserRepo.

OK. Ready?

Watch what happens when we slap an interface in between the two components make UserRepo implement an IUserRepo interface, and then point the UserController to refer to *that* instead of the UserRepo concrete class.

```
// repos/userRepo.ts  
  
/**  
 * @interface IUserRepo
```

```

 * @desc Responsible for pulling users from persistence.
 **/


export interface IUserRepo {           // Exported
    getUsers (): Promise<User[]>
}

class UserRepo implements IUserRepo { // Not exported
    constructor () {}

    getUsers (): Promise<User[]> {
        ...
    }
}

```

And update the controller to refer to the `IUserRepo` interface *instead* of the `UserRepo` concrete class.

```

// controllers/userController.ts

import { IUserRepo } from '../repos' // Good!

/**
 * @class UserController
 * @desc Responsible for handling API requests for the
 * /user route.
 **/

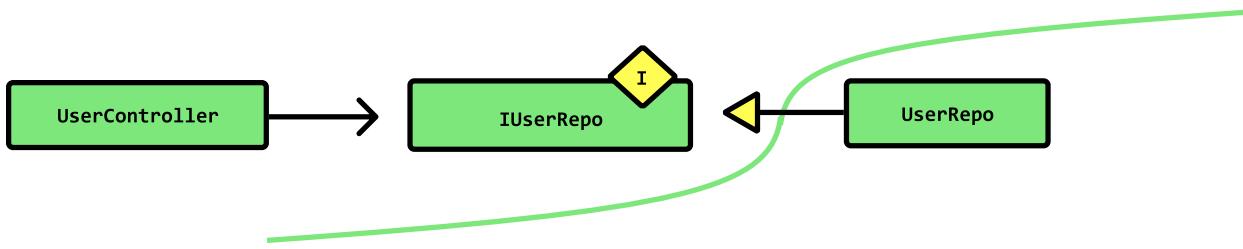
class UserController {
    private userRepo: IUserRepo; // like here

    constructor (userRepo: IUserRepo) { // and here
        this.userRepo = userRepo;
    }

    async handleGetUsers (req, res): Promise<void> {
        const users = await this.userRepo.getUsers();
        return res.status(200).json({ users });
    }
}

```

Now look at direction of the flow of dependencies.



You see what we just did? By changing all of the **references from concrete classes to interfaces**, we've just **flipped the dependency graph** and created an *architectural boundary* in-between the two components.

Design principle: Program against interfaces, not implementations.

Maybe you're not as excited about this as I am. Let me show you why this is so great.

Remember when I said that we wanted to be able to run tests on the UserController without having to pass in a UserRepo, solely because it would make the tests slow(UserRepo needs a db connection to run)?

Well, now we **can write** a MockUserRepo which implements IUserRepo and all the methods on the interface, and instead of using a class that relies on a slow db connection, use a class that contains an internal array of User[]s (much quicker! 🚀).

That's what we'll pass that into the UserController instead.

Using a mock object

```
// repos/mocks/mockUserRepo.ts
import { IUserRepo } from '../repos';

class MockUserRepo implements IUserRepo {
    private users: User[] = [];

    constructor () {}

    async getUsers (): Promise<User[]> {
        return this.users;
    }
}
```

Tip: Adding “`async`” to a method auto-wraps it in a `Promise`, making it easy to fake asynchronous activity.

We can write a test using a testing framework like **Jest**.

```
// controllers/userRepo.spec.ts
import { MockUserRepo } from '../repos/mock/mockUserRepo';

let userController: UserController;
```

```

const mockResponse = () => {
  const res = {};
  res.status = jest.fn().mockReturnValue(res);
  res.json = jest.fn().mockReturnValue(res);
  return res;
};

beforeEach(() => {
  userController = new UserController(
    new MockUserRepo() // Speedy! And valid since it inherits IUserRepo.
  )
});

test ("Should 200 with an empty array of users", async () => {
  let res = mockResponse();
  await userController.handleGetUsers(null, res);
  expect(res.status).toHaveBeenCalledWith(200);
  expect(res.json).toHaveBeenCalledWith({ users: [] });
})

```

Congrats. You (more or less) just learned how write testable code!

The primary wins of Dependency Inversion

Not only does this decoupling make your code *testable*, but it improves the following characteristics of your code:

- Testability: We can substitute expensive to infrastructure components for mock ones during testing.
- Substitutability: If we program against an interface, we enable a **plugin architecture** adhering to the Liskov Substitution Principle, which makes it incredibly easy for us to swap out valid plugins, and program against code that doesn't yet exist. Because the interface defines the *shape* of the dependency, all we need to do to substitute the current dependency is create a new one that adheres to the contract defined by the interface. See this article to dive deeper on that.
- Flexibility: Adhering to the Open Closed Principle, a system should be open for extension but closed for modification. That means if we want to extend the system, we need only create a new plugin in order to extend the current behavior.
- Delegation: **Inversion of Control** is the phenomenon we observe when we delegate behavior to be implemented by someone else, but provide the hooks/plugins/callbacks to do so. We design the current component to *invert* control to another one. Lots of web frameworks are built on this principle.

Inversion of Control & IoC Containers

Applications get much larger than just two components.

Not only do we need to ensure we're **referring to interfaces** and NOT concrete implemen-

tations, but we also need to handle the process of manually injecting *instances* of dependencies at runtime.

If your app is relatively small or you've got a style guide for hooking up dependencies on your team, you could do this manually.

If you've got a huge app and you don't have a plan for how you'll accomplish dependency injection within in your app, it has potential to get out of hand.

It's for that reason that **Inversion of Control (IoC) Containers** exist.

They work by requiring you to:

1. Create a container (that will hold all of your app dependencies)
2. Make that dependency known to the container (specify that it is *injectable*)
3. Resolve the dependencies that you need by asking the container to inject them

Some of the more popular ones for JavaScript/TypeScript are Awilix and InversifyJS.

Personally, I'm not a huge fan of them and the additional **infrastructure-specific framework logic** that they scatter all across my codebase.

Inversion of Control: Traditional control flow for a program is when the program only does what we tell it to do (*today*). Inversion of control flow is a common thing to enable in framework development and **plugin architecture** with areas of code that can be hooked into.

In these cases, we *might not know (today)* what we want the behavior to be, or - we wish to enable clients of our API, other developers, to make that decision on their own.

That means that every **lifecycle hook in React.js or Angular** is a good example of Inversion of Control in practice. IoC is also often referred to as the "Hollywood Design Principle": *Don't call us, we'll call you.*

Design by Contract (DBC)

Chapter 21 in The Pragmatic Programmer is really good for this!!!

Separation of Concerns

Related blog posts

CQS (Command Query Separation)

Principle of Least Surprise

Law of Demeter

Composition over Inheritance

YAGNI

KISS (Keep It Simple, Silly)

DRY (Don't Repeat Yourself)

The Four Primary Object-Oriented Design Principles

Composition over inheritance

Aim for shallow class hierarchies

Aim for shallow class hierarchies

Encapsulate what varies

Program to interfaces, not to implementations

Depend upon abstractions. Don't depend upon concrete classes.

Relationship to Ports and Adapters architecture

Relationship to Dependency Inversion Principle

Relationship to Ports and Adapters architecture

Relationship to Dependency Inversion Principle

The Hollywood Principle

Strive for loosely coupled design between objects that interact

All software is composition

Design patterns are complexity

Know of them, but know when you need them

Know of them, but know when you need them

Separation of Concerns

Example: overloaded controller

Example: overloaded controller

Here is some code that demonstrates what not to do. It's an example of a RESTful API controller class. In it, only the createUser method is shown, but given the class name AppController, it's rightful to assume that this controller would contain *all* of the methods for the app's REST api.

```
// AppController.ts

export class AppController {
  ...
  createUser (req: express.Request, res: express.Response): Promise<void> {
    // Get values from request
    const { username, email, password } = req.body;

    // Validate request values
    const isUsernameValid = TextUtils.isAtLeast(3, username)
      && TextUtils.isAtMost(30, username);
    const isEmailValid = TextUtils.isValidEmail(email)
    const isPasswordValid = TextUtils.isAtLeast(3, password)
      && TextUtils.isAtMost(25, password);

    // Check if username has already been taken
    const existingUserByUserName = await this.userRepo
      .getUserByUserName(username);

    if (existingUserByUserName) {
      return res.status(409).json({
        message: "Username already taken"
      });
    }

    // Check if email already exists
    const existingUserByEmail = await this.userRepo
      .getUserByEmail(email);

    if (existingUserByEmail) {
      return res.status(409).json({
        message: "User already exists. Try logging in."
      });
    }
  }
}
```

```

// Otherwise, create the user
const user = await this.userRepo.create({
  username,
  email,
  password
})

// Send email verification email
await this.emailService.sendVerificationEmail(user.email);

// Add email to mailing list
await this.mailingList.add(user.email);

// Do more stuff...
...
}
}

```

Without going on a tangent, this file has the potential to become very large.

In a request, there are several concerns to address:

1. Authentication — is the requester authorized?
2. **Controller — pull the args and data from the request object (and maybe even sanitize the data) then handle the response.**
3. Authorization — does the requester have access to this resource?
4. Use Case — what are the application-level business rules?
5. Domain Logic — what are the core business rules?

This is a **lot** to handle in a single controller method. The controller should only be doing #2 in the list provided above. Adhering to the Separation of Concerns principle is going to help keep files small by delegating concerns to the appropriate class.

Taking on only #2, here's what an altered version of the AppController *could* look like.

```

// AppController.ts
...
export class AppController extends BaseController {
  private createUserUseCase: CreateUserUseCase;

  constructor (createUserUseCase: CreateUserUseCase) {
    super();
    this.createUserUseCase = createUserUseCase;
  }

  async createUser (
    req: DecodedExpressRequest,
    res: express.Response
  ): Promise<any> {

```

```

let dto: CreateUserDTO = req.body as CreateUserDTO;

// It's OK for us to sanitize data coming in- that
// sounds like a controller concern.
dto = {
    username: TextUtils.sanitize(dto.username),
    email: TextUtils.sanitize(dto.email),
    password: dto.password
}

try {
    // All the business logic happens in the use case
    const result = await this.createUserUseCase.execute(dto);

    // If the operation failed, present the appropriate error
    // response.
    if (result.isLeft()) {
        const error = result.value;

        switch (error.constructor) {
            case CreateUserControllerErrors.UsernameTakenError:
                return this.conflict(error.errorValue().message)
            case CreateUserControllerErrors.EmailAlreadyExistsError:
                return this.conflict(error.errorValue().message)
            default:
                return this.fail(res, error.errorValue().message);
        }
    } else {
        // Otherwise, success!
        return this.ok(res);
    }
} catch (err) {
    return this.fail(res, err)
}
}

```

This example provides a class that handles everything that has to do with *being a controller*, and it does it *better* with proper error handling, and consistent responses too.

Separation of concerns

In the previous example, we fixed the AppController class to address the Separation of Concerns problem. What about the Single Responsibility Principle that we're still violating?

AppController is an object that has many reasons to change because it is relied on by several different roles. If we have Users, Notifications, Billing, Analytics and Media teams,

if each team relies on this *God-like* controller, everyone, even from different domains, will eventually find reasons for it to need to change. Code needing to change isn't necessarily the bad thing, but code changing for a reason that could inadvertently affect another team, *is* a bad thing. Understanding the single responsibility of a class and decoupling it to ensure that adheres to that contract is a way to insulate our code. It's also a way to reduce file size.

One solution the controller's responsibility down to only the Users subdomain would reduce the issue.

```
// modules/users/infra/http/userController.ts
export class UserController {
  ...
}
```

Even better, scoping the controller down to a *feature within* the Users subdomain, let's say in a folder called createUser, along with all the other files needed to make the feature work (input type, response type, potential error namespace, etc), we've can both improve the Single Responsibility **and** the Separation of Concerns.

```
// modules/users/useCases/createUser/createUserController.ts
export class CreateUserController {
  ...
}
```

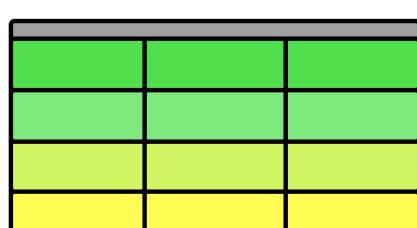
The Separation of Concerns and Single Responsibility Principle help to keep files small by indicating *how to split your files into smaller pieces instead of arbitrarily splitting them*.

The Relationship between SRP and SoC

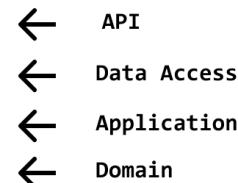
by Khalil Stemmler | @stemmlerjs

Single Responsibility (Role)

Admin Member Guest



Separation of Concerns



Single Responsibility Principle

A component should have one and one reason only to change. Based on Conway's Law, a component will only need to change if the role that utilizes the component determines change is necessary. Therefore, we should divide responsibility by role (vertically).

Separation of Concerns

To realize a feature, we need to rely on a mix of infrastructure, application, and domain layer logic. The SoC principle urges us to divide the "technical" implementation details of a feature. To do this, we can implement a layered architecture (horizontal).

Feature = createPost

Cross-cutting concerns

Principle of Least Surprise

Strive for loose coupling between objects that interact

Principle of Least Resistance

7. Design Patterns

Factory pattern

8. Architectural Principles

Component principles

Reuse-Release Equivalence Principle

Common closure principle (CCP)

The Common Reuse Principle (CRP)

Reuse-Release Equivalence Principle

“The granule of reuse is the granule of release.”

Excerpt From: Robert C. Martin. “Clean Architecture: A Craftsman’s Guide to Software Structure and Design (Robert C. Martin Series).” Apple Books.

Common closure principle (CCP)

“Gather into components those classes that change for the same reasons and at the same times. Separate into different components those classes that change at different times and for different reasons.”

Excerpt From: Robert C. Martin. “Clean Architecture: A Craftsman’s Guide to Software Structure and Design (Robert C. Martin Series).” Apple Books.

The Common Reuse Principle (CRP)

“Don’t force users of a component to depend on things they don’t need.”

Excerpt From: Robert C. Martin. “Clean Architecture: A Craftsman’s Guide to Software Structure and Design (Robert C. Martin Series).” Apple Books.

Conway's Law

The Dependency Rule

Boundaries

Coupling & cohesion

9. Architectural Styles

Structural

Component-based architectures

Layered Architectures

Monolithic architectures

Component-based architectures

Layered Architectures

Monolithic architectures

Message-based

Event-Driven architectures

Publish-Subscribe architectures

Event-Driven architectures

Publish-Subscribe architectures

Distributed

Client-server architectures

Peer-to-peer architectures

Client-server architectures

Peer-to-peer architectures

10. Architectural Patterns

Clean architecture

Layers

Similar architectures

Layers

Domain layer
Application layer
Infrastructure layer
Adapter layer

Domain layer

Application layer

Infrastructure layer

Adapter layer

Similar architectures

Ports & Adapters
Vertical-slice architecture

Ports & Adapters

Vertical-slice architecture

This is really cool
<https://jimmybogard.com/vertical-slice-architecture/>

Domain-Driven Design

Event Sourcing

Notes

Everything I've recorded about Event Sourcing so far

Everything I've recorded about Event Sourcing so far

About this

This is the most requested content on the blog so far.
I'm going to put all my notes and everything I know about this thus far, here:

Internal links

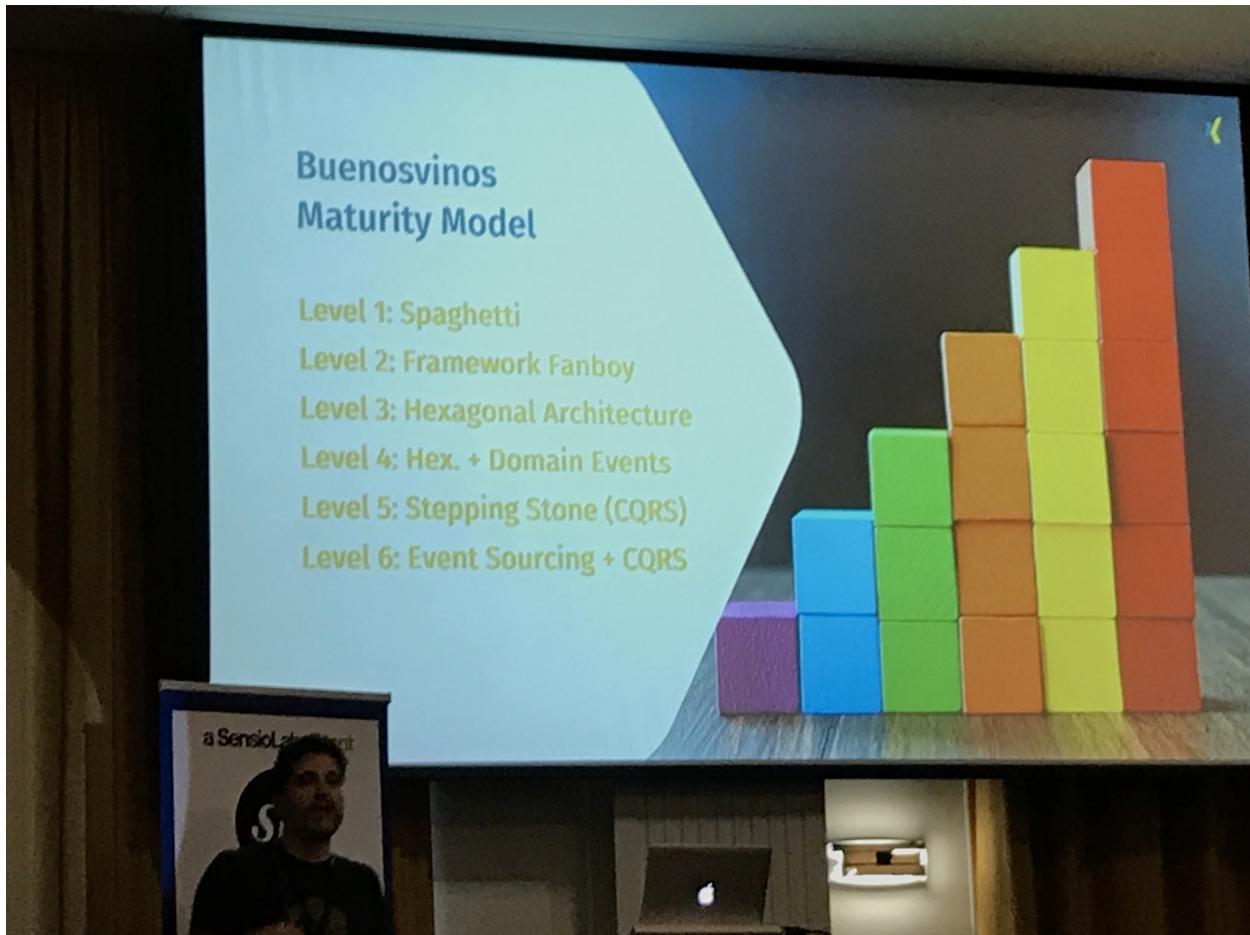
Copy of Best Places to Learn CQRS, Event Sourcing

Progression to Event Sourcing

In order to demonstrate how far we are in our journey, I want to present a map or something so that people know how far they are.

This tweet by @buenosvinos lays out the roadmap for software design really nicely in my opinion. I think that this is a great way to describe the roadmap from no DDD to DDD + Event Sourcing. I can come up with my own list, but using this as a resource would be helpful.

<https://twitter.com/JellyBellyDev/status/931153536171302922>



Sam Hotoum's Event Sourcing w/ TypeScript repo

Sam wrote this amazing baseline repository for event sourcing with TypeScript. We need to utilize this stuff for articles and all kinds of other thing.

Take a look at the description in the repo because it's excellent. It has a lot of content in there that we can write blog posts about and whatnot.

[stemmllerjs/typescript-event-sourcing](https://github.com/stemmllerjs/typescript-event-sourcing)

Here's a tweet I put out so far.

<https://twitter.com/stemmllerjs/status/1262365339578400769>

Why Event Sourcing?

So many reasons.

State management code can get messy

Structured data for state management introduces accidental complexity. Separating the write & read models solves this. You need ES/CQRS to do this. Greg Young's class is 6h. I think we can condense it in. Watch this space :)

All the reasons described in Why Event Based Systems

Why Event-Based Systems? | Enterprise Node.js + TypeScript | Khalil Stemmler

For the **Complexity** part,

This is how we're ensuring that we have event handlers

```
private applyChangeInternal(event: Event, isNew = false): void {
  if (!this[`apply${event.constructor.name}`]) {
    throw new Error(
      `No handler found for ${event.constructor.name}. Be sure to define a method called apply${event.constructor.name} on the aggregate.`
    )
  }

  this[`apply${event.constructor.name}`](event)
  this._version += 1

  if (isNew) {
    this._changes.push(event)
  }
}
```

Projections and deserializing events happens like this!

Sam Hatoum, 4 days ago | 1 author (Sam Hatoum)

```
export class FileTree {
    public files = {}

    constructor(private messageBus: IMessageBus) {
        messageBus.registerEventHandler(FileCreated, (e) => {
            const event = e as FileCreated
            this.files[event.aggregateId] = event
        })
        messageBus.registerEventHandler(FileRenamed, (e) => {
            const event = e as FileRenamed
            this.files[event.aggregateId].name = event.name
        })
    }
}
```

Copy of Best Places to Learn CQRS, Event Sourcing

Chapter

How to learn Event Sourcing [<https://github.com/MerrionComputing/EventsSourcing-on-Azure-Functions/wiki/Worked-Example--Bank-Account>]

Hey, Khalil: Re books. The appendix in the red DDD book from Vaughn Vernon, written by Rinat Abdullin is Gold for Aggregates + EventSourcing. Also covers this problem a bit.

Loads of online stuff from @gregyoung

@UdiDahan and many others. @mathiasverraes blogged about many messaging and ES patterns not long ago!!! I don't know a book that is very deep in detail here. But classic @ddddesign BC and context maps are still super important.

[<https://www.amazon.com/o/asin/0321200683/ref=nosim/enterpriseint-20>]

That is a good book - I'd add <http://dataintensive.net> for messaging/architecture related things and if you are in the Microsoft side of things then this <https://amazon.com/Exploring-Sourcing-Microsoft-patterns-practices-ebook> is a good resource for "event sourcing".

[https://www.youtube.com/watch?time_continue=1&v=kpM5gCLF1Zc&feature=emb_title]

Khalil's notes: Damn, the chapter in Vaughn Vernon's book is actually excellent. This code sample is the best example that I've seen so far: [<https://github.com/abdullin/iddd-sample>].

Other resources:

https://www.pluralsight.com/courses/modern-software-architecture-domain-models-cqrs-event-sourcing?fbclid=IwAR1EjdEPW32oNfyMItDEGcTPzhQo6J1tT2XoxEAARxGV AJV_96m4hPg5UcU

https://leanpub.com/implementing-ddd-cqrs-and-event-sourcing-in-nodejs?fbclid=IwAR2o-1mLodUpGHIZzvMw92P_pK_gfWQSbEs2CtFI3jl_g9-gtpZgcxyo-Zk

https://dev.to/heroku/best-practices-for-event-driven-microservice-architecture-2lh7?utm_source=additional_box&utm_medium=internal&utm_campaign=regular&boomster_org=heroku

DDD is very tough subject. So far I've read a few books about it and the only position that I can recommend for tech and non-tech people is "Patterns, Principles, and Practices of Domain-Driven Design" by Scott Millett and Nick Tune. They covered almost all aspects of DDD with examples and detailed explanations.

This book

Designing_Event_Driven_Systems (1).pdf

[Scott-Millett,-Nick-Tune]-Patterns,-Principles,-a(z-lib.org).pdf

Hands on Domain-Driven Design with .NET (epub, code, mobi, pdf)

9781788834094_HANDS_ON_DOMAIN_DRIVEN_DESIGN_WITH_NET_CORE.pdf

II. Building a Real-World DDD app

About this chapter

When I first launched solidbook.io, I chose to go about building it in a way that it could be most useful to its readers, as soon as possible.

That meant not writing an exorbitant amount of content upfront but instead asking readers to help me determine the trajectory of the book.

The most common feedback I got was that readers are most excited about **going from A to Z on their object-modeling skills and learning how to build real-world applications with Domain-Driven Design**.

For those of you that know me, you'd know that I'm *obsessed* with Domain-Driven Design and take any chance I have to chat with you about it.

That said, this is a book on software design and architecture *overall* - not a book solely devoted to Domain-Driven Design (which is a topic that truly deserves an entire book's worth of effort).

Eventually, I would love to create an approachable adaptation of the blue book and the red book for JavaScript and TypeScript developers interested in DDD.

I'm sure readers are satisfied with solidbook.io, I'll be working on creating a Domain-Driven Design with TypeScript course.

While I have no problem devoting a couple of chapters to DDD, it's important to note that **we are choosing a particular architectural pattern to focus on**.

It's essential to address this because, in Chapter 1, we learned that upon starting a new project, it helps to identify the **SQAs** most tethered to the project's success, *then* choose the

architectural pattern that has the potential to satisfy those SQAs best.

Depending on your project, that pattern might NOT be DDD.

However, in this chapter, I line you up with a project that *could benefit* from adopting DDD.

I think that learning DDD is the next logical step for developers comfortable with MVC.

This chapter may not be the *last* piece of literature or video you watch on DDD, but it should at least open your mind and answer several questions. My goal for this chapter is to provide you with a *really solid, hands-on, and practical* intro on my favored approach to solve complex software problems: Domain-Driven Design.

Chapter goals

This chapter is separated into 5 parts. Here's what's ahead. We will:

- Understand what Domain-Driven Design is and how it can address the shortcomings of MVC.
- Conduct a crash course on the essential Domain-Driven Design concepts: entities, value objects, aggregates, domain events, subdomains, bounded contexts, and two popular deployment styles for DDD projects (modular monoliths & distributed microservices).
- Learn about DDDForum.com, the real-world DDD app we're going to build.
- Learn different approaches for project planning before coding, such as imperative-driven design, use case modeling, Event Storming, and Event Modeling.
- Explore the DDDForum.com codebase, features, and design choices.

Domain-Driven Design

Domain-Driven Design is an approach to software development, and one of our *architectural patterns* meant to apply against projects with **lots of business logic complexity**.

Here's how it works:

- Discover the *domain model* by **interacting with domain experts** and agreeing upon a **common language** to refer to *processes, actors* and any other important phenomenon (like events and side-effects to events) that occur in your problem domain.
- Take those newly discovered terms and embed them in the code, creating a rich domain model that reflects the actual living, breathing business and its rules.
- Protect that (zero-dependency) domain model from all the other technical intricacies involved in creating a web application (like databases, web servers, etc)
- Continuously crunch domain knowledge into a software implementation of that knowledge.

You may first wish to read Chapter 5 - Object-Oriented Programming and Domain Modeling, a chapter that's in equal part theory and hands-on coding where we learn to write object-oriented code to create **domain models**.

Domain model: A domain model is a *declarative* layer of code (without dependencies to any upper-layer concerns) that encapsulates the business rules of a particular problem domain.

You could say that a domain model is the *solution space* to a problem domain.

Domain models are central to the architecture, hold the **highest-level policy**, are the **most stable** (since a drastic change of the domain model would mean a drastic change of the business itself - which is unlikely), and **may not rely on anything from a layer above it**, yet - *can be relied on* by any upper layer (such as application, adapter, and infra).

Fundamentally, if you can understand the problem domain (your real-life business), you can create the software version of it.

Ubiquitous Language

The **Ubiquitous Language** (which is a fancy DDD term for the common language that best describes the domain model concepts) is learned through conversation with domain experts.

To **communicate effectively**, the *code* must be based on the *same language* used to write the *requirements* - the *same language* that *developers speak* with each other and the *domain experts* - Eric Evans

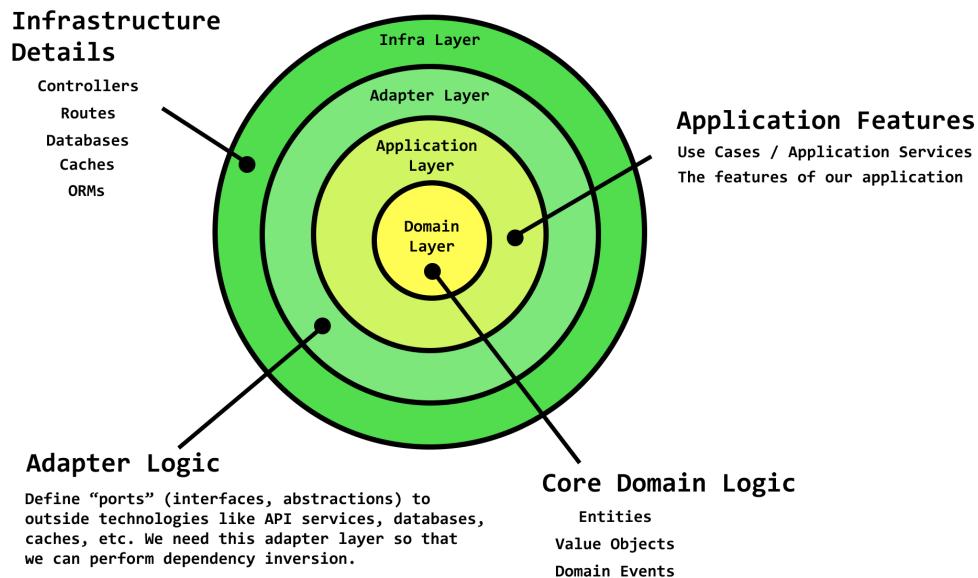
Once the common language is established and agreed upon, we use those words as our classes, use cases and types in the code.

It is not the *domain expert's knowledge* that goes to production, it is the *assumption* of the *developers* that goes to production - Alberto Brandolini

Implementing DDD & ensuring domain model purity

Protecting our domain model and keeping it pure is going to take some hard work and rely heavily on our use of:

- a Layered Architecture, most formally understood as the clean architecture (for separation of concerns)
- Dependency Inversion (to keep inner layer code testable)
- The Dependency Rule (to enforce the use of Dependency Inversion)



A layered architecture (also known as the “clean architecture”) is critical to our success towards keeping our domain model pure.

DDD addresses the shortcomings of MVC

You might remember that I started this book with a story about how I failed a job interview when I was asked the question, “how would you design your business-logic layer”?

Let me turn it around to you.

How would *you* design your business-logic layer?

If you’ve never worked on the backend of a challenging enterprise application, you might not have the answer to that question, like I didn’t.

Maybe you don’t understand the question like I didn’t.

Here’s what I mean: consider we’re working on a Vinyl-Trading application built using the popular Model-View-Controller pattern.

Let’s say that we need to implement some sort of **Role-based Access Management**. We want to restrict *who* has to access to *what*. Let’s say that Traders can only view their *own* Vinyl, while Admins can view *everyone’s* Vinyl.

If we’re thinking about building this app API-first, one of our goals might be to setup **retrieving Vinyl by id**.

Where do the business rules go?

How do we enforce:

- Traders only being able to view their own Vinyl

- Admins being able to view anyone's Vinyl

In the **view**? No, we're not supposed to put business logic in the view. We know that.

In the **controller**? No. That's just supposed to handle HTTP requests and pass off execution to something else.

In the **model**? The answer is actually "yes". But how?

Slim (Logic-less) Models

If you've worked with JavaScript or TypeScript over the last 4 years in a full-stack capacity, you might be familiar with at least one Node.js ORM (object-relational mapper).

ORM (Object-relational mapper): An ORM is a technique (though some people refer to the library that performs this technique as *an ORM*) to query and manipulate data from a database using object-oriented programming concepts. Some well-known ORMs in the Node.js world are Sequelize and TypeORM.

Early usage of these tools yields very **slim** models. They're just *definition files*. Their sole purpose is to define a schema and relationships between each other so that the object-relational mapper can do some *object-relational mapping magic* to map to real tables in a database.

For example, using the Sequelize ORM, defining a User model is done like this:

```
// models/user.js
// Sequelize ORM's way of creating a User model.

module.exports = (sequelize, type) => {
  return sequelize.define('user', {
    id: {
      type: type.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    firstName: {
      type: type.STRING,
      allowNull: true,
    },
    lastName: {
      type: type.STRING,
      allowNull: true,
    },
    age: {
      type: type.INTEGER,
      allowNull: true
    }
  })
}
```

Similarly, using TypeORM, we can define the same User model like this:

```
// models/user.ts
// TypeORM's way of creating a User model.

import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    firstName: string;

    @Column()
    lastName: string;

    @Column()
    age: number;

}
```

Just about anyone can build a simple CRUD-based MVC app today. Within a minimal Node.js + Express.js + Sequelize (or TypeORM) stack, everyone knows that the ORM handles the **model**.

While this is an excellent approach in order to launch a prototype quickly, it's not suitable for ambitious projects.

The problem lies in the fact that the M in MVC is responsible for too much, while tools like Sequelize and TypeORM makes it *seem like* the M is only responsible for the **shape of the data**.

What do developers *usually* do in this situation?

Pick your object-modeling poison

In my experience, this mismatch between the responsibility of the model in MVC and beginner-level tutorial code that promotes slim ORM models makes it hard to interpret where business logic should go.

This confusion manifests as one of three (poisonous) options:

A: Controller holds business logic - Put business logic in the controller (which isn't correct) and is only an option because putting it in the Sequelize ORM model is going to *feel dirty*.

B: Start creating “services” to hold business logic - It's a common (dangerous) thought pattern that anything that doesn't naturally fit within the confines of three constructs of MVC (model, view, controller), is a *service*. You might find that this approach is a quick train

ride to writing code that has no **Single Responsibility** and contributes towards a turning a codebase into an Anemic Domain Model, which is as bad as it sounds.

C: Do the dirty thing and **put the business logic inside the Sequelize ORM model** as an instance method. I don't recommend this. Sequelize is an infrastructure-layer concern and needs an active database connection to use. Putting the domain logic here means that any unit tests relying on Sequelize will be slow since we have to account for the additional overhead of setting up and tearing down tables and connections. This is not clean at all. Sequelize (and any other infrastructure-layer technology) should be far from the domain logic.

Choosing one of these options is problematic, indeed. None of them are great, yet the poison most developers pick is A: *controllers hold the logic*.

Let's entertain that:

```
// modules/vinyl/controller.ts

class VinylController {
  constructor (models: any) {
    this.models = models;
  }

  // HTTP GET /vinyl/:vinylId

  async getVinylById (req: express.Request, res: express.Response) {
    const { userId } = req.decoded as DecodedExpressRequest;
    const { Vinyl, User, Trader } = this.models
    const { vinylId } = req.params;

    // Get trader and associated user.
    const user = await User.findOne({
      where: { user_id: userId },
      include: [{ model: Trader, as: 'Trader' }]
    });

    const isUserAdmin = user.Trader.is_admin;

    // If the user isn't an admin, then we have to confirm this
    // vinyl is owned by the person requesting it.

    if (!isUserAdmin) {
      const traderThatOwnsVinyl = await Vinyl.findOne({
        where: { vinyl_id: vinylId },
        include: [
          { model: Trader, as: 'Trader', where: { user_id: userId } }
        ]
      });
    }
  }
}
```

```

const traderExists = !!traderThatOwnsVinyl === true;

if (!traderExists) {
  return res.status(403).json({
    message: "You don't have access to this"
  })
}

// Otherwise continue
const vinyl = await Vinyl.findOne({ where: { vinyl_id: vinylId }});

// Return the vinyl (raw)
return res.status(200).json({ vinyl })
}
}

```

What I just presented is how I wrote code for a **long time**. Unfortunately, there are several drawbacks:

- I have to repeatedly write this **permission logic** in **every single controller** to see if I have access to a resource.
- We're returning the entire vinyl object *raw*. If we were to add or change a column on the model, we've potentially broken someone's code that relied on this from the API response.
- Nowhere in the code do we represent the concept of Role in this supposedly role-based access management. The concept of a Role is not explicitly expressed; it's expressed inadvertently through the *absence* or *presence* of a database row. That kind of beating-around-the-bush makes it pretty challenging to follow along understand the **domain logic** quickly. It actually forces readers to read between the lines. The code should read like a book, and it doesn't.
- Although I named variables expressively, **low-level details** (Sequelize - data-access logic) are mixed in with **high-level rules** (role-based access control). This is a bad **separation of concerns**.

These are legitimate problems that need to be solved. And we'll solve 'em. That's why "**how would you design your business-logic layer**" is such an excellent question.

In fact, if I were to start interviewing candidates for a full-stack software development role, that might be one of the next questions I ask after "can you remember the **4 principles of Object-Oriented Programming?**".

"How would you design your business-logic layer" is a good litmus test for the scope of projects a developer has previously worked on.

When asked that question, most of the time, developers who have a) never had the chance to work on complex full-stack applications, or b) didn't know how to address a growing application's complex needs, tend to describe MVC .

Concerns of the unspecified layer in MVC

MVC is an excellent starting point for a lot of simple CRUD apps, but complex ones struggle to keep things under control because the M in MVC is responsible for too many things, and **there's no simple framework to tell us how we should approach designing our models.**

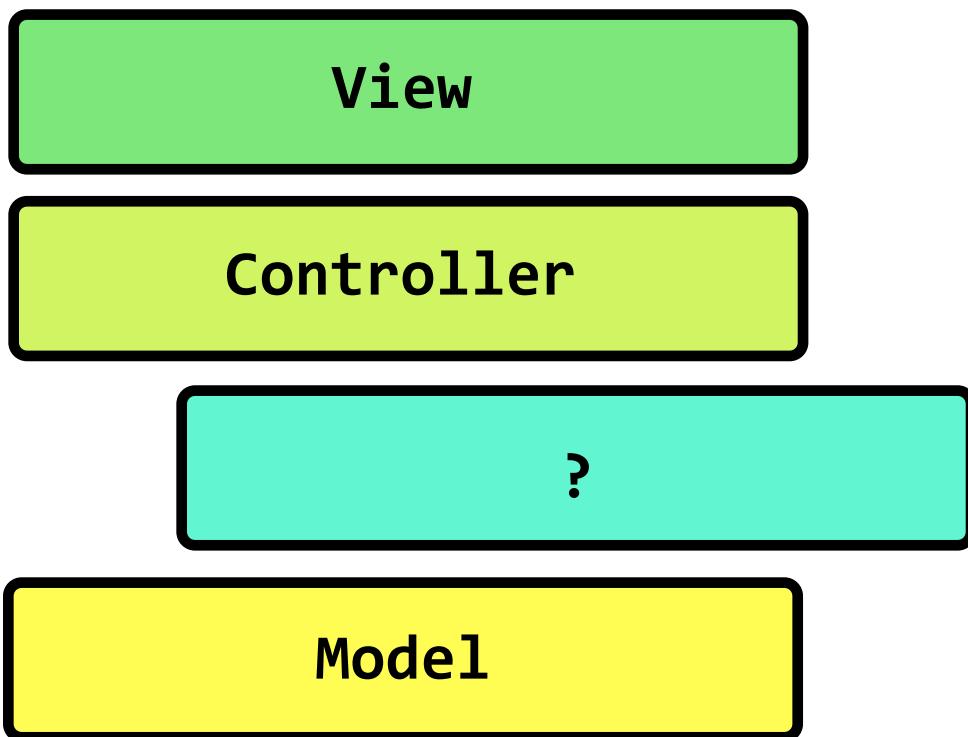
Let's say we extracted all of that access-control logic to some middleware functions.

Where in the **Model (M)** do we handle these things?:

- validation logic
- invariant rules
- domain events
- use cases
- complex queries
- and business logic

Those are a lot of concerns to just organize into functions and middleware.

Leaving all that important stuff to interpretation tends to find it placed in some *unspecified layer* between (what *should* be a rich) **model** and the **controller**.

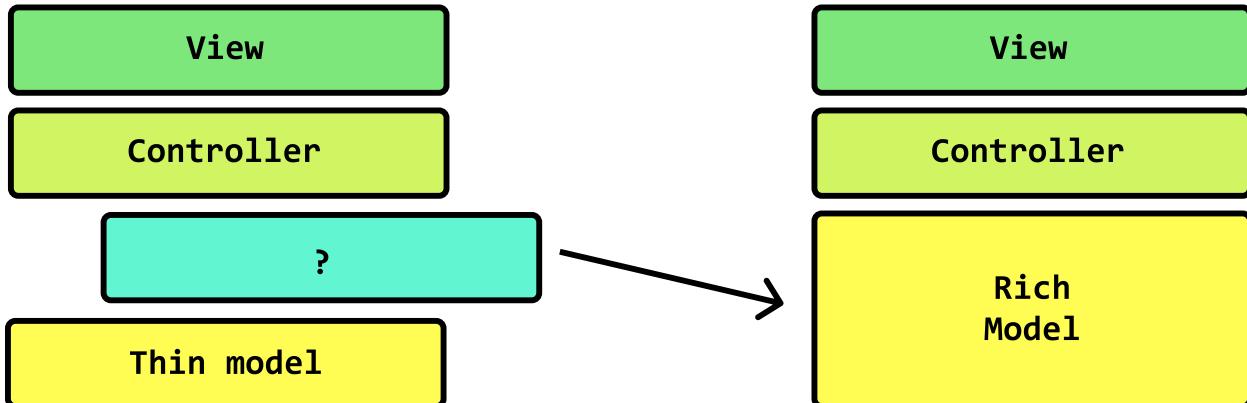


The missing layer in MVC.

Misguidedly trying to fit all these important concerns somewhere in between a *thin model* and the **controller** can feel like trying to push a square-shaped block inside a triangle-shaped one.

That **mystery layer** contains the *heart* of our software. It's the family jewels. It's the money-maker. It's the stuff that we **can't** just copy and paste or outsource. It's the *most challenging* and rewarding layer to master. It accounts for 90% of the code we *actually write*.

That's our *domain model*.



The missing layer in MVC is actually our domain model, which contains the business rules of our domain.

And it's what we should spend most of our time thinking about how to design effectively.

Undesirable side-effects with a lack of a domain model

If we fail to recognize when we need a domain model, we'll run into problems like:

- Missing abstractions and fostering a breeding ground of duplicate code
- Writing untestable and tightly-coupled classes
- Using computer-y sounding class names like 'handlers', 'factories', 'managers', 'interactors', making comprehension a real challenge for anyone else other than the original author
- Not caring about the actual problem domain and writing code that doesn't express the real-world problems it solves

Model behavior and shape

Ultimately, the model is responsible for both the *behavior* and the *shape* of our data, where the *behavior* is much more challenging to discover and represent.

And without a domain model, it's frequent that **behavior is an afterthought**.

In How to plan a new project, we introduce approaches to start a project by first identifying the behavior, averting out focus to the shape of the data second.

Technical Benefits

There are huge payoffs to DDD and domain modeling. When our code lines up with the real-life domain, we end up with a rich declarative design that enables us to make changes and add new features exponentially faster.

Projects that adopt DDD can expect the following technical benefits:

- Testable business-layer logic
- Less time fixing bugs
- A codebase that improves rather than degrades over time as code gets added to it

- Long life-spans

Technical Drawbacks

Domain modeling is time-consuming, takes repeated effort, and can be challenging.

Depending on the project, it might be more worthwhile to continue building an Anemic Domain Model.

Choosing DDD coincides with a lot of the arguments I made for when it's right to use TypeScript over JavaScript for your project. Use DDD for #3 of the *3 Hard Software Problems*: The Complex Domain Problem.

Alternatives to DDD

There are only two approaches. Either you write a *Transaction Script*, or you write a *domain model*.

If you've never written a domain model, you've been writing *Transaction Scripts*.

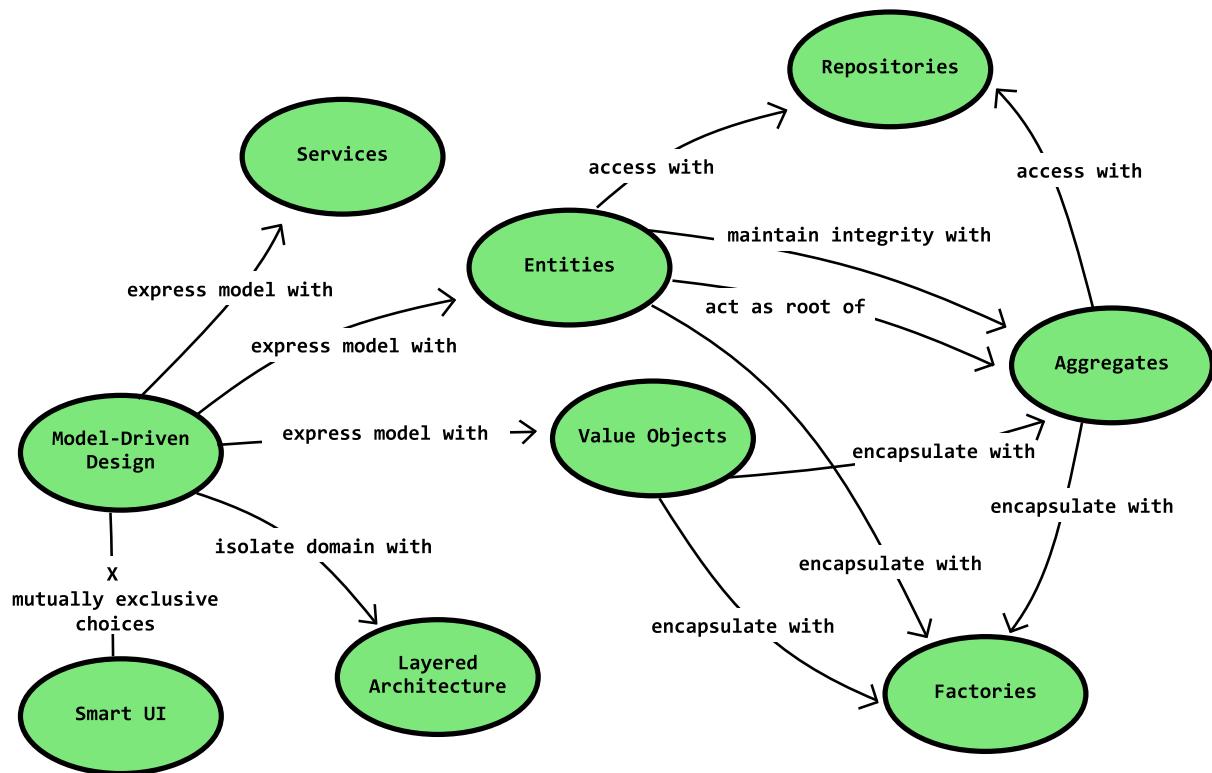
Transaction script: The simplest way to organize domain logic as possible. By using simple *if* and *else* control statements, we can express domain logic easily. This is perfect for simple applications without a huge amount of domain logic that can do without a lot of time spent on architecture.

Transaction Scripts are great for simple CRUD apps but for applications where the problem domain is complex, we need to break down the "model" part even further.

To do that, we use the **building blocks** of DDD.

DDD Building Blocks

Very briefly, these are the main technical artifacts involved in implementing DDD.



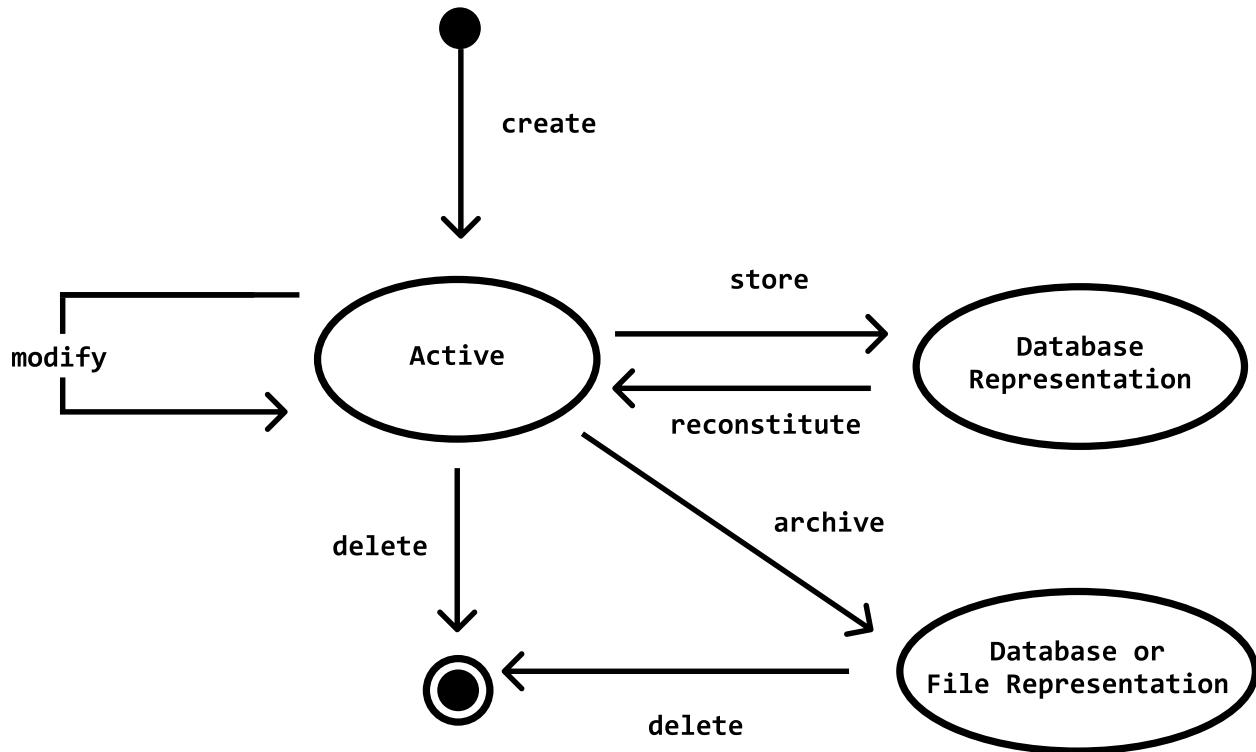
Entities

Domain objects that we care to *uniquely* identify.

Things like: User, Job, Vinyl, Post, Comment, etc.

Entities live a life enabling them to be created, updated, persisted, retrieved from persistence, archived, and deleted.

Entities are compared by their **unique identifier** (usually a UUID or Primary Key of some sort).



Resource: Read this article about Entities.

Value Objects

Value objects have no identity. They are *attributes* of Entities.

Think:

- Name as a Value Object on a User.
- JobStatus as a Value Object on Job
- PostTitle as a Value Object on Post

```
// A valid (yet not very efficient) way to compare Value Objects

const khalilName = { firstName: 'Khalil', lastName: 'Stemmler' };
const nick = { firstName: 'Nick', lastName: 'Cave' }

JSON.stringify(khalil) === JSON.stringify(nick) // false
```

Value Objects are compared by their **structural equality**.

Resource: Read this article about Value Objects.

Aggregates

An aggregate is a collection of entities bound together by an aggregate root. The aggregate root is the thing that we refer to for lookups. No members from within the aggregate boundary can be referred to directly from anything external to the aggregate. This is how the aggregate maintains consistency.

Every transaction that happens in our app happens against an aggregate - and it's the aggregate that protects against class invariants.

The **most powerful part about aggregates is that they dispatch Domain Events**, which can be used to decouple sequences of business logic so that they can be handled from the appropriate subdomain. ***

Resource: Read this article about Aggregates.

Domain Services

This is where we locate domain logic that doesn't belong to any one object conceptually.

Domain Services are most often executed by application layer Application Services / Use Cases. Because Domain Services are a part of the Domain Layer and adhere to the Dependency rule, they aren't allowed to depend on infrastructure layer concerns like Repositories to get access to the domain entities that they interact with. Application Services fetch the necessary entities, then pass them to Domain Services to run allow them to interact.

Check out PostService.ts, a Domain Service from DDDForum.com, the app we explore later in this chapter.

Repositories

We use repositories in order to retrieve domain objects from persistence technologies. Using software design principles like the Liskov Substitution Principle and layered architecture, we can design this in a way so that we can easily make architecture decisions to switch between an in-memory repository for testing, a MySQL implementation for today, and a MongoDB based implementation 2 years from now.

Resource: Read this article about implementing the repository pattern.

Factories

We'll want to create domain objects in many different ways. We map to domain objects using a factory that operates on raw SQL rows, raw json, or the Active Record that's returned from your ORM tool (like Sequelize or TypeORM).

We might also want to create domain objects from templates using the prototype pattern or through the use of an abstract factory.

Resource: Read this article about Static Factory Methods.

Domain Events

The best part of Domain-Driven Design.

Domain events are simply objects that define some sort of **event** that occurs in the domain **that domain experts care about**.

Typically when we're dealing with CRUD apps, we add new **domain logic** that we've identified by adding more if/else statements.

However, in complex applications that can become very cumbersome (think Gitlab or Netflix).

Using Domain Events, instead of *adding more and more if/else blocks* like this:

```
// UserController.ts
// Example of handling domain logic (transaction script-style).

class UserController extends BaseController {
  public createUser () {
    ...

    await User.save(user);

    // After creating a user, we handle both:

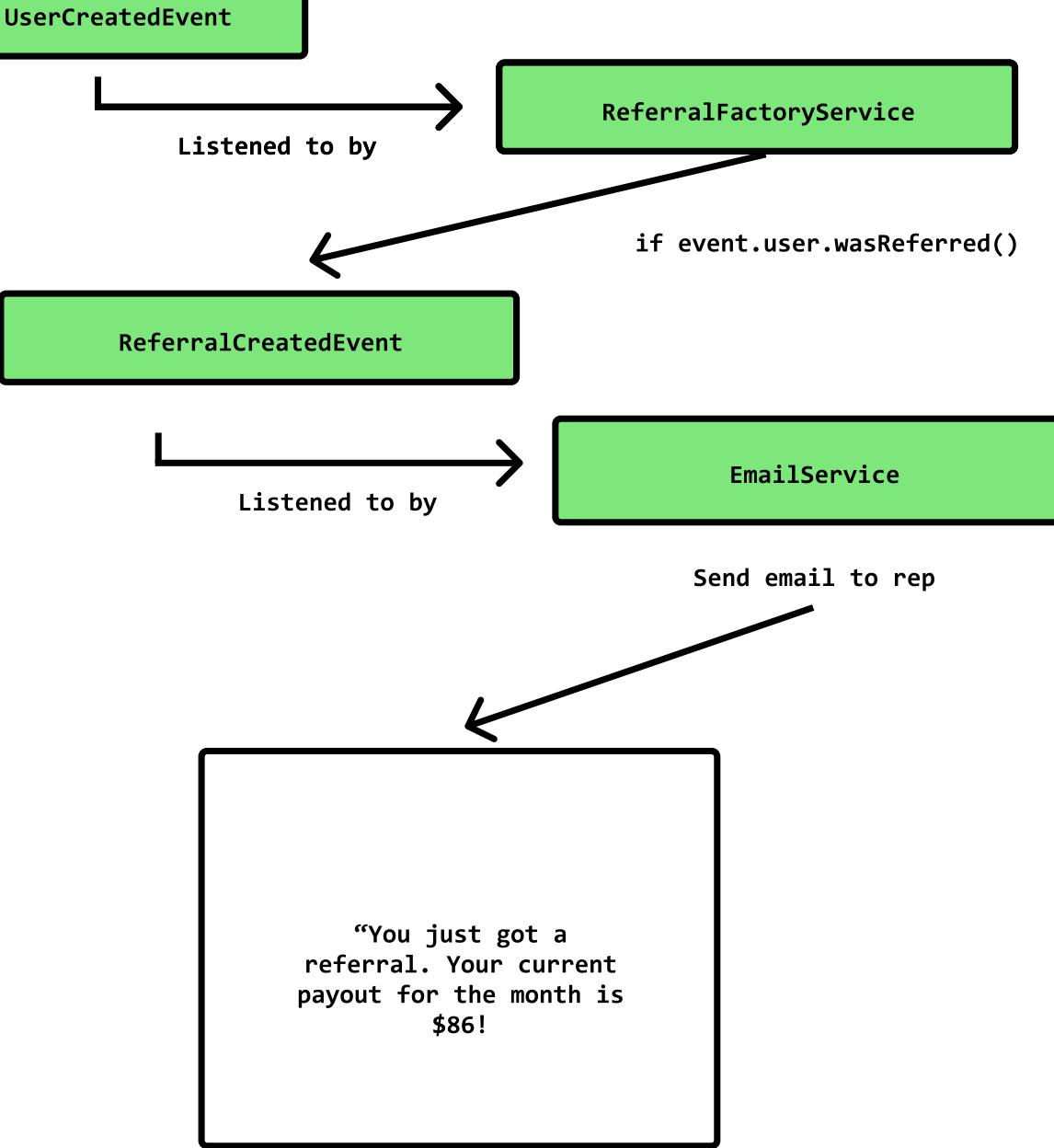
    // 1. Recording a referral (if one was made)
    if (user.referred_by_referral_code) {
      // calculate payouts
      // .. there could be a lot more logic here
      await Referral.create({
        code: this.req.body.referralCode,
        user_id: user.user_id
      });
    }

    // 2. Sending an email verification email
    EmailToken.createToken();
    await EmailService.sendEmailVerificationEmail(user.user_email);

    // mind you, neither of these 2 additional things that need to get
    // done are particularly the responsibility of the "user" subdomain

    this.ok();
  }
}
```

We can achieve something beautiful like this:



Using **domain services** and **application services**, Domain Events are an excellent way to *separate concerns* and decouple domain logic across DDD boundaries known as *Subdomains* and *Bounded Contexts* (read on).

Resource: For a resources on using Domain Events to decouple domain logic, read the “Decoupling Logic with Domain Events” guide and “Where Do Domain Events Get Created?”.

Architectural concepts

The two most important architectural concepts to grasp in Domain-Driven Design are **sub-domains** and **bounded contexts**. Subdomains are about creating logical boundaries and bounded contexts are about creating physical ones.

Subdomains

In DDD, a *Subdomain* is a **smaller piece (logical boundary) within the entire problem space.**

A *Subdomain* is a smaller piece (logical boundary) within the entire problem space.

What's a problem space?

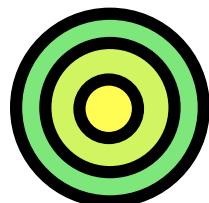
The *problem space* is the entirety of the things that a business is faced with solving.

For example, White Label, the app I'm working on for the upcoming DDD with TypeScript course, is about *trading vinyl*.

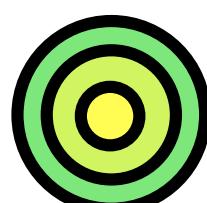
Unfortunately, trading vinyl isn't the *only* thing that needs my attention. There's much more that needs to be accounted for.

In addition to the trading aspect (Trading), the enterprise also has to account for several other concerns: identity and access management (Users), cataloging items (Catalog), billing (Billing), notifications (Notifications) and more.

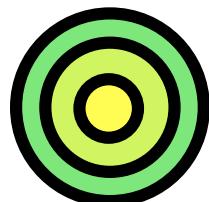
Each of these concerns are *subdomains*; decomposed logical slices of the entire problem domain.



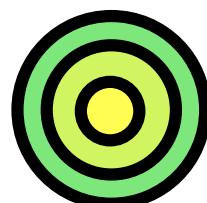
Billing
subdomain



Trading
subdomain



Catalog
subdomain

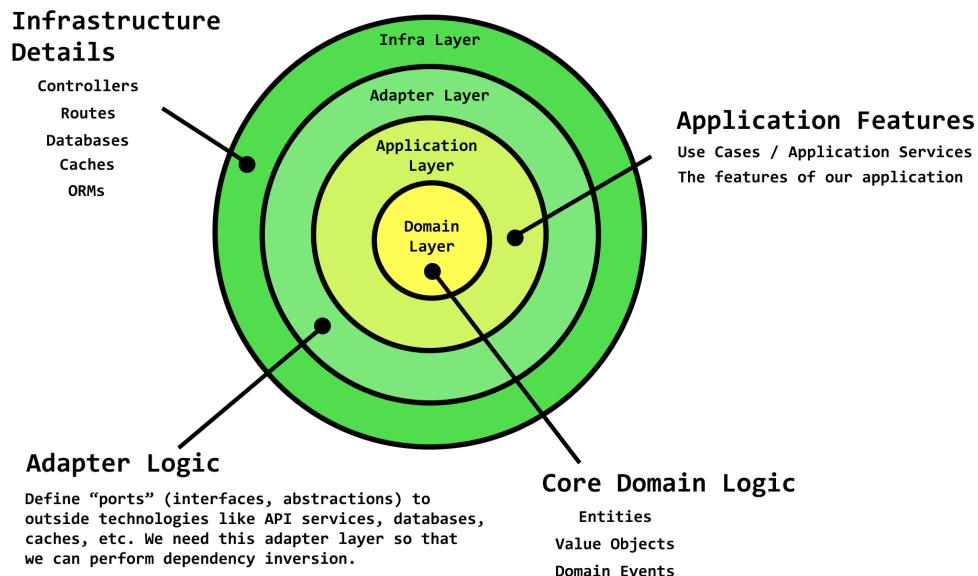


Users
subdomain

4 subdomains: Billing, Trading, Catalog, and Users from a vinyl-trading enterprise.

Each one of these *Subdomains* takes responsibility over a certain set of problems to be solved in the business. You can say that certain problems are more appropriate to be solved within the context of the *Billing Subdomain* (like making payments and creating subscriptions) while others are more appropriate to hand off to the *Users Subdomain* (like verifying an account or resetting a password).

Not only can we *horizontally* assign problems throughout our business to the appropriate subdomain, but we can *vertically* separate the concerns for each subdomain by each one implementing its very own *layered (clean)* architecture.



Types of subdomains

In Eric Evans' DDD book, he provides us with a few questions we can ask to determine which parts of our system are *core* to the domain. The questions are:

1. **What makes the system worth writing?**
2. Why not buy it off the shelf?
3. Why not outsource it?

If this subdomain is something that doesn't yet exist and we can't just buy it off the shelf, then it might be worth writing.

If access to the domain knowledge is not readily available and somehow, we're in possession of that knowledge and concerned about building a system around it *properly*, we might not want to outsource it.

If we know that we can make lots of money or help lots of people with it, then that might really make the system worth writing.

Generic subdomains

These are things that aren't *core* to the business. Yes, just about every SaaS application needs to figure out identity & authentication, but though that's important, it's likely that it's not the critical business problems that we're focused on addressing. We *could* outsource that job to someone like Auth0.

In White Label, the *generic* subdomains are: Users, Notifications, and Billing. That means there should be no explicit mention of anything regarding vinyl, traders, artists, or anything else from the core or supporting subdomains.

Supporting subdomains

These are parts of the application that are still necessary in helping the business do what it does, *yet we cannot just outsource it to some other service* because it's specific to the domain. In White Label, there's a subdomain responsible for Shipping vinyl. Shipping isn't the most important thing to us, but shipping *is* a necessary part of the business - though, even if not the *core*.

The *supporting* subdomains are: Shipping (arguably) and Catalog.

Core subdomain(s)

The core domain is the main thing that we're focused on. In White Label, that's *trading vinyl*: the trading subdomain. Evans says that "the Core domain should deliver about 20% of the total value of the entire system, be about 5% of the code base, and take about 80% of the effort".

Benefits of using subdomains

The main benefits of enforcing boundaries within your application is that they:

- a) Help to prevent domain concepts from other subdomains bleeding into your core one.
- b) Sometimes we refer to the same concept but from different contexts; subdomains act as a *safe space* to represent it in the way that makes the most sense per context (eg: Customer and Recipient are equivalent concepts but have different responsibilities depending on the *Subdomain*).
- b) Properly prepares you for scale.

There are two meanings to *scale*, at least how I mean:

1. Scaling the **size of your team** and your desire to be able to delegate ownership of sub-domains to a team.
2. Scaling as the **traffic demands** in your application have grown, and you now need to deploy more instances (or splice the application in some way) to keep up with traffic.

Logically organizing code into subdomains is the first step, *Bounded Contexts* are *what we actually deploy*.

Bounded Contexts

A *Bounded Context* is another *logical boundary* like *Subdomains*, but this time, it's a logical boundary around **all of the subdomains needed in order for an application achieve its goals**.

A *Bounded Context* is a *logical boundary* around **all of the subdomains needed in order for an application to achieve its goals**.

Some say that the *Bounded Context* is the *solution space* to the *problem space*. Technically, this checks out because we're identifying the actual *blocks* necessary to address the problem domain.

Let's look at an example.

Assume DDDForum.com was successful enough that the CEO decided they wanted to add several new services to their already very successful enterprise.

Depending on the features that each service needs and the problems that each existing subdomain is capable of addressing, it might mean we need to add more subdomains to address new problems.

Here are a couple ideas that the CEO came up with:

DDDDating (dating, users, location, billing, notifications): “Love is *ubiquitous*. Find your domain-modeling match today”.

This could be a dating app for anyone to find dates with people near them. The app would be part of the DDD-apps enterprise, but we might need some new subdomain. For example, there’s a feature that enables you to meet with people nearby, so location-tracking, computing distances, etc- might be a part of the a location subdomain. As well, this would likely be *paid* application, so we’ll need to figure out the billing subdomain as well as the dating subdomain to hold all data and operations specific to this application itself.

DDDMeetups (meetups, users, location, billing, notifications): “Find a local DDD Meetup near you”.

You know Meetup.com? We could create our own version of that strictly for the DDD community. For those of you who aren’t familiar with Meetup.com, it’s a platform where you can organize and find meetups based on things you’re interested in. In order to build this, we’d definitely need a new `meetups` subdomain in addition to `billing` if we wanted to charge event promoters to create events and notifications in order to send announcements to people before events.

DDDMerch (merch, inventory, billing, notifications, shipping, users): “Find all your *value objects* here on the world’s first e-commerce store dedicated to everything DDD! (so nerdy, I know)”

DDDMerch.com could be an e-commerce website selling t-shirts, sweaters, stickers, mugs, etc. If we built this entire thing from scratch, we’d need a `merch` subdomain (which might just be a CMS like Contentful) in order to change the text, promotions, and set coupons for the site, an `inventory` subdomain to keep track of all the items in stock, and also a `shipping` subdomain to handle tracking packages.

In order to create the solution space for each of our new *Bounded Contexts*, we can utilize several existing *Subdomains* that already solve those problems for us, while introducing a few new ones (such as `dating`, `meetups`, and `merch`) to solve problems not yet addressed.

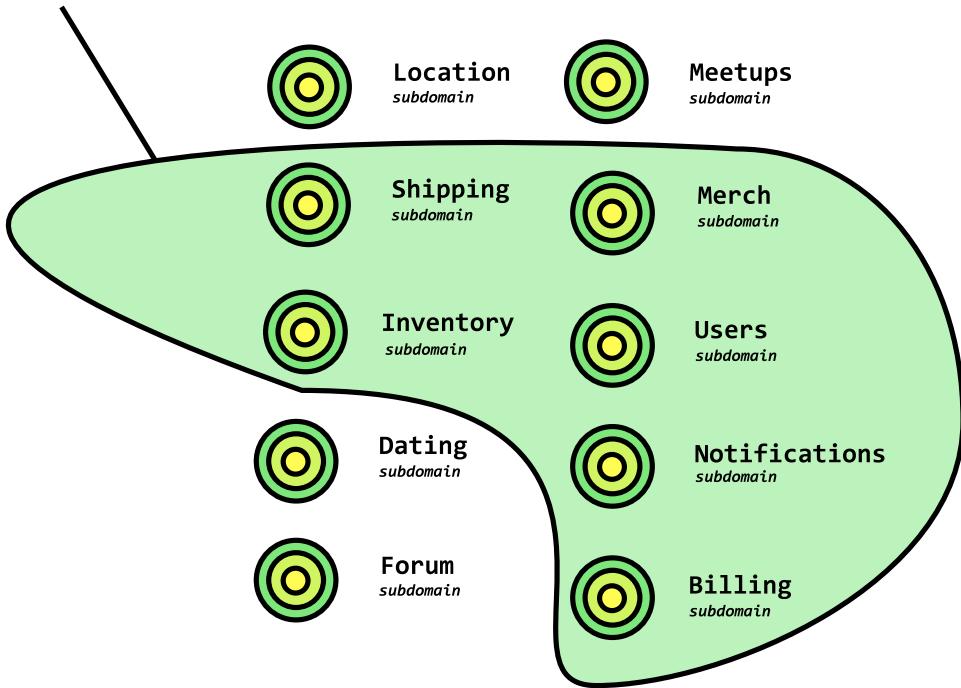


All subdomains in the DDD-apps enterprise.

Take note that not every *Subdomain* is required for each *Bounded Context*. For example, the only *Bounded Context* that needs access to the shipping subdomain is DDDMerch.

We could draw the boundary around all of the subdomains needed to power DDDMerch.

DDDMerch's Bounded Context



Bounded context for DDDMerch — it's a collection of several subdomains! The subdomains needed in order to achieve its goals are circled.

In the context of DDDMerch, the *problem space* is that we “need to sell merch, put on promotions, ship items, and track them”. The *subdomains* help to realize the solution.

Some subdomains make sense to use, and some of them don’t. There’s no reason for us to need to use the location, meetup, dating, or forum subdomains for DDDMerch.

As we continue, we realize that we have options for realizing our *Bounded Contexts*: **Modular Monoliths** and **Distributed Micro-services**.

Deployment as a Modular Monolith

Not all applications start out as micro-services. Many of them start as **well-organized Modular Monoliths** until they reach the critical mass at which it makes sense for us to break into separate teams and deployments.

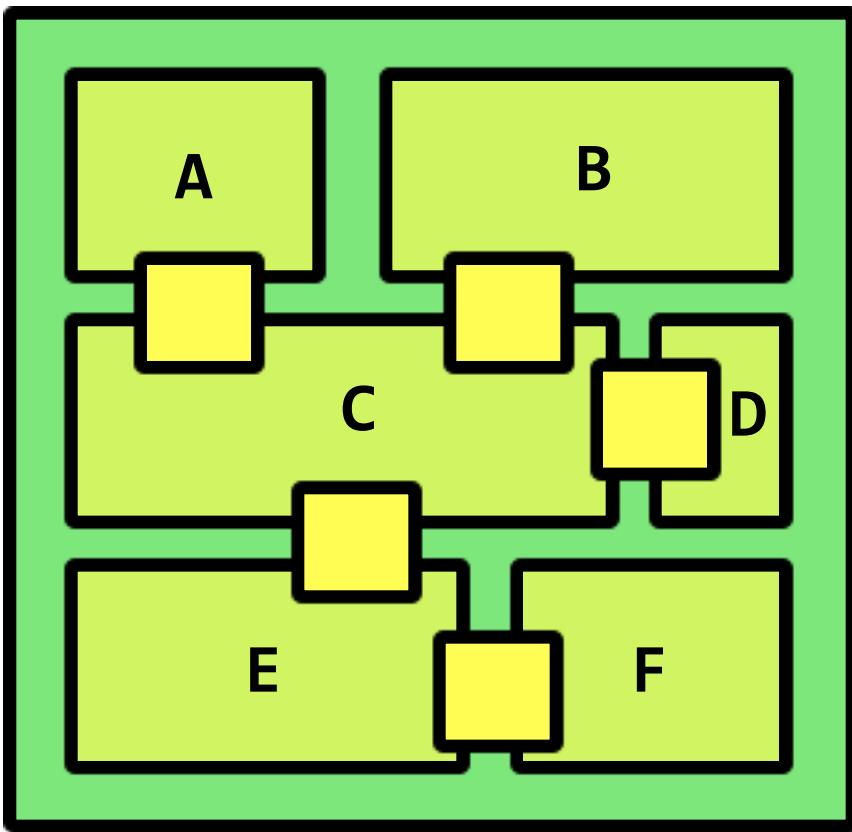
If we were to start building an application as a monolith, at the *folder* level, we could enforce those *Subdomain* boundaries by using the **package by module** principle.

```

src
  modules      # All subdomains
  ...
  billing      # Customers, subscriptions, invoices, etc
  catalog      # Vinyl, artists, albums, etc
  notifications # Email, push notifications, slack webhooks
  shipping      # Shipments, tracking, routes, etc
  trading       # Trades, offers, reputation, etc
  user          # Users, passwords, jwt, roles, groups, etc

```

In this case, each of the Subdomains for a one *Bounded Context* live within a single deployable unit.



Modular Monolith: a single Bounded Context with several Subdomains within it.

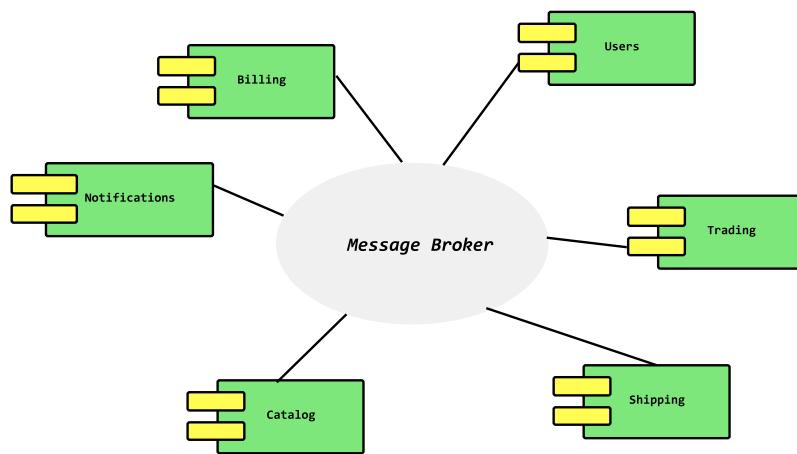
If we modeled **one of the DDD-apps** like DDDMerch as a *Modular Monolith*, what do we do when we want to also deploy another one like DDDDating or DDDForum?

What should we do if we want to utilize common *Subdomains* across *Bounded Contexts* (DDDDating, DDDForum both need users and notifications for example) but we **don't want to maintain duplicate Subdomain code across separate Bounded Contexts**? How do we avoid this turning into a development nightmare?

This would be an appropriate time to investigate *Distributed Micro-services*.

Deployment as Distributed Micro-services

As team sizes and traffic needs grow, and as more applications enter our enterprise, we can break *Subdomains* into their own *Bounded Contexts* so that they can be managed and deployed as *Distributed Micro-services*.



A generalization of a microservice architecture where each microservice is a subdomain from our problem domain.

There's a considerable amount of complexity added here, and that's not to be underestimated. As long as we have networking and ops under control, teams can be assigned to manage a bounded context and integrate with others. This *event-driven* architecture is arguably the best way to scale an application— both in terms of code size & complexity, and traffic.

Design principle: “Strive for loose coupling between objects that interact.”

Implementing the one database per service pattern, we can think of our enterprise as a **platform to be built on top of**. This removes the need to perform duplicate work and alter the existing platform, and shifts the work towards simply integrating with the existing services.

Note: If we can utilize our existing architecture to build new applications *on top of it with minimal-to-zero* modification necessary, that means we're applying the Open-Closed principle *architecturally*. That right there is a beautiful thing.

How to plan a new project

I love the process of starting a new project. After having come up with a plan, I get this efficiency and confidence boost, knowing that I'm not going to run into any huge surprises, and it feels empowering to know how I plan to use my energy in a targeted way.

Unfortunately, at least for me- I wasn't taught suitable formal methods for planning projects. Sure, I learned about UML Use Case and Class Diagrams, but they weren't useful in every scenario. They felt quite *ceremonious* for many projects.

As many readers of this book, I began with questions like “what's the *best way* to plan a project? Should you start from the database and go *upwards* in the stack? Should I start with the API first? Maybe the UI first”?

I eventually developed a personal preference and stuck to it for years. That got me through a lot of relatively simple projects for quite some time.

But it wasn't until I started working in a team setting on large, complex, challenging projects in *domains that I knew nothing about* that I realized I needed a different approach towards project planning.

With the pressure to deliver working code within strict time frames, it's not uncommon that you'll encounter pushback to shorten deadlines and eliminate upfront design. Because we know that the early design efforts in a project have the potential to influence its success or failure, we should never sacrifice.

In this section, I'll cover several conventional approaches to project planning, how to identify which approach would be most appropriate given the context, and two practical domain-driven approaches for project planning.

We'll learn about:

- Imperative design processes (Database-first, API-first, and UI-first) benefits and drawbacks.
- Project dimensions that influence which design approach makes the most sense.
- Planning a project using traditional use case design.
- How identifying roles and applying boundaries using Conway's Law can help to create high-quality designs.
- Event Storming: a design technique (focused more on the problem domain rather than the technology) that involves both Domain Experts and developers.
- Event Modeling: an emerging technique similar to Event Storming that combines and builds upon the best of over 40 years of design principles and best practices.

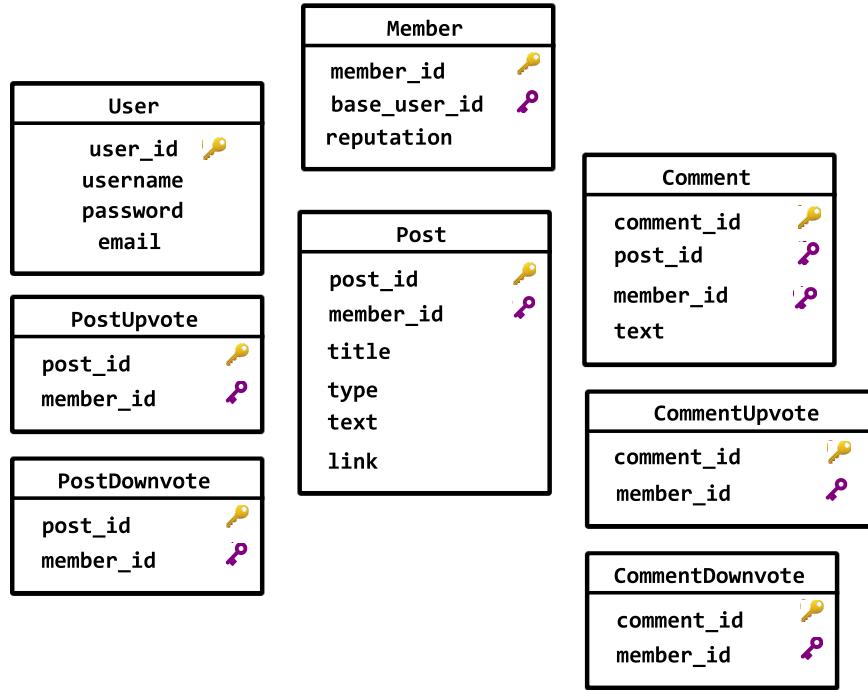
Imperative design

A trivial and imperative way to plan out the development of a project is to single out **one of the technologies required to complete the project** and code that out first. Components involved in building web apps using MVC are:

Database + RESTful API + Front-end application

Imperative design starts either database-first, UI-first, or API-first.

Taking a **database-first** approach, I might start by drawing out all of the tables I'm sure that I'll need, defining their relationships, and adding all of their columns.



Database-first design approach. Start with the database schema.

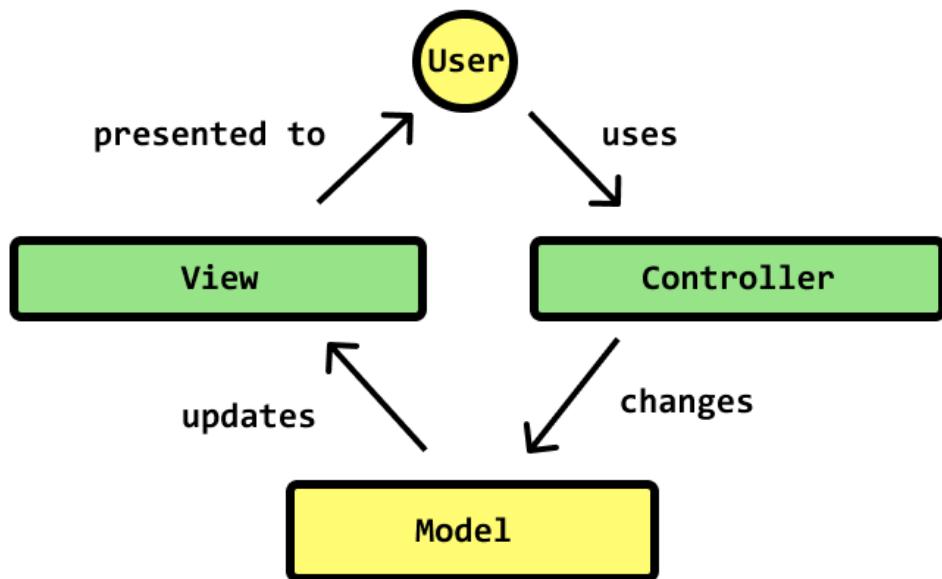
Doing design **UI-first** means that we start by creating all of the wireframes for the system. Then, we discover the API calls we need based on the functional requirements of the components. Additionally, the shape of the database is determined based on fields and relationships present in the UI.

API-first design involves identifying and enumerating all of the API calls up-front. Out of all three of these approaches, I like API-first design most. I like it most because it draws several parallels against starting with a Use Case Diagram. Naturally, that's because an API call *is* a use case and a use case is either a command OR a query. Identifying all of the use cases would help us understand at a high-level all of the application-specific functionality that we need to support — not a bad place to be.

I think the name **imperative design** is a proper name for this style of design because the process prioritizes merely completing all 3 parts of MVC architecture in order to produce a working, fully complete system.

Model View Controller

used to separate concerns between client & server



I don't think that's properly guided. Quite often, central business rules get displaced and put in places that tend to make it hard to maintain as projects grow in size and complexity. This imperative approach works well for certain types of applications: CRUD ones.

Imperative design approaches are for small, simple CRUD applications

Imperative design approaches work remarkably well for basic CRUD (Create Read Update & Delete) + MVC applications. That's because the *view is the UI*, the *controller is the API*, and the *model is the database*. Building all three components means we've completed the development of the application.

Code generators, realtime GraphQL APIs that try to be the entire M in MVC, and full-stack frameworks are great for simple CRUD-based imperative projects. Given that there are only three components to MVC, it's attractive to cut cost and time by having the framework do the *entire M* or the *entire C* for you. For simple CRUD apps, I say have at 'er. For anything else, *proceed with caution*.

Here's why.

Code quality quickly degrades on business-logic heavy projects designed and developed us-

ing one of these imperative design approaches.

Because of a lack of a domain model, codifying business rules becomes an afterthought. If business rules are present in the problem space, a common mistake is to *patch* duplicate logic throughout controller code or ineffectually locate it within an anemic service class.

In a nutshell, CRUD-based apps expect the M in MVC to represent the shape of the data simply. CRUD-based apps fail to account for designing the *behavior* of the data.

For more on this phenomenon, read “REST-first design is Imperative, DDD is Declarative” and “Knowing When CRUD & MVC Isn’t Enough”.

Dimensions that influence the design approach we should take

Ultimately speaking, the dimensions that are going to matter most in determining whether it’s OK to do imperative design are:

- If the project size is large.
- If the project *complexity* is high (we just discussed this one with CRUD apps).
- If the project is enterprise software (this is software serving an organization with several teams, where each team has a set of employees with roles that carry out a specific set of activities- think *major airports*, large e-commerce giants like *Amazon*, or *Wal-Mart*).
- If the team size or the number of teams working collaboratively on the project is significant.
- If we need to learn the problem domain from domain experts first.

If we can safely answer *no* to all of these, then there’s an excellent chance we’re dealing with a simple CRUD app, and we should be fine with the imperative approach.

Examples of CRUD projects that we’d be OK with designing imperatively: An admin dashboard to perform CRUD operations, todo apps, basic weather apps, a comment-moderation system, a home file sharing app, or other hobbyist projects.

If we answer *yes* to any of these, then its time to roll up our sleeves. It’s very likely we need to take a more involved approach to design because there’s a lot more is going against us to see to the success of the project.

Examples of ambitious projects that require better upfront design: A vinyl-trading application (like White Label), a large scale e-commerce website and fulfillment network (like Amazon.com), a source-code management platform (like GitHub).

Let’s look at conventional approaches to handle planning challenging software projects.

Use-case driven design

Programming can feel like a creative and never-ending art form. While I think there’s room for creative programming, in a professional setting, we want to know what we have to do, and what dictates a project being **functionally complete**.

Being use-case driven is arguably the best way to spend no more time on a project than is absolutely necessary. It helps you make better estimates, write more direct and intentional code, and plan out the tests you'll need in order to make sure your stuff works.

Resource: “Better Software Design with Application Layer Use Cases” is an essential read. It’s quite possibly my favourite essay. I highly recommend you take a read after you finish this section.

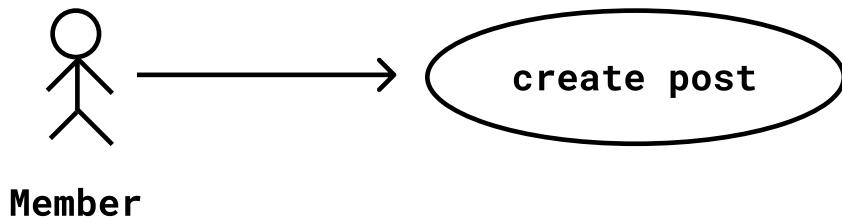
Use cases & actors

A **use case** is a high-level way to document behavior that an actor performs within a system.

An **actor** is a *role* played by either a specific type of user or a *system* itself (automations and external systems can be actors too — think payment processors like Stripe).

For example:

- If the domain is an **ecommerce application**, an *actor* might be a customer, and some of their use cases might be to `makePurchase`, `getOrders`, `search`, and `postReview`.
- If the domain is a **forum application**, an *actor* might be a member, and some of their use cases might be to `createPost`, `postComment`, `upvotePost`, and `getPopularPosts`.



A very simple use case diagram of a Forum system.

Use case design is one of the more traditional approaches of documenting the *functional requirements* of a system, usually before writing any code.

Applications are groupings of use cases

All applications can be thought of as **a grouping of use cases**. That's all apps are anyways.

For example, in a simple Todo app, the use cases that the **actor** needs to be able to accomplish are: `create todos`, `edit todos`, `delete todos`, and `get todos`.

One could argue, “*is it still a todo app if it doesn't have all the use cases that a todo app needs to have*”?

While it might pose as a light-hearted theoretical discussion for another day, we can at least agree that the app is *not complete* until all of the **agreed-upon** use cases are built and working.

You can see how this is useful if we need to scope out a project for a potential client, creating a proposal for the work we'll do.

This is why I love use case design. The work is done once we decide on all the use cases and implement them. Again, since software development can be a creatively unbounded practice, it's helpful to have a way to *objectively* understand what the completion state looks like.

A use case is a command or a query

The CQS (Command-Query Separation) principle from Chapter 6 - Design Principles says that every operation should be either a *command* (something that changes the system) or a *query* (something that returns data from the system), but never both.

Use cases follow this principle. Instead of designing a use case to perform both a command *and* a query like `createAndReturnPost`, we'd ensure two separate code paths for *writes* and *reads* by designing `createPost` (the command) and `getPostById` (the query).

We yield the same benefits in simplicity when applying CQS at the design-level with use cases documentation as we do at the:

- method-level: `createPost(post: Post) + getPostById(postId: PostId)`
- API-level: `POST /post/new + GET /post/:postId`
- and architectural-level: Post write model(aggregate) + PostDetails read model(DTO or GraphQL object type)

Use case artifacts

There are two artifacts that you can create from doing use case design:

1. **Use case diagrams** and
2. **Use case reports**

Diagrams help to understand at a high-level what *systems*, *actors*, and use cases exist.

Reports are good compliments to use case diagrams. They can contain quite a bit more detail and are most useful when they document the **functional requirements** and how the system should respond to different scenarios when the **preconditions** change.

Functional requirements document business logic

When creating a **use case report**, we can outline all of the **functional requirements**. These are our use cases. But how do we document business logic and how the system should respond in various scenarios?

Given-When-Then

In addition to documenting our use case itself, we can utilize both **preconditions** and **post-conditions** to provide additional context as to how the system should interact in certain scenarios.

This is often enough information for developers to translate it directly into failing BDD-style unit tests, and then write the code to make the tests pass.

Let me show you what I mean by translating a DDDForum.com requirement spoken in plain English to BDD-style unit tests.

Given an existing post that the member hasn't yet upvoted, when they upvote it, then the post's score should increase by one.

Use case name: Upvote Post.

Precondition(s): A Member exists. A Post created by a different member also exists. The Member hasn't yet upvoted the Post.

Postcondition(s): The Posts score increased by one.

To translate that into a failing test case, we could simply write enough code to express what should happen (not focused too much on the design at this point), creating any classes that we mention in the test case:

```
// A failing BDD-style unit test.

let post: Post;
let member: Member;
let postService: PostService;
let existingPostVotes: PostVote[];

describe("A post the member hasn't upvoted", () => {

  beforeEach (() => {
    post = null;
    member = null;
    existingPostVotes = [];
  })

  it ("When upvoted, it should upvote the post's score by one", () => {
    // Start out with a failing test.
    let initialScore: number;
    post = Post.create(...);
    initialScore = post.score;
    member = Member.create(...);
    postService.upvotePost(existingPostVotes, member, post);

    // Should fail since we haven't written any logic yet
    expect(post.score).toEqual(initialScore + 1);
    expect(post.votes.getNewItems().length).toEqual(1)
  })
})
```

The goal from here on would be to further flesh out the classes with domain logic and continue until all tests pass.

As we identify more test cases, we should aim to write tests for those as well. Tests are how we can tell if our use cases that utilize domain layer entities are correct.

Parallels with API-first design

Use cases help us understand what needs to happen at the *business level*, which is why I think API-first design is adequate for most scenarios.

With API-first design, we're *technically* discovering all the use cases. They're just masqueraded as API calls. If you were to do a API-first design, functionally, you're doing use-case driven design so long as you include Given-When-Then test cases for each API call in your planning.

Steps to implement use case design

If you want to use this approach on your next project, here's are a few steps that I use. Here's how to get started.

1 - Identify the roles of the **actors** using the system.

Simply put, *who* needs a system built? Figure out what the role of that person is.

2 - Understand their end **goal(s)**.

The end goal for someone who needs a todo app might sound like "I need to be more organized with my daily tasks".

The end goal for a vinyl enthusiast might be "I want to make some money off of my vinyl collection" or "I want to trade in some of my old records for better ones."

For DDDForum.com, the goal for someone interested in DDD might be that they "want a place to learn about DDD" and get their questions answered.

3 - Identify the **system(s)** that need to be constructed in order to enable the **actor(s)** to meet their goals.

For the todo app, a desktop or mobile app might do.

For White Label, the vinyl marketplace, we might need a web app for Traders to make trades, an admin panel for Admins to monitor activity and perform administrative tasks, and an application for Shipping Staff to pack items and track that they're delivered.

4 - For each actor in each system, list out all of the use cases involved in helping that actor meet their goal(s).

For example, in the vinyl-trading enterprise comprised of a) a trading web app, b) an admin panel, c) a packing and shipping application, Traders from the trading web app (a) requires several use cases like:

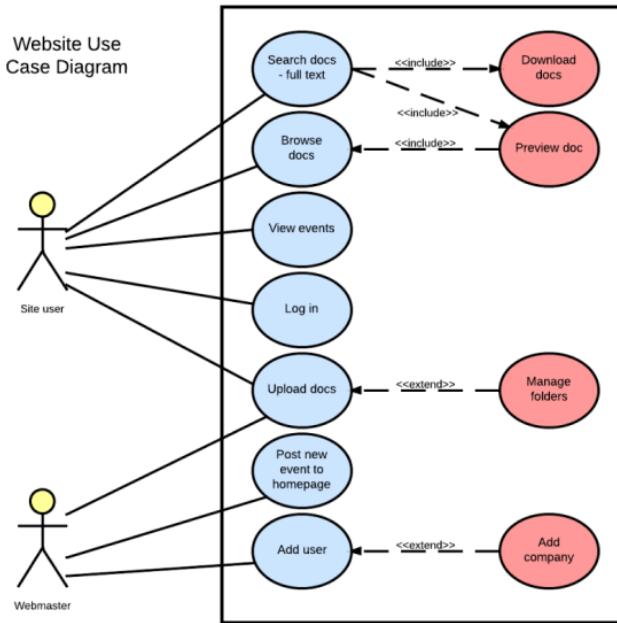
- postVinyl (details: VinylDetails): void
- getVinylDetails (vinylId: VinylId): VinylDetails
- makeOffer (tradeItems: TradeItems[]): void
- etc

Let's try this out with DDDForum by making some UML use case diagrams.

Planning with UML Use Case Diagrams

If you get tired of reading and want to learn more about how to design use case diagrams, here's an excellent free video tutorial. You'll want to stick around and read this section either way, because use case diagrams are helpful **up until a point**.

In use case diagrams, the *square* represents the system, the *stick-man* represents an actor of the system, and the *circles* represent the use cases. The use cases connect to the actor(s) that should be able to execute them.



A slightly more detailed use case diagram of a website.

I — Identifying the actors

Let's start with the *roles*. Who is involved?

We know that DDDForum.com is an online forum site where people can learn about DDD and get their questions answered, so we could start by identifying two roles: Members and Visitors.

Members are users who have registered to the site and can post questions, articles, comments, and cast votes while Visitors are anyone who hasn't created an account and is just an anonymous user.

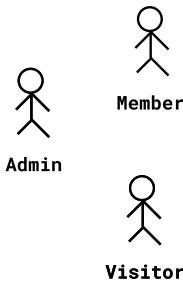


Member



Visitor

There's another type of role in this domain that could also be important, and that's the Admin role.



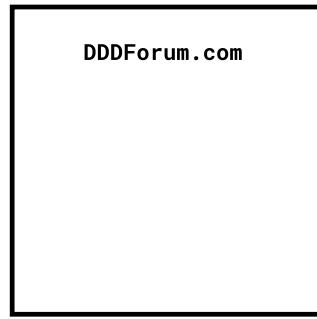
2 — Identifying the actor goals

We're creating an application for users to **learn about DDD and get their questions answered**. That's a common goal for Visitors and Members.

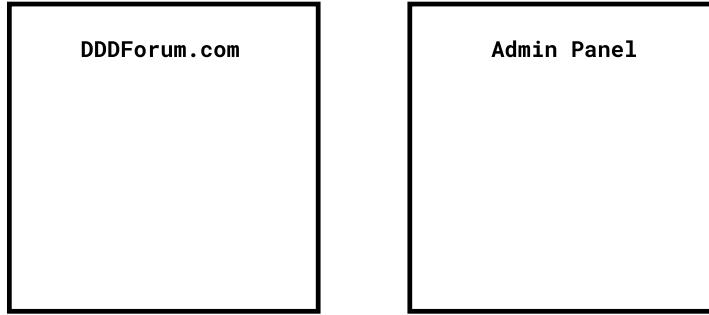
For Admin users, their goal is to **ensure that the DDDForum community is welcoming, helpful, and respectful**. To do that, they need the ability to **moderate users and content posted within the site**.

3 — Identifying the systems we need to create

It may not come as a surprise to you, but we need to build the actual forum site for Members and Visitors, so *DDDForum.com* is the first system we need to build.

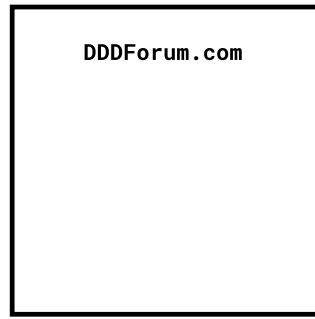


For Admin users, the *admin panel* could exist to serve their moderation needs.

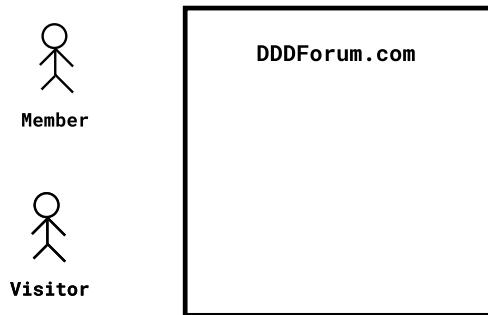


To contain the scope of our discussion in this chapter, we're just going to focus on DDDForum.com for Members and Visitors, leaving out the admin panel and Admins for now.

We might explore including the Admin Panel in this chapter in a future release of this book.



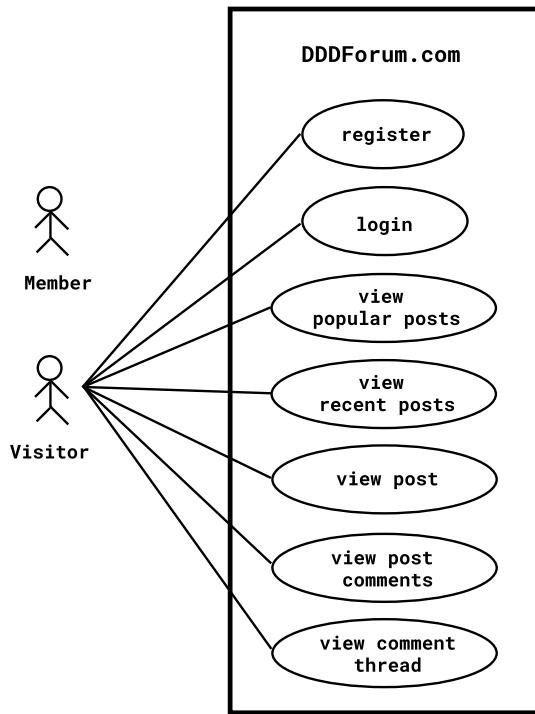
OK, so the DDDForum.com is a system that is going to serve the needs of Members and Visitors.



What can they do within the system? Let's think of a few use cases.

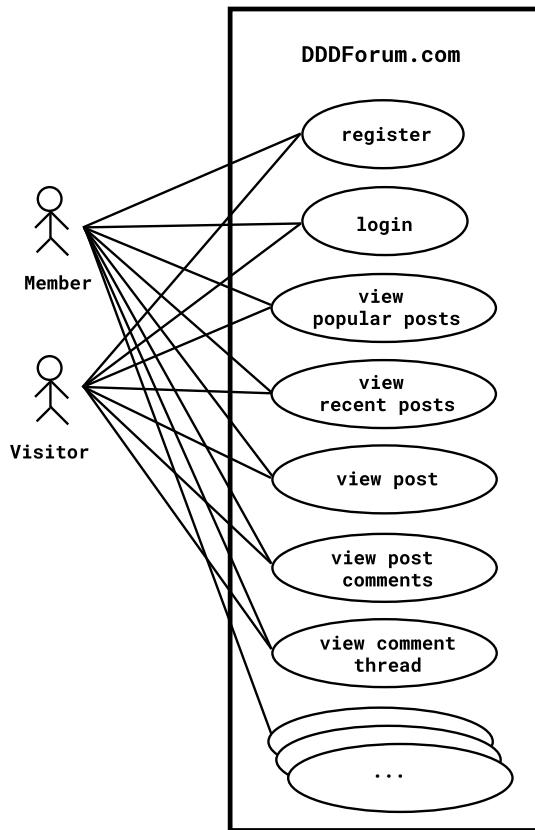
4 — Identifying the use cases for each role

Starting with Visitors, we can brainstorm use cases that describe the capabilities of a Visitor. I landed on facts that specify they can register, login, view popular posts, view recent posts, view a specific post, view the comments for the post, view a comment thread and...that might be just about it. Don't worry if you miss a few right now.

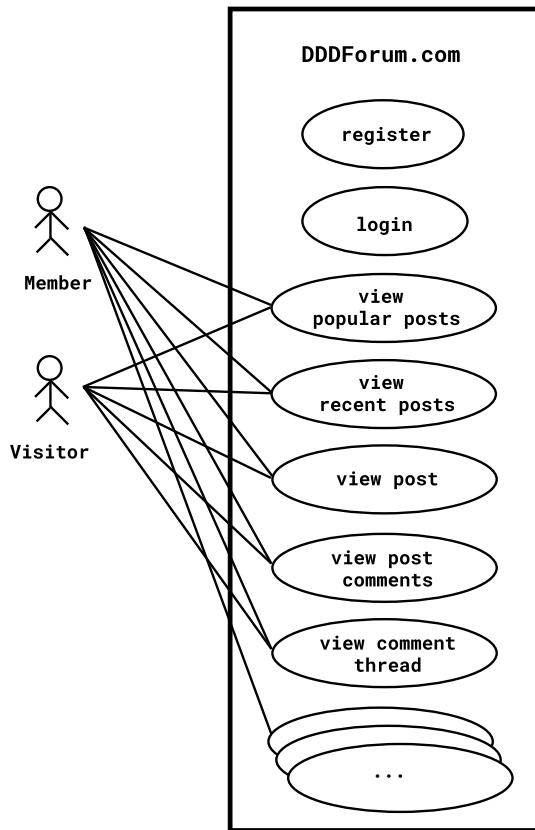


Remember that each of these use cases are either a **command** or a **query**. Try to identify which are which.

And let's hook up some of the use cases for Members. Members can do everything Visitors can do and more (we'll document all of those soon).



I know things are going well so far, but I'm about to plunge a stick in the spokes. I need to raise something important about the `register` and `login` use cases. Let's remove the lines that show that Members and Visitors can log in and register.



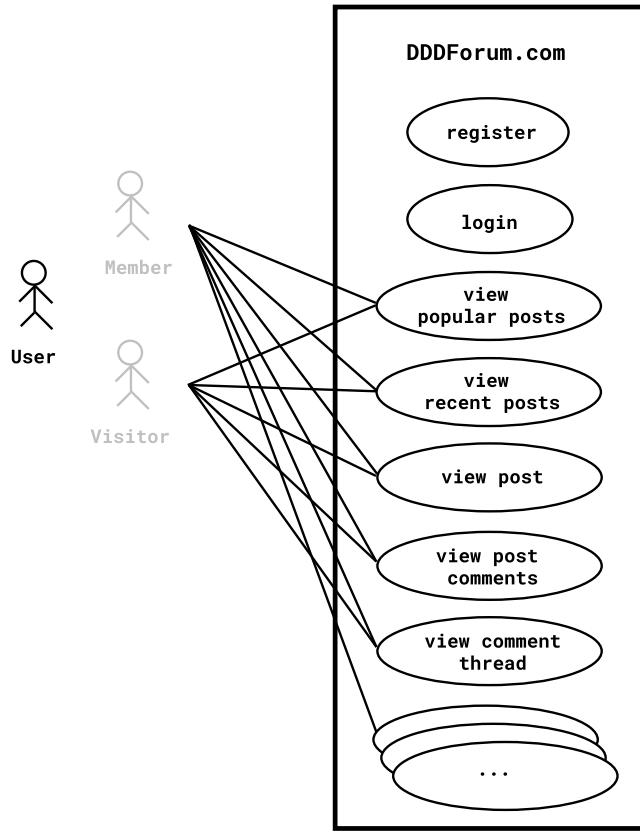
Why did I do that? Well, it's *complicated* and long-winded, but it has to do with *role, boundaries*, and, most importantly- *Conway's Law*. The detour is worthwhile (trust me).

Roles, boundaries, and Conway's Law in Use Case Design

Starting with *role*, let me ask you a question. If you were doing this by yourself for the first time, would you have labeled Visitor or Member as User? I know I would have.

So why didn't we?

Why didn't we just label User as the primary actor?



It's because *role* dictates *responsibility*.

Role dictates responsibility

While we *could* call everyone a User, that could be unproductive to the **ubiquitous language**.

Recall that in DDD, a considerable part of our domain modeling efforts is to identify and capture a *shared language* used between domain experts, programmers, and anyone in-between?

That language that we capture should appear everywhere: in conversation, in code, and technical documents like *this one*.

Because we're developers and developers are accustomed to calling everything a User, it's easy to fall into this trap. If we were working with domain experts, it would be more unlikely we'd hear them use the term User to describe the role of someone that they work with or manage in their day-to-day.

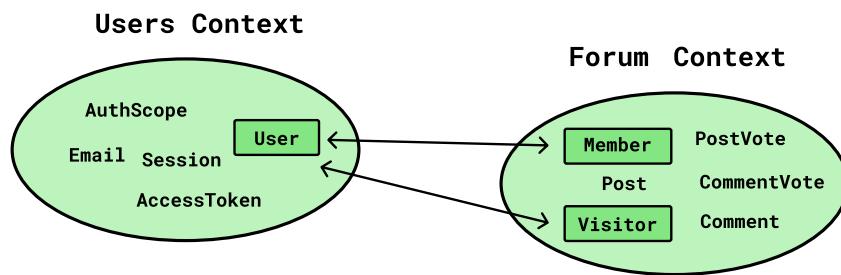
Normally, you'd work collaboratively with domain experts to identify the common language.

There is a time and place to call a user a User, such as in an **Identity & Access Management / Authentication** context (like if we were building services like Amazon IAM, Amazon Cog-

nito, or Autho). If our **core domain was IAM (auth)**, the primary actor type *is*, in fact, a User- since an Identity & Access Management (auth context) *makes no assumption about the role of users outside of its context*. Autho and other popular IAM services understand that developers integrate with a domain unrelated to their API, and the role of User is of a similar level of importance but has a different meaning in a separate context.

Comparing an **Authentication (Identity & Access Management)** context to a **Forum** context, the concept of **User** is the same, but *different* for each context.

Here's an example of a *Context Map* to illustrate what I mean.



Going beyond the Forum context, I can think of plenty of other alternatives to User depending on the context:

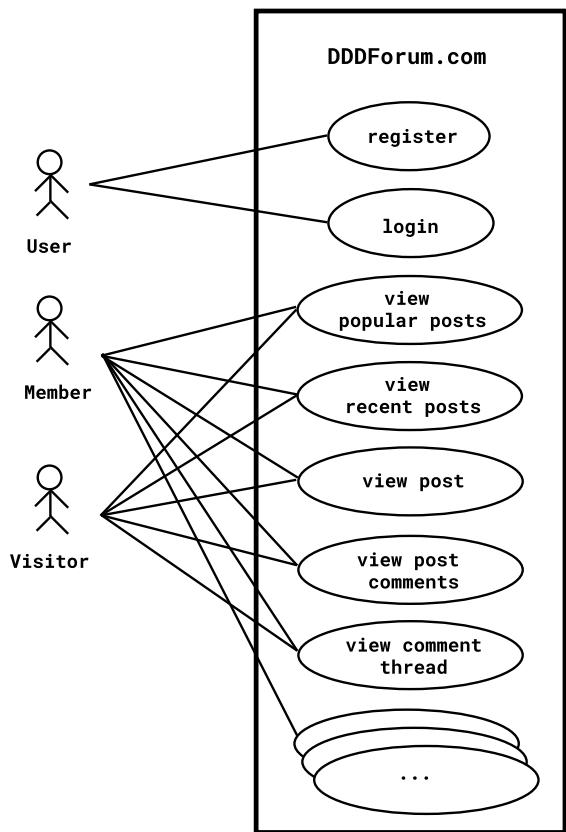
- A billing system: Customer, Subscriber, Accountant, Treasurer, Employee
- A blogging system: Editor, Reviewer, Guest, Author
- A recruitment platform: JobSeeker, Employer, Interviewer, Recruiter
- A vinyl-trading application: Trader, Admin
- An email marketing company: Contact, Recipient, Sender, ListOwner

Get the point? **Role** matters. When identifying **actors**, name them based on their *role*.

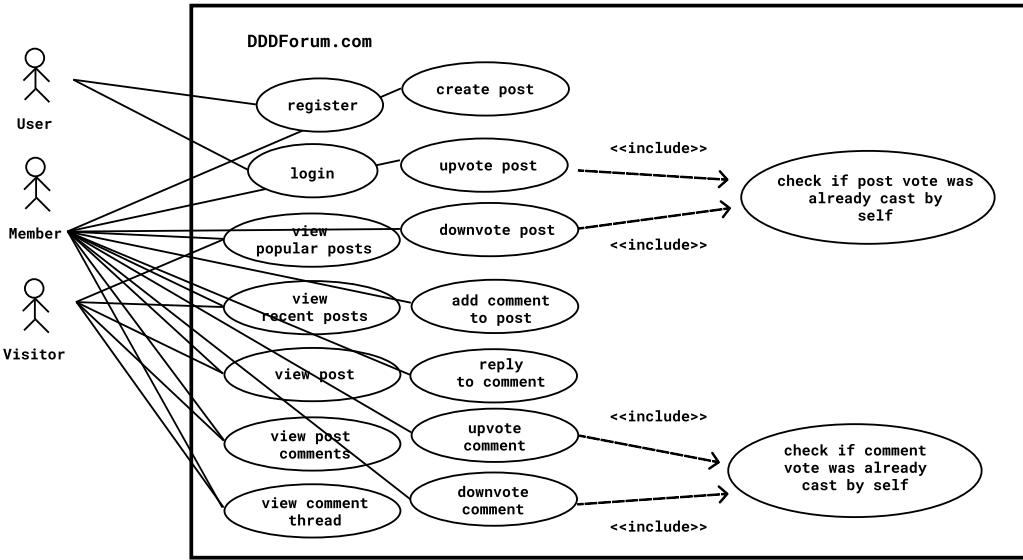
Because role dictates responsibility, sometimes when we uncover a responsibility (like log in and register), we have to ask ourselves if we've assigned it to the correct role. In this case, it's good that we were specific about Members and Visitors because it enabled us to understand the responsibilities that they hold from within the context of Forum.

Identifying actors in our systems by thinking about their role helps us **determine what their responsibilities are, and more importantly, what they're not**.

We should now understand that the register and log in use cases *are not* significant to a Member or Visitor, but they *are* significant to a User.



We could continue by adding as many other things that a Member could do, but using the Use Case format, it's challenging to do it in a way that doesn't become messy.



UML Use Case Diagram without a good separation of concerns

Yeah, that doesn't look great. And I didn't even get to add all of the use cases. There's a bunch of other things that a User is responsible for, like getting the user account, deleting their account, and so on. This clutter signifies a design problem with use cases.

Everything from auth use cases, to forum use cases, and perhaps even if we wanted to send notifications- notifications use cases, are all clumped together in this document.

We need a way to represent **boundaries**.

Boundaries

The biggest problem with use case diagrams is the lack of being able to represent *architectural boundaries*.

I understand boundaries as a *logical surface area where every construct is in the same context*.

Previously, we say that we had an Auth/Users context and a Forum context. Because use cases tend to be high-level documents, they have trouble representing boundaries. As a result, having all the use cases for a system with several boundaries within one grouping can make the design overwhelming.

You saw the mess we made a moment ago.

What we need is a good way to represent the *architectural boundaries* and the use cases that belong to those boundaries within our system(s).

In Domain-Driven Design, the concept of *subdomains* is equivalent to these boundaries.

Using subdomains to define logical boundaries in DDDForum

We're going to organize all the actors and their use cases into subdomains.

First, let's list all our use cases (commands & queries) out.

Use Cases

- Register
- Login
- Logout
- Get current user
- Get user by user name
- Refresh access token
- Verify email
- Delete user
- Create post
- Delete post
- Downvote post
- Upvote post
- Get popular posts
- Get recent posts
- Get post by slug
- Get comments for post
- Get comment thread
- Upvote comment
- Downvote comment
- Reply to comment
- Reply to post

And let's list all of the actors.

Actors

- User
- Member
- Visitors

I don't know if you can see this, but there are two **subdomains** that I see right away.

There is a forum subdomain, which appears to be our **core subdomain** that allows us to focus on posts, comments, votes, and such - an essential part of our application.

There's also a generic users subdomain which takes care of all of our identity and access management for users.

As well, the primary actor in the forum is the member, while the primary actor of the users subdomain is the user.



Users
subdomain



Forum
subdomain

Users (generic) and Forum (core) subdomains for DDDForum.com

We might also have *one more* subdomain for notifications.

You see, if we're going to be *sending emails* to do *email verification* or if we're going to be sending notifications to members if someone replies to their comment, we need another *supporting* subdomain to decouple the concerns of everything related to emails, notifications, etc.



Users
subdomain



Forum
subdomain

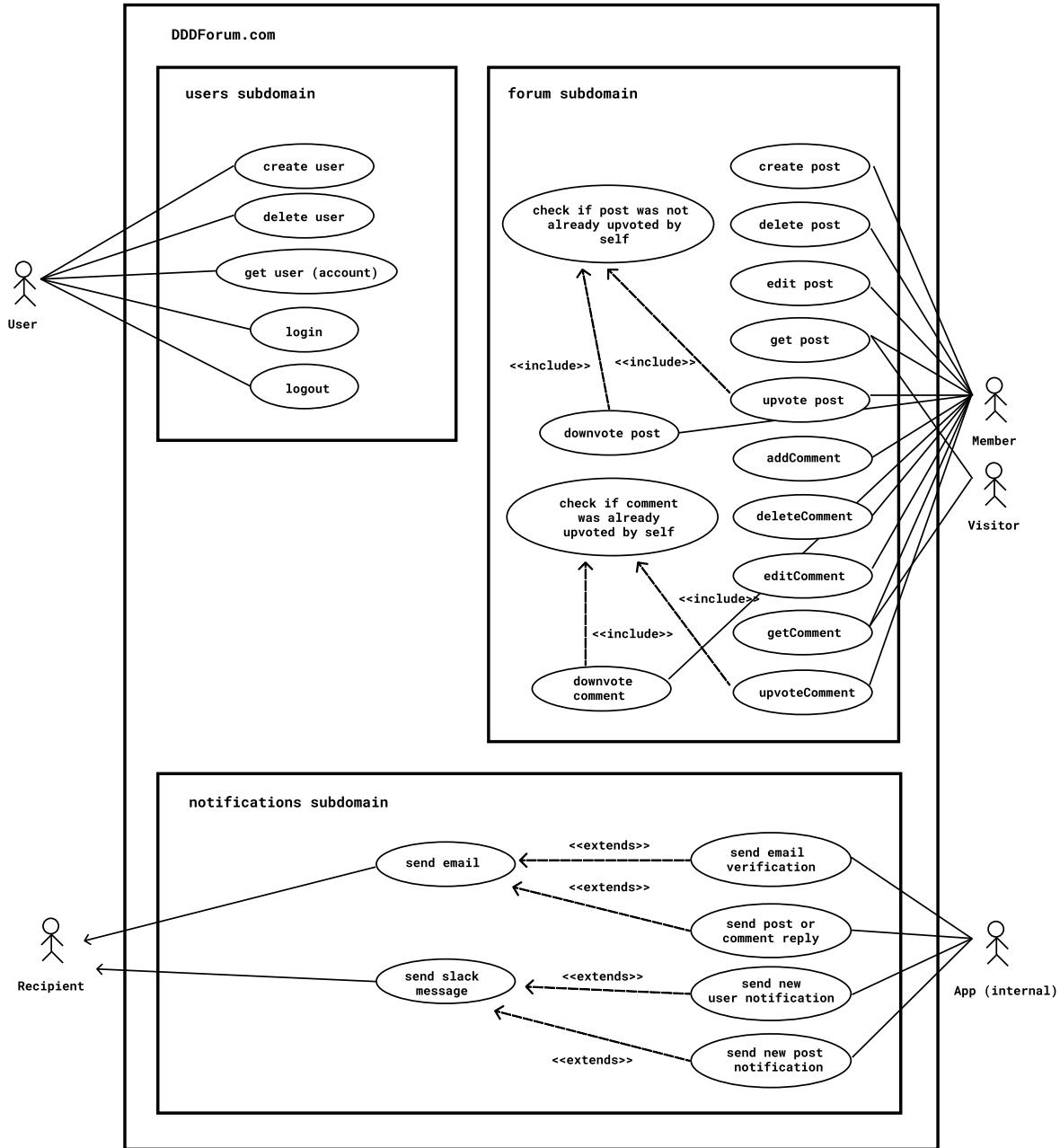


Notifications
subdomain

Users (generic), Forum (core) subdomain and Notifications (supporting) subdomains for DDDForum.com

Excellent.

Let's decompose the *system* from our use case diagram into smaller pieces based on our sub-domains. Here's the refactored use case diagram of DDDForum.com.



A use case diagram displaying the entire DDDForum.com system and the subdomains that it's comprised of.

I like this a lot more. Here's what we can take away from this now:

- We know that there are 3 subdomains: users, notifications, forum.
- We know the **specific subdomains** our system needs.
- We know the **actors for each subdomain**.
- We know all of the **use cases**, the **actors** that can execute them, and the **subdomain** they belong to.

Conway's Law

How did I know we needed a users, forum, and notifications subdomain?

Well, there's the fact that I've done this several times before, but there's also a useful fact.

A *law*, actually. It's called *Conway's Law*.

In 1967, Melvin Conway, a clever computer scientist and object-modeler, was credited with the following quote with respect to designing systems. He said:

"Organizations which design systems are constrained to produce designs which are *copies of the communication structures of these organizations*". — Wikipedia.

In my terms, Conway is saying:

When we build software, we need to know the different groups/teams/roles of people it serves, and divide the app up into separate parts, similar to how those groups of people *normally* communicate in real life.

Remember that this is the *first* step of building use case diagrams? Now we understand *why*.

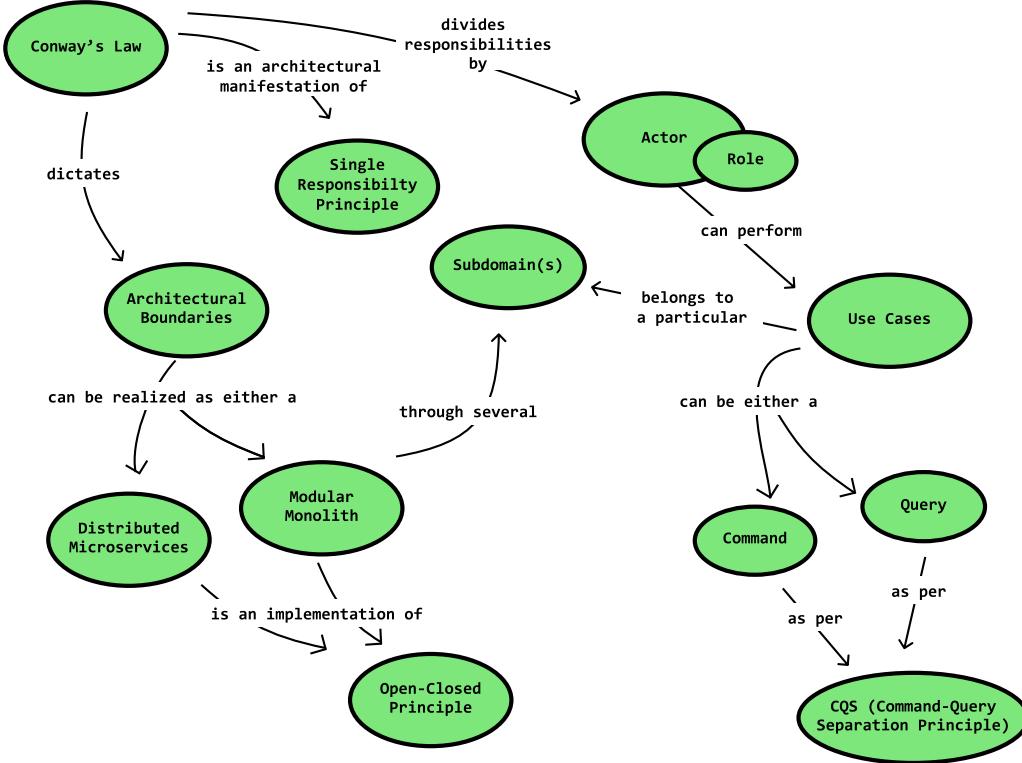
SRP (Single Responsibility Principle): Conway somehow discovered that if we allow each group to have their system, we constrain the possible surface area of required changes from one group rippling into another one as side effects, thus satisfying the **single responsibility principle** (before it was even discovered).

Boundaries: Conway's law dictates **architectural boundaries**, which informs architects how to split an application into either **distributed micro-services** (micro-services split up and networked together) or a **modular monolith** (several subdomains well-separated within one deployable unit of code).

So, to *discover the subdomains*, all we have to do is think about the **different teams that we could assemble** to take ownership over a *specific part* of the business.

And if we **know the teams (subdomains like users, forum, notifications)**, then every domain concept like an actor/role (and their use cases) belongs to a subdomain.

Tying everything together, here's an illustration of the influence Conway's Law has on architecture.



This is my favourite diagram in solidbook.io. Really think about these pathways and see if they makes sense to you.

Summary on use case diagrams

Use case diagrams and reports are pretty useful tools that you can use to document project requirements and business rules with test cases.

I would advocate for using use case diagrams when we understand the domain, and we're ok with doing the majority of the use case modeling work in isolation, away from domain experts who might not understand even the slightest semantics of use case diagrams.

However, it can be risky for developers to spend design time alone since we know that it's the initial design of a project that has the potential to have the most profound impact on the overall quality of the system.

There must be a design tool that involves both the developers and the domain experts in this process.

There is, and it's called **Event Storming**.

Event Storming

A group or workshop-based modeling technique that brings stakeholders and developers together in order to understand the domain quickly.



We use lots of sticky notes when we do Event Storming sessions.

A developer named Alberto Brandolini found himself short on time for a traditional UML use case design session, but improvised with some sticky notes, markers, and a whiteboard, inadvertently creating Event Storming.

Event Storming has become something of a staple in the DDD community. It's an interactive design process that engages both developers and business-folk to quickly and cheaply learn the business and create a shared understanding of the *problem domain*. The result is either:

- a) a **big-picture** understanding of the domain (less precise).
 - b) a **design-level** understanding (more precise), which yields software artifacts (*aggregates, commands, views, domain events*) agreed on by both developers *and* domain experts that can be turned into rich domain layer code.

It works by:

- **Getting the *right people* in the room.** Ideally, we want the developers building the system, developers responsible for other third party systems we need to integrate with (if any), and domain experts we're building the system for (the *actors* we need to serve). You'll need people to answer questions and help build the *ubiquitous language*.
 - **Finding a large erasable surface like a whiteboard or a wall to place stickies and draw on.** You can also use a long roll of paper, which may work as a better adhesive

for the stickies. Alternatively, if you work on a distributed team, you can use software like Miro.

- **Bringing a lot of colored stickies and markers.** We'll write the name of domain concepts we discover on these and place them on the wall.
- **Spending anywhere from a couple hours working on creating a model.** You'll want to take breaks and bring snacks. It can be pretty intense- that's because it's straight-up critical thinking. Not the type of activity you can do while knitting or playing an iPhone game. You can chunk it out over the period of a couple of days in one or two hour sessions, or do it all in a day. It might be a challenge to convince management to get everyone together for an extended amount of time; though, the time is well spent for the return on investment of a high-quality design for software that can last for years.

After the event storming session, you can:

- **Convert the design into code using the terminology agreed upon in the session.** When we use the ubiquitous language and a layered architecture, our domain layer code is declarative and can be understood by domain experts. Developers have a shared mental model of what the business is, and how we're representing the *solution space* in code.
- **Understand the relationships between your core domain and other supporting subdomains.** Sometimes we might not wish to code everything ourselves. Sometimes we might just want to *buy a tool* instead of developing it in house. Again, consider Auth0 for the Users subdomain, or Pusher for a Notifications subdomain. It's likely that Users and Notifications aren't part of your core domain. After event storming, we have a better understanding of those boundaries, and we can make an informed decision.
- **Run another event storming session to evolve the model.** When new business requirements or rules come into existence, it doesn't hurt to run another session to determine where in the timeline of events things need to change.
- **Take it further by defining scenarios with domain experts and then building a series of test cases to exercise that the model is working correctly.** Domain experts can help you ensure that you're using quality test data when testing against your scenarios. They're also essential in verifying that the model is correct since they are the ones that know the domain best.

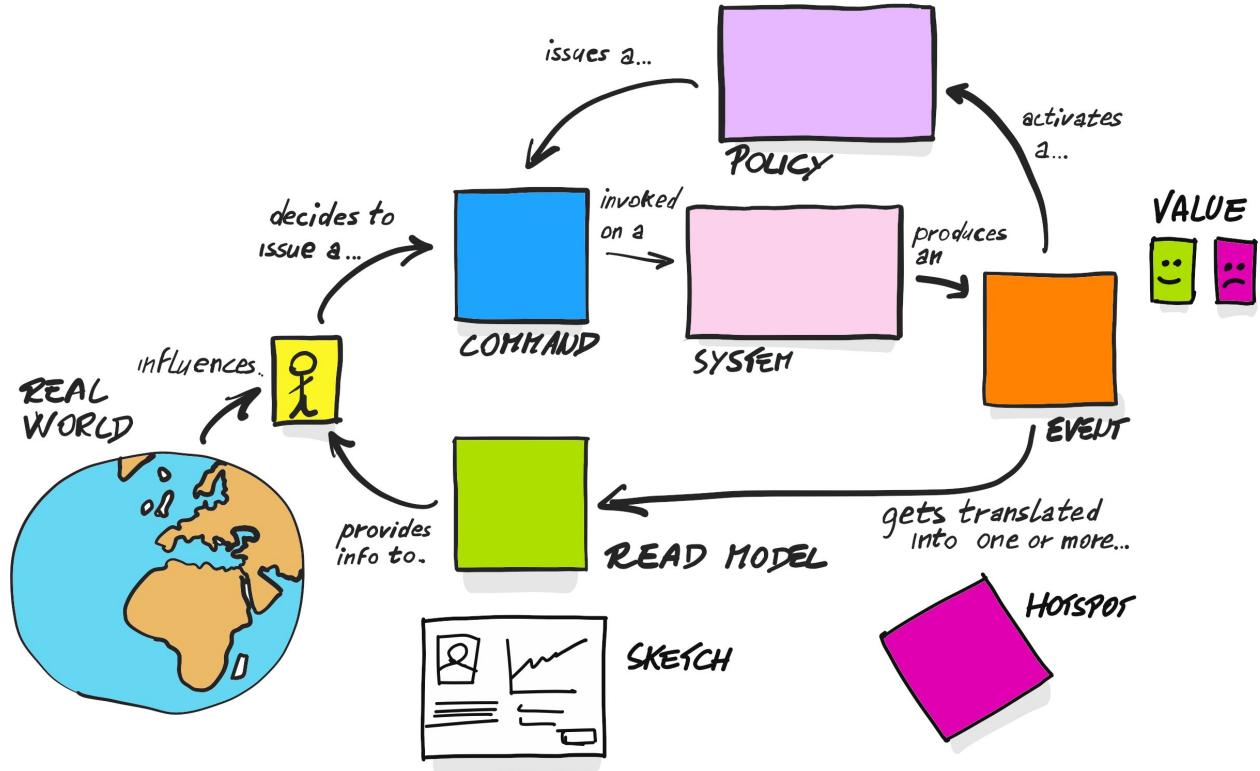
Why we need event storming

The domain experts are the ones that understand the business best because it's them, not the developers, that *live it*.

When design choices are made in isolation by developers, we may end up with software that doesn't fully meet the needs of its users.

Additionally, when developers code without fully understanding the domain, each developer is left with their *own singular understanding of the domain*, which may not line up with what is actually implemented. Brandolini says,

“Too many developers on the project make individual mental models; this makes the project unreliable” - via Twitter (@ziobrando - Oct 18th, 2019)



Event storming in a nutshell — via @ziobrando.

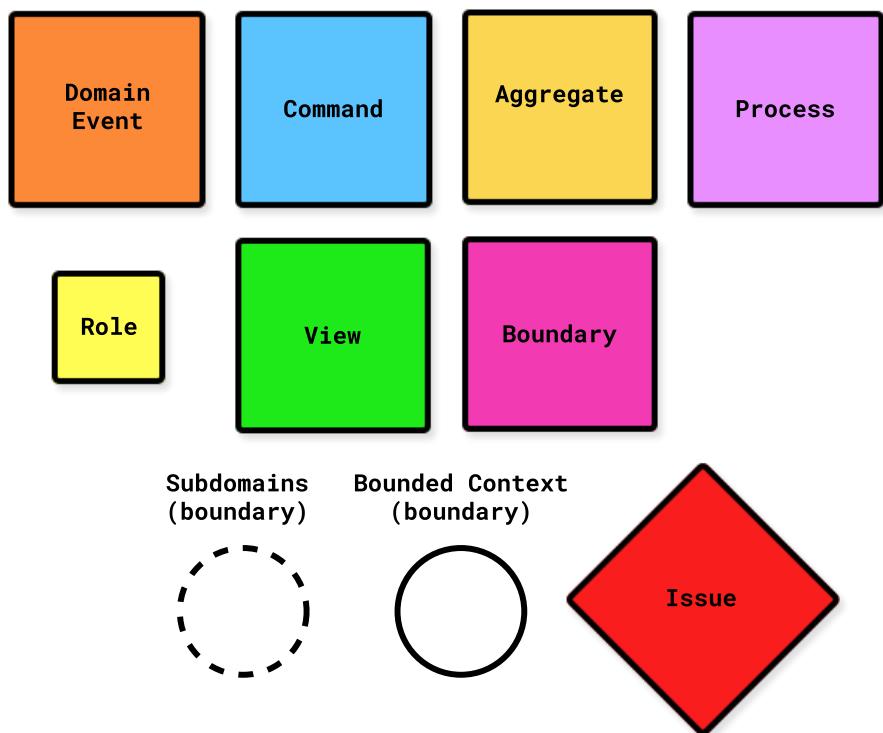
How to conduct an event storming session

Let's walk through each of the steps involved in holding an event storming session with your team.

Step 0 — Create a legend of all the event storming constructs

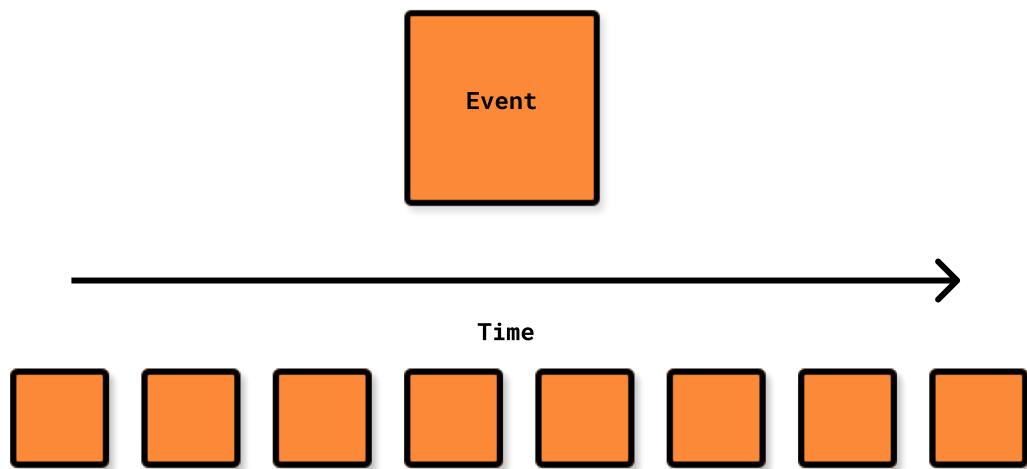
Before we even get started, Vaughn Vernon recommends we use a legend so that the event storming constructs and their color schemes are well understood by all participants.

- Domain Events — orange
- Commands — light blue
- Aggregates — yellow
- Issues — red
- Actors/Roles — light yellow or yellow with a stick figure
- Views — green
- Bounded contexts (boundary) - solid line + named w/ a pink sticky
- Subdomains (boundary) — dashed lines + named w/ a pink sticky
- Event Flow — arrows



Step 1 — Brainstorm Domain Events

From left to right, chronologically map out the *story* of the business using orange sticky notes.



To get started, everyone grabs some stickies, gets a sharpie, and works together to put the domain events on the board, making sure that the order of events is correct.

Thinking in terms of domain events is the closest we can get to expressing what happens in the real world. Non-technical folk can communicate the entire business process as a series of domain events.

```
UserRegistered -> EmailVerificationSent -> EmailVerified
```

A couple of things to note about this step:

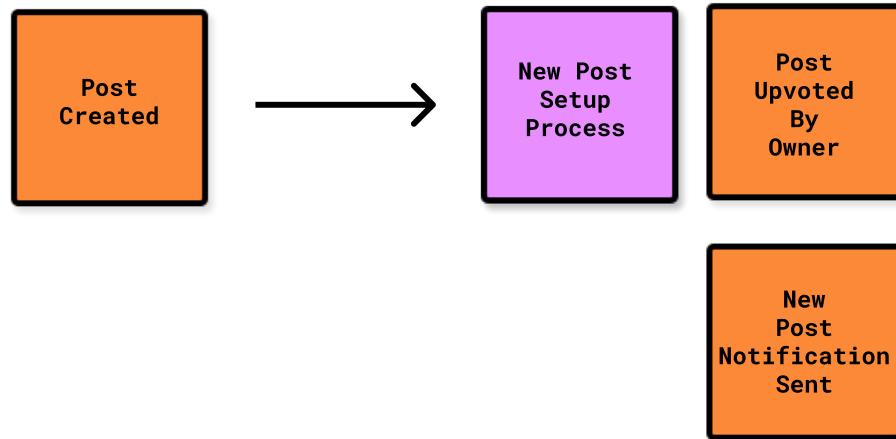
- **Domain events are *past-tense verbs*.** For subsequent steps, it's important to follow this convention. The challenge here is that domain experts often think of business processes as *tasks* rather than events, so we may have to warmly nurse them to get in the habit of using the past-tense format like PostCreated or CommentUpvoted.
- **Parallel or alternative domain events (like failure states) can be placed vertically.** For example, in an Orders domain, if we had a domain event called OrderPaid, it's also possible that the order could fail; that means we'd need an OrderPaymentFailed domain event as well. Because these are alternative outcomes, and one of these events occurs chronologically for some preconditions, we can place both outcomes on the board *vertically*.



Vertically placed domain events for scenarios where one of many meaningful domain events could occur.

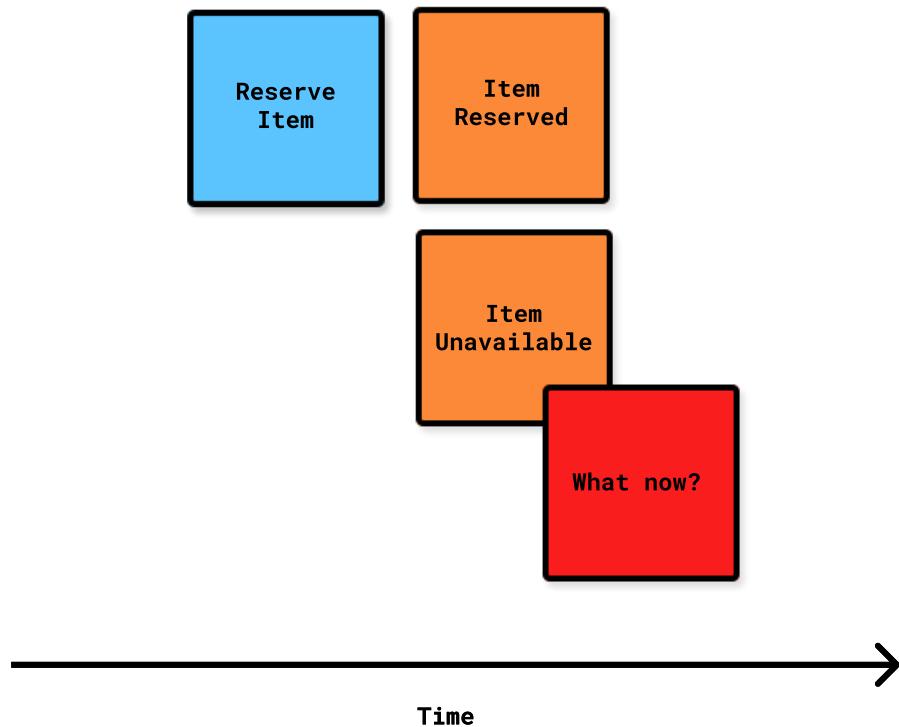
- **It's possible for domain events to be the result of something happening in another system that floats into this one.** In this case, we can still document the domain event.
- **When the outcome of a domain event is a process, document it with a purple sticky.** A process is any scenario where there is either a single step or multiple steps that we can collectively identify. If the process involves important domain events, we

can put stickies beside the process to represent them. Draw a line from the originating domain event to the process.



Example of a process started after the PostCreated domain event. PostUpvotedByOwner and NewPostNotificationSent are parallel domain events.

- If the process is something not important to the domain you're currently focused on (like an intensive user registration process, for example), we can just stick a simple domain event to represent the result of the entire process like UserRegistered.
- **Only list meaningful domain events.** Meaningful domain events are those that are followed up by a *process* or subsequent domain events. Refrain from listing domain events like PostCreationFailed where it's not followed by a process or a subsequent domain event.
- **If you find an area that's troublesome and you don't have all the answers to, name and document it with a red sticky note.** This might happen if there are still some unsolved problems or the right people aren't present. Marking this as a trouble spot reminds you to go back and research it later.

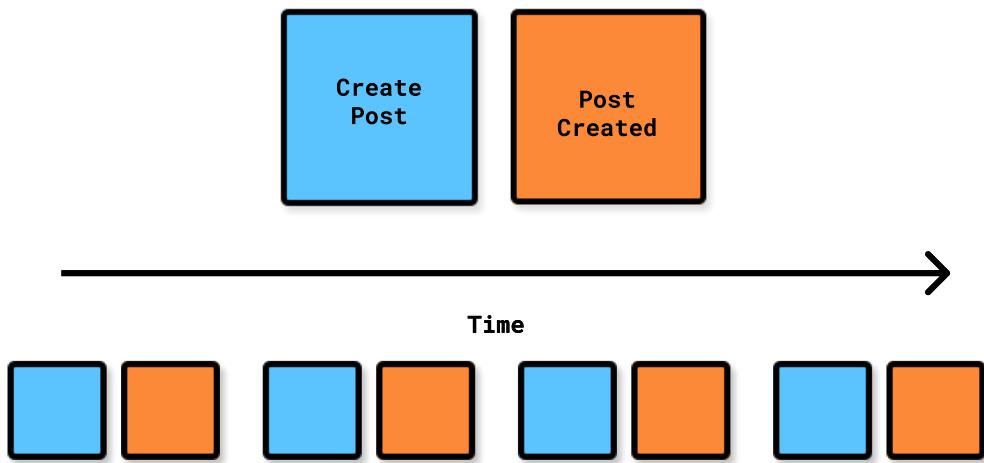


An example of encountering a meaningful alternative domain event that we would like to have a way to handle, but we don't have all the answers for at the moment.

If we can't think of any more domain events, it's time to move onto the next step: *commands*.

Step 2 — Create the Commands that cause Domain Events

For each domain event, write the command that causes it.



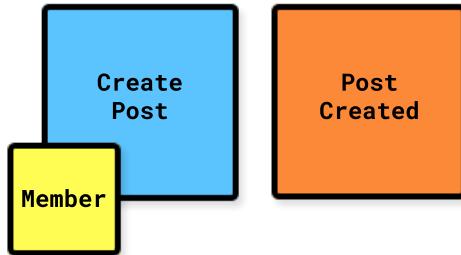
The *command* is exactly the same thing as commands from our UML use case diagrams and the CQS principle. You don't need to have done a UML use case diagram or anything in order to do this step. You can go through each domain event, and place the name of the command that creates the domain event, *before* each one.

For example, the *PostCreated* domain event was the result of the *CreatePost* command.

By the end of this step, each domain event should be accompanied by a command in *Command/Domain Event* format.

Other relevant things about this step:

- **(Optional) If you know the Actor/Role that issues the command, place a small yellow sticky on the bottom left of the command to document it.** For example, I know that a Member is responsible for the *CreatePost* command, so I place the small yellow sticky on the command.



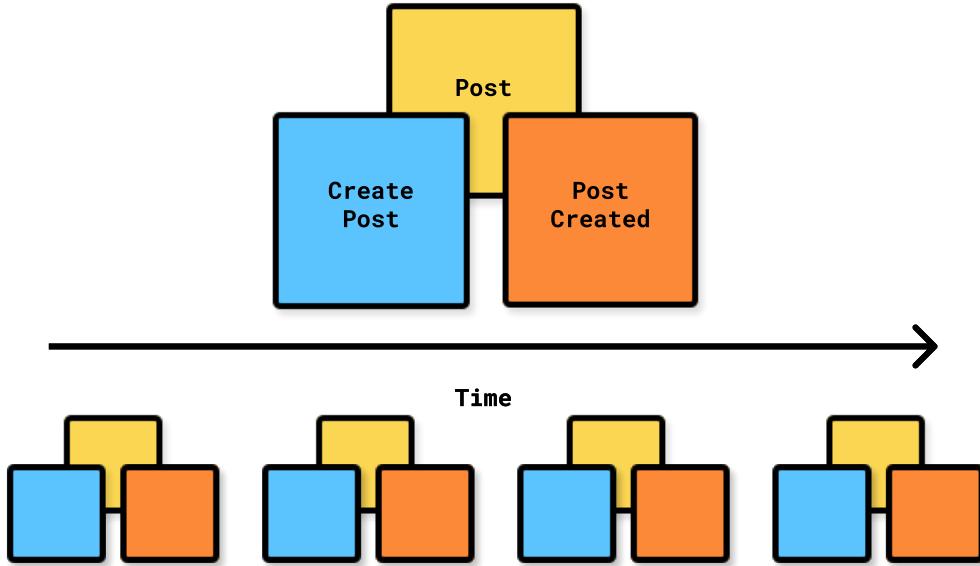
Example of documenting the *Actor/Role* for a command.

- You might find more domain events in the scenario that one command creates multiple domain events. In that case, just do the same thing: put the blue command sticky on the left, and put the domain events on the right of the command.

If we've found all our commands, we're ready for the next part.

Step 3 — Identify the Aggregate that the Command is executed against and the resulting Domain Event

For each *Command/Domain Event* pair, put a pale yellow sticky in slightly above and between them to represent the Aggregate.



Recall that in DDD, an *Aggregate* is a special type of *Entity*. *Aggregates* domain objects that protect **model invariants**. They are what we perform **commands** against.

Finally, we've gotten to the part where we identify the models that appear in our code. Notice that this is the *third step* of our analysis- where some approaches put this *first*.

In the database-first approach (or even *UML class diagrams*), we would have aimed to attempt to discover entities upfront.

The unique value of event-based design approaches like event storming is that we change the order of discovery to prioritize understanding the business. **We put the behavior of the business before the structure of the data in the business.**

Reasons why this approach is better:

- It's much easier to describe the data required after having discovered the behavior of the business, while the opposite is significantly more challenging.
- Business-folk don't understand UML and entity-relationship diagrams (it makes for poor discussion).

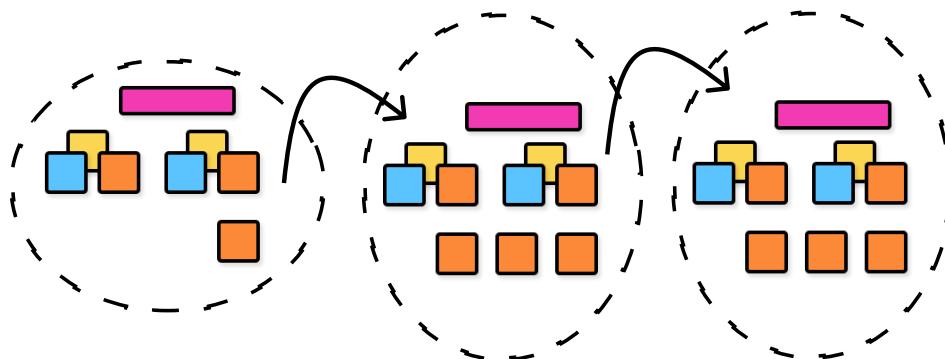
For this step, here are a few notes:

- **Use the word Entity or Data if Aggregate is confusing to others.** DDD is well known for the challenging names for all of its constructs. I wouldn't expect anyone to know what an *Aggregate* is, so it's recommended to use the word *Entity*.
- **If Aggregates are used multiple times, create copies and place them repeatedly on the timeline.**
- **If at any point you realize any more Domain Events or Commands, feel free to also document those.**

Next, we're onto boundaries.

Step 4 — Create Subdomain and Bounded Context boundaries

With all the current stickies on the board, it's time to establish the subdomain and bounded context boundaries.



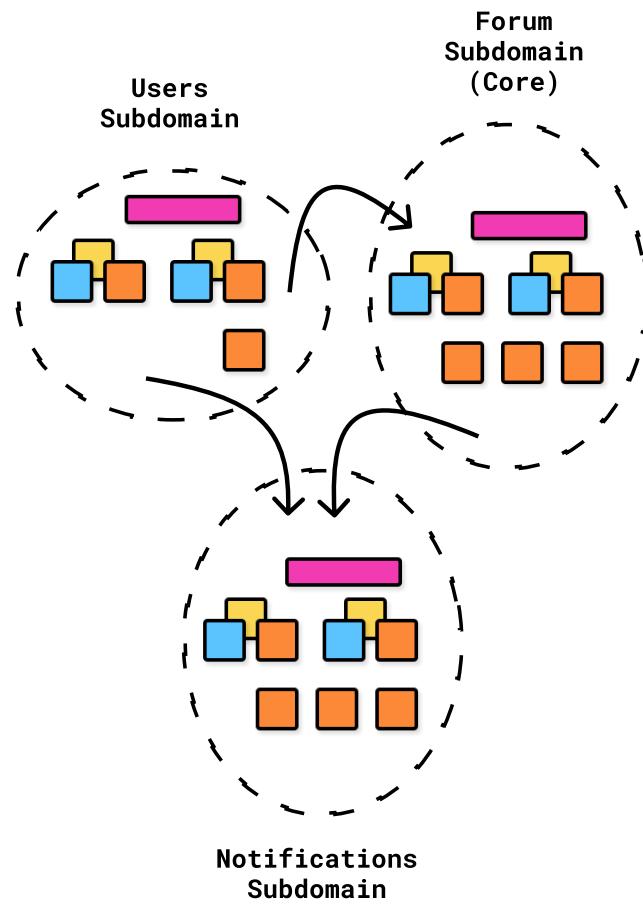
This step is asking you to apply **Conway's Law**.

This is likely the most challenging step of event storming because it requires having a good understanding of subdomains and bounded contexts.

Good things about being able to apply this are:

- **It becomes easy to see how some Domain Events end up within our core domain without needing a command to be invoked first; this is because we subscribe to them from another domain.**

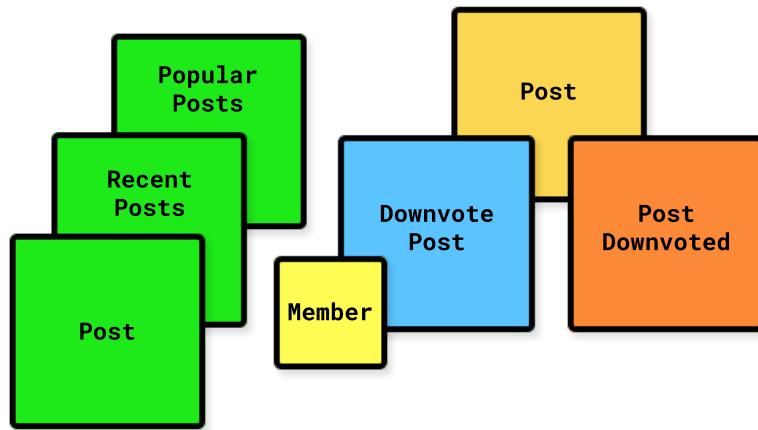
When we apply this step to DDDForum.com, we should end up with a similar diagram to the one we created when applying Conway's Law to our Use Case diagrams.



Subdomains within DDDForum.com (a single bounded context if built as a modular monolith) illustrating the direction of domain events that flow between subdomain boundaries.

Step 5 — Identify Views & Roles

For each *Command/Domain Event* pair, identify the view(s) needed to provide information before the *Role(s)* invoke the *Command*.



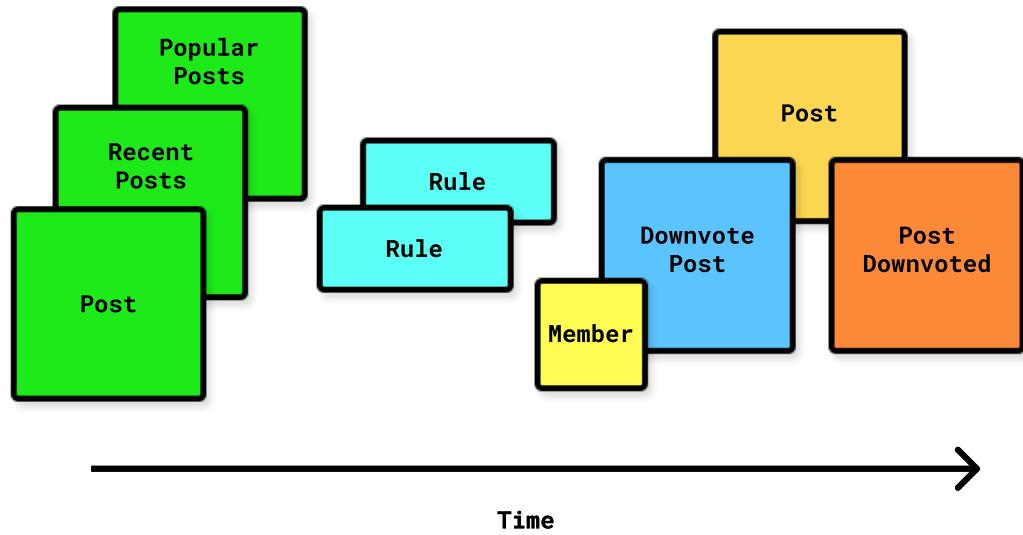
- If there are important roles, make sure to list them here.
- (Optional) Create wireframes or mockups to illustrate what the views might look like. You want to do whatever is going to be most useful for you and everyone else to understand what's necessary for the model.

Steps 0 to 5 are all of the absolutely necessary steps, but feel free to **invent constructs** to use. Everything is fair game here if it helps us improve the model.

For example, I like to document the preconditions that specify when and how a *Command* can be invoked.

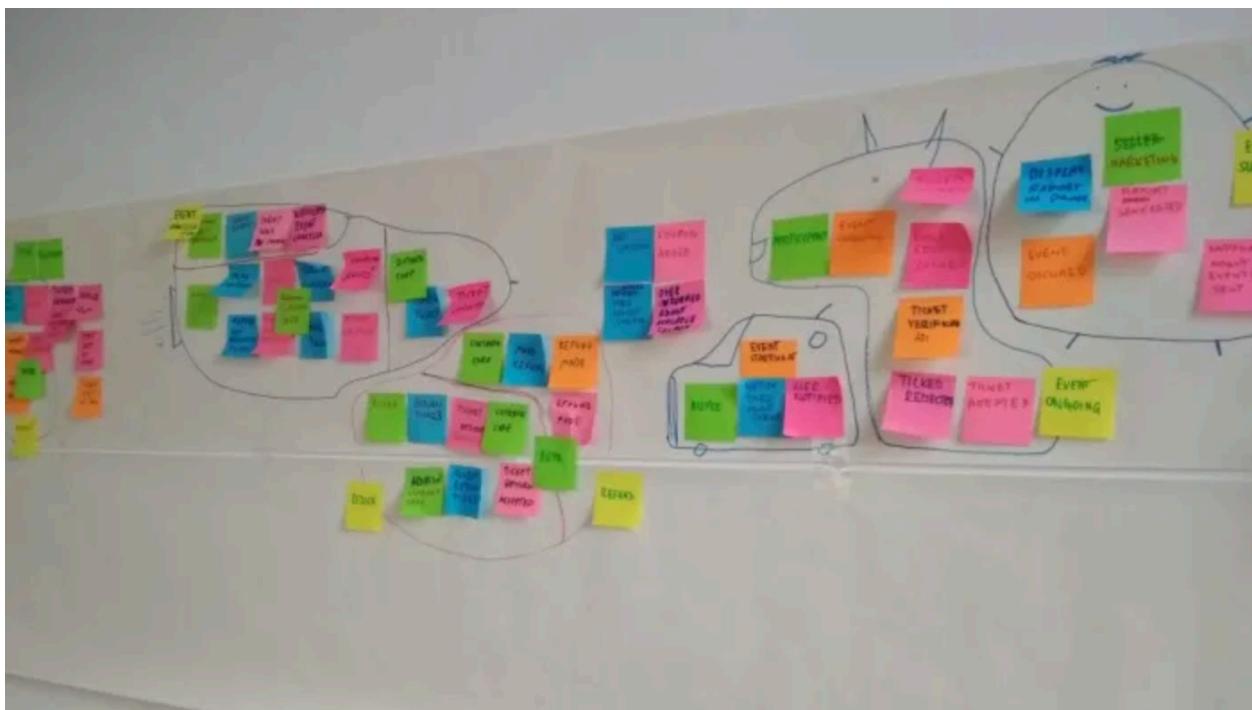
Step 6 (Optional) — Identify rules/policies

Place neon blue stickies before the *Command* and document the preconditions that allow or disallow the *Command* to be invoked.



Depending on how detailed you want to get and if you're doing big-picture event storming or design-level storming, documenting the rules within the event stormed model is an option.

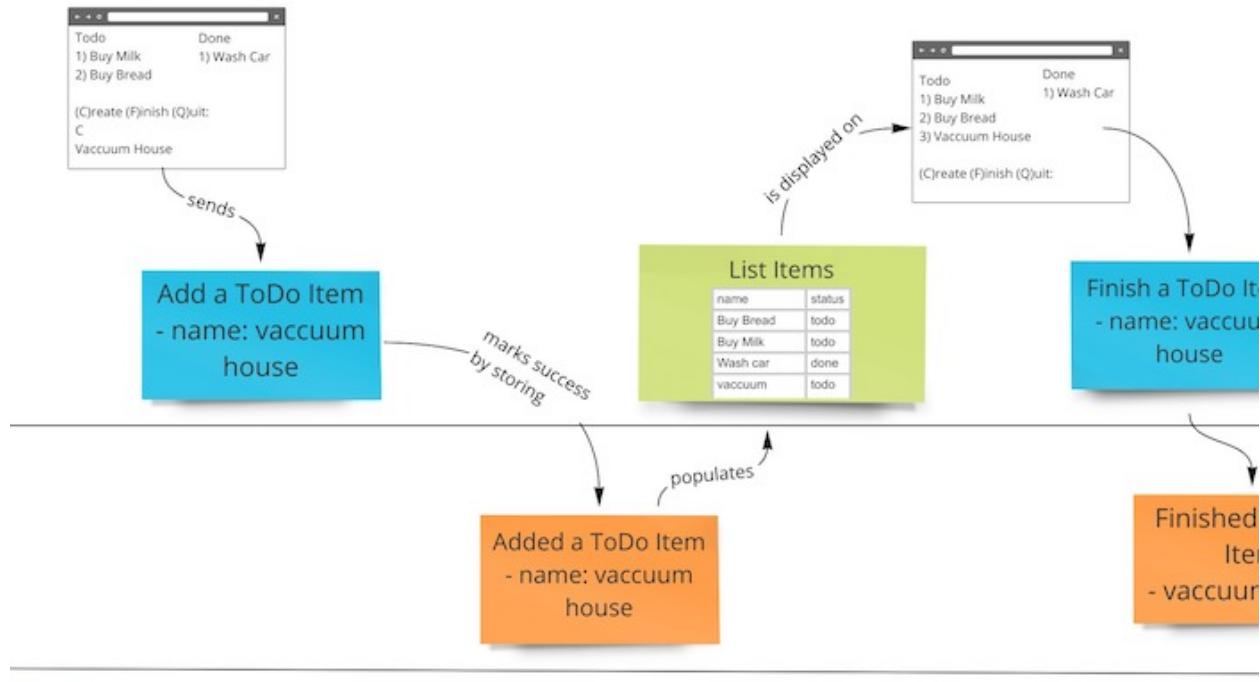
These rules are BDD-style **given X, then Y** policies.



Someone's productive Event Storming session.

Event Modeling

The final approach to planning a project that I want to mention briefly is actually the newest one: it's called *Event Modeling*.



Event modeling brings together all of the discoveries of Event Sourcing, Event Storming, DDD, Conway's Law, and Use Case design. Image courtesy of eventmodeling.org.

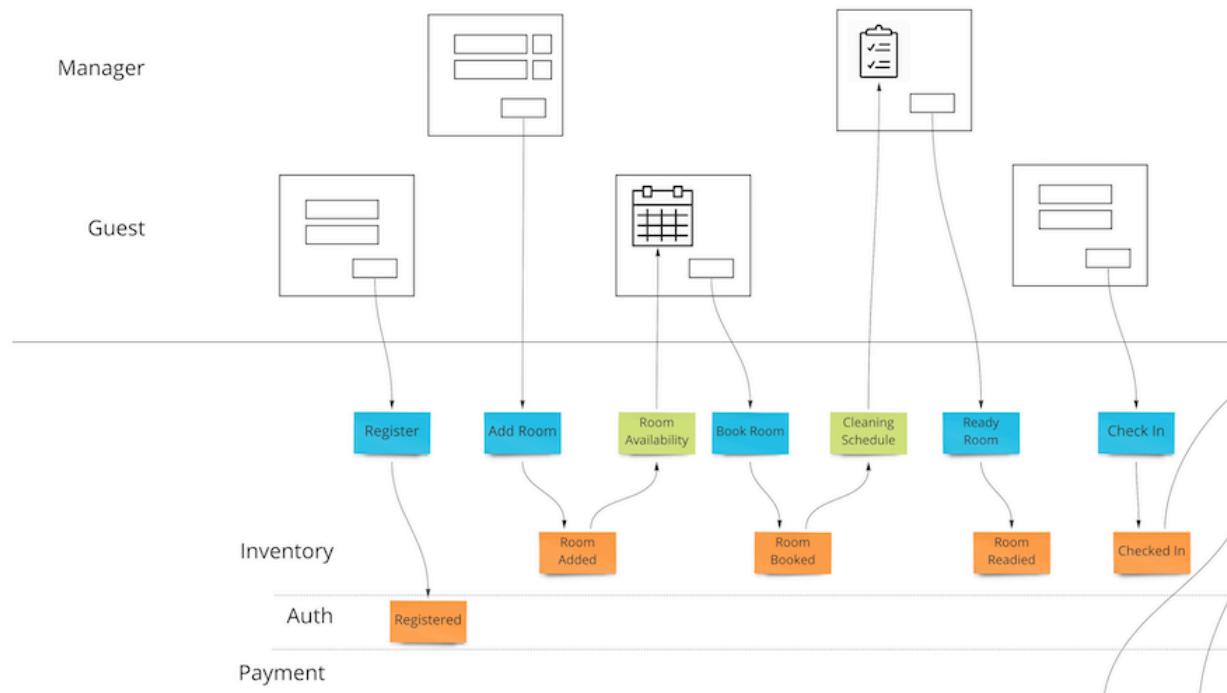
There's not much *new* about *Event Modeling*. It's more of a **formalization** of all the knowledge we've acquired about building event-driven systems.

This traces all the way back to around 2003 when Eric Evans released the original *Domain Driven Design* book, Fowler wrote about Event-Sourcing in 2005, Greg Young popularized CQRS and Event Sourcing from 2007 to 2012, and 2013 when **Brandolini** used Event Storming as a way to understand and plan a project around a problem domain.

In 2008, Canadian consultant, Adam Dymitruk formalized the work from 2003-2018 revolving around DDD and event-based systems, coining the term *Event Modeling*.

Event Modeling is another approach, very similar to *Event Storming*, that puts all the pieces of the puzzle together in order to plan a system before coding it.

While it shares its similarities with *Event Storming*, the main addition to *Event Modeling* that wasn't formally defined in *Event Storming* was the use of UI drawings in order to present the views more effectively.



Snippet of an Event Modeling session done on a Hotel Reservation domain. Image courtesy of eventmodeling.org.

Sometimes people in *Event Storming* sessions would do this anyways, but *Event Storming* formalizes the approach to create rough wireframes as part of the design process.

Here is an outline of the steps involved. Notice the similarity with *Event Storming*:

1. Identify events
2. Plot the events on a timeline
3. Create wireframes / mockups of the story
4. Identify inputs
5. Identify outputs
6. Apply Conway's Law
7. Elaborate on each scenario (BDD-style test cases)

Adam argues that the resulting approach is a more reasonable way to design an event-based system in order to:

- design for scalability
- achieve zero data loss
- achieve faster transactional performance
- keep the system model simple
- reduce development timelines

For more information, I recommend you check out “Event Modeling: What is it?” on EventModeling.org and get a feel for it yourself.

Building DDDForum

This is the part of the book where we apply everything we've learned, and you get to see if I practice what I preach.

After having come up with a design, whether it be using *Event Storming*, *Event Modeling*, or Event just boxes, shapes, and arrows, we're in a much better position to start coding up our project.

For DDDForum.com, before I started coding, I had:

- an *Event Stormed* model created that identified all of the events, commands, queries, aggregates, and views
- Several of the *policies/business rules* thought out and identified that would affect how and when specific commands and events occur.
- Wireframes created with Figma to verify the exact attributes that I'd need on each model

With that in place, we're informed and ready to build DDDForum.com.

Let me walk you through it.

View the code: You can find the code for DDDForum.com in its completion here on GitHub.

Project architecture

Some of the significant upfront decisions that we're making about this project are the following:

Decision 1: We're going to use Domain-Driven Design

That shouldn't be a surprise to you at this point in the chapter. Using Domain-Driven Design means that we're going to start our development journey by encoding the business rules within our domain models. The task is to define all the models, the relationships between them, the policies that govern when and how they can change, and **make it virtually impossible to represent any illegal state**.

For example, if we had a User entity (which we do in the users subdomain), consider the implications of having a getter and a setter for the userId property.

```
// users/domain/user.ts
export class User extends Entity<UserProps> {
    get userId () : UserId {
        return this.props.userId;
    }

    // set userId (userId: UserId) {
    //     this.props.userId = userId;
    // }
```

```
...  
}
```

Concerning the users subdomain, there's *no reason* why the `userId` should ever change to a new value. Doing so would break the relationships between `User` and any other subdomains that have a 1-to-1 relationship with `User`, like `Member` from the `Forum` subdomain.

So we remove the setter. We make it impossible to mutate `User` in a way that puts it in an invalid state.

```
// users/domain/user.ts  
export class User extends Entity<UserProps> {  
    get userId () : UserId {  
        return this.props.userId;  
    }  
  
    ...  
}
```

Therefore, our task is to create plain ol' TypeScript objects and ensure that they can *only perform valid operations*.

It's kind of like building your very own DSL (domain-specific language).

Domain objects have zero dependencies and only create source code dependencies to other domain objects. Because of this, we can write tests to ensure that the business logic contained in *entities*, *value objects*, and *domain services* are correct, and we can expect these tests to run very fast.

Decision 2: We're going to use a Layered Architecture

We've mentioned it before, but you'll find it challenging to implement Domain-Driven Design without some sort of *Layered Architecture*.

That's especially true because we need a way to isolate our domain layer from outer layer concerns like databases, controllers, web servers, and other things that might slow down our ability to run tests and clash with the *Ubiquitous Language*.

Because software doesn't do a whole lot unless we can connect the pieces, we can implement *Dependency Inversion* to bridge the gap between layers.

As a rule of thumb, the direction of source code dependencies must always point inwards, towards the domain-layer code. This rule of thumb is called *The Dependency Rule*.

Additional reading: If you're interested, you can read more about The Dependency Rule here.

Decision 3: We're going to deploy a Modular Monolith

Like we talked about in Deployments as a Modular Monolith, on new projects with a smaller team, it could be a good idea to start with a *Modular Monolith* instead of jumping to implementing *Micro-services* right away.

A monolithic application enables both the `Users` and `Forum` subdomains to live within the same codebase but from within separate modules.

In DDD, the way that subdomains or bounded contexts communicate with each other is through the publishing and subscribing of *Domain Events*.

Using Domain Events as the primary mechanism for messaging is an excellent way to foster *loose coupling* between modules.

In a real-world micro-service deployment, *Domain Events* get published to a queue and sent out across the network to subscribers.

In our project, we'll implement an **In-Memory Domain Events Queue** so that we can exchange messages between the subdomains in our modular monolith and maintain loose coupling.

Decision 4: We're going to use CQRS (Command Query Response Segregation)

When we first learn about DDD, it's common to also hear about concepts like CQRS (Command Query Response Segregation) as well.

Based on my experience, CQRS solves a lot of design issues for us. The most apparent design issue that it addresses is related to *Aggregate* design.

When we're building *Aggregates*, our goal is to design an object that enforces *model invariants* against operations that change the state of the system. Namely, writes. Write commands.

Our task becomes even more challenging when we also have to design aggregates to return enough information to build view models (or DTOs) from as well.

As a result of these two responsibilities living on the same object (writes & reads), we end up with an *Aggregate* model that becomes messy, unreasonable, volatile, and unclear of which properties are necessary for the sake of protecting invariants, and which are necessary for merely creating read models.

To address these challenges, we can adopt the CQRS pattern. Taking it one step further than the CQS (Command-Query Separation) pattern, CQRS implies that we have **separate models for reading and writing**. That is, for a `Post` aggregate, we have one *write model* and at least one *read model* opposed to having only one model responsible for both operations.

Decision 5: We're not going to use Event Sourcing

Event Sourcing is another approach to implementing DDD that comes up often in discussion.

In my opinion, Event Sourcing is *hard*. I wouldn't recommend using it on your first Domain-Driven Design project.

That said, there are incredibly valid reasons to use Event Sourcing.

Additional reading: Communication, auditing & reasoning, estimates, scalability, and taming complexity are good arguments for event-based systems. You can read "Why Event-Based Systems?" for more information on this.

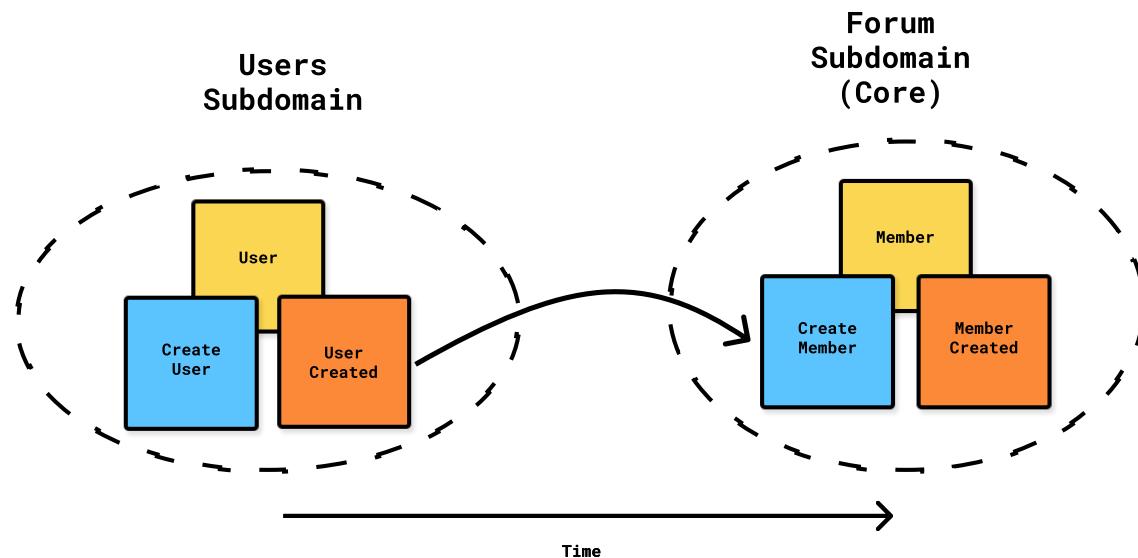
Because DDDForum.com is relatively simple, and because I don't want to expand too far past the scope of this section, we won't be implementing Event Sourcing today. All of what we'll learn how to do in DDD will be applicable when we finally get to Event Sourcing. And these are the basics. Let's start small.

Starting with the domain models

In the coming section **5. Object-Oriented Programming & Domain Modeling**, we will have thoroughly learned how to use the best of Object-Oriented Programming to create rich domain models. In this section, our focus is to *understand* at the class level how all of the concepts from Domain-Driven Design work together in conjunction to power a flexible, testable, and maintainable web application.

Let's start with the domain models in the Users subdomain because for us to even begin to have the equivalent of users in Forum, a UserCreated *Domain Event* needs to be emitted from the Users subdomain. The Users forum is where the first *Domain Events* occur in our timeline.

Here's the section of our Event Stormed model that depicts how Members from the Forum subdomain are created as a result of the UserCreated *Domain Event*.



The UserCreated domain event from the Users subdomain crossing the boundary and resulting in the dispatch of a CreateMember command.

So then it only makes sense to begin our journey from the User model.

Let's look inside of the modules/users/domain folder in DDDForum, where all of the domain layer concepts for the users subdomain live.

Modeling a User Aggregate

If I were to start from scratch on creating this User model, I'd start with trying to identify all of the different *things* or properties a User has.

I might start by creating an interface that holds all of the UserProps like so:

```
// users/domain/user.ts

interface UserProps {
  email: string
  username: string
  password: string
  isEmailVerified?: boolean
  isAdminUser?: boolean
  accessToken?: string
  refreshToken?: string
  isDeleted?: boolean
  lastLogin?: Date
}
```

This is great. We've identified pretty much all of the properties that we need in order to create a User.

Is there any way that we can improve this interface?

Why, yes, there is.

Part of the job in Domain-Driven Design is protecting against illegal states. When we use primitive types like string or number for properties that have **fundamental business rules encapsulated with them**, we're opening ourselves up to the possibility of having an object impossible to the domain.

I'm talking about using **Value Objects**.

The properties email, username, password, accessToken, and refreshToken all either:

- Have validation rules that dictate what makes it valid
- Or are important to nominally type so that it cannot be substituted for a type that is *similar*

For example, we want to prevent being able to pass in an emails that look like this:

```
test.com
imnotanemailaddresss@hello
khalilstemmler
```

And if we simply used a string primitive for email, it would be entirely possible to create an invalid User and have that floating around and persisted to the database.

Domain-Driven Design is not a String-ly typed affair :)

We can improve the design by promoting the primitives with business rules or with illegal substitutability to *Value Objects* like so.

```
// users/domain/user.ts

interface UserProps {
  email: UserEmail // value object
  username: UserName // value object
  password: UserPassword // value object
  isEmailVerified?: boolean
  isAdminUser?: boolean
  accessToken?: JWTToken // value object
  refreshToken?: RefreshToken // value object
  isDeleted?: boolean
  lastLogin?: Date
}
```

It's a wise idea to have unit tests against our *Value Objects* and ensure that they behave correctly.

```
// users/domain/userEmail.spec.ts

import { UserEmail } from "./userEmail"
import { Result } from "../../../../shared/core/Result"

let email: UserEmail
let emailOrError: Result<UserEmail>

test("Should be able to create a valid email", () => {
  emailOrError = UserEmail.create("khalil@apollographql.com")
  expect(emailOrError.isSuccess).toBe(true)
  email = emailOrError.getValue()
  expect(email.value).toBe("khalil@apollographql.com")
})

test("Should fail to create an invalid email", () => {
  emailOrError = UserEmail.create("notvalid")
  expect(emailOrError.isSuccess).toBe(false)
})
```

To restrict object creation and make sure that it's only possible in the case that we have valid User props, we can implement the Factory Pattern by placing the `private` keyword on our constructor. This forces everyone to use the static `create` method if we want to create a User. It also makes it impossible for you to create an invalid User.

```
// users/domain/user.ts

interface UserProps {
  email: UserEmail;
  username: UserName;
  password: UserPassword;
```

```

isEmailVerified?: boolean;
isAdminUser?: boolean;
accessToken?: JWTToken;
refreshToken?: RefreshToken;
isDeleted?: boolean;
lastLogin?: Date;
}

/**
 * User is an Aggregate Root since it's the
 * object that we perform commands against.
 */

export class User extends AggregateRoot<UserProps> {

    ...

    /**
     * Private constructor that disables us from
     * circumventing the creation rules by using
     * the `new` keyword.
     */
    private constructor (props: UserProps, id?: UniqueEntityID) {
        super(props, id)
    }

    /**
     * Static factory method that forces the creation of a
     * user by using User.create(props, id?)
     */
    public static create (props: UserProps, id?: UniqueEntityID): Result<User> {
        // Guard clause that fails if the required properties aren't
        // provided.

        const guardResult = Guard.againstNullOrUndefinedBulk([
            { argument: props.username, argumentName: 'username' },
            { argument: props.email, argumentName: 'email' }
        ]);

        if (!guardResult.succeeded) {
            return Result.fail<User>(guardResult.message)
        }
    }
}

```

```

const isNewUser = !!id === false;
const user = new User({
  ...props,
  // Assemble default props
  isDeleted: props.isDeleted ? props.isDeleted : false,
  isEmailVerified: props.isEmailVerified ? props.isEmailVerified : false,
  isAdminUser: props.isAdminUser ? props.isAdminUser : false
}, id);

if (isNewUser) {
  user.addDomainEvent(new UserCreated(user));
}

return Result.ok<User>(user);
}
}

```

We continue by writing getters and any appropriate setters on User (note: there are no setters for the User model).

```

// users/domain/user.ts

export class User extends AggregateRoot<UserProps> {

  ...

  get userId (): UserId {
    return UserId.create(this._id)
      .getValue();
  }

  get email (): UserEmail {
    return this.props.email;
  }

  get username (): UserName {
    return this.props.username;
  }

  get password (): UserPassword {
    return this.props.password;
  }

  ...
}

}

```

That's it for the User model for now. A few things to note:

- The User model has zero references to anything other than other plain ol' TypeScript objects. That makes the tests really fast.
- The User model extends an AggregateRoot class, which has some additional functionality that we are going to discuss immediately in the following section.
- We nest the props *within* the User model instead of declaring them directly on the class so that we can have control over other developers' ability to get and set properties on instances of User.
- Domain objects (*Aggregates*, *Entities*, and *Value Objects*) hold the highest level of policy in the entirety of our application. **Upper layer classes rely on it.** It's on one of these three objects that you want to aim to encapsulate business rules within first.
- The Stable Dependency Principle (SDP) says that all components should be in the direction of stability. Since these classes are depended on by upper layers, it needs to be the most stable. This is likely to happen naturally since domain layer classes mimic the business rules of the domain, and needing drastic changes to the domain code would be *unlikely* since it would mean a drastic change to the way the business fundamentally works.
- This isn't the only way to model a User model. This is what works for me, and it's ideal for teams to create their own core *Entity*, *Value Object*, and other important conceptual DDD classes that work for them and their understanding of how they work.

View the code: You can read `users/domain/user.ts` in its entirety here on GitHub.

Emitting Domain Events from a User Aggregate

You might have noticed that inside of the `create(props: UserProps, id?: UniqueEntityID)` method, there's a condition that determines when we should emit the `UserCreated` domain event.

```
// users/domain/user.ts

const isNewUser = !!id === false
const user = new User(
  {
    ...props,
    isDeleted: props.isDeleted ? props.isDeleted : false,
    isEmailVerified: props.isEmailVerified ? props.isEmailVerified : false,
    isAdminUser: props.isAdminUser ? props.isAdminUser : false,
  },
  id
)

if (isNewUser) {
  user.addDomainEvent(new UserCreated(user))
}
```

If you recall from Entities, we discuss the lifecycle of an *Entity*. An *Entity* doesn't have

an identifier until *after it's created*, that is, until after we invoke `User.create(props: UserProps, id?: UniqueEntityID)` and get a User back.

If we're *creating a User for the first time*, we won't pass in an `id: UniqueEntityID` because we don't have one yet.

In that scenario, we want to make sure we fire off the Domain Event to a *Subject* (*see Observer pattern*) so that when a transaction (or a Unit of Work) completes, we can propagate that *Domain Event* cross our enterprise and allow any subdomains or bounded contexts interested in that Event, to do something after having received it.

In a **monolithic application**, we pass messages between subdomains using a *class-level implementation* of the Observer Pattern. In a **micro-service application**, we pass messages between Bounded Contexts by using an *architecture-level implementation* of the Observer Pattern with *Message Queues*.

This is how a Member in the Forum subdomain gets created: in response to the `UserCreated domain event` from the `Users` subdomain.

Writing Domain Events

Most DDD developers will use a base domain events interface. The one shown below describes the contract for a domain event. It says that a domain event needs a `dateTimeOccurred` and it must define a function that knows how to get the aggregate id for the *Domain Event* in question.

```
// IDomainEvent.ts

import { UniqueEntityID } from "../UniqueEntityID";

export interface IDomainEvent {
  dateTimeOccurred: Date;
  getAggregateId (): UniqueEntityID;
}
```

A `MemberCreated` event taking shape could look like the following.

```
// forum/domain/events/memberCreated.ts

import {
  IDomainEvent
} from "../../../../../shared/domain/events/IDomainEvent";
import {
  UniqueEntityID
} from "../../../../../shared/domain/UniqueEntityID";
import { Member } from "../member";

export class MemberCreated implements IDomainEvent {
  public dateTimeOccurred: Date;
  public member: Member;
```

```

constructor (member: Member) {
    this.dateTimeOccurred = new Date();
    this.member = member;
}

getAggregateId (): UniqueEntityID {
    return this.member.id;
}

```

Building a Domain Events Subject

What happens when we say, `aggregate.addDomainEvent(event: IDomainEvent)` from an *Aggregate*?

Here's the base *AggregateRoot* class that we're using in DDDForum.

```

// shared/domain/AggregateRoot.ts

import { Entity } from "./Entity"
import { IDomainEvent } from "./events/IDomainEvent"
import { DomainEvents } from "./events/DomainEvents"
import { UniqueEntityID } from "./UniqueEntityID"

export abstract class AggregateRoot<T> extends Entity<T> {

    /**
     * All of the domain events for a subclass of AggregateRoot<T>
     * get added to this private array.
     */

    private _domainEvents: IDomainEvent[] = []

    get id(): UniqueEntityID {
        return this._id
    }

    get domainEvents(): IDomainEvent[] {
        return this._domainEvents
    }

    /**
     * @method addDomainEvent
     * @protected
     * @desc Called by a subclass in order to add a Domain Event
     * to the list of Domain Events currently on this aggregate
     * within a transactional boundary. Also notifies the DomainEvents
     * subject that the current aggregate has at least one Domain Event
     * that we will need to publish if the transaction completes.
    */
}
```

```

/*
protected addDomainEvent(domainEvent: IDomainEvent): void {
    // Add the domain event to this aggregate's list of domain events
    this._domainEvents.push(domainEvent)
    // Add this aggregate instance to the domain event's list of aggregates who's
    // events it eventually needs to dispatch.
    DomainEvents.markAggregateForDispatch(this)
    // Log the domain event
    this.logDomainEventAdded(domainEvent)
}

public clearEvents(): void {
    this._domainEvents.splice(0, this._domainEvents.length)
}

private logDomainEventAdded(domainEvent: IDomainEvent): void {
    const thisClass = Reflect.getPrototypeOf(this)
    const domainEventClass = Reflect.getPrototypeOf(domainEvent)
    console.info(
        ` [Domain Event Created]: `,
        thisClass.constructor.name,
        "=>",
        domainEventClass.constructor.name
    )
}
}
}

```

So the real magic that happens with this subclass is within the `addDomainEvent()` method. Not only do we add the `IDomainEvent` to a list of *Domain Events* currently on the *Aggregate* for the transaction, but we notify the `DomainEvents` subject that the current *Aggregate* should be marked for dispatch. This means that when the transaction for this *Aggregate* completes, we should publish the *Domain Events* attached to the *Aggregate*.

My implementation of the `DomainEvents` subject is something I ported to TypeScript from Udi Dahan's 2009 blog post about Domain Events in C#.

Here it is in its entirety.

```

// shared/domain/events/DomainEvents.ts

import { IDomainEvent } from "./IDomainEvent"
import { AggregateRoot } from "../AggregateRoot"
import { UniqueEntityID } from "../UniqueEntityID"

export class DomainEvents {
    private static handlersMap = {}
    private static markedAggregates: AggregateRoot<any>[] = []
}
```

```

/**
 * @method markAggregateForDispatch
 * @static
 * @desc Called by aggregate root objects that have created domain
 * events to eventually be dispatched when the infrastructure commits
 * the unit of work.
 */

public static markAggregateForDispatch(aggregate: AggregateRoot<any>): void {
    const aggregateFound = !!this.findMarkedAggregateByID(aggregate.id)

    if (!aggregateFound) {
        this.markedAggregates.push(aggregate)
    }
}

/**
 * @method dispatchAggregateEvents
 * @static
 * @private
 * @desc Call all of the handlers for any domain events on this aggregate.
 */

private static dispatchAggregateEvents(aggregate: AggregateRoot<any>): void {
    aggregate.domainEvents.forEach((event: IDomainEvent) =>
        this.dispatch(event)
    )
}

/**
 * @method removeAggregateFromMarkedDispatchList
 * @static
 * @desc Removes an aggregate from the marked list.
 */

private static removeAggregateFromMarkedDispatchList(
    aggregate: AggregateRoot<any>
): void {
    const index = this.markedAggregates.findIndex(a => a.equals(aggregate))

    this.markedAggregates.splice(index, 1)
}

/**
 * @method findMarkedAggregateByID
 * @static

```

```

 * @desc Finds an aggregate within the list of marked aggregates.
 */
private static findMarkedAggregateByID(
    id: UniqueEntityID
): AggregateRoot<any> {
    let found: AggregateRoot<any> = null
    for (let aggregate of this.markedAggregates) {
        if (aggregate.id.equals(id)) {
            found = aggregate
        }
    }

    return found
}

/**
 * @method dispatchEventsForAggregate
 * @static
 * @desc When all we know is the ID of the aggregate, call this
 * in order to dispatch any handlers subscribed to events on the
 * aggregate.
*/
public static dispatchEventsForAggregate(id: UniqueEntityID): void {
    const aggregate = this.findMarkedAggregateByID(id)

    if (aggregate) {
        this.dispatchAggregateEvents(aggregate)
        aggregate.clearEvents()
        this.removeAggregateFromMarkedDispatchList(aggregate)
    }
}

/**
 * @method register
 * @static
 * @desc Register a handler to a domain event.
*/
public static register(
    callback: (event: IDomainEvent) => void,
    eventClassName: string
): void {
    if (!this.handlersMap.hasOwnProperty(eventClassName)) {
        this.handlersMap[eventClassName] = []
    }
}

```

```

    }

    this.handlersMap[eventClassName].push(callback)
}

/** 
 * @method clearHandlers
 * @static
 * @desc Useful for testing.
 */

public static clearHandlers(): void {
    this.handlersMap = {}
}

/** 
 * @method clearMarkedAggregates
 * @static
 * @desc Useful for testing.
 */

public static clearMarkedAggregates(): void {
    this.markedAggregates = []
}

/** 
 * @method dispatch
 * @static
 * @desc Invokes all of the subscribers to a particular domain event.
 */

private static dispatch(event: IDomainEvent): void {
    const eventClassName: string = event.constructor.name

    if (this.handlersMap.hasOwnProperty(eventClassName)) {
        const handlers: any[] = this.handlersMap[eventClassName]
        for (let handler of handlers) {
            handler(event)
        }
    }
}
}

```

Marking an Aggregate that just created Domain Events

The DomainEvents subject needs a clean and clear way to hold onto the *Aggregates* that just created *Domain Events*.

The `markAggregateForDispatch()` method takes in the `AggregateRoot` that just created an event, and places it into an array of `markedAggregates`.

```
// shared/domain/events/DomainEvents.ts

export class DomainEvents {
    private static handlersMap = {};
    private static markedAggregates: AggregateRoot<any>[] = [];

    /**
     * @method markAggregateForDispatch
     * @static
     * @desc Called by aggregate root objects that have created domain
     * events to eventually be dispatched when the infrastructure commits
     * the unit of work.
    */

    public static markAggregateForDispatch (aggregate: AggregateRoot<any>): void {
        const aggregateFound = !!this.findMarkedAggregateByID(aggregate.id);

        if (!aggregateFound) {
            this.markedAggregates.push(aggregate);
        }
    }

    ...
}
```

How to signal that the transaction completed

When we're sure that the transaction has completed, the `DomainEvents` subject provides a method to notify all *Observers* of each *Domain Event* of its occurrence.

```
// shared/domain/events/DomainEvents.ts

export class DomainEvents {
    private static handlersMap = {};
    private static markedAggregates: AggregateRoot<any>[] = [];

    ...

    /**
     * @method dispatchEventsForAggregate
     * @static
     * @desc When all we know is the ID of the aggregate, call this
     * in order to dispatch any handlers subscribed to events on the
     * aggregate.
    */
```

```

public static dispatchEventsForAggregate(id: UniqueEntityID): void {
    const aggregate = this.findMarkedAggregateByID(id);

    if (aggregate) {
        this.dispatchAggregateEvents(aggregate);
        aggregate.clearEvents();
        this.removeAggregateFromMarkedDispatchList(aggregate);
    }
}

private static removeAggregateFromMarkedDispatchList(
    aggregate: AggregateRoot<any>
): void {
    const index = this.markedAggregates.findIndex(a => a.equals(aggregate));
    this.markedAggregates.splice(index, 1);
}

/**
 * @method dispatchAggregateEvents
 * @static
 * @private
 * @desc Call all of the handlers for any domain events on this aggregate.
 */
private static dispatchAggregateEvents(aggregate: AggregateRoot<any>): void {
    aggregate.domainEvents.forEach((event: IDomainEvent) =>
        this.dispatch(event)
    );
}

/**
 * @method dispatch
 * @static
 * @desc Invokes all of the subscribers to a particular domain event.
 */
private static dispatch(event: IDomainEvent): void {
    const eventClassName: string = event.constructor.name;

    if (this.handlersMap.hasOwnProperty(eventClassName)) {
        const handlers: any[] = this.handlersMap[eventClassName];
        for (let handler of handlers) {
            handler(event);
        }
    }
}

```

```
}
```

How to register a handler to a Domain Event?

To register a handler to Domain Event, from another class, we can pass a callback to the register method along with the *Domain Event* we're interested in being notified about.

```
// shared/domain/events/DomainEvents.ts

export class DomainEvents {
    private static handlersMap = {};
    private static markedAggregates: AggregateRoot<any>[] = [];

    ...

    public static register(
        callback: (event: IDomainEvent) => void,
        eventClassName: string
    ): void {
        if (!this.handlersMap.hasOwnProperty(eventClassName)) {
            this.handlersMap[eventClassName] = [];
        }
        this.handlersMap[eventClassName].push(callback);
    }

    ...
}
```

It accepts both a callback function and the eventClassName, which is the name of the class (we can get that using `Class.name`).

When we register a handler for a domain event, it gets added to the `handlersMap`.

For 3 different domain events and 7 different handlers, the data structure for the handler's map can end up looking like this:

```
// The handlersMap is an Identity map of Domain Event names
// to callback functions.

{
    "UserCreated": [Function, Function, Function],
    "UserEdited": [Function, Function],
    "PostCreated": [Function, Function]
}
```

Here's an example of a handler that subscribes to a domain event.

```
// modules/users/subscriptions/afterUserCreated.ts

import { IHandle } from "../../core/domain/events/IHandle"
```

```

import { DomainEvents } from "../../core/domain/events/DomainEvents"
import { UserCreated } from "../../users/domain/events/userCreated"
import { User } from "../../users/domain/user"

export class AfterUserCreated implements IHandle<UserCreated> {
    constructor() {
        this.setupSubscriptions()
    }

    setupSubscriptions(): void {
        // Register to the domain event
        DomainEvents.register(this.onUserCreated.bind(this), UserCreated.name)
    }

    private async onUserCreatedEvent(event: UserCreated): Promise<void> {
        const { user } = event

        /**
         * Do something with the domain event, like
         * invoke a use case
         */
    }
}

```

Who dictates when a transaction is complete?

This tends to be one of the more challenging things to understand. What should call `dispatchEventsForAggregate(aggregateId: UniqueEntityID)` method?

Should we call it at the end of every **application layer Use Case**?

Should we model the *Unit of Work* pattern and build it into that?

For most simple scenarios, I leave this single responsibility of knowing if the transaction was successful in the ORM being used in the project.

The thing is, a lot of these ORMs actually have mechanisms built in to execute code after things get saved to the database. They're usually called *hooks*.

For example, the Sequelize docs has hooks for each of these lifecycle events.

```

(1)
  beforeBulkCreate(instances, options)
  beforeBulkDestroy(options)
  beforeBulkUpdate(options)
(2)
  beforeValidate(instance, options)
(-)
  validate
(3)

```

```

afterValidate(instance, options)
- or -
validationFailed(instance, options, error)
(4)
beforeCreate(instance, options)
beforeDestroy(instance, options)
beforeUpdate(instance, options)
beforeSave(instance, options)
beforeUpsert(values, options)
(-)
create
destroy
update
(5)
afterCreate(instance, options)
afterDestroy(instance, options)
afterUpdate(instance, options)
afterSave(instance, options)
afterUpsert(created, options)
(6)
afterBulkCreate(instances, options)
afterBulkDestroy(options)
afterBulkUpdate(options)

```

We're interested in the ones in (5).

If this is the case, using Sequelize, we can define a callback function for each hook that takes the model name and the primary key field in order to dispatch the events for the aggregate.

```

// infra/sequelize/hooks/index.ts

import models from "../models"
import { DomainEvents } from "../../../../../core/domain/events/DomainEvents"
import { UniqueEntityID } from "../../../../../core/domain/UniqueEntityID"

const dispatchEventsCallback = (model: any, primaryKeyField: string) => {
  const aggregateId = new UniqueEntityID(model[primaryKeyField])
  DomainEvents.dispatchEventsForAggregate(aggregateId)
}

(async function createHooksForAggregateRoots() {
  const { User } = models

  User.addHook("afterCreate", (m: any) => dispatchEventsCallback(m, "user_id"))
  User.addHook("afterDestroy", (m: any) => dispatchEventsCallback(m, "user_id"))
  User.addHook("afterUpdate", (m: any) => dispatchEventsCallback(m, "user_id"))
  User.addHook("afterSave", (m: any) => dispatchEventsCallback(m, "user_id"))
  User.addHook("afterUpsert", (m: any) => dispatchEventsCallback(m, "user_id"))
}

```

```
})()
```

The benefit of this approach is its ability to keep the **infrastructural concerns** of a transaction out of the **application and domain layers**.

Want to learn more?: For a more detailed discussion on how to decoupling business logic, design transactions, and signal their completion using this approach, read “Decoupling Logic with Domain Events [Guide] - Domain-Driven Design w/ TypeScript”.

All of this might beg the question,

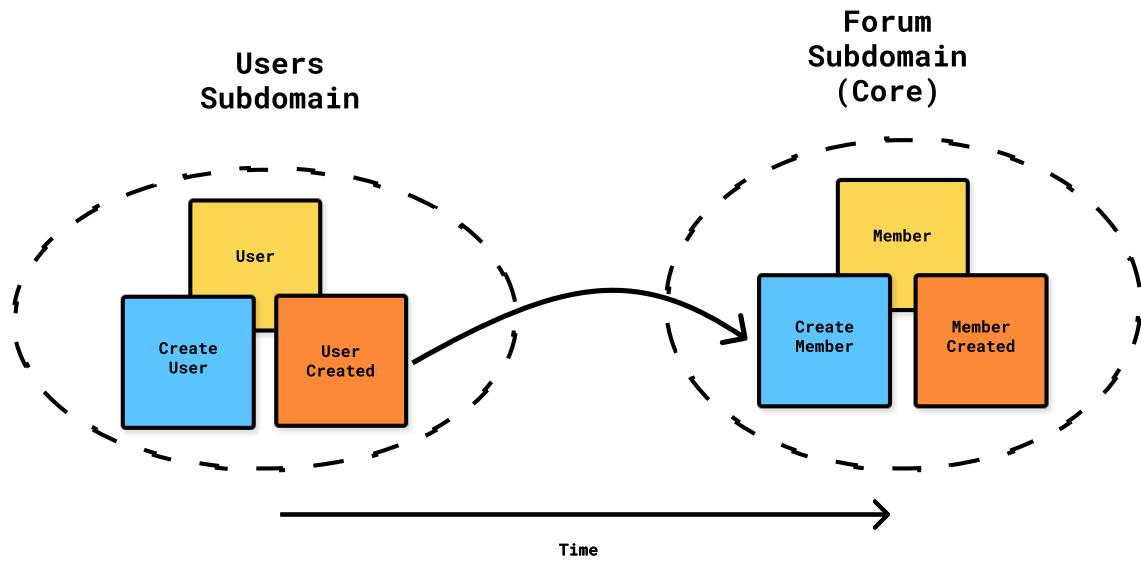
Why not use Node.js *EventEmitter*?

To be fair, you totally *could!* And I encourage you to build your own Domain Events Subject yourself (no copy-pasting). You may find that it works better for your needs. I enjoy the strict-typing and simplicity of Udi’s original approach. They’re just plain ol’ TypeScript objects.

Hopefully, you’re still following along and starting to understand how this all works. If not, hang in there. In the next section, I’ll take you through the program execution that describes exactly how a Member gets created.

Feature 1: Creating a Member

OK, so remember the *Event Stormed* model that we created earlier? Remember how a Member gets created? A Member gets created within the Forum subdomain when it hears a UserCreated event. This means that we’re going to need a way to publish domain events between subdomains. It also means we’re going to need a way to subscribe to Domain Events that we’re interested in so that we can chain the execution of commands.



Alright, enough theory. Let's get into some code.

Issuing an API request

The transaction starts right from when an API request comes into our system and routes to a controller or a resolver.

The transaction starts right from when an API request comes into our system and routes to a controller or a resolver.

```
# Using CURL to send an HTTP POST to a RESTful API

curl -d \
  '{"email":"khalil@apollographql.com", "password":"changeit"}' \
  -H "Content-Type: application/json" \
  -X POST https://api.dddforum.com/v1/users/new
```

If we're into GraphQL, it might look more like this.

```
# Using a GraphQL mutation to issue issue a command
# into our system.
```

```
mutation {
  createUser(email: "khalil@apollographql.com", password: "changeit") {
    accessToken
    refreshToken
  }
}
```

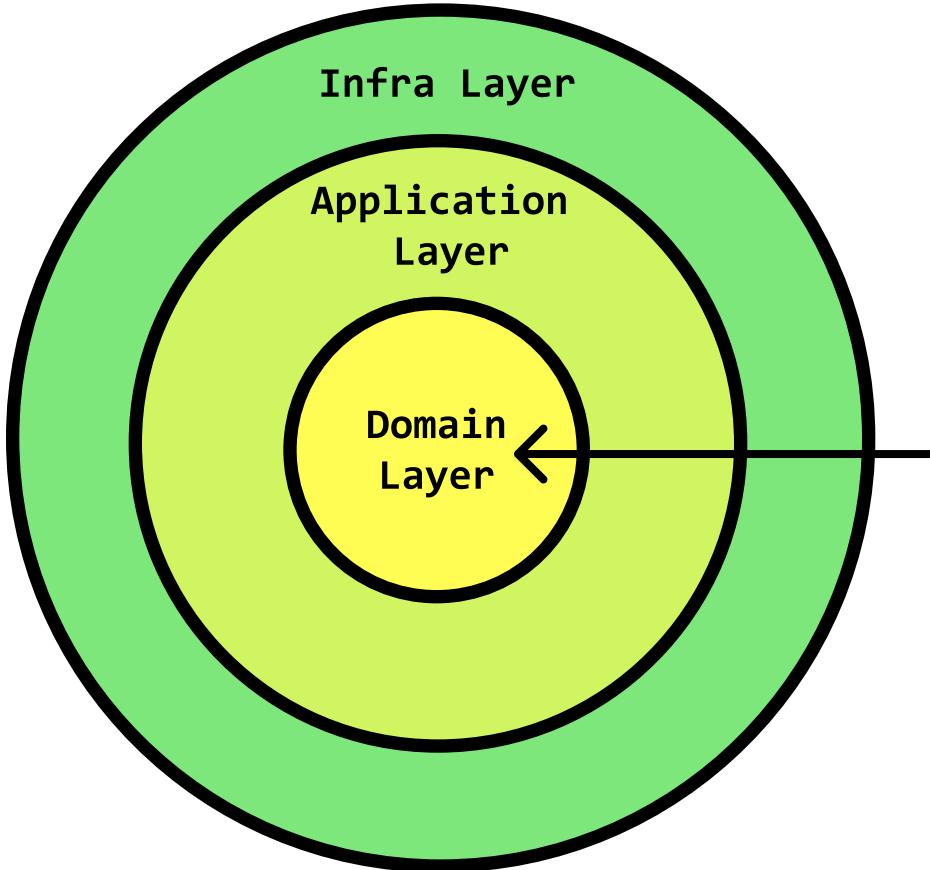
Nonetheless, because we're using a *Hexagonal Architecture*, the *type of API* that we use doesn't affect how our command works. It just affects *what calls* our command. Let's not forget that GraphQL and RESTful APIs, while we can spend a lot of time on their design, are simply API styles that belong to the *Infrastructure Layer* and should have *zero say* as to *how* code from within the critical layers of our project works (application & domain).

```
// Using GraphQL to execute the CreateUser use case.

import {
  createUserUseCase
} from '../users/useCases/createUser'
...

const server = new ApolloServer({
  context: () => ({ createUserUseCase }) ,
  resolvers: {
    Mutation: {
      createUser: (obj, args, context, info) => {
        // Pass execution off to a Use Case / Application Service
        return context.createUserUseCase.execute(args)
      }
    }
  }
})
```

Whether it be through an Express.js controller or a GraphQL resolver, whatever infrastructure object handles API requests, it passes execution off to an **Application Service** (also known as a *Use Case*).



The direction of dependencies flows inward, as does the flow of control in order to execute a command. The domain layer contains the highest policy level of policy.

Application Services/Use Cases

The Application Layer, which sits between the Domain and Infra layers, has a single responsibility.

The role of the application layer is handle all use cases for a particular subdomain

Use Cases (also known as *Application Services*) are objects that perform either a **COMMAND** or a **QUERY** against the system. Yes, these are the same commands and queries that we identified using *Event Storming*, *API-first design*, or *Use Case modeling* (as the name implies). We typically name a *Use Case* by the particular command or query it performs.

In DDDForum, you'll see several `useCases/` folders containing the *Use Cases* that we can perform from within that domain.

Here's what they look like in practice.

Use case interface

Use Cases are simple in principle. They have an optional request and response, and an execute method that takes in that request and returns a response.

```
export interface UseCase< IRequest, IResponse> {
    execute(request?: IRequest): Promise<IResponse> | IResponse
}
```

Creating a new Use Case is as easy as implementing this interface.

Adding the Command (request object)

Let's start by defining the request object (some call this the *Command* object).

Since we're working on *Create User*, we can make a new *Use Case* called *CreateUserUseCase* by implementing the *UseCase< IRequest, IResponse>* interface and temporarily setting the two generic types to any.

```
export class CreateUserUseCase implements UseCase<any, any> {
    public async execute(request: any): Promise<any> {
        return null
    }
}
```

IRequest, the *Command* object, is the first generic parameter to the *UseCase* interface. It allows us to define the shape of the data that we receive from the outside world.

Some also refer to this as the *DTO (Data Transfer Object)* since it's data that is being transferred from one system to another.

We need to create a *DTO/Command* that contains all of the properties required in order to execute the *CreateUserUseCase*. Since the task of this use case is simply to *create a user*, we'll go ahead and include all the stuff it takes in order to create a User.

In order to bring a new User into this world, it's mandatory that we include *username*, *email*, and *password*. This is something we'd already be acutely aware of due to our time spent writing unit tests that confirm we can only create a User with those three properties.

```
// The CreateUserDTO is the input object for our Use Case.
// Any infrastructure layer technology can execute our
// Use Case as long as it passes in these properties.

export interface CreateUserDTO {
    username: string
    email: string
    password: string
}
```

Let's add our new type to the first parameter of the *UseCase* to represent the *request* part of this contract. We'll also change the type for the parameter in the *execute()* method to represent *request DTO* as well.

```
// users/createUser/CreateUserUseCase.ts

export class CreateUserUseCase implements UseCase<CreateUserDTO, any> {
```

```

    public async execute(request: CreateUserDTO): Promise<any> {
        return null; // todo
    }
}

```

Great! We've got our `CreateUserUseCase` mostly set up, it is an Intention Revealing Interface, and all that's left for us to do is implement the logic that **creates a user**.

Fork in the road: Transaction Script vs. Domain Model

Let's pause.

From here, we can proceed in two ways.

The first is the *Transaction Script* approach.

This unhinges us to write the code however we want *in order to make it work*. As a result, the code we write is quite *imperatively* in nature. We write exactly what we want to happen without the sweetness of any abstraction to encapsulate the complexity. Our end goal is to make a User row appear in a database somewhere. Using this approach, one possible solution is to directly import Sequelize, TypeORM, or perhaps `mysqljs` in order to encode the steps that achieve our goal.

That final code to achieve our task might look a little something as follows:

```

// users/createUser/CreateUserUseCase.ts

// In Sequelize, all of your models are exported on a single
// models object.
import { models } from '../../../../../infra/sequelize/models';

export class CreateUserUseCase implements UseCase<CreateUserDTO, void> {

    public async execute(request: CreateUserDTO): Promise<void> {
        const isUsernameValid = !!request.username === true
            && UserUtils.isValidUsername(request.username);
        const isEmailValid = !!request.email === true
            && UserUtils.isValidEmail(request.email);
        const isPasswordPresent = !!request.password
            && UserUtils.isValidPassword(request.password);

        if (!isUsernameValid) {
            throw new Error("Username is not valid");
        }

        if (!isEmailValid) {
            throw new Error("Email is not valid");
        }

        if (!isPasswordPresent) {

```

```

        throw new Error("Password is not valid");
    }

    try {
        await models.User.create({
            username: request.username,
            email: request.email,
            password: UserUtils.hashPassword(request.password),
        });
    } catch (err) {
        throw new Error(`Sequelize error: ${err.toString()}`)
    }
}
}

```

Benefits of this approach:

- It's simple and easy to understand.
- It's fast to implement.

This code does exactly what you'd expect it to do. It makes a row appear in the database and does some validation checks before that. I call this the *brute force* of application layer use cases.

Now let me ask... Do you see anything *wrong* with this approach? Take a moment to think about it. Look at the code and see how many potential issues you can find.

There are 3 big ones.

Disadvantages of this approach:

- I. We've created a **hard source-code dependency** to the Sequelize models by referencing it directly.

```
// Hard source-code dependency to Sequelize models.
import { models } from '../../../../../infra/sequelize/models';
```

Because we reference our sequelize models directly like this, if we ever wanted to run tests against our CreateUserUseCase (which we most certainly will want to do), that'll make our class bring the **the entire database connection with** it every time we create an instance of CreateUserUseCase.

Khalil's Class-level Dependency Methodology: If, from one class, you need to refer another, be aware of the dependency relationship that it creates. Do so *only if* the dependency is either an abstraction (such as an interface or abstract class), is from an *inner layer* (as per The Dependency Rule), or is a stable dependency. Most importantly, abstain from using hard source-code dependencies on classes that you want to test. Read “How I Write Testable Code | Khalil’s Simple Methodology” for a more in-depth discussion.

2. Our **domain model is anemic** and fails to *encapsulate* mandatory validation logic to create a User.

A domain model is *anemic* when *services* (or any non-domain layer classes) contain all the domain logic, yet the domain objects themselves contain practically none. In the previous example, `UserUtils`, a utility class, contains the business rules that dictate how to create a `username`, `email`, and `password`.

This isn't ideal. Because of the lack of encapsulation, it opens up the surface area for someone to be able to create a `User` without adhering to the User validation/creation rules.

Consider if we next needed to develop the `EditUserUseCase`. How easy would it be to forget to first validate that the `username` was valid by utilizing `UserUtils.isValidUsername(name: string)`?

```
// users/editUser/EditUserUseCase.ts

// In Sequelize, all of your models are exported on a single
// models object.
import { models } from '../../../../../infra/sequelize/models';

export class EditUserUseCase implements UseCase<EditUserDTO, void> {

    public async execute(request: EditUserDTO): Promise<void> {
        // Username validation missing.

        const isEmailValid = !!request.email === true
            && UserUtils.isValidEmail(request.email);
        const isPasswordPresent = !!request.password
            && UserUtils.isValidPassword(request.password);

        // Throwing user validation error missing.

        if (!isEmailValid) {
            throw new Error("Email is not valid");
        }

        if (!isPasswordPresent) {
            throw new Error("Password is not valid");
        }

    }
}
```

With this approach, not only do we need to *remember* to include validation logic, but this mandatory business rule now needs to be maintained in *at least two* separate places. Repetition. Not good.

If we wanted to keep the code DRY and enforce the rules, the logic should be a part of a `User` domain model. However, in this case, there isn't even a `User` domain model to encapsulate rules within. There's just the raw Sequelize ORM model.

3. There are serious problems with error handling.

Throwing errors isn't always that helpful. But why? More on this in a moment.

The second approach is to use a *Domain Model*.

With this approach, we're going to respect the separation of concerns of the *Layered Architecture*, encapsulate business rules in models, and reduce the surface area of being able to write code that breaks those rules.

Before we continue, we should talk a little bit about *Errors* in a rich domain model.

What follows is an approach we can use to handle errors and represent them explicitly as domain concepts.

Handling errors as domain concepts

In most programming projects, there's confusion as to how and where errors should be handled.

Errors account of a large portion of our application's possible states, and more often than not, it's one of the last things considered.

When we encounter some code that will probably result in a non-optimal state, we often ask ourselves questions like:

- Do I throw an error and let the client figure out how to handle it?
- Do I return null?

Neither of these are *fantastic* approaches.

When we throw errors, we disrupt the flow of the program and make it trickier for someone to walk through the code, since breaking the natural flow of a program with errors shares similarities to the sometimes criticized GOTO command in older programming languages.

And when we return null, we're breaking the design principle that "**a method should return a single type**". Not adhering to this can result in *more errors, trust issues*, and can lead to the misuse of our methods from clients.

It's funny that we don't really know how to treat errors when they account for so much program behavior.

It's also very often that for a single use case, there are at least **one or more ways** that the use case could *fail*. For example, creating a user could fail for one of these two reasons:

1. The user already exists
2. The username has been taken

And that doesn't even account for unexpected errors or validation errors.

It's time to understand that some errors are domain concepts.

Errors have a place in the domain, and they deserve to be modeled as domain concepts

When we express our errors as domain concepts, our domain model becomes a lot richer, says a lot more about the actual problem domain, and we can construct a return type that forces the client to do something with an error.

That leads to improved readability and fewer bugs.

Here's how I like to model my errors expressively.

I create a base `UseCaseError` class that represents errors that could happen for any use case (obviously).

```
// shared/core/UseCaseError.ts

interface IUseCaseError {
  message: string
}

export abstract class UseCaseError implements IUseCaseError {
  public readonly message: string

  constructor(message: string) {
    this.message = message
  }
}
```

Then I create an error namespace, writing the use case-specific ways that a use case can fail. Check out what I mean below.

```
// users/useCases/createUser/CreateUserErrors.ts

import { UseCaseError } from "../../../../../shared/core/UseCaseError"
import { Result } from "../../../../../shared/core/Result"

export namespace CreateUserErrors {
  export class EmailAlreadyExistsError extends Result<UseCaseError> {
    constructor(email: string) {
      super(false, {
        message: `The email ${email} associated for this account already exists`,
        } as UseCaseError)
    }
  }

  export class UsernameTakenError extends Result<UseCaseError> {
    constructor(username: string) {
      super(false, {
        message: `The username ${username} was already taken`,
        } as UseCaseError)
    }
  }
}
```

With these errors defined, I can construct a return type for the `Use Case` that adequately says, “What you’re going to get back from this method is an object that is *either* one of

all possible **error** states OR the **success** value.”

Check it out.

```
import { Either, Result } from "../../../../../shared/core/Result";
import { CreateUserErrors } from "./CreateUserErrors";
import { AppError } from "../../../../../shared/core/AppError";

export type CreateUserResponse = Either<
  CreateUserErrors.EmailAlreadyExistsError |
  CreateUserErrors.UsernameTakenError |
  AppError.UnexpectedError |
  Result<any>,
  Result<void>
```

You probably have a lot of questions about how this works.

I'm going to direct you to *two* resources that I'd like you to read.

Resource 1: “Flexible Error Handling w/ the Result Class | Enterprise Node.js + TypeScript”. This explains the Result class, which is foundational to how we functionally handle errors.

Resource 2: “Functional Error Handling with Express.js and DDD | Enterprise Node.js + TypeScript”. This shows you how to utilize the Either monad to segregate success states from failure states.

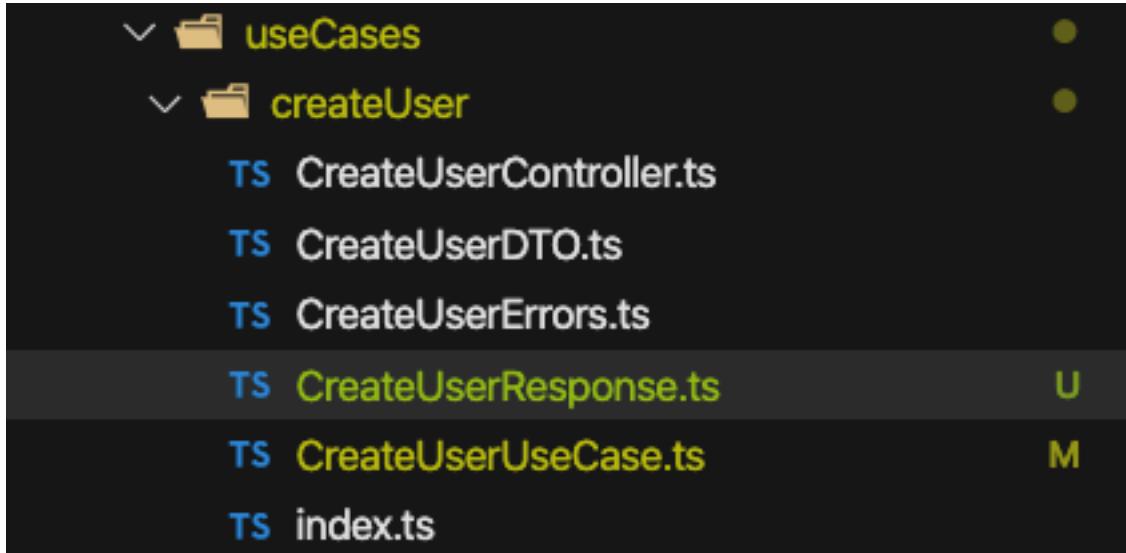
Read those first, then continue.

Summary on Use Cases/Application Services

Use cases are infrastructure-layer concern agnostic. If we design them as little modules that as long as we can provide the correct inputs, they'll know how to execute the features of our system, they can stay agnostic to what API style executes them.

You'll notice that in the DDDForum codebase, a single useCase/ folder has several files in it. Each folder has the *Controller*, the *DTO*, the *UseCaseErrors*, and the *Use Case* itself.

This type of co-location of files that are closely related to each other for a common task *is good* for several reasons.



A use case module contains all of the components that it needs in order to do its job (high cohesion).

1. High cohesion
2. Locate-ability
3. Improved maintainability

Because each of these files need each other in order to accomplish their task, having them close to each other *improves maintainability*. Think about other developers for a second. If we organized all of our code by *construct type*, like controllers/, errors/, dtos/, and had every single construct from our app in one of those folders, it would make maintaining the code a nightmare. We'd have to flip between folders several times in order to update a feature.

By *packaging by module*, we keep everything for a particular feature as close as possible.

And we export everything that the outside world needs to know about!

```
// The createUser folder acts as a module, exporting
// only what is necessary for the outside world to
// know about.

// useCases/createUser/index.ts

import { CreateUserUseCase } from "./CreateUserUseCase";
import { CreateUserController } from "./CreateUserController";
import { userRepo } from "../..../repos";

const createUserUseCase = new CreateUserUseCase(userRepo);
const createUserController = new CreateUserController(
  createUserUseCase
)

// Just these two things!
export {
```

```

    createUserUseCase,
    createUserController
}

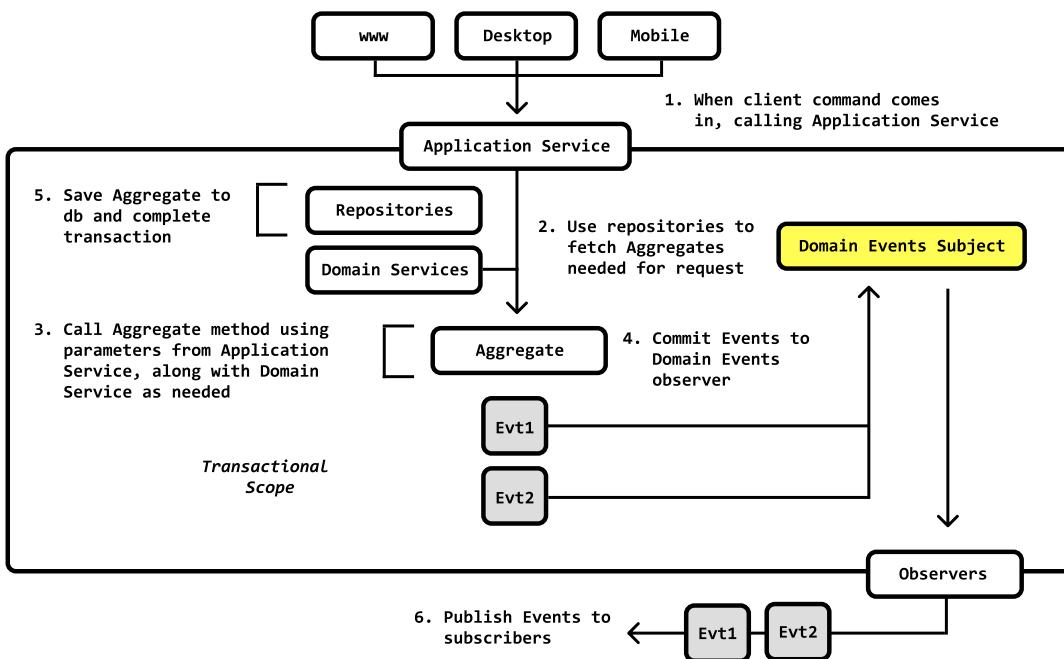
```

See also: “Why I Don’t Use a DI Container | Node.js w/ TypeScript”

Inside the CreateUser use case transaction

A look inside the CreateUserUseCase should help you to understand the big picture of how all the pieces of this architecture work together. Ultimately, the goal is to perform an operations against an *Aggregate* and publish *Domain Events* emitted from the *Aggregate* to subscribers interested in their occurrences.

The figure below is a depiction of how an Application Service/Use Case handles a request and facilitates a transaction against an *Aggregate*.



An Application Service/Use Case handling a request to perform a transaction against an Aggregate.

1. A client request can come from an API endpoint, passing off control to an Application Service/Use Case.
2. Application Service/Use Case uses the dependency injected repositories in order to retrieve any Aggregates needed to continue with the request.
3. Application Service/Use Case calls the Aggregate's method using parameters supplied by the Application Service. If we're dealing with more than one Aggregate in this request, and the domain logic for our transaction doesn't belong to a single *Aggregate*, we pass the parameters to a Domain Service instead (see *Feature 2 - Upvoting a Post* for example).

4. The *Aggregate*'s method dispatches *Domain Events* as a side effect. These Event are added to the *Domain Events Subject*, an implementation of the Observer Pattern.
5. The transaction is completed when the Aggregate is saved to the database successfully.
6. Upon successful database save, the *Domain Events Subject* notifies *Observers* by sending them the *Domain Events* for the *Aggregate* just persisted in the last transaction.

Finally, any other subdomains within our Monolithic application can subscribe to the Domain Events by name ahead of time in order to be notified when they're emitted.

Using an Express.js Route Handler to direct the request

For a RESTful API, an Express.js route handler is defined for the create user endpoint. The route handler passes control to a `createUserController` (one of the dependencies we export from within our `useCases/createUser` module).

```
// Express.js RESTful API uses an exported
// createUserController to handle a post to /users.

// users/infra/http/routes/index.ts

import express from 'express'
import {
  createUserController
} from '../../../../../useCases/createUser';

const userRouter = express.Router();

userRouter.post('/',
  (req, res) => createUserController.execute(req, res)
);
...
```

Handling the API request with an API Controller

Inside of the `users/useCases/createUser` folder, we find the `CreateUserController`.

The `CreateUserController` has a single dependency injected into it. The `CreateUserUseCase`. It's the Application Service/Use Case that does all the action.

The responsibilities of a controller are to:

- Invoke the `CreateUserUseCase`.
- Respond to the API call with the appropriate HTTP response code based on the result of the executed use case.

See the `CreateUserController` below.

```
// CreateUsersController - Handling the request data from
// the internet and passing it to the dependency injected
// use case class for execution.
```

```
// users/useCases/createUser/CreateUserController.ts

import { CreateUserUseCase } from "./CreateUserUseCase"
import { CreateUserDTO } from "./CreateUserDTO"
import { CreateUserErrors } from "./CreateUserErrors"
import { BaseController } from "../../../../../shared/infra/http/models/BaseController"
import { TextUtils } from "../../../../../shared/utils.TextUtils"
import { DecodedExpressRequest } from "../../../../../infra/http/models/decodedRequest"
import * as express from "express"

export class CreateUserController extends BaseController {
    private useCase: CreateUserUseCase

    constructor(useCase: CreateUserUseCase) {
        super()
        this.useCase = useCase
    }

    async executeImpl(
        req: DecodedExpressRequest,
        res: express.Response
    ): Promise<any> {
        let dto: CreateUserDTO = req.body as CreateUserDTO

        dto = {
            username: TextUtils.sanitize(dto.username),
            email: TextUtils.sanitize(dto.email),
            password: dto.password,
        }

        try {
            /**
             * Execute the CreateUserUseCase use case.
             */
        }

        const result = await this.useCase.execute(dto)

        if (result.isLeft()) {
            const error = result.value

            /**
             * This is an example of how one can standardize a
             * RESTful API's error messages.
             *
             * We utilize the Use Case Error types.
             */
        }
    }
}
```

```

        switch (error.constructor) {
            case CreateUserErrors.UsernameTakenError:
                return this.conflict(error.errorValue().message)
            case CreateUserErrors.EmailAlreadyExistsError:
                return this.conflict(error.errorValue().message)
            default:
                return this.fail(res, error.errorValue().message)
        }
    } else {
        return this.ok(res)
    }
} catch (err) {
    return this.fail(res, err)
}
}
}
}

```

Like this?: If you like the way this Express.js controller uses declarative response methods, check out “Clean & Consistent Express.js Controllers | Enterprise Node.js + TypeScript”.

Let's take a look at the use case now.

Invoking the Application Service / Use Case

In Domain-Driven Design, Application Services have a specific purpose.

The purpose of an *Application Services* is to get all the stuff needed in order for domain layer concerns to interact, than to save the result to persistence.

In the following example, using *Interface Adapters* (like `IUserRepo`), the responsibility is to:

- Fetch the *Aggregate(s)* and any other objects needed in order to execute the domain logic.
- Pass the objects to either an *Aggregate*'s method OR a *Domain Service*.
- Save the results of the transaction to persistence.

Essentially, an *Application Service* is the glue that interacts with databases, caches, and anything else needed in order to connect our fully encapsulated domain model to the real world.

For example, in our `CreateUserUseCase`, we need a way to determine if the User was already created or not, and we also need a way to determine if the username for the User we're trying to create has already been taken.

We need a *User Repository*. We can get one by referring to the interface, that is - the *Interface Adapter*. It doesn't matter to the Application Service, but we can implement one using Sequelize, TypeORM, MongoDB, etc - in the constructor for the `CreateUserUseCase`.

```
// useCases/createUser/CreateUserUseCase.ts

export class CreateUserUseCase
```

```

    implements UseCase<CreateUserDTO, Promise<CreateUserResponse>> {

        private userRepo: IUserRepo;

        // Dependency injected and inverted IUserRepo.
        constructor(userRepo: IUserRepo) {
            this.userRepo = userRepo
        }

        ...
    }
}

```

An important consideration about *Application Services* is that they should contain little to no domain logic at all within them. Anything that smells like *domain logic* should live in either an *Aggregate* or a *Domain Service*, but never an *Application Service*.

The reason why we do this is primarily to avoid code duplication. By encapsulating domain logic in the *Domain Layer*, anytime we want to execute a feature via the *Application Layer*, we have to offload the business logic to the *Domain Layer*.

Application Services, er- *Use Cases*, aren't *Domain Layer* constructs. They're *Application Layer* constructs. And if we create new applications, we'll have new *Use Cases/Application Services*. If there's a form of domain logic that's important to some of the new *Use Cases/Application Services* but it lives within an old *Application Service*, that logic will need to be duplicated. Again, that's an anemic domain model.

So, we have to keep an eye out for stuff that looks like domain logic, and aim to move it into the *Aggregate* or a *Domain Service* it best belongs within.

Application Service/Use Case design tip: Take a step back and look at the Cyclomatic Complexity **** of your Application Service/Use Cases. If you notice there's more than 2 layers of complexity (at least 2 nested control blocks like *if*, *while*, or *switch*), there's a good chance that we've got some leaky domain logic.

Here's what our *CreateUserUseCase Application Service/Use Case* looks like.

```

// users/useCases/createUser/CreateUserUseCase.ts

import { CreateUserDTO } from "./CreateUserDTO"
import { CreateUserErrors } from "./CreateUserErrors"
import { Result, left, right } from "../../../../../shared/core/Result"
import { AppError } from "../../../../../shared/core/AppError"
import { IUserRepo } from "../../repos/userRepo"
import { UseCase } from "../../../../../shared/core/UseCase"
import { UserEmail } from "../../../../../domain/userEmail"
import { UserPassword } from "../../../../../domain/userPassword"
import { UserName } from "../../../../../domain/userName"
import { User } from "../../../../../domain/user"
import { CreateUserResponse } from "./CreateUserResponse"

```

```

export class CreateUserUseCase implements UseCase<CreateUserDTO, Promise<CreateUserResponse>> {
    private userRepo: IUserRepo

    // Dependency injected and inverted IUserRepo.
    constructor(userRepo: IUserRepo) {
        this.userRepo = userRepo
    }

    async execute(request: CreateUserDTO): Promise<CreateUserResponse> {
        // Run validation logic on each of the inputs needed to create
        // a user by creating their Value Objects.
        const emailOrError: Result<UserEmail> = UserEmail.create(request.email)

        const passwordOrError: Result<UserPassword> = UserPassword.create({
            value: request.password,
        })

        const usernameOrError: Result<UserName> = UserName.create({
            name: request.username,
        })

        // Determine if they are valid.
        const dtoResult = Result.combine([
            emailOrError,
            passwordOrError,
            usernameOrError,
        ])

        if (dtoResult.isFailure) {
            return left(Result.fail<void>(dtoResult.error))
        }

        // If each value object is valid, we can continue with the
        // request. We pull the results out so that we can use them.
        const email: UserEmail = emailOrError.getValue()
        const password: UserPassword = passwordOrError.getValue()
        const username: UserName = usernameOrError.getValue()

        try {
            // Determine if the user already exists. And if it does,
            // the use case should fail with a EmailAlreadyExistsError
            const userAlreadyExists = await this.userRepo.exists(email)

            if (userAlreadyExists) {
                return left(new CreateUserErrors.EmailAlreadyExistsError(email.value))
            }
        }
    }
}

```

```

// Determine if the username is already in use. If it does,
// the use case should fail with a UsernameTakenError.
try {
    const alreadyCreatedUserByUserName = await this.userRepo.getUserByUserName(
        username
    )

    const userNameTaken = !!alreadyCreatedUserByUserName === true

    if (userNameTaken) {
        return left(new CreateUserErrors.UsernameTakenError(username.value))
    }
} catch (err) {}

// If all is well, we get to create the user.
// Behind the scenes, a UserCreated domain event is created
// and the User aggregate is marked.
const userOrError: Result<User> = User.create({
    email,
    password,
    username,
})

if (userOrError.isFailure) {
    return left(Result.fail<User>(userOrError.error.toString()))
}

const user: User = userOrError.getValue()

// Finally, we can complete the request by passing it off to a
// repository to save it to persistence.
await this.userRepo.save(user)

    return right(Result.ok<void>())
} catch (err) {
    return left(new AppError.UnexpectedError(err))
}
}
}

```

First thing the *Application Service/Use Case* does is create *Value Objects* from the string properties passed in via the `CreateUserDTO`. If any of those *Value Objects* fail to pass the encapsulated validation rules, they return a **failed** `Result<T>` object.

Otherwise, if all of the *Value Objects* look to be in order, we continue first determining if the User has already been created, or if the username has already been taken.

If the User is completely new, we'll go ahead and create the User domain object by passing in all of the previously created and valid *Value Objects*.

In the background, when we create a User, it:

- Creates the unique identifier for the User (UUID).
- Creates a UserCreated *Domain Event* and notifies the Domain Events subject that it was created.

Lastly, we complete the transaction by passing the User domain object with a *Domain Event* attached to it to the UserRepo to save to persistence.

Saving the Aggregate with Sequelize

Inside the save method for the SequelizeUserRepo that implements IUserRepo, we use a Mapper in order to convert a User into the shape needed to persist it to Sequelize, as per the Sequelize docs.

```
// users/repos/implementations/sequelizeUserRepo.ts

import { UserMap } from '../mappers/userMap'

export class SequelizeUserRepo implements IUserRepo {
    private models: any;

    constructor(models: any) {
        this.models = models;
    }

    async exists(userEmail: UserEmail): Promise<boolean> {
        ...
    }

    async save(user: User): Promise<void> {
        const UserModel = this.models.BaseUser;
        const exists = await this.exists(user.email);

        if (!exists) {
            // Create a JSON representation { username, password, userId, email }
            const rawSequelizeUser = await UserMap.toPersistence(user);
            // Save the user model
            await UserModel.create(rawSequelizeUser);
        } else {
            // Update logic
            ...
        }

        return;
    }
}
```

```
}
```

Note: Notice that we import UserMap directly? Normally, we wouldn't do that.
But UserMap is a stable dependency.

Here's what the UserMap looks like. Pay particular attention to the toPersistence method where we take in a User and convert it to an untyped object. That untyped object is what Sequelize needs to save the user to the database.

```
// users/mappers/userMap.ts

import { Mapper } from '../../shared/infra/Mapper'
import { User } from '../domain/user';
import { UserDTO } from '../dtos/userDTO';
import { UniqueEntityID } from '../../shared/domain/UniqueEntityID';
import { UserName } from '../domain/username';
import { UserPassword } from '../domain/userPassword';
import { UserEmail } from '../domain/userEmail';

export class UserMap implements Mapper<User> {
    public static toDTO (user: User): UserDTO {
        return {
            username: user.username.value,
            isEmailVerified: user.isEmailVerified,
            isAdminUser: user.isAdminUser,
            isDeleted: user.isDeleted
        }
    }

    public static toDomain (raw: any): User {
        const userNameOrError = UserName.create({ name: raw.username });
        const userPasswordOrError = UserPassword.create({ value: raw.user_password, hashed: raw.user_password hashed });
        const userEmailOrError = UserEmail.create(raw.user_email);

        const userOrError = User.create({
            username: userNameOrError.getValue(),
            isAdminUser: raw.is_admin_user,
            isDeleted: raw.is_deleted,
            isEmailVerified: raw.is_email_verified,
            password: userPasswordOrError.getValue(),
            email: userEmailOrError.getValue(),
        }, new UniqueEntityID(raw.base_user_id));

        userOrError.isFailure ? console.log(userOrError.error) : '';
        return userOrError.isSuccess ? userOrError.getValue() : null;
    }
}
```

```

public static async toPersistence (user: User): Promise<any> {
    let password: string = null;
    if (!user.password === true) {
        if (user.password.isAlreadyHashed()) {
            password = user.password.value;
        } else {
            password = await user.password.getHashedValue();
        }
    }

    return {
        base_user_id: user.userId.id.toString(),
        user_email: user.email.value,
        is_email_verified: user.isEmailVerified,
        username: user.username.value,
        user_password: password,
        is_admin_user: user.isAdminUser,
        is_deleted: user.isDeleted
    }
}
}

```

Mapper<T> classes are solely responsible for mapping entities between Domain, Persistence, and DTO format.

Back in the SequelizeUserRepo, we then complete the transaction using that raw object we get back from the UserMap with UserModel.create(rawSequelizeUser).

```

// users/repos/implementations/sequelizeUserRepo.ts

// Save the user model
await UserModel.create(rawSequelizeUser);

```

That's all we need to complete the entire transaction.

But what about the *Domain Events*?

Notifying subscribers and dispatching Domain Events from Sequelize Hooks

With Sequelize, recall that we can make use of the lovely Hollywood Principle (*"don't call us, we'll call you"*), by defining callbacks on the lifecycle hooks that get called when we perform operations against the database.

```

// shared/infra/database/sequelize/hooks/index.ts

import models from "../models"
import { UniqueEntityID } from "../../../../../domain/UniqueEntityID"
import { DomainEvents } from "../../../../../domain/events/DomainEvents"

```

```

const dispatchEventsCallback = (model: any, primaryKeyField: string) => {
  // Get the aggregate id from the Sequelize model just saved/updated.
  const aggregateId = new UniqueEntityID(model[primaryKeyField])
  // Dispatch any domain events on that aggregate from a previous transaction.
  DomainEvents.dispatchEventsForAggregate(aggregateId)
}

(async function createHooksForAggregateRoots() {
  const { BaseUser, Member, Post } = models

  // Notify subscribers when the User aggregate transactions complete
  BaseUser.addHook("afterCreate", (m: any) =>
    dispatchEventsCallback(m, "base_user_id")
  )
  BaseUser.addHook("afterDestroy", (m: any) =>
    dispatchEventsCallback(m, "base_user_id")
  )
  BaseUser.addHook("afterUpdate", (m: any) =>
    dispatchEventsCallback(m, "base_user_id")
  )
  BaseUser.addHook("afterSave", (m: any) =>
    dispatchEventsCallback(m, "base_user_id")
  )
  BaseUser.addHook("afterUpsert", (m: any) =>
    dispatchEventsCallback(m, "base_user_id")
  )

  // Notify subscribers when the Member aggregate transactions complete
  Member.addHook("afterCreate", (m: any) =>
    dispatchEventsCallback(m, "member_id")
  )
  Member.addHook("afterDestroy", (m: any) =>
    dispatchEventsCallback(m, "member_id")
  )
  Member.addHook("afterUpdate", (m: any) =>
    dispatchEventsCallback(m, "member_id")
  )
  Member.addHook("afterSave", (m: any) =>
    dispatchEventsCallback(m, "member_id")
  )
  Member.addHook("afterUpsert", (m: any) =>
    dispatchEventsCallback(m, "member_id")
  )

  // Notify subscribers when the Post aggregate transactions complete
  Post.addHook("afterCreate", (m: any) => dispatchEventsCallback(m, "post_id"))
})

```

```

Post.addHook("afterDestroy", (m: any) => dispatchEventsCallback(m, "post_id"))
Post.addHook("afterUpdate", (m: any) => dispatchEventsCallback(m, "post_id"))
Post.addHook("afterSave", (m: any) => dispatchEventsCallback(m, "post_id"))
Post.addHook("afterUpsert", (m: any) => dispatchEventsCallback(m, "post_id"))

console.log("[Hooks]: Sequelize hooks setup.")
})()

```

We use the hooks to tell our Domain Events subject that we should notify all subscribers to the particular *Domain Event* (in this case, the *UserCreated* event).

Read the docs: If you're using Sequelize, can read the documentation on Sequelize hooks here.

Not using Sequelize? Using raw SQL? Write 'yer own dang hooks library: In this case, you get the opportunity to write this great example of *inversion of control* yourself. Using the Hollywood Principle, you could wrap each create, insert, delete, or update with pre-hooks and post-hooks where you accept a callback function for future logic to get written. Don't forget to supply useful metadata to the callback so that we can do something similar to what we've done using Sequelize.

Chaining the CreateMember command from the Forum subdomain

The transaction that **created a user** in the Users subdomain has finished executing. The last piece to the puzzle in decoupling business logic between subdomains in a modular monolith is getting the Forum subdomain to react to the UserCreated event.

From the Forum subdomain, we can create a class to act as a subscriber to the UserCreated event. For scan-ability at the folder level, my personal preference is to name these like After-[event name].

Take a look at the AfterUserCreated class.

```

// forum/subscriptions/afterUserCreated.ts

import { UserCreated } from "../../users/domain/events/userCreated"
import { IHandle } from "../../shared/domain/events/IHandle"
import { CreateMember } from "../useCases/members/createMember/CreateMember"
import { DomainEvents } from "../../shared/domain/events/DomainEvents"

export class AfterUserCreated implements IHandle<UserCreated> {
    private createMember: CreateMember

    constructor(createMember: CreateMember) {
        this.setupSubscriptions()
        this.createMember = createMember
    }

    setupSubscriptions(): void {
        // Register to the domain event
    }
}

```

```

        DomainEvents.register(this.onUserCreated.bind(this), UserCreated.name)
    }

private async onUserCreated(event: UserCreated): Promise<void> {
    const { user } = event

    try {
        await this.createMember.execute({ userId: user.userId.id.toString() })
        console.log(
            ` [AfterUserCreated]: Successfully executed CreateMember use case AfterUserCreated`
        )
    } catch (err) {
        console.log(
            ` [AfterUserCreated]: Failed to execute CreateMember use case AfterUserCreated.`
        )
    }
}
}

```

This class implements the `IHandle<T>` interface, which is really just an Intention Revealing Interface. It doesn't do much other than help to signal to the reader what the class is for, and reminds us to implement the `setupSubscriptions` method.

```

// shared/domain/events/IHandle.ts

import { IDomainEvent } from "./IDomainEvent";

export interface IHandle<IDomainEvent> {
    setupSubscriptions(): void;
}

```

In the constructor for `AfterUserCreated`, we make sure to import the `UseCase<T, U>` from the `Forum` subdomain that we want to invoke *after the user is created*. That happens to be the `CreateMember` use case.

```

// forum/subscriptions/afterUserCreated.ts

export class AfterUserCreated implements IHandle<UserCreated> {
    private createMember: CreateMember

    constructor(createMember: CreateMember) {
        this.setupSubscriptions()
        this.createMember = createMember
    }
    ...
}

```

In the `setupSubscriptions` method, we set up a subscription to the `UserCreated Domain Event` by using the Domain Event subject's `register(callback: Function, eventName:`

string) method. It's important to bind to this so that we can execute the `createMember` use case, a property of the `AfterUserCreated` class.

```
// forum/subscriptions/afterUserCreated.ts

export class AfterUserCreated implements IHandle<UserCreated> {
    ...
    setupSubscriptions(): void {
        // Register to the domain event
        DomainEvents.register(this.onUserCreated.bind(this), UserCreated.name)
    }
    ...
}
```

Finally, when the Domain Events subject invokes the callbacks for all subscribers to the `UserCreated` *Domain Event*, it'll call this `onUserCreated` method.

It's here that we can follow up with an invocation of `createMember`.

```
// forum/subscriptions/afterUserCreated.ts

export class AfterUserCreated implements IHandle<UserCreated> {
    ...
    private async onUserCreated(event: UserCreated): Promise<void> {
        const { user } = event

        try {
            await this.createMember.execute({ userId: user.userId.id.toString() })
            console.log(`[AfterUserCreated]: Successfully executed CreateMember use case AfterUserCreated`)
        }
        } catch (err) {
            console.log(`[AfterUserCreated]: Failed to execute CreateMember use case AfterUserCreated`)
        }
    }
    ...
}
```

Queues and Event Stores?: What happens if we fail to follow up with the `CreateUser` use case in response to the `UserCreated` *Domain Event*? Is there a way to *re-try*? Maybe after some amount of time, like an exponential backoff? There are plenty of patterns we can use, like the Transactional Outbox. Just be warned, this kind of thing is called an *Enterprise Integration Pattern*, and the rabbit hole runs deep. I didn't realize it until later, but EIPs are the terminal point of complexity in Software Design and Architecture for web applications. Dealing with events, messaging, and the design around networking, contingency, and convergence is *if you ask me, the hardest problem*. It's also the stuff we need to solve if we decide to take on Event Sourcing, which is why we've scoped this chapter down a bit. I plan on

learning and distilling this complex realm of architecture in a follow up chapter in 2021. If you're keen on learning more today, check out [EnterpriseIntegrationPatterns.com](#) and the book of the same name.

Of course, in order to get our subscriptions up and running, we'll need to do a little bit of plumbing. If our `npm run start` script uses `src/index.ts` as the entry point to our app, since Node.js imports are singletons, we need to make sure to *start* our subscriptions by mentioning the name of whatever creates our subscription classes.

```
// src/index.ts

// Infra
import "./shared/infra/http/app"
import "./shared/infra/database/sequelize"

// Subscriptions
import "./modules/forum/subscriptions";
```

Following `modules/forum/subscriptions`, we setup all of our Forum subdomain subscriptions with the following code.

```
// modules/forum/subscriptions/index.ts

import { createMember } from "../useCases/members/createMember";
import { AfterUserCreated } from "./afterUserCreated";
import { AfterCommentPosted } from "./afterCommentPosted";
import { updatePostStats } from "../useCases/post/updatePostStats";
import { AfterCommentVotesChanged } from "./afterCommentVotesChanged";
import { updateCommentStats } from "../useCases/comments/updateCommentStats";
import { AfterPostVotesChanged } from "./afterPostVotesChanged";

// Subscriptions
new AfterUserCreated(createMember);
new AfterCommentPosted(updatePostStats);
new AfterCommentVotesChanged(updatePostStats, updateCommentStats);
new AfterPostVotesChanged(updatePostStats);
```

Looking at the `CreateMember` use case that gets called in response to the `UserCreated Domain Event`, you'll notice that it follows the same structure as the `CreateUser` use case, and executes similar logic.

```
// forum/useCases/members/createMember/CreateMember.ts

import { UseCase } from "../../../../../shared/core/UseCase"
import { IMemberRepo } from "../../../../repos/memberRepo"
import { CreateMemberDTO } from "./CreateMemberDTO"
import { IUserRepo } from "../../../../users/repos/userRepo"
import { Either, Result, left, right } from "../../../../../shared/core/Result"
import { AppError } from "../../../../../shared/core/AppError"
```

```

import { CreateMemberErrors } from "./CreateMemberErrors"
import { User } from "../../../../../users/domain/user"
import { Member } from "../../../../domain/member"

type Response = Either<
  | CreateMemberErrors.MemberAlreadyExistsError
  | CreateMemberErrors.UserDoesntExistError
  | AppError.UnexpectedError
  | Result<any>,
  Result<void>
>

export class CreateMember
  implements UseCase<CreateMemberDTO, Promise<Response>> {
  private memberRepo: IMemberRepo
  private userRepo: IUserRepo

  constructor(userRepo: IUserRepo, memberRepo: IMemberRepo) {
    this.userRepo = userRepo
    this.memberRepo = memberRepo
  }

  public async execute(request: CreateMemberDTO): Promise<Response> {
    let user: User
    let member: Member
    const { userId } = request

    try {
      try {
        user = await this.userRepo.getUserById(userId)
      } catch (err) {
        return left(new CreateMemberErrors.UserDoesntExistError(userId))
      }

      try {
        member = await this.memberRepo.getMemberById(userId)
        const memberExists = !!member === true

        if (memberExists) {
          return left(new CreateMemberErrors.MemberAlreadyExistsError(userId))
        }
      } catch (err) {}

      // Member doesn't exist already (good), so we want to create it
      const memberOrError: Result<Member> = Member.create({
        userId: user.userId,

```

```

        username: user.username,
    })

    if (memberOrError.isFailure) {
        return left(memberOrError)
    }

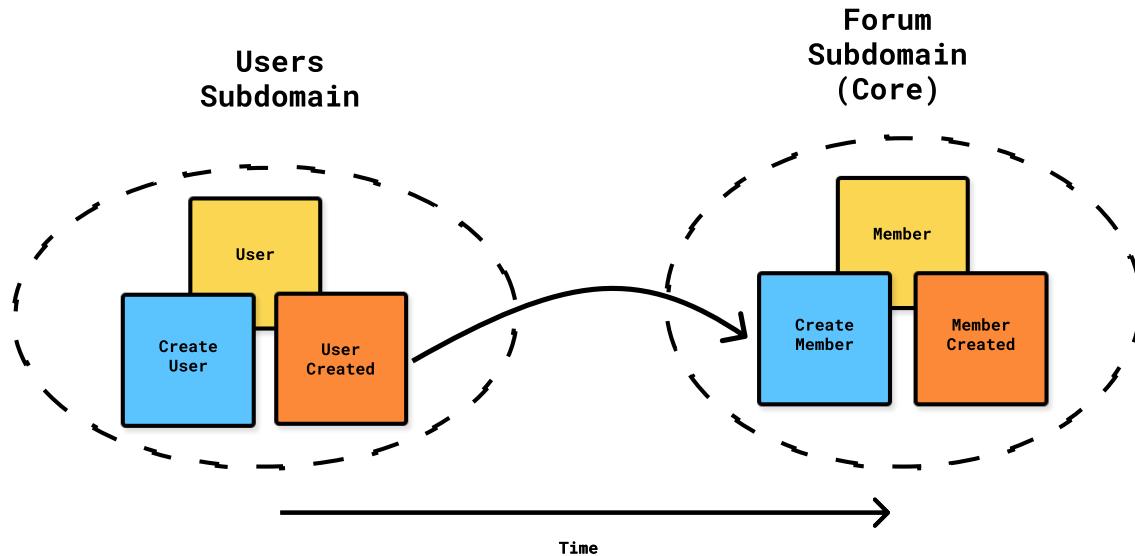
    member = memberOrError.getValue()

    await this.memberRepo.save(member)

    return right(Result.ok<void>())
} catch (err) {
    return left(new AppError.UnexpectedError(err))
}
}
}
}

```

That's it! We just went through the entire process of handling a *Command*. Our first *Use Case* in DDDForum. We saw how to dispatch *Domain Events* and react to them from separate subdomains within a *Modular Monolith* (single bounded context). For a reminder, check the image below to see how far we've come so far.



The UserCreated domain event was dispatched from the Users subdomain and reacted to from within the Forum subdomain by invoking the CreateMember command.

Feature 2: Upvote a post

In the previous example, we looked at the lifecycle of a *Create* operation in a *Clean Architecture/DDD* application using the *Ports and Adapters* approach. We saw how to handle a request from the moment it comes in through the public API to the moment it passes off execution to the application and domain layer constructs.

In this next feature, which is to *Upvote a Post*, I believe you'll get a better appreciation for the way DDD enables us to encapsulate complex domain logic.

In this feature, we'll have a brief discussion about how we design aggregates, what belongs in them, and how to use them to perform state changes. Lastly, and most importantly, we'll get an opportunity to learn about the utility of *Domain Services* by seeing one in action.

Understanding voting domain logic

Authenticated Users can both **upvote** and **downvote** posts and comments.

During an *Event Storming* session, we discovered the Upvote Post, Upvote Comment, Downvote Post, and Downvote Comment commands, but given the nature of *Event Storming*, it's unlikely we'd have had the time to sit down with each *Command* and expand on the scenarios.

All *Commands* invoke change, it's essential to document the preconditions that dictate **when**, and rules that specify **how** the system may change. Let's use the Given-When-Then style test specifications to plan this out.

Gherkin test specifications

Gherkin is a domain-specific language made for describing the business behavior without needing to into specifics of implementation details.

Since each *Command* is a *feature*, we define the criteria describing how the feature works. When we write tests for the criteria, and they pass, we're done the feature and onto the next one!

Here's an example of a Gherkins test specification for Upvote Post:

```
Feature: Upvoting a post
```

```
Upvoting posts is a primary feature of DDDforum that
can be done by users who have created an account
```

```
Scenario: Upvoting a post I've already submitted
  Given I am authenticated, I have submitted a post,
  and I previously upvoted it
  When I upvote the post
  Then the post should not change
```

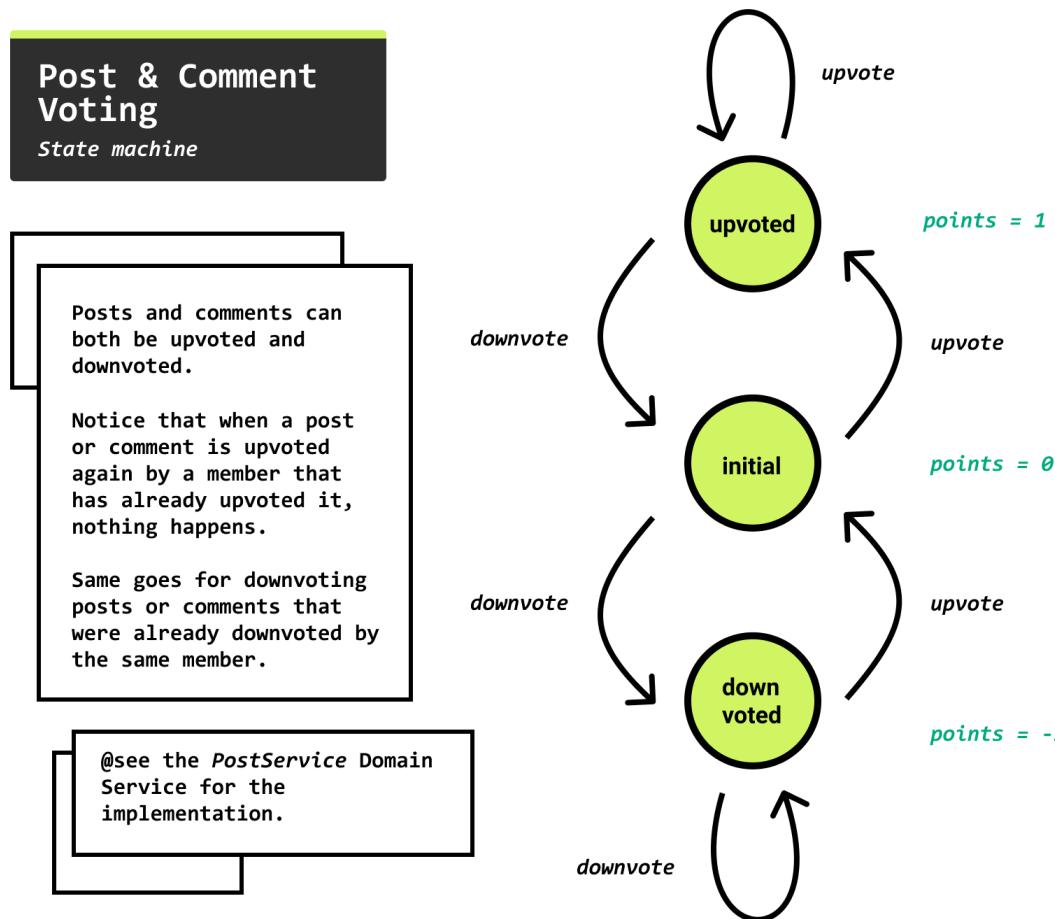
```
# ...
```

```
# Come up with as many scenarios necessary to test that
# the feature works the way it should
```

Using the keywords Feature, Given, When, and Then, we can line up a feature, describe the many scenarios that capture rules and quirks of that feature, specify the inputs, and the outcomes.

Before you write a single line of code, it's usually a good idea to have all of these figured out ahead of time; this is what the TDD-practitioners tell us.

The image below is a **state machine** that describes the potential states a post can be in for any given user: upvoted, initial, and downvoted. Also, notice how the state machine defines the legal *state transitions* (a post can't go directly from upvoted to downvoted).



Post & comment state machine.

State machine: There are many ways to interpret how systems work. One way to interpret how a system works is to consider it a state machine. A state machine is an abstract mental model of a system with several states, but can only be in one state at a time. It also defines from which states, it also graphically de-

scribes how the machine may change states. If you're intrigued, read "The Rise of The State Machines" via Smashing Magazine.

Handling the upvote post request

Let's get into it. Same as before, the request comes into our application through the API, whether it be a GraphQL mutation resolver or a RESTful API controller.

This time we'll use the RESTful API. As always, we inject the controller with the upvotePost *Use Case/Application Service*.

```
// post/useCases/upvotePost/index.ts

// Setting up a controller by passing in the Use Case.

import { UpvotePost } from "./UpvotePost";
import { memberRepo, postRepo, postVotesRepo } from "../../repos";
import { postService } from "../../domain/services";
import { UpvotePostController } from "./UpvotePostController";

const upvotePost = new UpvotePost(memberRepo, postRepo, postVotesRepo, postService);

const upvotePostController = new UpvotePostController(
  upvotePost
)

export { upvotePost, upvotePostController };
```

In the controller, we do the same thing as before. Pass in the relevant *Use Case*, call the *Use Case*'s execute method with the *Input DTO/Command*, then handle the result.

```
// post/useCases/upvotePost/upvotePostController.ts

// UpvotePostController calling execution of the
// upvotePost use case.

import { BaseController } from "../../../../../shared/infra/http/models/BaseController";
import { UpvotePost } from "./UpvotePost";
import { DecodedExpressRequest } from "../../../../../users/infra/http/models/decodedRequest";
import { UpvotePostDTO } from "./UpvotePostDTO";
import { UpvotePostErrors } from "./UpvotePostErrors";
import * as express from 'express'

export class UpvotePostController extends BaseController {
  private useCase: UpvotePost;

  constructor (useCase: UpvotePost) {
    super();
    this.useCase = useCase;
```

```

}

async executeImpl (req: DecodedExpressRequest, res: express.Response): Promise<any> {
  const { userId } = req.decoded;

  const dto: UpvotePostDTO = {
    userId,
    slug: req.body.slug
  }

  try {
    const result = await this.useCase.execute(dto);

    if (result.isLeft()) {
      const error = result.value;

      switch (error.constructor) {
        case UpvotePostErrors.MemberNotFoundError:
        case UpvotePostErrors.PostNotFoundError:
          return this.notFound(res, error.errorValue().message)
        case UpvotePostErrors.AlreadyUpvotedError:
          return this.conflict(error.errorValue().message)
        default:
          return this.fail(res, error.errorValue().message);
      }
    } else {
      return this.ok(res);
    }
  } catch (err) {
    return this.fail(res, err)
  }
}
}

```

Inside the Upvote Post use case

Our first task with the *Use Case/Application Service* is to fetch all of the *Aggregates* and entities necessary for us to upvote a Post.

Given the first scenario from our test specifications (upvoting a post when one has already upvoted it), which *Aggregates* and *entities* are involved?

- The Post to be upvoted needs to be fetched.
- The Member about to do the upvoting - we need that too.
- Any PostUpvotes that were already cast by the Member against this Post.

Let's dependency inject the *Repositories* necessary for us to get these things:

```
// post/useCases/upvotePost/upvotePost.ts

export class UpvotePost implements UseCase<UpvotePostDTO, Promise<UpvotePostResponse>> {
    private memberRepo: IMemberRepo;
    private postRepo: IPostRepo;
    private postVotesRepo: IPostVotesRepo;
    private postService: PostService;

    constructor (
        memberRepo: IMemberRepo,
        postRepo: IPostRepo,
        postVotesRepo: IPostVotesRepo,
        postService: PostService
    ) {
        this.memberRepo = memberRepo;
        this.postRepo = postRepo;
        this.postVotesRepo = postVotesRepo;
        this.postService = postService;
    }

    public async execute (req: UpvotePostDTO): Promise<UpvotePostResponse> {
        let member: Member;
        let post: Post;
        let existingVotesOnPostByMember: PostVote[];

        ...

    }
}
```

Next, we fetch all of these things, throwing the appropriate errors if they're not found.

```
// post/useCases/upvotePost/upvotePost.ts

export class UpvotePost implements UseCase<UpvotePostDTO, Promise<UpvotePostResponse>> {
    private memberRepo: IMemberRepo;
    private postRepo: IPostRepo;
    private postVotesRepo: IPostVotesRepo;
    private postService: PostService;

    constructor (
        memberRepo: IMemberRepo,
        postRepo: IPostRepo,
        postVotesRepo: IPostVotesRepo,
        postService: PostService
```

```

) {

    this.memberRepo = memberRepo;
    this.postRepo = postRepo;
    this.postVotesRepo = postVotesRepo
    this.postService = postService;
}

public async execute (req: UpvotePostDTO): Promise<UpvotePostResponse> {
    let member: Member;
    let post: Post;
    let existingVotesOnPostByMember: PostVote[];

    try {

        try {
            member = await this.memberRepo.getMemberByUserId(req.userId);
        } catch (err) {
            return left(new UpvotePostErrors.MemberNotFoundError())
        }

        try {
            post = await this.postRepo.getPostBySlug(req.slug);
        } catch (err) {
            return left(new UpvotePostErrors.PostNotFoundError(req.slug));
        }

        existingVotesOnPostByMember = await this.postVotesRepo
            .getVotesForPostById(post.postId, member.memberId);

        // implement logic

        ...

        return right(Result.ok<void>())

    } catch (err) {
        return left(new AppError.UnexpectedError(err));
    }
}
}

```

OK, so we've got the Post and Member *Aggregates*, and possibly a PostUpvote.

Question for you, out of all these entities...

Which is responsible for holding the upvote post logic? The Post? The Member?

Should we do `post.upvote(member)` on the Post *Aggregate*? Should we do `member.upvotePost(post)`?

The answer is *neither*. Mentioning the member *Aggregate* from post or vice-versa will break the encapsulation of the Aggregate and also break one of several *Aggregate Design Principles*: an *Aggregate* may only refer to other *Aggregates* by id.

OK, so we've got the Post and Member *Aggregates*, and possibly a PostUpvote.

Aggregate design principles

Here are a few principles popularized in the DDD community about how to design *Aggregates*.

Rule #1 - All transactions happen against Aggregates

The *Aggregate* is responsible for *Command* decision-making logic involving a single entity. Each *Entity* or *Value Object* within an *Aggregate* should be related to that singular purpose: making decisions against *Commands*.

To decide whether we should allow the *Command* transaction to complete, or if we should cancel with a failed `Result<T>` instead requires us to have instances of all the *Entities* and *Value Objects* that belong to this *Aggregate* and play some part in making the decision.

For example, when we want to `editPost`, there are **business rules we need to implement**.

In the Gherkins specification for `editPost`, if a Post is a *link post* and we wish to update it, we can only do so if it doesn't have Comments on it already. See the code below.

```
// forum/domain/post.ts

// Attempting to update the link on a post that already
// has comments will result in a PostSealedError.

export type UpdatePostOrLinkResult = Either<
    EditPostErrors.InvalidPostTypeOperationError |
    EditPostErrors.PostSealedError |
    Result<any>,
    Result<void>
>

export class Post extends AggregateRoot<PostProps> {

    ...

    public updateLink (postLink: PostLink): UpdatePostOrLinkResult {
        if (!this.isLinkPost()) {
            return left(new EditPostErrors.InvalidPostTypeOperationError())
        }

        if (this.hasComments()) {
            return left(new EditPostErrors.PostSealedError())
        }
    }
}
```

```

    }

    const guardResult = Guard.againstNullOrUndefined(postLink, 'postLink');

    if (!guardResult.succeeded) {
        return left(Result.fail<any>(guardResult.message))
    }

    this.props.link = postLink;
    return right(Result.ok<void>());
}
}

```

Rule #2 - Design Aggregates to be as small as possible

We keep the *Aggregates* small because we want to keep our writes fast, and that becomes hard to do if we continually add other entities not necessary for decision-making logic to our *Aggregate*.

If using CQRS, only encapsulate entities and value objects necessary for protecting model invariants.

If we're not using CQRS, encapsulate everything necessary to protect model invariants, but also locate everything required to create views (potentially expensive design).

Rule #3 - You may not alter entities within the aggregate's transaction boundary without going through the aggregate

When we build an *Aggregate*, we clump together all of the related *Entities*, promote one of them to act as the *Aggregate Root*, and we must make all of our transactions by going through the identifier for the *Aggregate Root*.

With this design, it becomes impossible to circumvent important invariants on *Entities* within the *Aggregate*. Each *Entity* within the *Aggregate* is only allowed to change, given the *Aggregate Root* deems it appropriate for it to do so, and it keeps track of those state changes.

Using a Domain Service

When you're unable to locate some domain logic within an *Aggregate* because the *Command* involves several *Entities*, and assigning the task to one of the involved entities would break encapsulation, use a *Domain Service*.

Implementing the Upvote Post logic in a Domain Service

To implement the upvote post logic, pass all of the entities fetched by the application service to the PostService *Domain Service*.

```
// domain/services/postService.ts
```

```

...
export type UpvotePostResponse = Either<
  UpvotePostResponse.MemberNotFoundError |
  UpvotePostResponse.AlreadyUpvotedError |
  UpvotePostResponse.PostNotFoundError |
  AppError.UnexpectedError |
  Result<any>,
  Result<void>
>

export class PostService {
  ...
  public upvotePost (
    post: Post,
    member: Member,
    existingVotesOnPostByMember: PostVote[]
  ): UpvotePostResponse {

    // If already upvoted, do nothing

    // If downvoted, we need to remove the downvote

    // Otherwise, add an upvote

    return right(Result.ok<void>());
  }
}

```

Firstly, if the post was already upvoted, do nothing.

```

// domain/services/postService.ts

...

export class PostService {
  public upvotePost (
    post: Post,
    member: Member,
    existingVotesOnPostByMember: PostVote[]
  ): UpvotePostResponse {

    // If already upvoted, do nothing
    const existingUpvote: PostVote = existingVotesOnPostByMember
      .find((v) => v.isUpvote());

    const upvoteAlreadyExists = !!existingUpvote;
  }
}

```

```

    if (upvoteAlreadyExists) {
        return right(Result.ok<void>());
    }

    // If downvoted, we need to remove the downvote

    // Otherwise, add an upvote

    return right(Result.ok<void>());
}

}

```

If downvoted, remove the downvote.

```

// domain/services/postService.ts

...

// If downvoted, remove the downvote
const existingDownvote: PostVote = existingVotesOnPostByMember
    .find((v) => v.isDownvote());

const downvoteAlreadyExists = !existingDownvote;

if (downvoteAlreadyExists) {

    // Signal that the vote was removed (we'll look into this)
    post.removeVote(existingDownvote);
    return right(Result.ok<void>());
}

...

```

Otherwise, add the upvote to Post and then return with a successful Result<T>.

```

// domain/services/postService.ts

...

// Otherwise, add upvote
const upvoteOrError = PostVote
    .createUpvote(member.memberId, post.postId);

if (upvoteOrError.isFailure) {
    return left(upvoteOrError);
}

```

```

const upvote: PostVote = upvoteOrError.getValue();

// Signal that the vote was added (we'll look at this)
post.addVote(upvote);

return right(Result.ok<void>());
...

```

A couple of notes about the design here:

- **The CQS principle is at play:** Notice that this method doesn't return anything? That's because an operation (methods too) is either a command or a query. This one, `upvotePost`, is a *Command*, so it changes `Post` in some way but doesn't return a value, because that would break the CQS principle that a Command changes the system but returns no value.
- **We're using Dependency Injection without Mocking:** In our pursuits of writing testable code, we often implement Dependency Inversion by referring to abstractions over concrete classes. We've done this in our *Use Case/Application Service* by referring to interfaces of *Repositories* instead of concrete ones. Notice here that we require things that *come from* Repositories, yet we haven't related to a Repository. Some say that mocking is a code smell. Mocking is necessary with interfaces and abstract classes.

Persisting the upvote post operation

When implementing DDD without using *Event Sourcing*, in a transaction, the *Aggregate* gets mutated, and it needs to know *how* it was mutated so that we can issue the correct persistence commands to reflect the way it has changed.

This is one of the disadvantages of not using *Event Sourcing*. In *Event Sourcing*, we persist the **state changes** themselves, whereas with how we're doing it, we update and overwrite the existing state with the new state.

At some point, we must answer the hard questions about persisting *Aggregates* this way.

Signaling relationship changes

How do you implement one-to-one, one-to-many, and many-to-many relationships? How do you signal that a new entity in a collection was created? How do you mark it deleted?

For example, when we do `post.addUpvote(vote)` or `post.removeUpvote(vote)`, we're adding or removing an entity (`PostUpvote`), where a `Post` can have many `PostUpvotes`, from `Post`.

To solve this particular problem, I rolled myself a `WatchedList<T>` base class that keeps track of the initial, new, and deleted items in a collection of entities, like the collection of `PostUpvotes`.

```
// shared/domain/watchedList.ts
```

```

export abstract class WatchedList<T> {

    public currentItems: T[];
    private initial: T[];
    private new: T[];
    private removed: T[];

    /**
     * When we first create a WatchedList<T>, the items passed in
     * initially via the constructor become the initial and current
     * items.
     */

    constructor (initialItems?: T[]) {
        this.currentItems = initialItems ? initialItems : [];
        this.initial = initialItems ? initialItems : [];
        this.new = [];
        this.removed = [];
    }

    abstract compareItems (a: T, b: T): boolean;

    public getItems (): T[] {
        return this.currentItems;
    }

    public getNewItems (): T[] {
        return this.new;
    }

    public getRemovedItems (): T[] {
        return this.removed;
    }

    private isCurrentItem (item: T): boolean {
        return this.currentItems
            .filter((v: T) => this.compareItems(item, v)).length !== 0
    }

    private isNewItem (item: T): boolean {
        return this.new
            .filter((v: T) => this.compareItems(item, v)).length !== 0
    }

    private isRemovedItem (item: T): boolean {
        return this.removed

```

```

    .filter((v: T) => this.compareItems(item, v))
    .length !== 0
}

private removeFromNew (item: T): void {
    this.new = this.new
        .filter((v) => !this.compareItems(v, item));
}

private removeFromCurrent (item: T): void {
    this.currentItems = this.currentItems
        .filter((v) => !this.compareItems(item, v))
}

private removeFromRemoved (item: T): void {
    this.removed = this.removed
        .filter((v) => !this.compareItems(item, v))
}

private wasAddedInitially (item: T): boolean {
    return this.initial
        .filter((v: T) => this.compareItems(item, v))
        .length !== 0
}

public exists (item: T): boolean {
    return this.isCurrentItem(item);
}

public add (item: T): void {
    if (this.isRemovedItem(item)) {
        this.removeFromRemoved(item);
    }

    if (!this.isNewItem(item) && !this.wasAddedInitially(item)) {
        this.new.push(item);
    }

    if (!this.isCurrentItem(item)) {
        this.currentItems.push(item);
    }
}

public remove (item: T): void {
    this.removeFromCurrent(item);
}

```

```

    if (this.isNewItem(item)) {
        this.removeFromNew(item);
        return;
    }

    if (!this.isRemovedItem(item)) {
        this.removed.push(item);
    }
}

```

Subclassing WatchedList<T> gives the new class all the capabilities of the WatchList<T>. For example, I did this very thing with PostVotes, using the PostVote entity as the generic.

```

// forum/domain/postVotes.ts

import { PostVote } from "./postVote";
import { WatchedList } from "../../../../../shared/domain/WatchedList";

export class PostVotes extends WatchedList<PostVote> {
    private constructor (initialVotes: PostVote[]) {
        super(initialVotes)
    }

    public compareItems (a: PostVote, b: PostVote): boolean {
        return a.equals(b)
    }

    public static create (initialVotes?: PostVote[]): PostVotes {
        return new PostVotes(initialVotes ? initialVotes : []);
    }
}

```

From the Post class, instead of referring to the upvotes as PostVote[] like so...

```

// forum/domain/post.ts

export interface PostProps {
    ...
    votes?: PostVote[]
}

```

We refer to an abstraction of the PostVote[] collection.

```

// forum/domain/post.ts
export interface PostProps {
    ...
    votes?: PostVotes; // collection
}

```

Adding a vote to PostVotes can now be done using the add(t: T) method from the WatchedList<T> base class. The abstraction **keeps track of new items added**.

```
// forum/domain/post.ts

export class Post extends AggregateRoot<PostProps> {
  ...
  public addVote (vote: PostVote): Result<void> {
    this.props.votes.add(vote);
    this.addDomainEvent(new PostVotesChanged(this, vote));
    return Result.ok<void>();
  }
}
```

Inside the repository, having received a *dirtied* Post Aggregate, it passes off the post's postVotes to a separate PostVotesRepo sub-repo for persistence.

```
// forum/repo/implementations/sequelizePostRepo.ts

export class SequelizePostRepo implements PostRepo {
  ...

  private savePostVotes (postVotes: PostVotes) {
    return this.postVotesRepo.saveBulk(postVotes);
  }

  public async save (post: Post): Promise<void> {
    const PostModel = this.models.Post;
    const exists = await this.exists(post.postId);
    const isNewPost = !exists;
    const rawSequelizePost = await PostMap.toPersistence(post);

    if (isNewPost) {

      try {
        await PostModel.create(rawSequelizePost);
        await this.saveComments(post.comments);
        await this.savePostVotes(post.getVotes());
      } catch (err) {
        await this.delete(post.postId);
        throw new Error(err.toString())
      }
    } else {
      // Save non-aggregate tables before saving the aggregate
      // so that any domain events on the aggregate get dispatched
    }
  }
}
```

```

        await this.saveComments(post.comments);
        await this.savePostVotes(post.getVotes());

        await PostModel.update(rawSequelizePost, {
            // To make sure your hooks always run, make sure to include this in
            // the query
            individualHooks: true,
            hooks: true,
            where: { post_id: post.postId.id.toString() }
        });
    }
}

```

In PostVotesRepo's `saveBulk` method, we make use of the ability to get all items removed and all new items by calling `votes.getRemovedItems()` and `votes.getNewItem()`.

```

// forum/repos/implementations/sequelizePostVotesRepo.ts

export class PostVotesRepo implements IPostVotesRepo {
    ...

    async save (vote: PostVote): Promise<any> {
        const PostVoteModel = this.models.PostVote;
        const exists = await this.exists(vote.postId, vote.memberId, vote.type);
        const rawSequelizePostVote = PostVoteMap.toPersistence(vote);

        if (!exists) {
            try {
                await PostVoteModel.create(rawSequelizePostVote);
            } catch (err) {
                throw new Error(err.toString());
            }
        } else {
            throw new Error('Invalid state. Votes arent updated.')
        }
    }

    public async delete (vote: PostVote): Promise<any> {
        const PostVoteModel = this.models.PostVote;
        return PostVoteModel.destroy({
            where: {
                post_id: vote.postId.id.toString(),
                member_id: vote.memberId.id.toString()
            }
        })
    }
}

```

```

async saveBulk (votes: PostVotes): Promise<any> {
  for (let vote of votes.getRemovedItems()) {
    await this.delete(vote);
  }

  for (let vote of votes.getNewItemss()) {
    await this.save(vote);
  }
}

...
}

```

The Composite Design Pattern: Wrapping a collection of objects and treating it as if it's a single object. Use this when you need custom logic or statefulness around how and when encapsulated collection changes.

Persisting complex aggregates using database transactions

What were to happen if we were saving an *Aggregate* like Post, and we were able to save only a *part* of it, like the nested PostVote entity? Should the entire transaction fail? Should we rollback?

Yeah, ideally, this is the best option. We don't want to leave our database in an inconsistent state, so we should ensure that if anything fails within the aggregate consistency boundary, the entire transaction fails.

Sequelize and most popular ORM or database adapters for Node.js come with the ability to tie several operations to a single transaction.

Sequelize comes with the ability to use **Unmanaged transactions**, which means that you include a reference to the transaction with every operation. When you've called all your operations, you `commit` the transaction if it was successful, and you `rollback` the transaction if it wasn't.

```

// First, we start a transaction and save it into a variable
const t = await sequelize.transaction();

try {

  // Then, we do some calls passing this transaction as an option:

  const user = await User.create({
    firstName: 'Bart',
    lastName: 'Simpson'
  }, { transaction: t });

  await user.addSibling({

```

```

    firstName: 'Lisa',
    lastName: 'Simpson'
  }, { transaction: t });

  // If the execution reaches this line, no errors were thrown.
  // We commit the transaction.
  await t.commit();

} catch (error) {

  // If the execution reaches this line, an error was thrown.
  // We rollback the transaction.
  await t.rollback();

}

```

The semantics of how transactions work may be different per library, but it should be possible to design a Repository to handle operations against complex aggregates using transactions.

Remind me: I've got it on my list to put together an open-source project demonstrating how to design repositories to use transactions instead.

Feature 3: Get Popular Posts

We know how *Commands*, now let's talk about the other side of the fence: *Queries*.

In CQRS, for any Aggregate, there exists a model for writing (the *Aggregate* itself), and at least one model for reading (the view model).

For example, in the Forum subdomain, we have the Post *Aggregate* as the **write** model. For the **read** model, we just need a plain ol' TypeScript object that contains all the fields required in the Presentation layer.

Read models

When people use an application, they typically have the ability to view data. Sometimes there are several ways to represent that data. Sometimes it depends on the role you are or auth scope you have.

In CQRS, a read model is a simple object intended for the Presentation layer.

There are two ways to model read models.

1. As domain concepts using Value Objects
2. As raw data with a loose shape

Modeling read models as domain concepts

Read models can be thought about as domain concepts, and can be represented as such. We can model them as Value Objects and enforce creation through factory methods.

Here's an example of a PostDetails read model properties interface.

```
// forum/domain/postDetails.ts

interface PostDetailsProps {
  member: MemberDetails;
  slug: PostSlug;
  title: PostTitle;
  type: PostType;
  text?: PostText;
  link?: PostLink;
  numComments: number;
  points: number;
  dateTimePosted: string | Date;
  wasUpvotedByMe: boolean;
  wasDownvotedByMe: boolean;
}
```

And here's what it looks like when we implement the interface as a generic of ValueObject<T>.

```
// forum/domain/postDetails.ts

import { ValueObject } from "../../shared/domain/ValueObject";
import { PostLink } from "./postLink";
import { PostText } from "./postText";
import { PostType } from "./postType";
import { PostTitle } from "./postTitle";
import { PostSlug } from "./postSlug";
import { MemberDetails } from "./memberDetails";
import { Result } from "../../shared/core/Result";
import { IGuardArgument, Guard } from "../../shared/core/Guard";
import { Post } from "./post";

interface PostDetailsProps {
  member: MemberDetails;
  slug: PostSlug;
  title: PostTitle;
  type: PostType;
  text?: PostText;
  link?: PostLink;
  numComments: number;
  points: number;
  dateTimePosted: string | Date;
  wasUpvotedByMe: boolean;
  wasDownvotedByMe: boolean;
}

export class PostDetails extends ValueObject<PostDetailsProps> {
```

```
get member (): MemberDetails {
    return this.props.member;
}

get slug (): PostSlug {
    return this.props.slug;
}

get title (): PostTitle {
    return this.props.title;
}

get postType (): PostType {
    return this.props.type;
}

get text (): PostText {
    return this.props.text;
}

get link (): PostLink {
    return this.props.link;
}

get numComments (): number {
    return this.props.numComments;
}

get points (): number {
    return this.props.points;
}

get dateTimePosted (): string | Date {
    return this.props.dateTimePosted;
}

get wasUpvotedByMe (): boolean {
    return this.props.wasUpvotedByMe;
}

get wasDownvotedByMe (): boolean {
    return this.props.wasDownvotedByMe;
}

private constructor (props: PostDetailsProps) {
    super(props);
```

```

}

public static create (props: PostDetailsProps): Result<PostDetails> {
  const guardArgs: IGuardArgument[] = [
    { argument: props.member, argumentName: 'member' },
    { argument: props.slug, argumentName: 'slug' },
    { argument: props.title, argumentName: 'title' },
    { argument: props.type, argumentName: 'type' },
    { argument: props.numComments, argumentName: 'numComments' },
    { argument: props.points, argumentName: 'points' },
    { argument: props.dateTimePosted, argumentName: 'dateTimePosted' },
  ];

  if (props.type === 'link') {
    guardArgs.push({ argument: props.link, argumentName: 'link' })
  } else {
    guardArgs.push({ argument: props.text, argumentName: 'text' })
  }

  const guardResult = Guard.againstNullOrUndefinedBulk(guardArgs);

  if (!guardResult.succeeded) {
    return Result.fail<PostDetails>(guardResult.message);
  }

  if (!Post.isValidPostType(props.type)) {
    return Result.fail<PostDetails>("Invalid post type provided.")
  }

  return Result.ok<PostDetails>(new PostDetails({
    ...props,
    wasUpvotedByMe: props.wasUpvotedByMe ? props.wasUpvotedByMe : false,
    wasDownvotedByMe: props.wasDownvotedByMe ? props.wasDownvotedByMe : false
  }));
}
}

```

Modeling read models as raw data

Since there's no real reason for us to enforce **model invariants** on *read* operations, one might question — why bother enforcing object creation?

That's a great point.

If you want, you can relax creating read models and construct them by going to your data store directly.

You can create a read model using:

- a raw SQL query
- a method call from your ORM's API
- any other method of retrieving data from your data source.

CQRS

This is one of the great benefits of CQRS. The stack we use for **reads** doesn't have to be the same stack we use for **writes**. They are independent operations and we can scale them separately from each other.

If it's more efficient to use raw SQL queries, that's an option we can take.

In **Event Sourcing**, we usually have at least two databases. The first database is an Event Store, and it saves the *Domain Events* that occur as a result of a successful **command**. Those events are *projected* and the data is written to a second database, which is usually a **relational or object database**. This second database is typically optimized for **reads**. All **queries** are resolved from this second database. This is why sometimes it takes a second for Twitter to update your notifications after someone tweets you — the *read database* has to catch up to the *write database*.

Handling an API request to Get Popular Posts

Similar to last time, we can handle the request through either a RESTful API controller or a GraphQL resolver.

Here's an example of a RESTful API controller that handles the request. The `GetPopularPostsRequestDTO` type describes anything we'd like to use as an input to the `GetPopularPosts` use case.

```
// useCases/getPopularPosts/GetPopularPostsController.ts

export class GetPopularPostsController extends BaseController {
  private useCase: GetPopularPosts;

  constructor (useCase: GetPopularPosts) {
    super();
    this.useCase = useCase;
  }

  async executeImpl (req: DecodedExpressRequest, res: express.Response): Promise<any> {

    const dto: GetPopularPostsRequestDTO = {
      offset: req.query.offset
    }

    try {
      const result = await this.useCase.execute(dto);

      if (result.isLeft()) {
        const error = result.value;
        res.status(500).json({ error: error.message });
      } else {
        res.json(result.value);
      }
    } catch (error) {
      res.status(500).json({ error: 'Internal Server Error' });
    }
  }
}
```

```

        switch (error.constructor) {
            default:
                return this.fail(res, error.errorValue().message);
        }

    } else {
        const postDetails = result.value.getValue();
        return this.ok<GetPopularPostsResponseDTO>(res, {
            posts: postDetails.map((d) => PostDetailsMap.toDTO(d))
        });
    }

} catch (err) {
    return this.fail(res, err)
}
}

}

```

The response type of this DTO looks like the following.

```

// post/dtos/GetPopularPostsResponseDTO.ts

import { PostDTO } from "../../../../../dtos/postDTO";

export interface GetPopularPostsResponseDTO {
    posts: PostDTO[];
}

```

Using a repository to fetch the read models

The repository often contains a variety of service methods that perform different queries against the model.

For example, in the `IPostRepo` interface, we define methods for **getting all recent posts** by calling `getRecentPosts (offset?: number)`, and a similar one for **getting popular posts** as well.

These methods exist solely so that the `Query` use case can do its job.

```

// forum/repos/postRepo.ts

import { Post } from "../domain/post";
import { PostId } from "../domain/postId";
import { PostDetails } from "../domain/postDetails";

export interface IPostRepo {
    getPostDetailsBySlug (slug: string): Promise<PostDetails>;
    getPostBySlug (slug: string): Promise<Post>;
    getRecentPosts (offset?: number): Promise<PostDetails[]>;
}

```

```

    getPopularPosts (offset?: number): Promise<PostDetails[]>;
    getNumberOfCommentsByPostId (postId: PostId | string): Promise<number>;
    getPostByPostId (postId: PostId | string): Promise<Post>;
    exists (postId: PostId): Promise<boolean>;
    save (post: Post): Promise<void>;
    delete (postId: PostId): Promise<void>;
}

```

In each of these methods, you'd implement the database query logic that does what the method says it will do.

Implementing pagination

There are two ways that I know to implement pagination:

- Offset-based
- Cursor-based

In **offset-based** pagination, we pass in the absolute index of all the results for a search, and we only see results from then forward. It's pretty straightforward to implement with basic SQL or to use an ORM's API.

Here's an example of offset-based pagination with Sequelize.

```
// repos/implementations/sequelizePostRepo.ts

export class SequelizePostRepo implements IPostRepo {
  ...
  public async getPopularPosts (offset?: number): Promise<PostDetails[]> {
    const PostModel = this.models.Post;
    const detailsQuery = this.createBaseDetailsQuery();
    detailsQuery.offset = offset ? offset : detailsQuery.offset;
    detailsQuery['order'] = [
      ['points', 'DESC'],
    ];

    const posts = await PostModel.findAll(detailsQuery);
    return posts.map((p) => PostDetailsMap.toDomain(p))
  }
  ...
}
```

In **cursor-based** pagination, we use a cursor to keep track of where the next items should be fetched from. This works by referring to the ID of the last object fetched, and defining the search criteria that led that search.

Where to go from here?

There's so much more to explore! If you're into DDD, I'd recommend:

- Peeking around the DDDForum.com codebase a little bit
- Implementing your Domain-Driven application
- Reading the original Domain-Driven Design book
- Sending me an email if you have questions on how I can help you!

In future revisions of this book, I'm going to include sections on:

- Upgrading to Event Sourcing for scalability
- Adding a cache in order to speed up queries
- Using an external message queue instead of an in-memory implementation

Resources

- The Domain-Driven Design Series @ khalilstemmer.com

References

- Conway's Law. 5 Dec. 2019.
- Jonathan Oliver, and Jonathan Oliver. "DDD: Strategic Design: Core, Supporting, and Generic Subdomains · Jonathan Oliver." Jonathan Oliver, 4 Apr. 2009, <https://blog.jonathanoliver.com/ddd-strategic-design-core-supporting-and-generic-subdomains/>.
- Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software.
- Vernon, V. (2016). Implementing Domain-Driven Design.
- Chapter 7 of Vernon, V. (2016). Domain-Driven Design Distilled.
- How to squash big design up front with Event Sourcing
- Awesome Event Storming
- Transaction Script
- Domain Events by Udi Dahan
- Supporting & Core Subdomains
- Database Per Service Pattern
- "Use Case." Wikipedia, Wikimedia Foundation, 18 Dec. 2019, https://en.wikipedia.org/wiki/Use_case
- "Hexagonal Architecture (Software)." Wikipedia, Wikimedia Foundation, 6 Dec. 2019, [https://en.wikipedia.org/wiki/Hexagonal_architecture_\(software\)](https://en.wikipedia.org/wiki/Hexagonal_architecture_(software))
- "Event Modeling." What Is It?, 23 June 2019, <https://eventmodeling.org/posts/what-is-event-modeling/>.