





*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

<b>About this online book</b>	<b>32</b>
<b>This book is continuously updated</b> . . . . .	33
Downloading a copy . . . . .	33
Timeline . . . . .	33
Updates . . . . .	33
Accessing the book . . . . .	33
Feedback . . . . .	33
 <b>Introduction</b>	 33
My story . . . . .	33
My early research . . . . .	34
Why I wrote this book . . . . .	35
Practical design skills aren't being taught . . . . .	35
Bad design is extremely expensive . . . . .	36
There's a lot to learn from the past . . . . .	36
How I wrote this book . . . . .	37
Two types of wisdom: Sophia & Phronesis . . . . .	37
Sophia . . . . .	37
Phronesis . . . . .	37
This book's approach to wisdom . . . . .	38
How to master software design . . . . .	39
Step 1: Identify the <b>purpose</b> of code . . . . .	39
Step 2: Identify the <b>qualities of code</b> that make it accomplish its purpose well . . . . .	39
Step 3: Master the techniques of those who are doing it well . . . . .	40
Step 4: Develop our own principles by <b>practicing and building experience</b> . . . . .	40
How this book will benefit you . . . . .	40
How this book is organized . . . . .	41
TypeScript . . . . .	41
Resources . . . . .	42
Articles . . . . .	42
 <b>Part I: Up to Speed</b>	 42

1. Complexity & the World of Software Design . . . . .	42
Chapter goals . . . . .	42
The goal of software design . . . . .	42
Complexity . . . . .	42
What is complexity in software? . . . . .	43
Complexity is an incremental phenomenon . . . . .	43
Types of complexity . . . . .	43
Essential complexity . . . . .	44
Accidental complexity . . . . .	45
Causes of accidental complexity . . . . .	45
Getting the requirements wrong . . . . .	45
Dependencies (tight coupling) . . . . .	45
Obscurity (low cohesion) . . . . .	47
More developers . . . . .	48
Language and API capabilities . . . . .	48
Premature optimization . . . . .	49
Other causes . . . . .	49
How to detect complexity . . . . .	49
Ripple . . . . .	49
Cognitive load . . . . .	49
Poor discoverability . . . . .	50
Poor understandability . . . . .	50
Mitigating complexity . . . . .	50
Tactical vs. strategic programming . . . . .	50
Extreme Programming: The original Agile development methodology . . . . .	51
Technical practices . . . . .	51
Values (from Extreme Programming) . . . . .	52
Feedback . . . . .	52
Simplicity . . . . .	54
Communication . . . . .	54
Summary . . . . .	55
Exercises . . . . .	55
References . . . . .	55
Articles . . . . .	55
Books . . . . .	56
Papers . . . . .	56
2. Software Craftsmanship . . . . .	56
Chapter goals . . . . .	56
Professionalism . . . . .	56
A brief history of software development . . . . .	58
Programming picking up speed (50s) . . . . .	59
The software crisis of the 60s-80s . . . . .	60
Dot-com bubble, OOP, and Extreme Programming (1995 – 2001) . . . . .	61
Agile (2001 — today) . . . . .	63
The Agile Manifesto . . . . .	65
The (Misled) Era of Agile . . . . .	65
Why didn't Agile work? . . . . .	66

Software Craftsmanship (2006 — today) . . . . .	67
The Software Craftsmanship Manifesto . . . . .	68
Back to Basics (XP) . . . . .	68
Craftsmanship: Professionalism in software development . . . . .	69
Definition . . . . .	69
Are you a software craftsperson? . . . . .	69
Understanding the manifesto . . . . .	69
<b>Not only working software</b> , but also well-crafted software . . . . .	69
<b>Not only responding to change</b> , but also steadily adding value . .	69
<b>Not only individuals and interactions</b> , but also a community of professionals . . . . .	69
<b>Not only customer collaboration</b> , but also productive partnerships .	70
Craftsmanship principles . . . . .	70
To write well-crafted software... . . . . .	70
To steadily add value... . . . . .	71
Engage in the community... . . . . .	72
Consider yourself a partner... . . . . .	72
How to make habits out of craftsmanship principles . . . . .	73
Summary . . . . .	76
Exercises . . . . .	76
References . . . . .	76
Books . . . . .	76
3. A 5000ft View of Software Design . . . . .	76
Chapter goals . . . . .	76
Levels of design . . . . .	76
The Software Design and Architecture Stack & Roadmap . . . . .	77
Resource: The Stack . . . . .	77
Resource: The Map . . . . .	78
Step 1: Clean Code . . . . .	79
Step 2: Programming paradigms . . . . .	80
Step 3: Object Oriented Programming and Domain-Modeling . . . . .	82
Step 4: Design Principles . . . . .	83
Step 5: Design Patterns . . . . .	84
Design pattern criticisms . . . . .	85
Step 6: Architectural Principles . . . . .	85
Step 7: Architectural Styles . . . . .	86
Structral . . . . .	87
Message-based . . . . .	88
Distributed . . . . .	89
Step 8: Architecture Patterns . . . . .	89
Step 9: Enterprise Patterns . . . . .	90
Summary . . . . .	91
Exercises . . . . .	91
References . . . . .	91
Articles . . . . .	91
Books . . . . .	91

<b>Part II: Humans &amp; Code</b>	<b>91</b>
4. Demystifying Clean Code . . . . .	91
Chapter goals . . . . .	92
What is clean code anyway? . . . . .	92
According to the community . . . . .	92
According to the experts . . . . .	94
Clean coding standards . . . . .	94
What is a coding standard? . . . . .	95
Why do we need coding standards? . . . . .	95
Simple Design . . . . .	96
Emergence . . . . .	98
Structure vs. Developer Experience . . . . .	99
Structure . . . . .	99
Developer experience . . . . .	100
Developer experience for developer tooling companies . . . . .	100
APIs aren't just URLs . . . . .	101
Developer experience is important for all companies . . . . .	101
Balancing structure vs. developer experience . . . . .	103
Structure vs. Developer Experience in Practice: Angular and React .	104
Structure vs. Developer Experience in Practice: Object-Oriented	
vs. Functional Programming . . . . .	104
Conclusion . . . . .	104
Summary . . . . .	105
Exercises . . . . .	105
References . . . . .	105
Articles . . . . .	105
Books . . . . .	105
5. Human-Centered Design For Developers . . . . .	105
Chapter goals . . . . .	106
Human-Centered Design . . . . .	106
What is it? . . . . .	106
How is this helpful for us? . . . . .	107
<b>Developer use cases</b> . . . . .	107
How We Figure Things Out — The Psychology of Human Action . . . . .	108
Discoverability & understanding . . . . .	108
The 7 Stages of Action . . . . .	109
Fundamental Principles of Design . . . . .	113
Knowledge in the Head vs. World . . . . .	114
Knowledge in the Head . . . . .	115
Knowledge in the World . . . . .	116
How is this helpful for us? . . . . .	117
Affordances . . . . .	118
Real-life examples . . . . .	119
Why is this useful? . . . . .	119
Affordances in programming languages . . . . .	122
Affordances in design patterns . . . . .	123
How to do affordances well . . . . .	123

Signifiers . . . . .	I23
Real-life examples . . . . .	I24
Why is this useful? . . . . .	I24
<b>Intentional signifiers in software development</b> . . . . .	I25
<b>Accidental signifiers in software development</b> . . . . .	I25
How to use signifiers well . . . . .	I26
Constraints . . . . .	I26
Real-life examples (physical constraints) . . . . .	I26
Real-life examples (cultural constraints) . . . . .	I27
Real-life examples (semantic constraints) . . . . .	I27
Real-life examples (logical constraints) . . . . .	I27
Constraint examples in software development . . . . .	I27
Thin interfaces . . . . .	I27
Other ways to use constraints well . . . . .	I28
Mapping . . . . .	I28
Real-life examples . . . . .	I28
Why is this useful? . . . . .	I30
How to do mappings well . . . . .	I31
<b>Grouping</b> . . . . .	I31
<b>Proximity</b> . . . . .	I32
<b>Grouping in software development</b> . . . . .	I33
<b>Proximity in software development</b> . . . . .	I34
Feedback . . . . .	I35
Types of errors . . . . .	I35
Real-life examples . . . . .	I36
Why is this useful? . . . . .	I36
Feedback in software development . . . . .	I37
How to do feedback well . . . . .	I38
Conceptual Models . . . . .	I38
Real-life examples . . . . .	I38
Why is this useful? . . . . .	I38
Conceptual models in software development . . . . .	I39
How to do conceptual models well . . . . .	I39
Testing your code for cleanliness . . . . .	I40
<b>Ask: What does my code do?</b> . . . . .	I40
<b>Ask: Find the code that needs to be changed</b> . . . . .	I41
<b>Ask: Change this code without introducing bugs</b> . . . . .	I41
Summary . . . . .	I41
A philosophy for human-friendly code . . . . .	I41
Exercises . . . . .	I42
Consider the fundamentals of interaction when designing software	I42
Learn more about Human-Centered Design . . . . .	I42
Resources . . . . .	I42
Articles . . . . .	I42
Books . . . . .	I42
6. Organizing things . . . . .	I42
Chapter goals . . . . .	I43

Symptoms of poor project structure . . . . .	143
Forgetting what the system does . . . . .	143
Hard to locate features . . . . .	143
Energy spent flipping back and forth between files . . . . .	143
Common project structure approaches . . . . .	143
Just evolve it over time . . . . .	144
Package by infrastructure/technology/type . . . . .	144
Package by domain . . . . .	145
Features (use cases) . . . . .	146
Features are vertical slices . . . . .	146
Features are the entry-point . . . . .	147
Screaming architecture & feature-driven folder structure . . . . .	148
Feature-driven front-end project structure . . . . .	149
Pages . . . . .	149
Project structure . . . . .	153
Feature-driven backend project structure . . . . .	154
Use cases . . . . .	154
Shared content . . . . .	154
Benefits of a feature-driven project organization . . . . .	154
Project organization rules . . . . .	155
Use top-level conventions . . . . .	155
It belongs to a feature or it's shared . . . . .	155
Group files related to a particular feature . . . . .	156
Fight the framework . . . . .	156
Summary . . . . .	156
References . . . . .	157
Articles . . . . .	157
Books . . . . .	157
Exercises . . . . .	157
7. Documentation & Repositories . . . . .	157
Chapter goals . . . . .	158
User goals and questions to address in a repository . . . . .	158
Push complexity downwards . . . . .	158
Tests as documentation . . . . .	158
Summary . . . . .	159
Exercises . . . . .	159
Resources . . . . .	159
Articles . . . . .	159
8. Naming things . . . . .	159
The seven principles of naming . . . . .	159
Naming Principle #1. Consistency & uniqueness . . . . .	160
Consistency in naming . . . . .	161
Consistency with everything . . . . .	161
Uniqueness . . . . .	162
Rule: <b>Avoid using similar words to express the same concept (thesaurus names)</b> . . . . .	163

Rule: Follow programming language and project (naming) coding conventions . . . . .	163
Rule: Avoid very similar variable names by mis-spelling (or using correct, alternate spellings) . . . . .	164
Rule: Don't use the same name to express different concepts from within the same namespace . . . . .	164
Rule: Don't recycle variable names . . . . .	164
Rule: Names should be unique, regardless of the case . . . . .	165
Naming Principle #2: Understandability . . . . .	166
Knowledge in the world . . . . .	166
Representing real-world concepts . . . . .	166
Domain-specific names are a long term investment . . . . .	167
The domain layer describes core business rules . . . . .	167
Naming things in more technical layers . . . . .	169
Using architecture & frameworks to dictate how to name things . . . . .	169
There's a construct for everything . . . . .	169
Rule: Don't randomly capitalize syllables within words . . . . .	170
Rule: Avoiding names with digits . . . . .	170
Rule: Use pronounceable names . . . . .	170
Rule: Use the CQS (Command Query Separation) principle for naming methods . . . . .	170
Rule: Document side effects in methods with several notable side effects . . . . .	171
Rule: Use domain concepts to refer to things from the business . . . . .	171
Rule: Use technical concepts to express technical things . . . . .	171
Rule: Use simple (grammatically correct) English (without spelling errors) . . . . .	171
Rule: Don't omit vowels or unnecessarily abbreviate words . . . . .	171
Rule: Avoid misleading names . . . . .	171
Rule: Avoid using negatives in methods that return boolean . . . . .	171
Rule: Avoid irrelevant names . . . . .	172
Make meaningful distinctions . . . . .	172
Naming Principle #3: Specificity . . . . .	172
Over-specifying . . . . .	173
Multiple variants . . . . .	174
Lacking necessary context . . . . .	174
Under-specifying . . . . .	175
Optional type annotations . . . . .	176
React hooks API . . . . .	176
Rule: Singular / plural naming . . . . .	177
Rule: Avoid referring to things as variables, classes or methods in the name . . . . .	178
Rule: Name people by their roles . . . . .	178
Rule: Specificity against unions . . . . .	178
Rule: Avoid blob parameters . . . . .	178
Rule: Avoid number series parameters . . . . .	179
Rule: Utilize namespaces . . . . .	179

Rule: Use abbreviations sparingly . . . . .	180
Naming Principle #4: Brevity . . . . .	180
Compression . . . . .	180
Context . . . . .	180
The law, reiterated . . . . .	181
Rule: Use concise English . . . . .	182
Rule: Refrain from non-conventional single-letter variables . . . . .	182
Rule: Grouping indicates context . . . . .	182
Rule: The operation and resource name must be in context . . . . .	183
Rule: Don't use unnecessary member prefixes . . . . .	184
Rule: Refactor member prefixes out of objects . . . . .	184
Naming Principle #5: Search-ability . . . . .	185
Rule: Avoid using numeric constants . . . . .	186
Rule: Keep documentation up to date . . . . .	186
Naming Principle #6: Pronounceability . . . . .	186
Rule: Very standard abbreviations don't have to be pronounceable .	186
Rule: Use camel case to signal word breaks in variables . . . . .	186
Rule: Booleans should ask a question or make an assertion . . . . .	187
Rule: Don't omit vowels . . . . .	187
Naming Principle #7: Austerity . . . . .	187
Not everyone has the same sense of humor as you . . . . .	187
Rule: Don't use temporarily relevant concepts . . . . .	187
Rule: Avoid being cute, funny, clever . . . . .	188
Rule: Don't include popular culture references . . . . .	188
Summary . . . . .	188
Exercises . . . . .	188
Resources . . . . .	188
Articles . . . . .	188
Books . . . . .	188
9. Comments . . . . .	188
Code explains what and how, comments explain why . . . . .	188
Comments clutter code . . . . .	191
<b>Turning comments into clear, explanatory, declarative code</b> . .	191
<b>Bad comments</b> . . . . .	192
When to write comments . . . . .	193
Demonstration . . . . .	193
Example: Adding additional context . . . . .	196
Summary . . . . .	196
Exercises . . . . .	196
Resources . . . . .	197
Books . . . . .	197
10. Formatting & Style . . . . .	197
Chapter goals . . . . .	197
Objective readability truths . . . . .	197
Whitespace . . . . .	198
<b>Use obvious spacing rules</b> . . . . .	199
Keep code density low . . . . .	200

Break horizontally when necessary . . . . .	202
Prefer smaller files . . . . .	203
Consistency . . . . .	203
Capitalization . . . . .	203
Consistent whitespace . . . . .	205
Storytelling . . . . .	206
Newspaper Code and the Step-down Principle . . . . .	206
Maintaining a consistent level of abstraction . . . . .	209
Code should descend in abstraction towards lower-level details . . . . .	210
Keeping related methods close to each other . . . . .	211
Enforcing formatting rules with tooling . . . . .	212
ESLint . . . . .	212
Prettier . . . . .	213
Husky . . . . .	214
Summary . . . . .	215
Exercises . . . . .	215
Resources . . . . .	215
Articles . . . . .	215
II. Types . . . . .	215
Chapter goals . . . . .	215
Understanding types . . . . .	215
Static vs. dynamic . . . . .	216
Strong (or strict) vs. weak . . . . .	217
Why statically-typed languages? . . . . .	218
Catch silly mistakes . . . . .	218
Abstraction techniques . . . . .	219
Enforce policy . . . . .	220
Make the implicit, explicit . . . . .	220
Communicate design intent easier . . . . .	222
Non-typed languages don't scale very well . . . . .	224
TypeScript: Our statically-typed language . . . . .	224
The "type" annotation . . . . .	225
Types are optional . . . . .	225
Types in TypeScript . . . . .	225
Convenient Implicit Types . . . . .	225
Explicit Types . . . . .	226
Structural types . . . . .	227
Nominal typing . . . . .	227
Duck typing . . . . .	228
Ambient types . . . . .	230
Basic TypeScript language features . . . . .	231
Primitive types . . . . .	231
Object-oriented programming style in TypeScript . . . . .	232
Classes . . . . .	232
Class inheritance . . . . .	233
Static properties . . . . .	234
Instance variables . . . . .	234

<b>Access Modifiers</b>	235
<b> Readonly Modifier</b>	236
Interfaces	236
Generics	238
Special types	240
Type assertions	240
The “type” keyword	241
Type Aliases	243
Union Type	243
Intersection Type	243
Enum	244
Any	245
Void	246
Inline & Literal Types	246
Type Guards	247
Summary	249
Exercises	249
Resources	249
Articles	249
<b>12. Errors and exceptions</b>	249
Chapter goals	250
Error & exception-handling follies	250
Return null	250
Log and throw	251
Problems with these approaches	252
Understanding errors & exceptions	252
Errors	252
Exceptions	253
A philosophy for error handling	253
Features & use cases	253
One happy path, multiple sad paths	253
Aggregate errors with unions	254
Building error-handling infrastructure	254
What do we want in our error-handling API?	255
Introducing the Either type	255
Modelling the response type	256
Hooking it up	257
Encapsulating error message construction in classes	259
A philosophy for exception handling	261
Dealing with other people’s code	261
Wrap I/O code in a try/catch block	262
<b>Turn exceptions into meaningful errors</b>	262
Deciding when to throw exceptions	264
Throw exceptions when a consumer should be forced to fix the problem	264
Handling nothingness	264
Summary	265
Exercises	265

Resources . . . . .	265
Articles . . . . .	265
Books . . . . .	266
<b>Part III: Phronesis</b>	<b>266</b>
13. Features (use-cases) are the key . . . . .	266
Chapter goals . . . . .	268
Code-first approaches to software development . . . . .	268
Tactical programming . . . . .	268
Imperative programming . . . . .	269
Data, behavior, and namespaces . . . . .	272
Drawbacks of an API-first approach . . . . .	273
A feature/use-case driven philosophy for software design . . . . .	278
Feature = use case . . . . .	279
The anatomy of a feature . . . . .	279
Data . . . . .	279
Behaviour . . . . .	281
Name-spacing . . . . .	283
The lifecycle of a feature . . . . .	284
Discovery . . . . .	284
Planning . . . . .	284
Estimation . . . . .	285
Architecture . . . . .	285
Acceptance tests . . . . .	286
Design & implementation . . . . .	286
How to solve our goals in software design . . . . .	287
Building sources of feedback (mistake proofing) . . . . .	289
Questions . . . . .	290
Isn't this just XP? . . . . .	290
Do I have to do everything here? . . . . .	290
Summary . . . . .	290
References . . . . .	291
Books . . . . .	291
Articles . . . . .	291
14. Planning . . . . .	291
Chapter goals . . . . .	292
The most common reason projects fail . . . . .	292
Bill of rights . . . . .	294
For customers . . . . .	294
For developers . . . . .	294
Stay on script . . . . .	294
Why plan? . . . . .	294
Prioritize — Do the most important stories first . . . . .	295
Coordinate — Keep everyone synced up and in the know . . . . .	295
Recalibrate — Get back on track when we get off track . . . . .	296
How planning works . . . . .	296
I — Scoping a project . . . . .	296

2 — Negotiating the first release plan . . . . .	298
3 — Executing the first plan . . . . .	300
4 — Measuring velocity . . . . .	301
Why this approach works . . . . .	302
Summary . . . . .	302
Exercises . . . . .	303
References . . . . .	303
Articles . . . . .	303
Books . . . . .	303
Papers . . . . .	303
15. Customers . . . . .	303
Chapter goals . . . . .	304
Driving vs. steering . . . . .	304
Who is the customer? . . . . .	305
The person who makes business decisions . . . . .	305
A domain expert . . . . .	305
Must be available for questions . . . . .	305
Responsible for the success or the failure of the project . . . . .	305
Often an entire team . . . . .	306
Locating a customer . . . . .	307
What if you can't find a customer? . . . . .	307
Summary . . . . .	308
Exercises . . . . .	308
References . . . . .	308
Articles . . . . .	308
Papers . . . . .	308
Books . . . . .	308
16. Learning the Domain . . . . .	309
Chapter goals . . . . .	309
Where are we so far? . . . . .	309
Domain-Driven Design . . . . .	310
A shared model . . . . .	310
The domain . . . . .	310
Learning the domain without a shared model . . . . .	311
Benefits of a shared model . . . . .	314
Patterns, principles, practices . . . . .	314
Building a shared model with Event storming . . . . .	315
White Label: An example domain . . . . .	317
1 — Plot the domain events . . . . .	318
Sort the events chronologically . . . . .	320
Push events to the edges . . . . .	320
Benefits to modelling with events . . . . .	322
2 — Identify commands . . . . .	323
3 — Identify aggregates (optional) . . . . .	324
4 — Decomposing to subdomains . . . . .	326
Subdomains . . . . .	326
Context maps — documenting the relationships between subdomains	328

Core, generic, supporting domains . . . . .	329
5 — Utilize the ubiquitous language everywhere . . . . .	330
What about the queries? . . . . .	330
Event Modelling . . . . .	331
Summary . . . . .	333
Domain-Driven Design concepts . . . . .	333
Wrapping up . . . . .	333
References . . . . .	334
Articles . . . . .	334
Books . . . . .	334
17. Stories . . . . .	334
Chapter goals . . . . .	335
User stories . . . . .	335
Developer responsibilities . . . . .	336
Customer responsibilities . . . . .	336
INVEST: Story-writing principles . . . . .	337
I — Stories must be independent . . . . .	337
N — Stories must be negotiable . . . . .	337
V — Stories must provide value . . . . .	337
E — Stories must be estimable . . . . .	338
S — Prefer short stories . . . . .	338
T — Stories must be testable . . . . .	338
Story-writing formats . . . . .	338
Command/query format . . . . .	339
As a [role], I want [feature], so that [value] . . . . .	340
Writing stories for non-functional requirements . . . . .	341
Summary . . . . .	341
Exercises . . . . .	342
References . . . . .	342
Articles . . . . .	342
Books . . . . .	342
18. Estimates & Story Points . . . . .	342
Chapter goals . . . . .	343
About estimates . . . . .	343
Ideal time (effort) . . . . .	343
Story points . . . . .	344
How to make estimates using the recommended story points table . . . . .	345
Refactoring stories . . . . .	345
Merging — when stories can be consolidated . . . . .	345
Splitting — when a story is too big . . . . .	346
Splitting — when a story contains functional and non-functional re- quirements . . . . .	346
Spike — when there's uncertainty . . . . .	347
Team estimation techniques . . . . .	348
Flying fingers . . . . .	348
Planning poker . . . . .	348
Ways to handle non-unanimous estimates . . . . .	348

How to improve estimates . . . . .	349
When to update estimates . . . . .	349
Summary . . . . .	349
References . . . . .	349
Articles . . . . .	349
Books . . . . .	349
Videos . . . . .	350
19. Release Planning . . . . .	350
Chapter goals . . . . .	350
What is release planning?	350
For the customer . . . . .	351
For developers . . . . .	351
Ordering stories . . . . .	352
Iteration size . . . . .	352
Example release plan . . . . .	352
Plan stability . . . . .	353
Events that effect the plan . . . . .	354
Regular rebuilds . . . . .	354
Dealing with bugs . . . . .	354
Planning infrastructure . . . . .	354
Walking the skeleton . . . . .	355
Iteration zero: Zero-functionality iteration . . . . .	355
Pizza party iteration (or mob programming) . . . . .	355
Summary . . . . .	356
Exercises . . . . .	356
References . . . . .	356
Books . . . . .	356
20. Iteration Planning . . . . .	356
Chapter goals . . . . .	357
Iteration Planning Meeting . . . . .	357
Understanding a story . . . . .	357
Listing the tasks . . . . .	358
Technical tasks . . . . .	358
Signing up for tasks . . . . .	358
Keeping track of the iteration plan . . . . .	358
Example iteration plan . . . . .	359
Summary . . . . .	359
Exercises . . . . .	359
Resources . . . . .	359
Articles . . . . .	359
Books . . . . .	360
21. Understanding a Story . . . . .	360
Chapter goals . . . . .	360
Interviewing a domain expert . . . . .	361
Rules for a good interview . . . . .	361
Documenting with pseudocode . . . . .	361
Feature Summary . . . . .	362

Domain Primitives . . . . .	363
Feature Workflow . . . . .	363
Step 1 — Get a high-level understanding of the entire workflow . . . . .	364
Input and output data . . . . .	364
Success states, failure states and dependencies . . . . .	365
Probe edge cases with hypothetical scenarios . . . . .	368
Step 2 — Document the domain primitives (data types) . . . . .	369
Constraints . . . . .	369
Variants & state machines . . . . .	370
Step 3 — Document the steps . . . . .	372
Use case algorithm: The highest level of abstraction . . . . .	372
Sub-steps . . . . .	373
Summary . . . . .	375
Exercises . . . . .	375
References . . . . .	375
Articles . . . . .	375
Books . . . . .	375
22. Acceptance Tests . . . . .	376
Chapter goals . . . . .	376
What is acceptance testing?	376
A way to contractualize and exercise <b>the success and failure scenarios</b> . . . . .	377
Understanding (and appreciating) acceptance tests . . . . .	377
The road to acceptance tests (late 90s until now) . . . . .	377
A division of responsibilities . . . . .	377
Early acceptance test tooling . . . . .	377
Failure to gain ubiquity . . . . .	378
The need for TDD best practices . . . . .	378
BDD (behavior-driven development) . . . . .	379
Given-When-Then (example) . . . . .	380
Writing acceptance tests . . . . .	382
Tooling . . . . .	382
Format #1: Single-line tests . . . . .	383
Format #2: Given-When-Then style Jest tests . . . . .	383
Format #3: Given-When-Then feature tests (preferred) . . . . .	384
Frequently asked questions . . . . .	386
Who does what again? . . . . .	386
Cadence . . . . .	386
Summary . . . . .	387
Exercises . . . . .	387
References . . . . .	387
Articles . . . . .	387
23. Programming Paradigms . . . . .	388
Chapter goals . . . . .	390
Structured programming . . . . .	390
Mathematical proofs . . . . .	391
Putting a stop to GOTOs . . . . .	392

Functional decomposition . . . . .	392
The birth of testing as we know it today . . . . .	392
What's the correct conceptual model? . . . . .	393
Problem #1: Shared mutable state (coupling) . . . . .	394
Problem #2: Cyclomatic complexity . . . . .	394
Problem #3: Unsafe polymorphism . . . . .	398
Object-oriented . . . . .	400
The main idea: A web of objects . . . . .	400
Emergence . . . . .	403
Sequence, selection, iteration, and indirection . . . . .	406
How does OO solve the problems structured programming faced? .	407
Inheritance — how to mess up doing OO . . . . .	409
How to do OO properly . . . . .	410
Functional . . . . .	4II
A completely different conceptual model . . . . .	4II
Function basics . . . . .	4I2
Not the same sequence, selection, iteration and indirection . . . . .	4I4
Composition . . . . .	4I5
Using composition to build real-world systems . . . . .	4I6
Why functional programming? . . . . .	4I7
Design principles are paradigm-agnostic . . . . .	4I8
Choosing a programming paradigm . . . . .	4I9
Summary . . . . .	4I9
Exercises . . . . .	420
Resources . . . . .	420
Articles . . . . .	420
Books . . . . .	420
Videos . . . . .	420
24. An Object-Oriented Architecture . . . . .	420
Chapter goals . . . . .	422
Deciding on an architecture . . . . .	422
1. Use the requirements . . . . .	423
2. Consult design principles . . . . .	425
3. Consult architectural principles . . . . .	426
4. Consult architectural patterns . . . . .	427
5. Draw it out on a whiteboard . . . . .	428
Improving your architecture skills . . . . .	428
An object-oriented architecture for business applications . . . . .	428
MVC: The trivial layered architecture . . . . .	429
What's the problem with MVC? . . . . .	430
You can't unit test infrastructure . . . . .	43I
Core code vs. infrastructure code . . . . .	434
How requests work in an OO architecture: Transaction scripts vs. domain models . . . . .	435
Dependency inversion & the dependency rule . . . . .	437
Why decouple core from infrastructure code again? . . . . .	440
Deployment as a modular monolith . . . . .	440

Summary . . . . .	441
References . . . . .	441
Articles . . . . .	441
25. Testing Strategies . . . . .	441
Chapter goals . . . . .	442
Test types . . . . .	442
Unit . . . . .	442
Integration . . . . .	443
End-to-end . . . . .	444
Acceptance . . . . .	445
Concerns: What do we need to test? . . . . .	447
Features (queries) . . . . .	447
Features (commands) . . . . .	447
Core code . . . . .	447
Infrastructure . . . . .	448
Database (infra) . . . . .	448
<b>GraphQL or RESTful API (infra)</b> . . . . .	448
Shared infrastructural components (infra) . . . . .	448
External APIs and integrations (infra) . . . . .	449
Testing strategy for a decoupled application . . . . .	449
Unit test all domain layer code . . . . .	449
Use case tests: acceptance test application layer features as unit tests	450
Integration test input adapters . . . . .	451
Integration test outgoing adapters . . . . .	452
End-to-end test for queries and additional confidence . . . . .	453
Summary . . . . .	453
References . . . . .	454
Books . . . . .	454
Articles . . . . .	454
26. The Walking Skeleton . . . . .	454
Chapter goals . . . . .	455
<b>What is a walking skeleton?</b> . . . . .	455
Why end-to-end? . . . . .	456
Exposing deployment and connection issues early . . . . .	456
What do you mean by production-like environment? . . . . .	457
Why automated builds? . . . . .	457
What do we do about the components not covered by end-to-end test? . . . . .	458
High-level demonstration (example) . . . . .	459
Small steps . . . . .	460
Choose the simplest scenario(s) . . . . .	460
A failing end-to-end test . . . . .	461
Use Page Objects to write declarative E2E tests . . . . .	462
Fetching the non-existent spots . . . . .	464
Backend: the bare minimum . . . . .	465
Summary . . . . .	466
References . . . . .	466
Articles . . . . .	466

Books . . . . .	466
Videos . . . . .	466
<b>27. Pair Programming . . . . .</b>	<b>466</b>
Chapter goals . . . . .	467
What is it? . . . . .	467
Roles . . . . .	468
Switching techniques . . . . .	468
Using a clock . . . . .	468
Ping pong/popcorn . . . . .	468
Strong-style pairing . . . . .	468
What's the output of a pair programming session? . . . . .	469
How often should you pair program? . . . . .	469
Considerations . . . . .	469
What a good pair looks like . . . . .	470
Summary . . . . .	470
Resources . . . . .	470
Articles . . . . .	470
Books . . . . .	470
<b>28. Test-Driven Development Workflow . . . . .</b>	<b>471</b>
Chapter goals . . . . .	471
Test-Driven Development rules & considerations . . . . .	471
Red-Green-Refactor . . . . .	471
Don't get ahead of yourself (the three laws of TDD) . . . . .	472
Have your tests running alongside you as you code . . . . .	473
<b>Committing after a passing test</b> . . . . .	474
TDD Prerequisites . . . . .	474
Reasonable testing strategy thought out . . . . .	474
Create a build-test-deploy test architecture . . . . .	474
Separate scripts for acceptance/unit, integration, and E2E tests . . . . .	475
Can seamlessly switch between environments . . . . .	475
Why real-world TDD is so hard . . . . .	475
Different TDD challenges in front-end vs. back-end development . . . . .	475
As an industry, we struggle to agree on testing terminology . . . . .	476
Many fail to recognize that TDD starts with architecture . . . . .	476
Understanding what to test (and how to test it) is hard . . . . .	476
Double Loop TDD & the two TDD schools of thought . . . . .	477
<b>Classic (Inside-Out/Chicago) and Mockist (Outside-In/London)</b>	
<b>TDD</b> . . . . .	477
Filling in gaps: Adding E2E and Integration Tests . . . . .	478
Integration tests . . . . .	478
E2E tests . . . . .	480
Refactoring is where design happens . . . . .	481
Summary . . . . .	482
In conclusion of Phronesis . . . . .	482
References . . . . .	482
Articles . . . . .	482

<b>Part IV: Test-Driven Development Basics</b>	<b>483</b>
29. Getting Started with Classic Test-Driven Development . . . . .	483
Chapter goals . . . . .	483
About Classic TDD . . . . .	483
How is Classic TDD different from Mockist TDD? . . . . .	483
<b>Why start with Classic TDD?</b> . . . . .	484
TDD is mostly a design tool . . . . .	484
You must do the examples . . . . .	484
The core TDD mechanics (recap) . . . . .	485
Design happens during refactoring . . . . .	485
The rule of three . . . . .	486
How to push the design forward with TDD . . . . .	487
Making a test pass (red to green) . . . . .	487
Writing the next test (green to red) . . . . .	488
Triangulation . . . . .	488
Naming tests . . . . .	489
Basic testing pattern . . . . .	489
Write your tests based on behavior, not implementation . . . . .	490
Prefer concrete examples instead of abstract statements . . . . .	491
One example per test . . . . .	491
Exercises . . . . .	492
Palindrome (partial demonstration) . . . . .	492
Fizz Buzz . . . . .	501
Nth Fibonacci . . . . .	502
Summary . . . . .	502
TDD considerations so far . . . . .	502
Current test-driven workflow and design rules . . . . .	502
References . . . . .	504
Articles . . . . .	504
Books . . . . .	504
30. Working Backwards using Arrange-Act-Assert . . . . .	504
Chapter goals . . . . .	505
Three likely challenges you'll encounter . . . . .	505
<b>Arrange-Act-Assert</b> . . . . .	505
Benefits of organizing tests into act-arrange-assert . . . . .	506
Do I have to do this? . . . . .	506
Keep it clean — comments not necessary . . . . .	506
Writing your tests backwards . . . . .	506
Exercises . . . . .	507
Stats calculator . . . . .	507
Password validator . . . . .	508
Summary . . . . .	508
TDD considerations so far . . . . .	508
Current test-driven workflow and design rules . . . . .	508
Resources . . . . .	5II
Articles . . . . .	5II
Books . . . . .	5II

<b>31. Avoiding Impasses with the Transformation Priority Premise . . . . .</b>	<b>511</b>
Chapter goals . . . . .	512
Transformations . . . . .	512
Revisiting how we go from red to green . . . . .	513
Problems with obvious implementation . . . . .	513
<b>Transformation Priority Premise (TPP) . . . . .</b>	<b>514</b>
TPP table . . . . .	514
How this works — constraint upon the obvious implementation . .	514
TPP transformations . . . . .	514
Applying TPP to the Fibonacci exercise . . . . .	517
<b>Exercises . . . . .</b>	<b>518</b>
Fizz Buzz . . . . .	518
Recently Used List . . . . .	518
Tennis . . . . .	519
<b>Summary . . . . .</b>	<b>519</b>
TDD considerations so far . . . . .	519
Current test-driven workflow and design rules . . . . .	520
What's next? . . . . .	522
<b>References . . . . .</b>	<b>522</b>
Articles . . . . .	522
Books . . . . .	523
<b>Part V: Object-Oriented Design With Tests . . . . .</b>	<b>523</b>
<b>32. Why &amp; How to Learn Object-Oriented Software Design . . . . .</b>	<b>523</b>
Chapter goals . . . . .	525
Current challenge: design . . . . .	525
Tests aren't design . . . . .	525
Introducing object design . . . . .	526
Analysis . . . . .	527
Non-functional requirements: evaluation vs. execution qualities .	528
Techniques for analysis . . . . .	528
Design . . . . .	529
Responsibility Driven Design . . . . .	531
Object design is about closing the gap . . . . .	533
You'll do a lot of design outside of the editor (CRC cards, whiteboards, etc) . . . . .	535
Programming . . . . .	536
Guide your code with tests . . . . .	537
Why object design? It's foundational to our continued journey towards mas- tery . . . . .	538
How to learn object design: The Object Design Mastery Framework . . . . .	538
Phase 1: Awareness . . . . .	538
Phase 2: Quality stewardship . . . . .	539
Phase 3: Test dexterity . . . . .	539
Phase 4: Pattern recognition . . . . .	540
Phase 5: Naturalization . . . . .	540
Phase 6: System architecture . . . . .	541

Summary . . . . .	541
Resources . . . . .	542
Books . . . . .	542
Articles . . . . .	542
33. Responsibility Driven Design 101 - Rough Notes . . . . .	542
Introduction . . . . .	542
Chapter goals . . . . .	543
Object basics . . . . .	543
Object capabilities . . . . .	544
Objects come and go, filling in roles . . . . .	545
Object models violate real-world physics . . . . .	546
Core design concepts . . . . .	548
Responsibilities . . . . .	549
Roles . . . . .	552
Collaborations . . . . .	552
Object stereotypes . . . . .	554
1. The Information Holder . . . . .	554
2. The Structurer . . . . .	555
3. The Service Provider . . . . .	556
4. The Coordinator . . . . .	557
5. The Controller . . . . .	558
6. The Interfacer . . . . .	559
Considerations on stereotypes . . . . .	560
From design to code . . . . .	562
Contracts . . . . .	563
Contract technology . . . . .	563
Concretions . . . . .	564
Classes . . . . .	565
Classes as object factories . . . . .	566
Classes as collaborators . . . . .	566
Fulfilling a contract . . . . .	567
Composition . . . . .	569
Interfaces . . . . .	569
Growing neighborhoods . . . . .	570
Object neighbourhoods . . . . .	570
Components . . . . .	571
Designing control between neighborhoods . . . . .	573
Control center . . . . .	573
Control styles . . . . .	573
1. Centralized . . . . .	573
2. Delegated . . . . .	575
3. Dispersed . . . . .	575
Patterns . . . . .	577
Architecture . . . . .	577
Filling in the gaps . . . . .	577
Frameworks . . . . .	578
Libraries/packages . . . . .	578

Exercises . . . . .	578
Summary . . . . .	579
Resources . . . . .	581
<b>34. The RDD Process By Example - Rough Notes . . . . .</b>	<b>582</b>
Introduction . . . . .	582
Chapter goals . . . . .	582
How to find objects . . . . .	582
How to find responsibilities . . . . .	582
How to identify collaborations . . . . .	582
How do you test code? When does that happen? At which point in the process? . . . . .	583
If you're not sure where to start . . . . .	583
If you get stuck . . . . .	583
If it's hard to test . . . . .	583
Resources . . . . .	583
<b>Show techniques for this . . . . .</b>	<b>585</b>
<b>List out other ways to find responsibilities (from the RDD book) . . . . .</b>	<b>586</b>
<b>35. Mapping Concepts &amp; Designs to Object-Oriented Code - Rough Notes . . . . .</b>	<b>586</b>
Introduction . . . . .	587
Chapter goals . . . . .	588
The Four Principles of Object-Oriented Programming . . . . .	589
Principle #1 — Abstraction . . . . .	589
Mapping roles . . . . .	589
Classes . . . . .	589
Static factory method . . . . .	594
Scope . . . . .	595
Interfaces & abstract classes . . . . .	595
Mapping responsibilities . . . . .	596
Constructors . . . . .	596
State . . . . .	596
Methods & messages . . . . .	596
Generalization (generics) . . . . .	596
Mapping collaborations . . . . .	596
Composition . . . . .	596
Dependencies . . . . .	596
Object-oriented programming principles . . . . .	598
<b>Why is it important to know this stuff? . . . . .</b>	<b>599</b>
Abstraction . . . . .	599
Inheritance . . . . .	599
Encapsulation . . . . .	599
Polymorphism . . . . .	599
Summary . . . . .	600
<b>33. Demystifying The Fundamental Concepts of Classic OO . . . . .</b>	<b>603</b>
<b>36. Better Objects with Object Calisthenics - Rough Notes . . . . .</b>	<b>604</b>
Chapter goals . . . . .	604
Design . . . . .	604
Object calisthenics . . . . .	605
The rules . . . . .	605

1 — Only one level of indentation per method . . . . .	605
What to do? . . . . .	606
Benefits . . . . .	607
2 — Don't use the ELSE keyword . . . . .	608
What to do? . . . . .	608
Promotes a main execution lane with few special cases. • Suggests polymorphism to handle complex conditional cases, making the code more explicit (for example, using the State Pattern). • Try using the Null Object pattern to express that a result has no value. . . . .	609
3 — Wrap all primitives and strings . . . . .	610
4 — First class collections . . . . .	610
5 — One dot per line . . . . .	612
6 — Don't abbreviate . . . . .	613
7 — Keep all entities small . . . . .	613
8 — No classes with more than two instance variables . . . . .	613
9 — No getters, setters or properties . . . . .	615
Exercises . . . . .	617
Summary . . . . .	617
Moving forward . . . . .	617
References . . . . .	617
Articles . . . . .	617
37. Refactoring - Rough Notes . . . . .	618
Chapter goals . . . . .	618
About refactoring . . . . .	618
What refactoring is not . . . . .	618
When should we refactor? . . . . .	618
Types of refactoring operations . . . . .	619
Refactoring principles . . . . .	619
Humans first, design second . . . . .	619
Stay in the green . . . . .	619
Use the method template . . . . .	620
A good method template . . . . .	620
precondition checks . . . . .	620
Categories of refactorings . . . . .	621
Rename method . . . . .	622
A plan for applying the most common refactorings . . . . .	623
Start with cleanup . . . . .	623
Extract things . . . . .	624
Rename things . . . . .	624
Create abstractions . . . . .	624
How to master refactoring . . . . .	625
Exercises . . . . .	625
Summary . . . . .	625
References . . . . .	625
Books . . . . .	626
Websites . . . . .	626
38. Detecting Code Smells & Anti-Patterns - Rough Notes . . . . .	626

Chapter goals . . . . .	626
Understanding complexity, code smells, and anti-patterns . . . . .	626
Symptoms of complexity . . . . .	626
Code smells . . . . .	626
Anti-patterns . . . . .	626
The five categories of code smells . . . . .	627
Bloated . . . . .	627
Object-orientation abusers . . . . .	627
Change preventers . . . . .	627
Dispensables . . . . .	627
Couples . . . . .	627
Using object calisthenics to prevent code smells . . . . .	627
Demonstration . . . . .	627
Violation consequences . . . . .	627
Use refactoring techniques to fix code smells . . . . .	628
Data clumps (bloater) . . . . .	628
Switch statements (object-orientation abuser) . . . . .	628
Shotgun surgery (change preventer) . . . . .	628
Comments (dispensable) . . . . .	628
Inappropriate intimacy (coupler) . . . . .	628
The five most common code smells . . . . .	628
1 - Duplication . . . . .	629
2 - Long method . . . . .	629
3 - Large class . . . . .	629
4 - Long parameter list . . . . .	629
5 - Primitive obsession . . . . .	629
6 - Feature envy . . . . .	629
7 - Message chains . . . . .	629
Summary . . . . .	629
Exercises . . . . .	629
Tic Tac Toe (with code smells) . . . . .	629
References . . . . .	629
Articles . . . . .	629
Books . . . . .	629
Web . . . . .	629
39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion -	
Rough Notes . . . . .	630
Chapter goals . . . . .	630
Where are we so far? . . . . .	630
Core code is pure, infrastructure is not . . . . .	630
Test core code with unit tests . . . . .	631
Dependency inversion revisited . . . . .	631
Common core-infrastructure coupling scenarios . . . . .	631
Databases: Dealing with persistence . . . . .	631
Network: Communicating over the network using an API . . . . .	632
Statefulness: Relying on existing test state . . . . .	633
System APIs: Using the system clock . . . . .	635

Environment: Relying directly on environment variables . . . . .	635
Demonstration . . . . .	636
Examples . . . . .	636
Summary . . . . .	636
Picture? Where we are now . . . . .	636
Resources . . . . .	636
Articles . . . . .	636
40. Using Test Doubles - Rough Notes . . . . .	636
Chapter goals . . . . .	637
What's missing in our feature verification? . . . . .	637
Verifying state . . . . .	637
Verifying behavior . . . . .	637
Classic verifies state, mockist verifies behavior . . . . .	637
Test doubles . . . . .	638
Behavior (commands & queries) . . . . .	639
Guidelines . . . . .	639
Mocks . . . . .	640
Spies . . . . .	641
Stubs . . . . .	641
Fakes . . . . .	641
Dummies . . . . .	641
Exercises . . . . .	641
Demonstration . . . . .	641
Summary . . . . .	642
Show where we are . . . . .	642
Testing objects in a layered architecture . . . . .	642
References . . . . .	642
Articles . . . . .	642
41. Testing Legacy Code - Rough Notes . . . . .	642
<b>Part VI: Design Patterns</b>	<b>643</b>
<b>Part VII: Design Principles</b>	<b>643</b>
Coupling, cohesion & connascense . . . . .	643
SOLID . . . . .	643
Single Responsibility Principle . . . . .	643
Open-Closed Principle (OCP) . . . . .	643
Liskov Substitution Principle (LSP) . . . . .	643
Interface Segregation Principle . . . . .	643
Dependency Inversion Principle (DIP) . . . . .	643
Terminology . . . . .	644
Components . . . . .	644
Dependency Injection . . . . .	644
Dependency Inversion . . . . .	647
Using a mock object . . . . .	648
The primary wins of Dependency Inversion . . . . .	649
Inversion of Control & IoC Containers . . . . .	650

Recollection . . . . .	651
GRASP . . . . .	651
The Four Elements of Simple Design . . . . .	651
Chapter goals . . . . .	651
The four elements of simple design . . . . .	651
Runs all the tests . . . . .	651
No duplication . . . . .	651
Maximizes clarity . . . . .	651
Minimizes the # of elements . . . . .	652
Other simple design definitions . . . . .	652
Intention . . . . .	652
Acceptance Test-Driven . . . . .	653
Coupling & cohesion . . . . .	653
Minimal . . . . .	653
Diving into design . . . . .	653
Shifting from technique to design . . . . .	653
Coupling & cohesion is the north star of design . . . . .	655
Summary . . . . .	655
Conclusion . . . . .	656
Resources . . . . .	656
Programming by Wishful Thinking . . . . .	656
The Least Principles . . . . .	656
Principle of Least Effort . . . . .	656
Principle of Least Astonishment (Surprise) . . . . .	656
Law of Demeter (Principle of Least Knowledge) . . . . .	656
Design by Contract (DBC) . . . . .	656
Separation of Concerns . . . . .	656
CQS (Command Query Separation) . . . . .	656
YAGNI . . . . .	656
KISS (Keep It Simple, Silly) . . . . .	656
DRY, WET, Rule of Three . . . . .	656
Composition over inheritance . . . . .	656
Aim for shallow class hierarchies . . . . .	656
Encapsulate what varies . . . . .	656
Program to interfaces, not to implementations . . . . .	656
Relationship to Ports and Adapters architecture . . . . .	656
Relationship to Dependency Inversion Principle . . . . .	656
The Hollywood Principle . . . . .	657
All software is composition . . . . .	657
Design patterns are complexity . . . . .	657
Know of them, but know when you need them . . . . .	657
Separation of Concerns . . . . .	657
Example: overloaded controller . . . . .	657
Separation of concerns . . . . .	660
Strive for loose coupling between objects that interact . . . . .	661
Principle of Least Resistance . . . . .	661
Tell, Don't Ask . . . . .	661

No And's, Or's, or But's . . . . .	661
<b>Part VIII: Architecture Essentials</b>	<b>661</b>
8. Architectural Principles . . . . .	661
Contracts . . . . .	662
Component principles . . . . .	662
Reuse-Release Equivalence Principle . . . . .	662
Common closure principle (CCP) . . . . .	662
The Common Reuse Principle (CRP) . . . . .	662
Stable Dependency Principle . . . . .	662
Volatile Components . . . . .	662
Conway's Law . . . . .	663
The Dependency Rule . . . . .	663
Boundaries . . . . .	663
Cross-cutting concern . . . . .	663
The Principles of Economics . . . . .	663
The Principle of Opportunity Cost . . . . .	663
The Principle of Last Responsible Moment . . . . .	663
Separation of Concerns . . . . .	663
9. Architectural Styles . . . . .	663
Structural . . . . .	663
Component-based architectures . . . . .	663
Layered Architectures . . . . .	663
Monolithic architectures . . . . .	663
Message-based . . . . .	663
Event-Driven architectures . . . . .	663
Publish-Subscribe architectures . . . . .	663
Distributed . . . . .	664
Client-server architectures . . . . .	664
Peer-to-peer architectures . . . . .	664
10. Architectural Patterns . . . . .	664
Layered (n-tier) architecture . . . . .	664
Layers . . . . .	664
Domain layer . . . . .	664
Application layer . . . . .	664
Infrastructure layer . . . . .	664
Adapter layer . . . . .	664
Similar architectures . . . . .	664
Ports & Adapters . . . . .	665
Vertical-slice architecture . . . . .	665
Event sourcing . . . . .	665
Microkernels . . . . .	665
Microservices . . . . .	665
Space-based . . . . .	665
<b>Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS</b>	<b>665</b>

II. Building a Real-World DDD app . . . . .	665
About this chapter . . . . .	665
Chapter goals . . . . .	666
Domain-Driven Design . . . . .	666
Ubiquitous Language . . . . .	667
Implementing DDD & ensuring domain model purity . . . . .	667
DDD addresses the shortcomings of MVC . . . . .	668
Slim (Logic-less) Models . . . . .	669
Pick your object-modeling poison . . . . .	670
Concerns of the unspecified layer in MVC . . . . .	672
Undesirable side-effects with a lack of a domain model . . . . .	674
Model behavior and shape . . . . .	674
Technical Benefits . . . . .	674
Technical Drawbacks . . . . .	675
Alternatives to DDD . . . . .	675
DDD Building Blocks . . . . .	675
Entities . . . . .	676
Value Objects . . . . .	676
Aggregates . . . . .	677
Domain Services . . . . .	677
Repositories . . . . .	677
Factories . . . . .	677
Domain Events . . . . .	678
Architectural concepts . . . . .	679
Subdomains . . . . .	679
Bounded Contexts . . . . .	682
The Project: DDDForum — a Hackernews-inspired forum app . . . . .	687
About the project . . . . .	687
The code . . . . .	694
How to plan a new project . . . . .	694
Imperative design . . . . .	695
Imperative design approaches are for small, simple CRUD applications . . . . .	697
Dimensions that influence the design approach we should take . . . . .	698
Use-case driven design . . . . .	698
Use cases & actors . . . . .	699
Applications are groupings of use cases . . . . .	699
A use case is a command or a query . . . . .	700
Use case artifacts . . . . .	700
Functional requirements document business logic . . . . .	700
Parallels with API-first design . . . . .	702
Steps to implement use case design . . . . .	702
Planning with UML Use Case Diagrams . . . . .	703
1 — Identifying the actors . . . . .	703
2 — Identifying the actor goals . . . . .	704
3 — Identifying the systems we need to create . . . . .	704
4 — Identifying the use cases for each role . . . . .	706

Roles, boundaries, and Conway's Law in Use Case Design . . . . .	709
Summary on use case diagrams . . . . .	718
Event Storming . . . . .	718
Why we need event storming . . . . .	720
How to conduct an event storming session . . . . .	721
Event Modeling . . . . .	733
Building DDDForum . . . . .	735
Project architecture . . . . .	735
<b>Decision 1: We're going to use Domain-Driven Design patterns</b> . . . . .	735
<b>Decision 2: We're going to use a Layered Architecture</b> . . . . .	736
<b>Decision 3: We're going to deploy a Modular Monolith</b> . . . . .	736
<b>Decision 4: We're going to use CQRS (Command Query Response Segregation)</b> . . . . .	737
<b>Decision 5: We're not going to use Event Sourcing</b> . . . . .	737
Starting with the domain models . . . . .	738
Modeling a User Aggregate . . . . .	739
Emitting Domain Events from a User Aggregate . . . . .	743
Writing Domain Events . . . . .	744
Building a Domain Events Subject . . . . .	745
Marking an Aggregate that just created Domain Events . . . . .	749
How to signal that the transaction completed . . . . .	750
How to register a handler to a Domain Event? . . . . .	752
Who dictates when a transaction is complete? . . . . .	753
Feature 1: Creating a Member . . . . .	755
Issuing an API request . . . . .	756
Application Services/Use Cases . . . . .	758
Inside the CreateUser use case transaction . . . . .	766
Feature 2: Upvote a post . . . . .	783
Understanding voting domain logic . . . . .	783
Handling the upvote post request . . . . .	785
Inside the Upvote Post use case . . . . .	786
Aggregate design principles . . . . .	789
Using a Domain Service . . . . .	790
Persisting the upvote post operation . . . . .	793
Feature 3: Get Popular Posts . . . . .	800
Read models . . . . .	800
Handling an API request to Get Popular Posts . . . . .	804
Using a repository to fetch the read models . . . . .	805
Implementing pagination . . . . .	806
Where to go from here? . . . . .	806
Resources . . . . .	807
References . . . . .	807
Context Mapping a Modular Monolith . . . . .	807
Shared Kernel pattern for shared infrastructure . . . . .	807
Partnership pattern for subdomain communication . . . . .	808
Open-Host Service pattern . . . . .	808

Anticorruption Layer for communicating with external models . . .	808
<b>Patterns . . . . .</b>	<b>808</b>
Use Case . . . . .	809
Repository . . . . .	809
Domain model . . . . .	809
Transaction Script . . . . .	809
Domain Event . . . . .	809
Transactional Outbox . . . . .	809
Unit of Work . . . . .	809
Aggregates . . . . .	809
Commands . . . . .	809
Queries . . . . .	809
Value objects . . . . .	809
Entities . . . . .	809
Event handlers . . . . .	809
Clock (time) . . . . .	809
<b>Part X: Advanced Test-Driven Development . . . . .</b>	<b>810</b>
Test against behaviour, not implementation . . . . .	810
Use Case Testing . . . . .	810
End-To-End Testing Principles . . . . .	810
Notes . . . . .	810
Resources . . . . .	810
Articles . . . . .	8II
Building an End-to-End Testing Rig . . . . .	8II
Stable End-to-End tests with the Page Object Pattern . . . . .	8II
Testing Managed vs. Unmanaged Dependencies . . . . .	8I2
Integration Testing Input Adapters: GraphQL, REST, CLI . . . . .	8I2
Unit Testing Principles . . . . .	8I2
Acceptance Testing Principles . . . . .	8I2
Dealing with Legacy Code . . . . .	8I2
<b>Part XI: Above and Beyond . . . . .</b>	<b>8I2</b>
DevOps . . . . .	8I2
Testing Cloud Infrastructure . . . . .	8I2
<b>Conclusion . . . . .</b>	<b>8I2</b>

## About this online book

Thanks for reading this book! It's currently being worked on by me and shaped into something awesome (with the feedback from the book's pre-sale buyers and reviewers).

## This book is continuously updated

I believe it was Julian Shapiro that said books are a “poor medium for education and discussion.” They quickly become outdated, contain filler to reach word counts, and it’s hard to link supplementary content that might be valuable for learning.

That’s one of the main reasons why I’d like the primary source of this book to be this online, living, breathing artifact.

If you **bought this book**, you were emailed the current version with **lifetime access** and the ability to download and read it in a variety of formats: here (web), PDF, and EPUB.

## Downloading a copy

To download the most recent version of the book, go to the downloads page at [wiki.solidbook.io/downloads](https://wiki.solidbook.io/downloads) and choose between PDF or EPUB.

## Timeline

**Release timeline** — You can keep on track of the timeline for the book in the updates page. I intend to finish writing solidbook.io by August 2021.

## Updates

**Receiving updates** — I’ll send you an email every time there’s an update. You can find updates on the **updates page** on the site too. You can also follow me on Twitter for updates.

## Accessing the book

**Reading this book online** — To read this book online, use the *magic link* that was sent to your email. Need to resend the link to yourself? No problem, just head to [wiki.solidbook.io](https://wiki.solidbook.io) and enter the email that you bought the book with. You should get an email containing your link in a few moments.

## Feedback

**Submitting feedback** — There are several ways to submit feedback. Use the button, send me an email, or reach out to me on Twitter.

*The best way to send me feedback is to write an email to [khalil@khalilstemmler.com](mailto:khalil@khalilstemmler.com) or DM me on Twitter.*

## Introduction

### My story

It all started when my interviewer asked me, “How would you design your business-logic layer?”

I'm sitting there in the middle of a dimly lit room, in front of three senior software developers, (none of them looking very pleasant I might add), hoping that not a single one of them catches a glimpse of the bead of sweat I just felt roll down the side of my head.

Business-logic layer... That's practically another language to me. I hadn't the slightest idea of how to answer it.

After a few seconds of deliberation, I started to say words. I probably said something about MVC (model-view-controller), how I organize my folders, and maybe something about how I structure my controllers in a Node.js & Express.js project.

Every word leaving my mouth was an extra bit of dirt flung into the hole I was digging for myself. As I was speaking, I was realizing more and more that nothing I said had anything to do with "business-logic".

Eventually, the sense came over me to stop speaking.

I was then met with a *very* uncomfortable couple of seconds of silence. After a head nod from one of the fellows and some glances at each other, one interviewer said "OK, I think that'll be about it. Thank you, do you have any questions for us?"

*Aaaaaaab-solutely not, I'll see myself out* — said my brain. Trying to erase that moment from my memory as soon as possible, I probably asked "So, what's the culture like here?"

Safe to say, I didn't get the job. My recruiter even dropped me. Thinking back to what I did to prepare myself for the interview, I remember studying:

- Whatever I found off of the "how to prepare for a full-stack web developer interview" Google search
- The algorithms in *Cracking the Coding Interview*
- Javascript and all its silly language quirks like closures, IIFEs, and passing by reference vs. value.

Not only that, but I had been working on building my own startup company during the time I was in school, and I thought that all that code I wrote would have translated into some serious experience.

I may have completely bombed that interview, but looking back today — that was one of the best things that could have happened to me. It was that moment that I realized there was an entire world of software design and architecture that I **needed** to teach *myself*. I became incredibly interested in this topic. I was going to learn. And I wasn't going to fail the next interview.

## My early research

Mid-2018, in the middle of my research, I really needed to pay the bills, so I ended up landing a job as a Frontend Consultant. I was mostly looking for a low-stress job I could perform to hack my career 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 code.

Every evening after work for 8 months, I read books and wrote code. I reviewed everything they taught me in school, but this time, learning from the experts' books.

I started with Object-Oriented Programming. What were the 4 principles of OOP again? What *does* an abstract class do anyway? Why would you ever want to use a static method? I heard that inheritance was a bad thing, right? Why should I avoid that?

As I was learning, I wrote down all of the concepts that I've **heard of but never really cared to try to understand** like POJOs, dependency injection, dependency inversion, inversion of control, concrete classes, design patterns, and principles, etc.

I revisited ideas that I *knew about* but never fully understood in depth like coupling, cohesion, managing dependencies, and separation of concerns. Soon, I was learning a lot of new things like the hexagonal architecture, Conway's law, Use-case driven development, 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. I quit my job, refactored Univjobs' codebase using Domain-Driven Design practices, and began to regularly share what I was learning about software design and architecture with my peers online on my personal blog.

Today, thousands of readers every month are learning how to write **testable, flexible, and maintainable** code @ [khalilstemmler.com](http://khalilstemmler.com).

## Why I wrote this book

Two years later, after about seventy-or-so blog posts, ten thousand newsletter subscribers, and the dissolution of Univjobs, I started collecting emails for a Domain-Driven Design course. As the most popular series topic on the blog, my plan was to teach developers how to use this technique to build high-quality software.

As I started to learn more about who was interested in this course, I realized that the skills gap in software design skills was *deep*.

## Practical design skills aren't being taught

The truth is, *not a lot has changed about the fundamentals of software design* over the past 60 years, but there's a **huge lack of training** on it.

By and large — junior and intermediate developers are still left confused and with a lack of guidance on how to confidently and consistently develop high-quality software.

From simple stuff like how to structure your project or name things well so that things can be understood, found and changed, to how to decide on an architectural style and *organize your business logic*, we're leaving developers out to dry.

## No one is teaching developers essential software design and architecture fundamentals

I tend to agree with Eric Elliot, who says that "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 a 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".

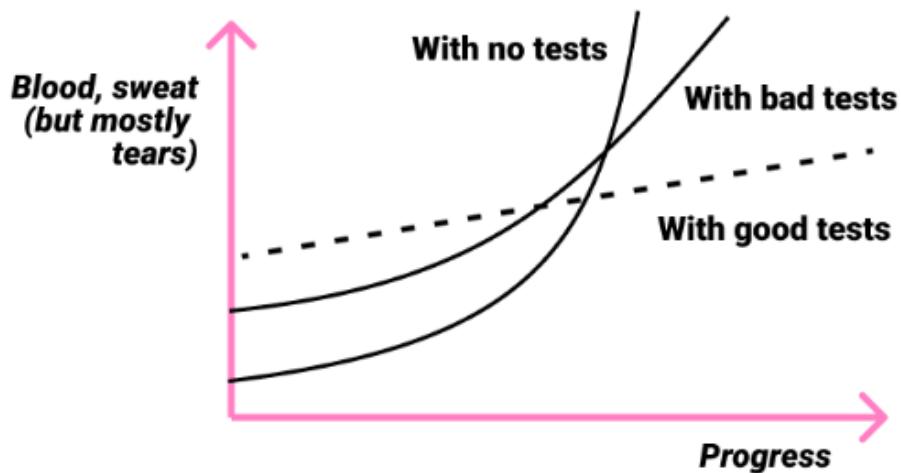
### Bad design is extremely expensive

Bad design has some seriously costly consequences. According to CNBC, in September 2018:

Approximately \$85 billion is spent yearly dealing with bad code.

That blew me away. I wrote an article about this remarkable phenomenon and why I think that's happening, but the executive summary of where I think this comes from is:

- Developers aren't being taught **essential software design skills**
- Most companies *say they practice* Agile
- *Actually* practicing Agile means tight feedback loops and the ability to change and refactor 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**
- Therefore, productivity plummets over time



How projects with good tests, bad tests, and no tests progress over time. Without tests (or with bad tests), it's hard to continue to safely make progress because new changes introduce regressions. Refactoring without tests is a great way to introduce regressions.

### There's a lot to learn from the past

Every so few years, a popular technology is released on top of previous ideas from the past. Redux is the observer pattern. GraphQL is architectural dependency inversion.

Any JavaScript developer from the 2010s knows how frustrating it can be to need to stay up to date on the latest libraries and today's best way to handle state, etc — only to have new best practices and tools emerge a few months later.

As an industry, we're very quick to forget our roots and reinvent the wheel.

Eventually, it got to a point where I just decided I wasn't going to chase the newest tools and technologies anymore. I decided I was going to learn the things that the new tools and technologies are influenced by and built from.

It's Santayana's curse after all:

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

## How I wrote this book

### Two types of wisdom: Sophia & Phronesis

Aristotle was one of the most influential teachers in history. He wrote about physics, logic, metaphysics, ethics, music, politics, linguistics and more. Aristotle exerted a unique influence on almost every form of knowledge in the West.

In his book, *The Nichomachean Ethics*, he distinguishes between two different types of wisdom: *sophia*, meaning “factual wisdom” \*\*and *phronesis*, meaning \*\*\*practical wisdom”.

#### Sophia

Ever heard of a topic called *philosophy*? The originating Greek word, *philo*\*-*sophia*\*, stands for “love of wisdom”.

Sophia is the type of knowledge concerned with the structurally teachable, composable, logical, factual knowledge that we often equate to science.

For example, a book titled “1000 Facts About Sharks” is likely going to be a \*sophia-\*sided book.

\*Sophia-\*sided programming books (and documentation) are generally good for learning new languages, platforms, and frameworks. They teach you new facts and **show you what you can do**.

#### Phronesis

*Phronesis*, which is practical wisdom — is wisdom acquired through experience.

This type of wisdom is more concerned with practical action, and how best to act in scenarios that call for it.

Traditionally, this type of knowledge is harder to teach. It's typically taught through mentorships and hands-on learning from the *experts themselves*, but often, the experts write books and create assets that learners can take and lead themselves faster down the path to acquiring practical wisdom.

\*Phronesis-\*sided programming books are the influential, timeless, classic books that withstand software trends. These are the books that **show you what you should do**.

## This book's approach to wisdom

An early version of this book was more within the vein of *sophia*-sided wisdom and followed a *first-principles* approach to mastering software design. The approach relied upon the idea that if we could understand how the low-level topics in design worked, we could compose those facts into larger pieces that made sense. In Chapter 3, we examine this in more detail, looking at how design patterns at the class-level can also be applied at the larger architectural level.

While this is a neat way to learn *facts* about software design, it's not going to help us master the practical act of writing quality code.

Instead, **we're using a method developed by Aristotle over 2000 years ago for being successful at anything (and everything) in life.**

Here's how it works.

Step 1: Identify the **purpose** of the thing we want to use. Knowing the purpose of something allows us to evaluate the ways we can use it.

- e.g. knives are for cutting

Step 2: Identify the **qualities** that make it accomplish its purpose well.

- e.g. if knives are for cutting, a sharp knife is a **good** knife, because it cuts **well**.

Step 3: Mimic **others** who accomplish the goal well

- e.g. we **find people who are good** at cutting and **observe them**.

Step 4: Develop our own principles through **practice**

We figure out the principles guiding their success.

- e.g. we **learn what makes them good** at cutting.

This might be the halfway point between two extremes.

- e.g. to cut something, the knife **must be neither too long nor too short**. (*middle point*)

It might also be a minimum or a maximum.

- e.g. to cut something, the knife **must be sharp enough**. (*minimum*)
- e.g. to cut something, the knife **must not be too** \*\*heavy. (*maximum*)

After enough experience, we build habits and we master what we're practicing.

- e.g. we **practice** cutting **until** cutting **well** is **easy and pleasurable**.

That's the approach.

Essentially, we do what the pros are doing and accumulate our own set of principles through experience. This gives us the ability to determine the right principles and correct amount thereof for any given situation.

## How to master software design

Let's apply Aristotle's method to the act **writing code** to learn how to master software design.

### Step 1: Identify the purpose of code

What is the purpose of code? I hear all sorts of arguments for code being an art, science, math, and so on.

It certainly depends on who we ask, doesn't it? People write code for different reasons. Interaction designers make generative art, data scientists practice exploratory coding, and I'm sure mathematicians have their own use cases as well. Even musicians code!

Let's consider the purpose of code as an art for a moment.

- e.g. code is for creating art

If that's the case, what qualities would be necessary for code to be considered *good art*? Perhaps code needs to be clever, interesting, engaging, unique, inspiring, polarizing, or evokes an emotion (sadness, nostalgia, love, angst, anguish). These are all potential qualities for good art (well, at least the weird stuff that I'm into).

It's clear that we could go down this route (and that would definitely be fun), but it's not in line with what the majority of us are using code for.

For the majority of us that work as skilled tradespeople for hire, the **purpose of code is to create products for customers**.

- e.g code is for creating products for customers

### Step 2: Identify the qualities of code that make it accomplish its purpose well

If we're using code to create products for customers, what are some qualities of *good* code written to serve this purpose?

Code has two consumers: **customers** and **the developers that maintain it**. Therefore, some of the essential qualities to accomplish its purpose well are:

- **Serves the needs of the customer:** The product has requirements — functional and non-functional. Are we meeting them?
- **Flexibility:** We want to be able to add new features by merely adding new code, and changing as little existing code as possible.
- **Testability:** When requirements change and we have to add new features, can we continue to do so without breaking existing features? To stay productive, we need tests. Tests have a lot of benefits, but the important ones to list here are that they prove business correctness, prevent regressions, and give us the confidence to safely refactor.
- **Maintainability:** In actuality, there are many more code qualities that we could list, like consistency, number of elements, clarity, simplicity — but ultimately, it is our ability to *understand* code that most effects its maintainability. Why is this so important? Well, imagine you're a customer with a budget. Now imagine that you're paying for time and materials (meaning — you're paying for the spent by the developers

working on the project). How long does it take the developers to produce a new feature that adds value to the project? A week? Two weeks? A month? Two months? That's a lot of time. And time is money. Code needs to be designed in a way such that it can be cost-effectively changed.

Therefore,

■ **The goal of code:** Code is for creating products for customers. Testable, flexible, and maintainable code that serves the needs of the users is **good** because it can be cost-effectively changed by developers.

### Step 3: Master the techniques of those who are doing it well

If *phronesis* is “practical wisdom”, then the *phronimos* are the wise developers; the ones with practical experience building software with the qualities that we want.

Among my personal *phronimos*-list are the following developers:

Martin Fowler, Robert C. Martin, Kent Beck, Rebecca Wirfs-Brock, John Ousterhout, Scott Wlaschin, Vaughn Vernon, Steve Freeman, Matthias Noback, Greg Young, Eric Evans, Dan North, Gregor Hohpe, Udi Dahan, Gerard Meszaros, Kevin Henney, Michael Feathers, Pedro Moreira Santos, Vladimir Khorikov, Marco Consolaro, Alessandro Di Gioia, Thomas Pierrain, Adam Dymitruk, and Kent C. Dodds.

The next step is to read their books and blog posts, watch their talks, take their courses, hands-on trainings, and chat with them on Twitter to learn their techniques and philosophies for software design.

### Step 4: Develop our own principles by practicing and building experience

Lastly, *practice*. Follow the techniques that the pros use and master them. Come to your own conclusions and develop principles on when to use them based on your own experience.

Decide to break the rules once you've mastered them.

## How this book will benefit you

This book is for any developer that feels like their code gets worse instead of better over time. What you're reading is a handbook that teaches **professional software developers** the **essential software design and architecture best practices** they didn't teach you in school.

This book is for:

- Bootcamp grads, junior, intermediate, and even senior developers — basically, any developer that wants to learn to master software design.
- Developers frustrated with writing buggy code and having things break over time.
- Developers interested in the battle-tested techniques to confidently, repeatedly, and reliably ship code without introducing regressions.
- Developers who want to learn how to design large-scale applications.
- Developers that want to learn how to write clean, **flexible, testable, and maintainable software**.

- Developers who care about shipping quality code and want to learn how to write code that can actually be tested.

This book has two goals. The first is show you the common *unknown unknowns* of software design and architecture from clean code to micro-services and from object design to Domain-Driven Design. The second part is to present you with the most influential ideas and **practical, actionable wisdom** from the phronimos — our teachers— on how write testable, flexible, maintainable code.

Depending on your experience, some of these ideas might be new to you, but in reality, very little of these topics and techniques are *new*. The vast majority of what you'll read are techniques that have been around for a long time. It is, however, likely you haven't been exposed to some of them or merely haven't been shown how to do them thoroughly.

What I'm offering is a synthesis of techniques from a number of developers much wiser than I. If you find something particularly clever or handy, I have to credit the experience of my predecessors. If you find an error, you can assume that I'm at fault.

## How this book is organized

I've broken this book up into several parts.

Parts I through III gets you up to speed on the unknown unknowns of software design, teaches you how to make more *human-friendly* design decisions, and summarizes all the habits and techniques that experts are using to take a software project from inside the customers' heads to deployed in production.

Parts IV to VII focuses on object-oriented programming mechanics, design principles, patterns, and how to incrementally ship simple, testable code using Test-Driven Development.

In VIII and IX, we shift our focus to architecture, learning different architectures to help better inform the walking skeleton we decide on. It's here that we learn how to construct a bullet-proof web application architecture using Domain-Driven Design, Hexagonal Architecture, and CQRS.

In the final chapters of the book, we discuss advanced testing topics like mocking, BDD, and techniques to get the best return on investment from your testing architecture on backend *and* frontend applications. We wrap up by briefly discussing how software scales in the real world, both from a traffic and people perspective.

Some chapters contains exercises to test your understanding and build experience with the techniques we cover. We'll use a mixture of reflections questions, coding katas, and projects to keep you progressing.

At the end of each chapter, I also include links to references so that if you want to go deeper by reading the books, blogs, papers, and watching the videos that informed the chapter.

## TypeScript

The language we use for examples in this book is TypeScript. If you're not too familiar with it, don't worry, I'll teach you what you need to know as we go along.

■ **Get started with TypeScript:** To get download TypeScript and set up a project, visit “How to Setup a TypeScript + Node.js Project”.

## Resources

### Articles

- <https://howtobeastoic.wordpress.com/2016/09/20/sophia-vs-phronesis-two-conceptions-of-wisdom/>
- <https://en.wikipedia.org/wiki/Phronesis>
- [https://medium.com/@\\_ericelliott/if-training-is-not-realistic-youre-in-the-wrong-industry-32e488b864ad](https://medium.com/@_ericelliott/if-training-is-not-realistic-youre-in-the-wrong-industry-32e488b864ad)

## Part I: Up to Speed

■ Programming is the art of telling another human what we want the computer to do. More specifically, as *paid* tradespeople, our mission is to write testable, flexible, and maintainable code. How do we do this well? The last 40 years have introduced a number of influential movements (XP, Agile, Craftsmanship), programming techniques (TDD, Pair programming, CI/CD), and relied on the discovery of patterns and architecture styles (like DDD, layered architectures, etc) to develop testable, flexible, and maintainable software. Let's get up to speed on the world of software design.

### I. Complexity & the World of Software Design

Complexity is the main challenge of software design; it cripples codebases and makes them difficult to maintain. We are in search of simple, battle-tested, repeatable ways to conquer complexity.

#### Chapter goals

- Understand complexity and how it finds its way into software
- Learn how to recognize complexity in code
- Learn practical techniques to mitigate and overcome complexity in software projects

#### The goal of software design

In the Introduction, we learned that, as paid software developers, the goal of software design is:

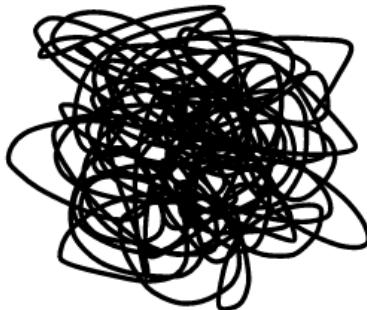
To build products that **serve the needs of the customer** and can be **cost-effectively changed by developers**.

The enemy standing in our way to accomplishing this is *complexity*.

#### Complexity

The default state of the world is complexity. We see this everywhere. Think about how little effort it takes for your room to get messy, for the dishes to pile up, or for your floors to get

nasty over time. It seems to be that when you exert the *least* amount of energy, the world tends towards complexity — or as physicists like to call it, *entropy*: randomness, uncertainty, or a state of disorder.



Entropy was first used in the study of thermodynamics but to us — software developers — entropy is at the heart of our greatest limitation as software developers. To understand.

Simply put: if a system is too complex for us to understand, we can't maintain it.

### **What is complexity in software?**

Complexity is anything that makes the system hard to **understand and modify**.

While it may have once been easy to make changes to the codebase towards the start of the project, over time, complexity strangles our codebase — hurting our ability to understand it, and suddenly, it is now much harder to make changes. New features take longer to build; they require more effort, energy, and *money*.

### **Complexity is an incremental phenomenon**

If we do nothing but add new features, even despite our best efforts, the potential complexity of a system increases incrementally.

Without the proper maintainability structures in place, more code means there are more things to understand, more ways it can break, and more things to account for when adding or changing features.

### **Types of complexity**

There are two types of complexity we need to concern ourselves with: essential complexity (complexity that we *can't* control) and accidental complexity (complexity that we *can* control).

## Essential complexity

Essential complexity is the type of complexity we can't do anything about. This is the type of complexity that's a part of the very nature of the system.

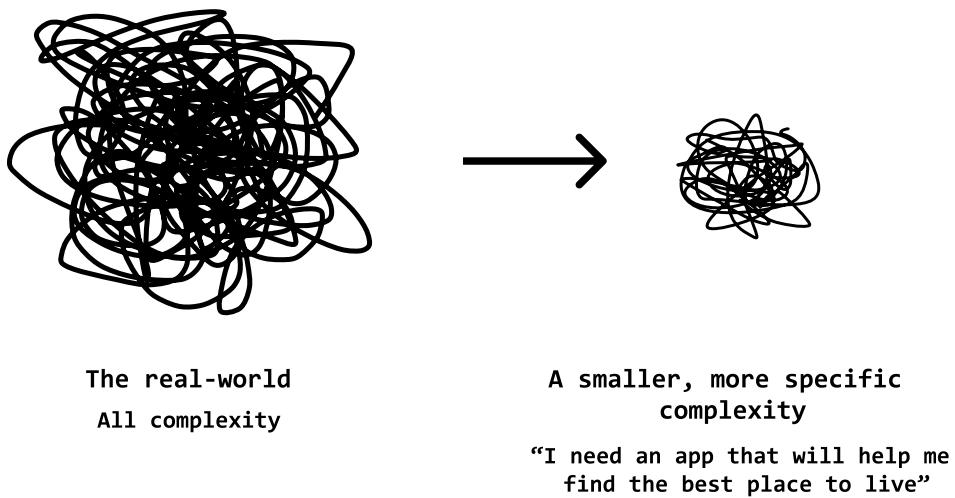
To better understand essential complexity, we can examine the *real world*.

The *real world* is a patchwork of systems. To apply for a loan, improve your friendships, make more money, or marry your partner, there are systems we follow.

Some of these systems are very well-known and standardized like checking in for your dentist appointment. Other systems are less well-known and albeit probably more subjective, like making friends in a new city.

When we're tasked with the challenge to **develop a product for a customer**, what we're doing is **improving a real-world system by developing an automated or better way to get to the desired end result**.

For example, if you were hired to build an app that "helps people find the best place to live", then you're potentially improving the existing system that involves browsing the internet for hours looking at rentals for condos, apartments, and houses in various cities.



A smaller (yet still complex) complexity of the world.

For our example rental application, the *essential complexity* that we're adopting is, in total:

- the features (use cases)
  - eg. sign up, create profile, search, apply to rental
- the data (state) necessary to fulfill the user stories
  - eg. "When displaying a rental, users need to see the square footage, number of rooms, and pictures of the rooms."
- the behaviour (business logic and application rules) that governs the data

- eg. “If I set the ideal rental range in my profile to 50KM, I don’t want to be recommended places outside of that range.”
- and ignoring UI cosmetics (you know, to make things look beautiful), semantically, what we need to present to the user
  - eg. “After logging in, I should see my dashboard with the latest rental recommendations.”

If these features are what the customer is asking for, and we’re sure that each of these features is necessary for the user to accomplish their goal, then we have the *essential complexity* of the application.

**This is the bare minimum complexity. It’s the floor. It’s declarative. This is what we need to build, but not how it’s built.**

## Accidental complexity

Accidental complexity is introduced as soon as we start to build the system. You can think of accidental complexity as **virtually any other complexity that is not the essential complexity**.

This is the *implementation* space. It’s the imperative.

Every time we make a design decision, whether it’s to create a class here, use a function there, loop over this here — we’re introducing some sort of complexity.

In reality, there is *some* accidental complexity that **can be avoided** — like duplication, dead code, and outdated comments. But there are also other subtle forms of complexity that are so foundational to the nature of modern object-oriented programming like *state* and *control* that, unless we adopt a purely functional approach to software design, we’ll find it very hard to ever completely remove.

Purely functional approaches exist, and we discuss them in 23. Programming Paradigms but there are always going to be tradeoffs. It’s what Fred Brooks said in 1987, “there are no silver bullets”.

## Causes of accidental complexity

### Getting the requirements wrong

One of the first possible ways to introduce *accidental complexity* is to just get the requirements wrong. There’s what we *think* we need to build, and then there’s what the customer actually wants. Failing to properly turn conversations with the customer into both *functional* and *non-functional requirements* is a recipe for disaster.

### Dependencies (tight coupling)

One of the first things we learn how to do when we start to program is breaking the problem down into smaller pieces. This is **decomposition**. Decomposition is great because instead of focusing on the entire problem, we can laser-focus our efforts on smaller chunks (components).

It's a great technique, but since software doesn't do a whole lot until we connect the pieces together, we're forced to wire things together. Depending on the strategies we use when we do this, we may introduce **concrete dependencies**.

A *dependency* is a component needed for another component to work.

Take the following TypeScript code for example:

```
class UserService {  
    constructor (  
        private db: MySqlConnection,  
        private logger: Logger  
    ) {}  
  
    createUser () { ... }  
}
```

To create a `UserService`, we need to pass in both `db` and `logger` as dependencies (concrete dependencies, to be more specific).

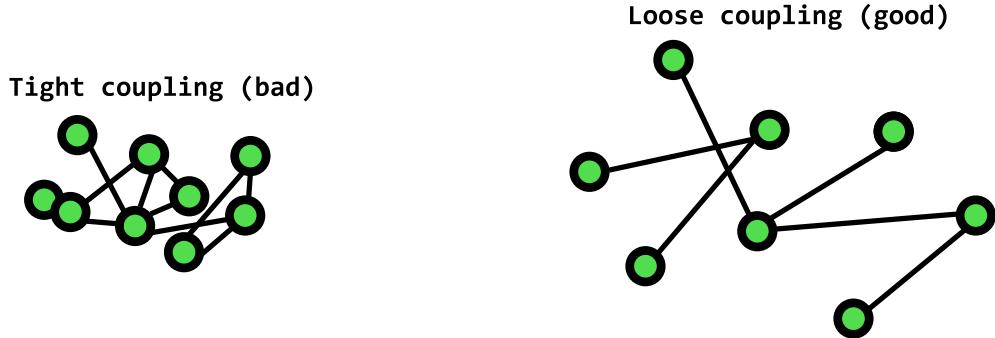
■ **Concretions:** A concrete object is a literal object created using the `new` keyword. The alternative to concretions are abstractions (interfaces, abstract classes, types).

This may not seem like such a bad thing, but as our codebase grows in size, we really need to rely on tests to ensure that things still function the way we want them to.

In the example above, there is no way to test a `UserService` in isolation. To test a `UserService` means we need to create and pass in both a `MySqlConnection` and a `Logger`. If your `MySqlConnection` object connects to a real, live database, then it'll make your tests slow. And if your `Logger` object prints a lot of noise to the console, then your tests will be especially chatty and hard to read.

Because we can't test our `UserService` in isolation, we can say it's tightly *coupled*.

■ **Coupling:** Dependencies are a form of *coupling*. Coupling is a measure of how intertwined two components are. A component could refer to a routine, method, function, class, module, or even an entire architectural component like the database or web app. We need to understand how components collaborate with each other and strive for *loose* coupling between them.



There are other forms of coupling as well. To name a few, there is:

- **Subclass coupling** — When a child class is connected to the parent.
  - eg.: SomeClass extends BaseClass
- **Temporal coupling** — When two actions are bundled together into one module because they happen at the same time.
  - eg.: Since after we perform CreateUser, we need to perform the SendAccountVerificationEmail operation, we'll just put them both in the same CreateUserAndSendEmail function and call that function instead.
- **Dynamic coupling** — When the order that things execute in are important.
  - eg.:

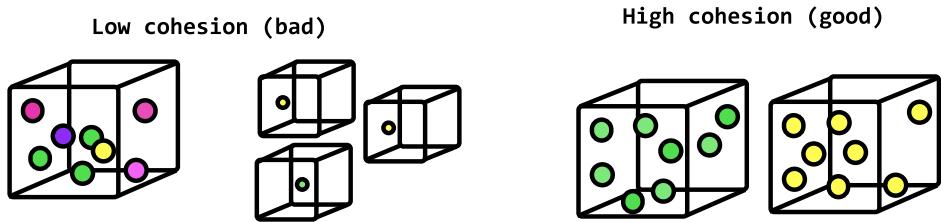
```
email = Email()
email.setRecipient("foo@example.comp")
email.setSender("me@mydomain.com")
email.send() // This would be wrong to do at this time
email.setSubject("Hello World")
```

We can't avoid coupling altogether, but we **can** strive for *loose* coupling as a way to minimize complexity.

### **Obscurity (low cohesion)**

If we're looking at some code and we're finding that it's too hard to understand *what the code does*, or *what we need to know about the code in question*, then it's said to be *obscure* and to have **low cohesion**.

■ **Cohesion:** Cohesion is a measure of how related components are within a particular module. It's a measure of how much they belong together. Do you have stuff in here mostly focused on the topic at hand? Or are there constructs in here that probably belong elsewhere?



Examples of obscurity and low cohesion are:

- **Useless information** — comments that don't provide anything of value to the reader
- **Poorly named variables, methods, classes, etc** — perhaps they say one thing but do another, they're not written using terms from the domain, they're overly terse or ambiguous
- **Poor packaging** — when related things aren't packaged together
- **Incorrect (or confusing) abstractions** — it could be the case that the abstraction you're creating is not readily understandable to the next person.
- **Unnecessary (early) abstractions** — it could also be the case that you create an abstraction too early when it's not yet needed — this has the effect of making your code more complex than it readily needs to be
- **Missed abstractions** — and there's also the case where you probably *should* abstract a lot of the work to another layer of code, because it could make the code a lot easier to read.

Ideally, we want to strive for high cohesion because it makes code more understandable.

## More developers

As we learned in The Mythical Man-Month, adding more developers to a project has the adverse effect of introducing more complexity.

With more brains, opinions, and abstraction-makers, a consistent coding style and disciplined technical practices are paramount to keep complexity from exploding exponentially.

## Language and API capabilities

The more power and options your programming language, library, framework, or API has, the larger the surface area to create accidental complexity is.

Tools with fewer options are less likely to be misused in disastrous ways. If you've ever felt overwhelmed or discouraged from using object-oriented programming, I can't say I'm surprised. OOP comes with great power. And with great power comes *the possibility you may shoot yourself in the foot*. While functional programming can appear to be more appealing because the surface area of what is possible is much smaller, purely functional programming in the real world introduces a much different set of challenges.

## Premature optimization

The heavily quoted Donald Knuth says that:

Premature optimization is the root of all evil

90% of the time, optimization means introducing more state. We add a cache, some database tricks, mess around with Kubernetes, etc. In Ben Moseley's influential *Out of the Tar Pit* paper, he argues that *non-essential state* is one of the leading causes of software complexity. One of the better things we can do is to avoid it until it is absolutely necessary.

## Other causes

There are plenty of other forms of complexity, but in some way, I believe that they all negatively impact coupling and (or) cohesion: the two best measures we have of how well we're taming complexity.

- Duplication — coupling (we need to change code in several places)
- Dead code — cohesion (obscures our understanding of what we should do)
- Poor modularity — cohesion (there is a mix of high-level and low-level abstraction which makes it hard to understand the purpose of the code)
- Poor documentation — cohesion (again, it's hard to know what to do)

■ These are all code smells. We learn about other common ones and their refactorings in Part V: Object-Oriented Design With Tests.

## How to detect complexity

Now that we have a decent idea of what complexity looks like, let's discuss the symptoms.

### Ripple

When what appears to be a simple code change actually means you have to make modifications in several other parts of your code, you may be experiencing *ripple*.

Ripple signals that aspects of your code may be tightly coupled, that there's a lack of encapsulation, or that dependencies know too much about each other.

### Cognitive load

The more elements that you need to hold in your brain to complete a particular task, the higher the cognitive load. Going the *other* way that ripple does, cognitive load may signal that we have *too* much abstraction and encapsulation. This happens when we create too many elements, layers, classes with too many methods, use a lot of global variables, fail to

enforce consistent design decisions, and organize the files in our code by infrastructure (or type) rather than by feature — forcing us to flip between files.

## Poor discoverability

Let's say you're working on a project and have to add some additional validation logic to an EditUser feature. How long does it take for you to find the code where you need to make the change?

Discoverability is about identifying where things are, what things are, and *what we can do* with them — the capabilities of the codebase, an object, a function or method.

## Poor understandability

Now let's imagine we've found the area of code involved with implementing the EditUser feature. We're almost there. We want to add the validation logic, but looking at the code — we have **no idea how to do it**.

Understanding is about **knowing how to perform the action, performing it**, and then **confirming that we've succeeded** by noticing that we got the intended result.

For example, well-designed elevators simplify understanding by presenting merely two options: up and down. When we press one of the buttons, elevators light up and make a noise, giving immediate feedback and letting us know that we've carried out the correct action sequence.

Tools with poor understandability make it hard to:

- Understanding *how to use* something.
- Understand if we're using something the *correct way*. When tools provide too many options to do the same thing, it leaves us to question whether we made the correct decision or not.
- Understand if what we did *worked*.

■ **Leaky abstractions:** Have you ever run into a snag with React, Angular, or perhaps an ORM you like? Whether we design our own abstractions or use libraries or frameworks, “all non-trivial abstractions eventually leak”. A leaky abstraction occurs when a tool was supposed to abstract away all the hard stuff, but instead — we end up in a situation where we're forced to learn some of the internal details of the tool in order to make it work the way we intended.

## Mitigating complexity

### Tactical vs. strategic programming

There are two general approaches to sitting down and writing code: the *tactical* approach and the *strategic* approach.

In the **tactical approach**, we brute force our way through a solution until it's complete. We're not too concerned with the implications of our design decisions — we're just coding and using whatever works best until it works the way we want it to.

The tactical approach has its practical use cases. When you're building proof of concepts, you're essentially running an experiment. When you're doing exploratory programming, you're submerging yourself in the code, seeing what you can do, and coming back up for air. Both of these use cases are great when you want to see what you can do with code. The tactical approach just makes sense when you're building things that you're confident aren't going to be long-lived.

The tactical side is 0% focused on the maintainability aspect of programming. So if we were to apply this approach to a real-life project, yes — we might get it done the first time around, but it's likely that it won't be easy for you or anyone else on your team to make future changes.

With a **strategic approach**, there's a 50/50 split between writing code for the **customer** and writing code for **our maintainers**. It's like when we build a marketplace; we do activities that seed both sides of the marketplace because they're equally important, and we can't succeed unless we have both.

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

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

Therefore, we should be *investing* in the codebase by following techniques that will enable us (and future maintainers down the road) to maintain it and keep complexity at bay.

## **Extreme Programming: The original Agile development methodology**

Extreme Programming is the OG Agile development methodology. Created by Kent Beck during the dot-com bubble, it's a framework of best practice techniques known to create higher quality software that withstands the test of time.

When companies realized that they could generate revenue streams by shipping websites to market faster, the race was on.

Kent, who was working on a project for Chrysler, became the leader for a substantial project in 1996. As the project leader, he took careful note of the development methodologies he was using on his team, replacing things that didn't work with things that worked really well, and eventually — he formalized what became known as *Extreme Programming*: an *extreme* set of best practices for consistently shipping quality code on time.

The influence of Extreme Programming (also known as XP) on our industry has been monumental. It set the bar for how professional software developers turn customer needs into products relied on by users.

## **Technical practices**

This is the original set of technical practices from Extreme Programming.

### **Feedback**

- Pair programming — when multiple programmers

- Planning game — Release planning (with the customer) and iteration planning (with your team)
- Test-driven development (TDD) — new functionality is added by first writing a failing test, making it pass, then contemplating the design
- Whole team — the *customer* is a part of the team; they need to be on hand for questions at any point in time

## Continuous process

- Continuous integration — Prevent significant code conflicts with other developers by constantly uploading your code every few hours
- Design improvement/Refactor — When we start to see symptoms of complexity, we should consider refactoring to make the system simpler and more generic
- Small releases — Frequent, small releases that deliver value

## Shared understanding

- Coding standard — decide on a *consistent* set of coding conventions that you and your team will use throughout the project
- Collective code ownership — While teams can form, everyone is responsible for all of the code in general.
- Simple design — The simplest code possible has tests, contains no duplication, maximizes clarity, and minimizes the number of elements
- System metaphor (Domain-Driven Design) — Use the language from the domain to name features, methods, variables, classes, and so on.

## Programmer welfare

- Sustainable pace — Developers shouldn't work more than 40 hours a week

We can learn a lot from these practices. In this book, we'll discuss the majority of them but focus specifically on the following:

- Pair Programming
- TDD
- Refactoring
- and Domain-Driven Design

## Values (from Extreme Programming)

Values dictate real-world practical action. We call such behavior *practice*.

We spend plenty of time on the aforementioned *practices*, but know that they stem strongly from Kent Beck's original values in the Extreme Programming methodology.

## Feedback

Bad design is truly complexity of our own making.

When we write code and create abstractions that are more trouble to understand and work with than need be, we should take care to fix them.

Sometimes though, we miss bad abstractions, get too far ahead, and it's much too challenging and cumbersome to fix, so we just decide to live with our mistakes.

What if we could catch bad designs *earlier*?

An important value from XP is to **introduce feedback loops into our processes of writing code**.

Feedback loops force us to stop for a moment and look at the results of what we just did. It gives us a chance to see our progress. If we're prone to pause, it's more likely that we'll catch bad designs before they get to the point where we can't maintain them anymore.

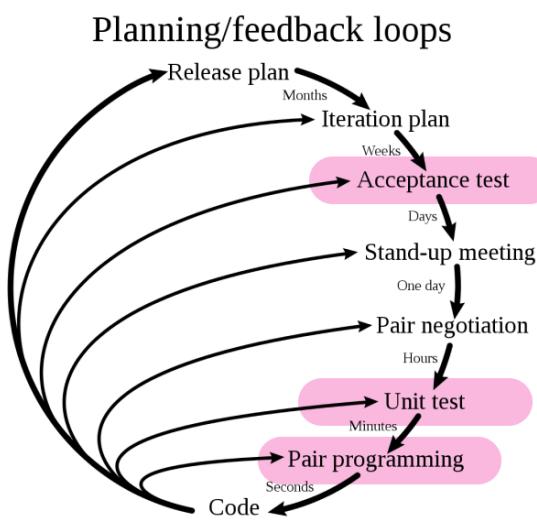
One form of feedback that most professional developers do at their daily jobs are **code reviews**. For those that don't perform code reviews, they're discussions you have with your team before your code gets merged into production. Code reviews give your teammates a chance to critique your code for quality.

If you're doing this, you're probably working on a *team* that values *feedback*.

However, this is the **last line of defence for feedback**.

Have you ever gotten notes on your code review where the changes you realized you had to make were *structural*?

What if I told you that there are other feedback techniques you can use to get feedback? Techniques to get feedback *much earlier* in the process. Even *while* you're coding.



In the diagram above, we can see various feedback loops that exist in the Extreme Programming methodology. In the context of this book, the ones that we'll focus on are:

- Pair programming
- TDD
  - Acceptance tests

- and Unit tests

## Simplicity

Our goal should be to build the simplest possible thing that works and is understandable to developers.

If there was a way to take the *formal specification* of what we're being asked to build — all the user stories and use cases (the features — that is), and then hand it to a no-code builder and have it cleanly developed into a solution for us, that would likely be the simplest possible thing that could work.

Unfortunately (or fortunately — for us), these tools don't exist (or they *do*, but they're not great at what they do yet).

So we have to build it ourselves.

Let's imagine two developers sitting down separately, both tasked with building the same feature.

To build it, they create abstractions.

They write classes, functions, objects and model the features in code.

Consider that each developer has their own approach to implementing the features. If neither of the developers' solutions makes sense to each other, who is in the wrong?

Neither one of them. This is hard stuff. It takes *years* for developers to understand how to design good abstractions and recognize bad ones. Crafting good abstractions straight from your brain on the first attempt is very unlikely to happen.

Obviously, we don't have years to master that. We need to be productive *today*. We need a repeatable, practical way to develop the simplest possible thing.

In Part III: Phronesis, we learn about all of the *feature-driven* techniques that the experts use in real-world Agile projects. TDD (Test-Driven Development) is one of those techniques. Perhaps the best for developing the simplest possible thing that works. Along with techniques like the Transformation Priority, Design Patterns, and Object Calisthenics, we define ways to gradually invent simple, necessary abstractions with TDD.

We learn the basics of TDD in Part IV: Test-Driven Development Basics, master the advanced stuff in Part X: Advanced Test-Driven Development, and learn how to incrementally improve our object-oriented designs in Part V: Object-Oriented Design With Tests, Part VII: Design Principles, and Part VI: Design Patterns.

## Communication

It's worth repeating that if 50% of what we're doing is building products that serve the needs of the user, then the other 50% is attempting to write code easily understood by other developers.

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

If we value communication, we want to use the most effective way to communicate program intent. Does that mean writing documentation? It *could*, but to me — the most effective documentation is the code itself.

If we write code in a declarative style, there's always going to be a single highest layer of abstraction that represents the code as close to the *essential complexity* as possible. In an object-oriented context, this will be our *acceptance test* code.

Our test code acts as a mounting point for other developers to learn about the domain, what our code does, and enables them to descend a layer deeper into a slightly more complex level of abstraction. Complexity gets pushed down.

In Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, we learn how to use a small set of architectural patterns to encapsulate complexity and represent the domain/essential complexity in the most declarative way possible in an object-oriented context.

A consistent *coding standard* is an equally important matter if we want to write code that communicates its intent well. Prior to writing our first test, in Part II: Humans & Code, we'll discuss how to develop (and enforce) a coding standard \*\*that results in code that humans enjoy reading and working with.

## Summary

- The goal of software design is to build products that **serve the needs of the customer** and can be **cost-effectively changed by developers**.
- Anyone can learn how to write code and build a product, but what separates the amateur from the professional is the ability to conquer complexity or succumb to it.
- XP is probably the best development methodology for reducing complexity in object-oriented software projects, and the XP values dictate our techniques and principles.
- Technical practices like TDD (see Part IV: Test-Driven Development Basics) and pair programming (see Part III: Phronesis). They'll help us write better code and catch bad designs before they have a chance to solidify themselves in our code.
- We'll start by treating these as *rules* because we don't have experience with them yet. In Part III: Phronesis, we learn the guiding principles behind expert use of these techniques like "Build the walking skeleton in Sprint 0" and "Get the customer to write the acceptance tests"
- Over time, treated as rules — practice the techniques and you'll develop your own set of principles. Then, you decide for yourself if and went to use them or not.

## Exercises

■ Coming soon

## References

## Articles

- [https://en.wikipedia.org/wiki/No\\_Silver\\_Bullet](https://en.wikipedia.org/wiki/No_Silver_Bullet)
- <https://khalilstemmler.com/wiki/leaky-abstraction/>

- [https://en.wikipedia.org/wiki/Extreme\\_programming](https://en.wikipedia.org/wiki/Extreme_programming)
- [https://en.wikipedia.org/wiki/Extreme\\_programming\\_practices](https://en.wikipedia.org/wiki/Extreme_programming_practices)
- [https://en.wikipedia.org/wiki/Chrysler\\_Comprehensive\\_Compensation\\_System](https://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System)
- [https://en.wikipedia.org/wiki/Coding\\_conventions](https://en.wikipedia.org/wiki/Coding_conventions)

## Books

- A Philosophy of Software Design by John Ousterhout
- Extreme Programming Explained: Embrace Change by Kent Beck
- The Mythical Man-Month by Fred Brooks
- Robert L. Glass: Facts and Fallacies of Software Engineering; Addison Wesley, 2003.

## Papers

- Out of the Tar Pit by Ben Moseley & Peter Marks

## 2. Software Craftsmanship

The Software Craftsmanship movement is about professionalism in software development. As James Clear writes in his popular best-seller *Atomic Habits*, if we want to see better results and stick to our habits, we have to start with our identity.

### Chapter goals

- Learn a brief history of software development and the big challenges up until this point
- To learn why software craftsmanship is the solution to a misled era of Agile
- Learn the main ideas behind software craftsmanship
- Learn a technique for adopting lifelong software craftsmanship habits

### Professionalism

Before I was a programmer, I was a landscaper.

When I was in my teens, with my stepdad's help, I started my own lawn care business to raise funds for my first few years of university. I called it Steps Lawn Care. It was a proper name for the company since my stepdad would teach me the basics and Shepard me to a level where I felt comfortable enough to do jobs independently.

I started not knowing the first thing about landscaping.

For the first few weeks, we went out and did jobs together. I watched how he worked, helped out where I could, and after time, I was doing it all by myself.

Eventually, through guidance and correction, I learned how to find clients, scope a job, provide an estimate for the work, and satisfy the customer by getting the job done *well* and on time.

My stepdad taught me **how to be a professional** in that industry.

I've realized that mentorship, the way I learned the trade of landscaping, **has been practiced for generations**, and stems back to medieval Europe with *masters* and *journeymen*.



Le charpentier.

Der Zimmermann.

The carpenter.

"The carpenter" - by Anonymous artist (<http://www.digibib.tu-bs.de/?docid=00000286>, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=981584>)

Back in those days, the *master* demonstrates the correct way of completing a task; then, the *journeymen* attempt to imitate the master's skills and get corrected for any mistakes.

It's interesting to note that this model, the **apprenticeship model**, is still widely used in traditional trades like steel-working, baking, plumbing, and many more.

Some traditional trades value **professionalism** so much so that due to trade regulation, it may be *illegal* to go out on your own until after you've completed an *apprenticeship* program: a certain number of practice hours with a more experienced member.

---

Programming, on the other hand, is an incredibly young (and turbulent) trade.

We expect new developers to learn what it takes to be a professional independently, and

often, there's detachment from the older generation of programmers to the newer ones.

Not only that, but we're still in the process of agreeing upon a standard procedure for reliably producing quality software on time (I believe the answer here is Agile, but with the proper technical practices, such as the ones advocated for in XP - Extreme Programming).

Imagine if plumbers were still trying to figure out how to fix leaky pipes.

It's not a surprise that so many developers are left confused, unequipped, and unguided to work *professionally*, because we, as an industry, haven't converged on what it means to work professionally.

Now, let's flip it around for a second. Imagine that you're a professional developer and you know the right things to do like TDD, Pair Programming, refactoring, and the like. That's great, but your life is *still* hard. Non-technical project managers often have a tendency to push back on these things. They don't see the value. These practices look like *not-working-on-features*.

Ah, challenges.

We might never regulate the industry of professional programming, companies will still be stringent to hire professional trainers, and it's unlikely that a non-technical manager will urge you to implement technical best practices. Even more, we might never introduce a concept like mandatory *formal* apprenticeships in our industry (some have tried - see 8th Light).

Well, is there anything we can do today?

Yes.

Today, what we *can do* is **adopt a craftsmanship mentality**.

Software craftsmanship is professionalism in software development.

*Software craftsmanship* is an emerging mindset shared by a growing community of developers dedicated to **raising the bar**.

While this guide is about software craftsmanship, another title could have been "how to be a professional software developer".

Being a professional (or craftsman — I consider them to mean the same thing) means taking **responsibility** for our careers, clients, and community. It also means taking **pride** in our solutions, working with **pragmatism**, and seeking to improve our reputations with the excellent work we do continually.

Professionalism is paramount to writing clean code.

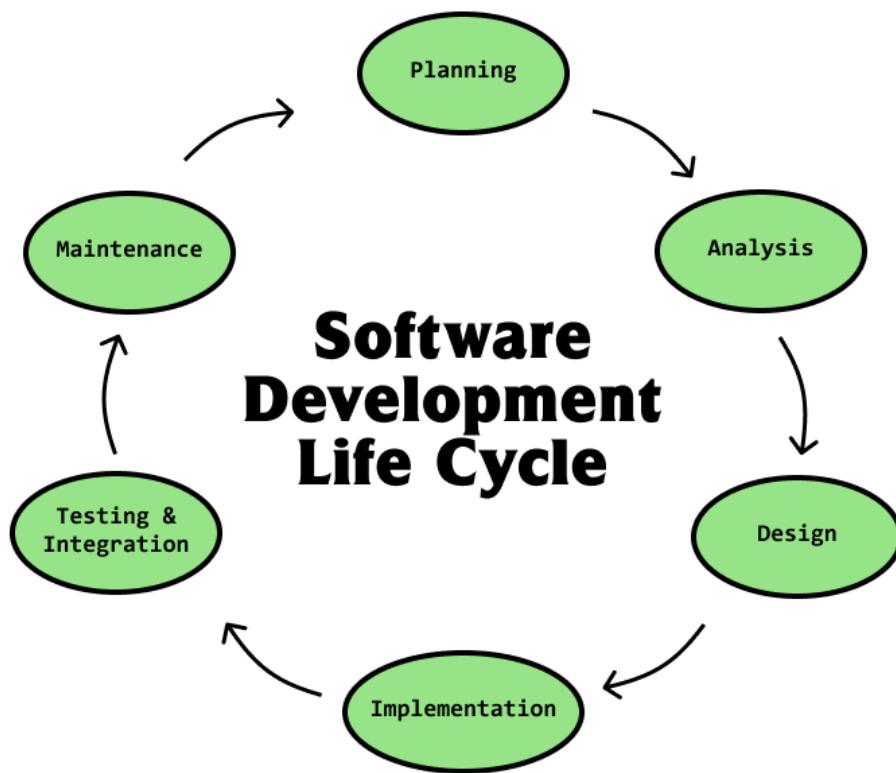
## A brief history of software development

First, to fully understand the origin of craftsmanship and respect the urgency of why you should strive to become one today, we need to discuss significant changes in our industry that left us where we are today.

## Programming picking up speed (50s)

In the 50s, computer science picked up. Around this time, we started to understand the lifecycle of how software is *planned*, *designed*, *built*, and *maintained*.

We called this the **SDLC (Software Development Life Cycle)**.



The Software Development Life Cycle demonstrated the general cycle that software projects operate in.

Until then, we had considered software development to be the act of writing code merely and fixing bugs. There wasn't any formal process around how this worked. But in 1956, we came up with the Waterfall model: an iterative approach to building software.

In later years (towards the 80s and 90s), industry professionals would come up with other, again opinionated ways to work, like XP (Extreme Programming), FDD (Feature-Driven Development), Scrum, and so on. We call these *software development methodologies*, and they act as a **framework to move through each phase in the SDLC**, specifically designed to circumvent bad coding practices and pitfalls. They give you rules to follow.

For example, one rule in XP is that *the customer writes acceptance tests*, that way, developers know what they should build and can define completion more precisely.

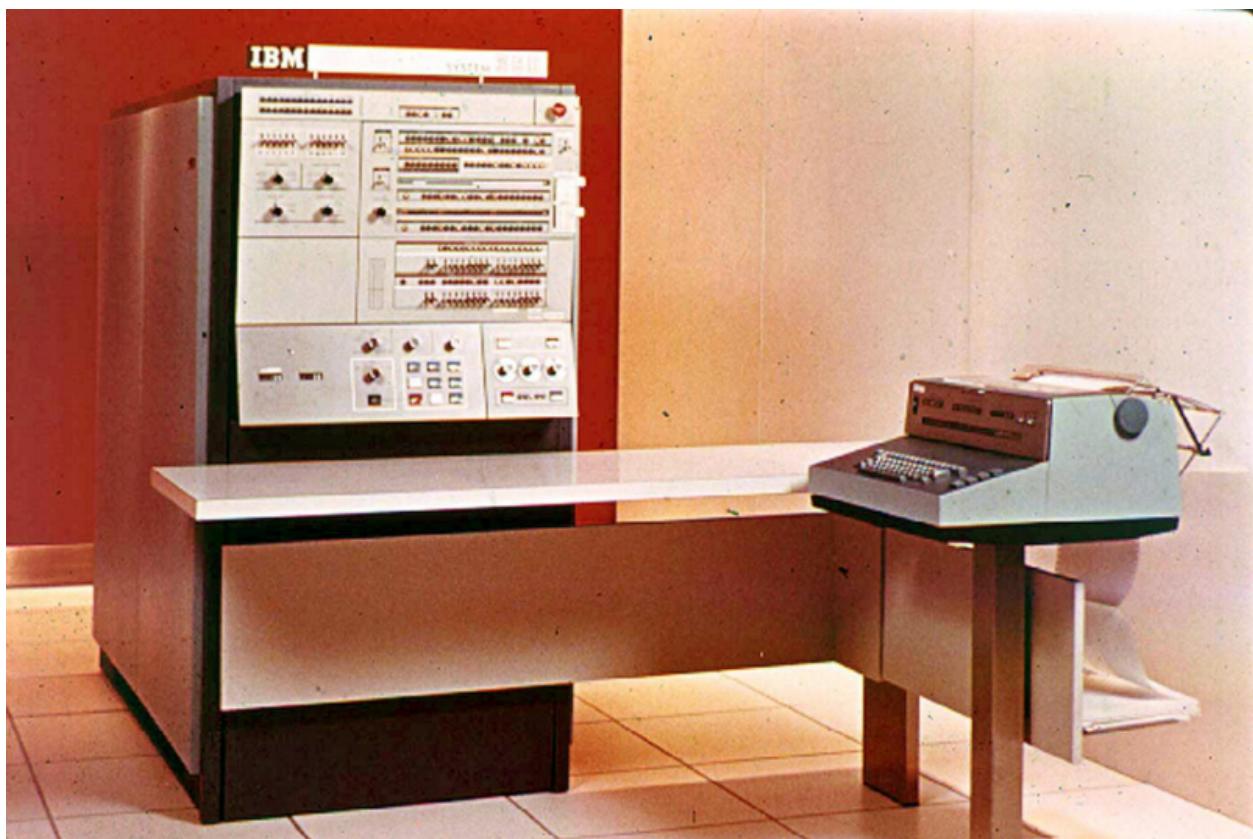
■ **Software Development Methodology:** An approach to software development, more specific than the Software Development Life Cycle (SDLC) that formalizes rules, strategies, and practices for delivering software projects. Examples: XP, FDD, Scrum, Waterfall.

But back in the 50s and 60s, processes were pretty stripped down. And so we went on to learn some tough lessons.

### The software crisis of the 60s-80s

From the 60s to the 80s, we discovered most of the **major problems** in software engineering that would continue to plague us today; they are based predominantly around *productivity* and *quality*.

I'll give you an example of each.



The IBM System/360

In 1964, IBM announced the *IBM System/360*: the first general-purpose computer system with cross-compatibility between models. Cross-compatibility was huge back then; the ability to *finally* run the same programs on every system instead of needing to stick to older, slower models to run the programs you needed for everyday use was revolutionary.

While spectacular when it was finally released, the **productivity failure** was legendary. It took IBM **an entire decade** and **1000 developers** to complete the system because they didn't develop a sound and understandable architecture. The influential book "*The Mythical*

*Man-Month*", written by Fred Books, is based on his observations managing the IBM system's development. By 1975, after completing the project and publishing the book, we, as an industry, came to learn that "adding manpower to a late software project makes it later". This is one of many things we had to learn the hard way, as there were few projects as large as this before.



The Therac-25: a machine that killed four people, and left two with lifelong injuries

One of the most famous **quality failures** is the 1986-87 story of the Therac-25: a buggy radiation therapy machine that malfunctioned and delivered ten times the amount of radiation acceptable, killing four people and leaving two patients with lifelong injuries.

What happened here? Lots. The architecture was doomed from the start. The original developer who wrote the core of the application had never done concurrency programming, so the foundation — the entire system design, was made extremely hard to understand after he had left the company. Further, to cut costs and make the code easier to reason about, developers removed the *hardware safety features* on previous models, replacing them to be controlled entirely by *software safety features* instead. That meant that when things went wrong because of software bugs, and the machine expelled way more radiation than it ever should have, there was no hardware detector to shut the machine down. Developers were too over-confident. They didn't write and run tests capable of testing currency. Even when operators discovered that people were dying due to what appeared to be a critical bug, the engineers repeatedly shifted the blame off themselves. They insisted that the deaths were due to "operator error".

Eventually, a dedicated staff physicist dug deep, found the bug, and reproduced it. AECL, the company behind the machine, then released updates to fix the bug. But even after they fixed it, some months later, someone else died from a bug caused by an *entirely different issue*: a counter overflow.

There's a certain legacy to this event. It's a perfect demonstration of how poor software practices, a lack of technical excellence, and *professionalism* to take responsibility can result in terrible (sometimes fatal) outcomes.

■ Read more about the Therac-25 in this article titled, "Killed By A Machine".

## Dot-com bubble, OOP, and Extreme Programming (1995 – 2001)

We called the 90s the **dot-com boom** because the internet completely took over. At some point, companies realized that if they could deliver internet-based products faster, they'd hold a massive competitive advantage.

Unfortunately, many teams were still using *iterative* software development methodologies like Waterfall, which, in this new climate of needing fast, quick, and snappy changes (which we continue to expect today), typically led to both low *productivity and quality*.

Many Waterfall projects started strong, but as customer requirements changed, projects felt more like death marches.

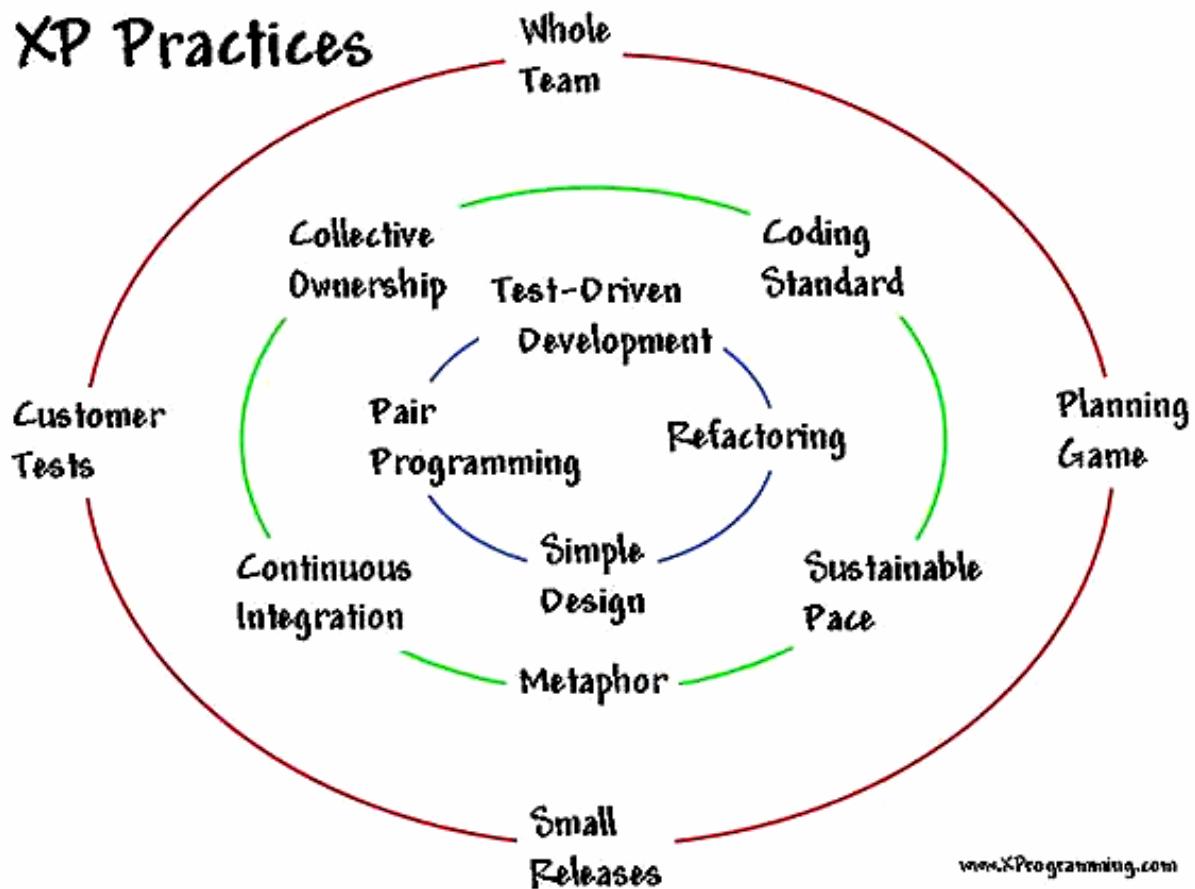
When developers progressed through the Planning phase, to Analysis, Design, and Implementation, if new requirements came in late, we didn't know how to best deal with it other than finish what we were working on, and then start all over. Iterative approaches did not help us keep up with the demands of our customers. We often delivered half-baked things that weren't really what our customers wanted.

Two other significant things happened around this time in the 90s.

1. Object-oriented programming became the most popular programming paradigm, surpassing procedural programming
2. Kent Beck formalized Extreme Programming

OOP became the go-to for many developers. There were a lot of developer experience benefits, but one large one was that it was now easier to create *domain models* that more acutely captured the essence of the business. Techniques were introduced that enabled us to strategically change code when new requirements came in.

# XP Practices



The “Circle of Life” — technical practices involved in implementing Extreme Programming (XP).

And it was around this time that Extreme Programming (XP), the **most well-defined and complete Agile process** (before Agile was even a thing), was formalized by Kent Beck. XP was born out of the dot-com era of quickly changing requirements.

## Agile (2001 — today)

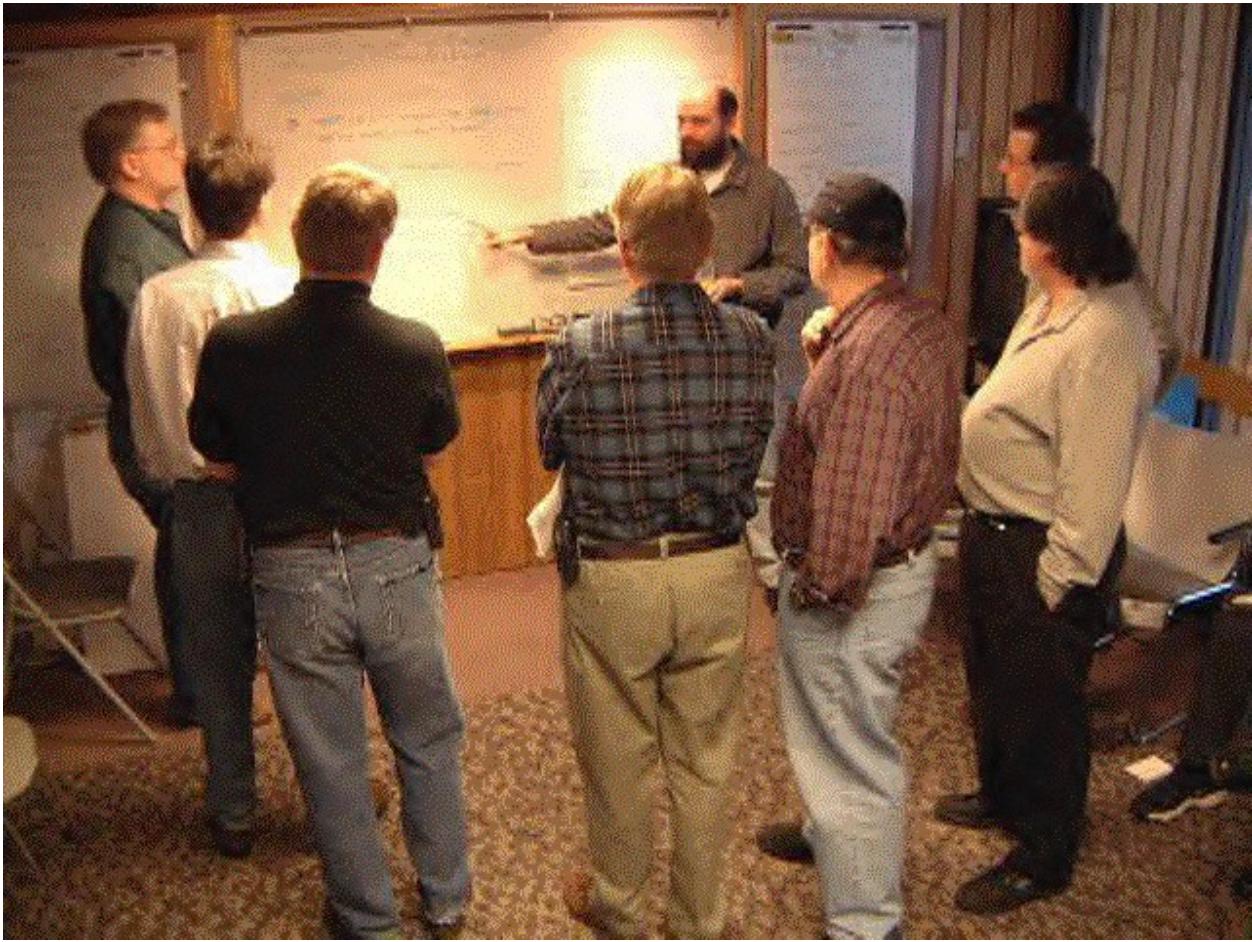
Agile is about delivering value incrementally instead of all at once.



Even though XP was very promising, we used and relied on other methodologies, too (i.e., *Scrum*, the *Dynamic Systems Development Model (DSDM)*, *Adaptive Software Development*, *Crystal*, *Feature-Driven Development*, and *pragmatic programming*).

In February 2001, seventeen influential developers, including those who created those methodologies, agreed to meet up at a ski resort to discuss better software delivery. The goal was to see if they could agree on something to be written down.

The list of developers were Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas.



The 2001 ski trip meeting in Utah that birthed the Agile Manifesto and began a new era of software development.

Miraculously, by the end of the trip, the group had agreed upon what would be called the **Agile Manifesto**, hence, formalizing Agile and changing the trajectory of our industry forever.

### **The Agile Manifesto**

The Agile Manifesto 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

That is, while there is value in the items on the right, we value the items on the left more."

### **The (Misled) Era of Agile**

There were notable changes to the industry because of Agile.

The first is the **introduction of Agile coaches**. Companies would hire Agile coaches to come to their site and help them perform an *Agile transformation* by introducing processes and Agile frameworks like Scrum. It's important to note that these Agile coaches did nothing to help *developers* change how they write code — they just introduced process.

The second change is that **developers became generalists**. Agile introduced the need for well-rounded developers. Today, developers need to know how to design, develop across the stack, set up continuous integration, deployment, databases, and know how to speak with customers and gather feedback.

While Agile promised the resolution of *quality* and *productivity* issues, and many companies dived right in, fixing their *process*, but still felt **plagued with both quality and productivity problems**. And so came the Agile-haters, more death marches, and more failed projects.

### **Why didn't Agile work?**

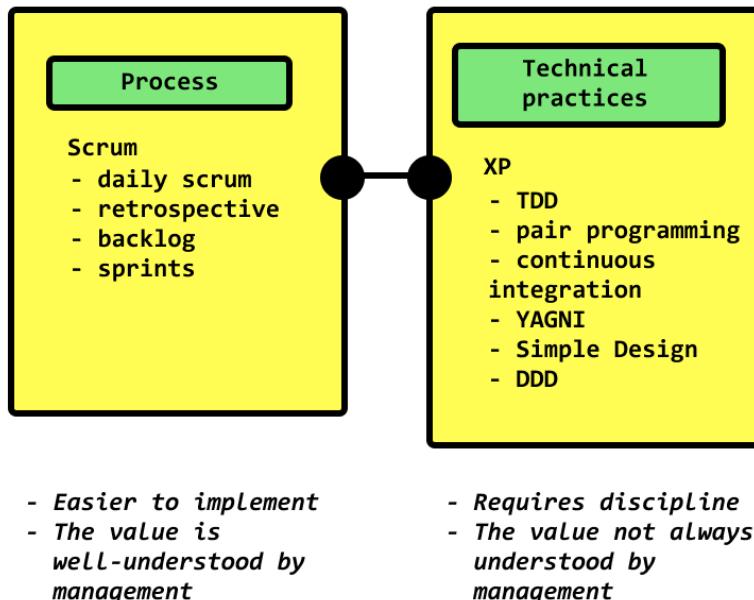
A lot of companies failed to realize that for Agile to work, two things are required:

1. **process**
2. **and technical practices**

If Agile didn't work, its because #2 — technical practices — were missing. But what was really missing was *craftsmanship*.

# Agile

**Both process & technical practices are mandatory to deliver value incrementally instead of all at once**



Teams were too focused on the process-oriented frameworks like Scrum but paid little attention to technical practices in XP. Not only that, but non-technical managers didn't see the value in XP practices like *pair programming* and *Test-Driven Development*.

"Why have two developers working on the same thing when you could be getting more done working separately?".

"And you're telling me it might take longer for you *initially* if you do TDD? Then don't do it."

The top-down, *I'm smarter than you, so do what I say, industrial, factory-worker* attitude applied to software developers enabled non-technical leaders to successfully push back on things developers see the long-term value in, but they, themselves do not value as much in as much as *delivering on time*.

For companies to properly migrate to Agile, they needed to **commit** to technical best practices. We needed Software Craftsmanship: the missing link.

## Software Craftsmanship (2006 — today)

There had been some talk about *Software Craftsmanship* before. The first book to hint that software development might be more of a trade than a science was "*The Pragmatic Programmer: From Journeyman to Master*." It drew comparisons between how developers learn and gain se-

niority to the apprenticeship model in medieval Europe, and was one of the first texts to attempt to distill some sense of professionalism toward software developers.

In 2006, Peter McBreen published the book *Software Craftsmanship*, solidifying the name and explaining many of today's professional practices. Unfortunately for the craftsmanship community around this time, Agile had the limelight. We considered Agile to be the solution to our biggest problems, giving little need for this thing called *Software Craftsmanship*.

To nudge craftsmanship back into the spotlight, in 2008, Uncle Bob proposed a fifth value for the Agile Manifesto: "**craftmanship over execution**".

Since things weren't going the way we wanted them to with Agile, and because we deviated from technical Agile frameworks like XP, in 2009, a group of developers aimed to produce something to be written down that could concisely describe the craftsmanship movement. Eventually, we created the **Software Craftsmanship Manifesto**. The manifesto, printed below, took the *Agile Manifesto*'s values and pushed it *further*, promoting technical excellence and professionalism.

## The Software Craftsmanship Manifesto

The manifesto 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 the following:

- **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"

## Back to Basics (XP)

There's still a lot of work to be done, but today, we're seeing some positive changes in the software industry around Agile and Software Craftsmanship.

**The software craftsmanship community is growing.** Developers worldwide are organizing meetups, signing the original manifesto, and spreading the word of professionalism. This community may very well be the one that comes together to popularize and standardize the approach to solving that 60-year old *productivity* and *quality* problem.

Also, in 2019, a frustrated Uncle Bob published a book titled "*Clean Agile: Back to Basics*", which is almost entirely a **walkthrough of the technical practices of XP**.

If you ask me about my future assumptions, I would say that in the next decade or so, we're going to see the craftsmanship and Agile communities converge. It's only a matter of time before this happens because they share the same goal: **to deliver high-quality software on time**. I can see what it will take for us to have that full Agile transformation. We've seen now that we can't leave out technical practices, and that we need to be more disciplined about sticking to those practices.

Now it's up to us, the software craftspeople, to do our part and set an example.

## Craftsmanship: Professionalism in software development

### Definition

We'll repeat the *best* definition for software craftsmanship.

Craftsmanship is professionalism in software development.

Software Craftsmanship is a *professionalism mindset*. It's not about if you have a certification, if you're a senior developer, or if you practice TDD all the time.

It's about taking **responsibility** for our careers, clients, and community. It also means taking **pride** in our solutions, working with **pragmatism**, and seeking to constantly improve our reputations with the good work we do.

### Are you a software craftsperson?

If you do the things mentioned above, you're already a software craftsperson. There's no magic to it. Even if you don't like the label and don't want to call yourself a craftsman, that's totally cool. What's important is that you do these things.

### Understanding the manifesto

#### Not only working software, but also well-crafted software

It's important to remember that **working software** can also be software that's:

- hard to change
- difficult to understand
- slow

But *well-crafted* software is software that is testable, flexible, and maintainable. It's also software that gets better over time — growing in value as you put more time and care into it.

#### Not only responding to change, but also steadily adding value

Adding value isn't just adding new features or fixing bugs. You can consider adding value to be continually improving the structure of the code, keeping it clean, testable, flexible, and maintainable. It costs a lot of money to perform re-writes, and though every application has a life-span, usually — if that's the only feasible option on a relatively young application, it means we've failed.

#### Not only individuals and interactions, but also a community of professionals

We're not alone in this. Your fellow software developers are going through the same challenges you are. To push this industry forward, we need to share knowledge, learn from others, inspire, mentor and prepare the next generation of craftsmen.

Because software development is so young, and because there *isn't yet a standard* for many things, knowledge sharing and mentorship are incredibly important. We don't want to lose what we've discovered over the last 40 years. We want to keep quality high, and teaching less experienced developers how to work better is a form of leverage that will pay off in years to come. We all have this responsibility.

## **Not only customer collaboration, but also productive partnerships**

We don't believe in an employer-employee relationship. That doesn't exist in software craftsmanship. Whatever is written on your contract is a formality. Instead, **treat your employer as your customer**, the same way you would if you were just a consultant or a contractor. Some important things to remember:

- Your employer isn't responsible for ensuring you grow professionally — you are.
- Your employer isn't responsible for ensuring you get a book budget, training, or sent to conferences and seminars — you are.
- Your responsibility is to **treat your employer as a customer and provide them with excellent service (advice, consulting, development) and help them achieve their goals by performing high-value activities, even if that sometimes doesn't mean coding**. Sometimes it means giving a talk, a presentation, or writing a blog post.

Unfortunately, not *all* companies have a management structure that lets you work this way, and it can be hard to push back on what you believe is best for the business when the other party doesn't see you as a partner.

## **Craftsmanship principles**

Staying true to the manifesto, we're ready to discuss actionable things you can do to become a better software craftsman *today*.

### **To write well-crafted software...**

**Care about what you do.** The world relies on the work you and I do. There are at least 4-10 different computer systems (your earbuds, your phone, computer, appliances) in any given room. Put love and care into what you're doing, because merely enjoying the work you do increases the likelihood of doing a good job.

**Use Agile technical best practices.** There are some things that all professional software developers should know how to do, and *when*. TDD, refactoring, pair programming, and simple design are all essential technical **Agile practices** from XP. We learn how to master each of these in this book. Following that, we improve our design skills with design principles, patterns, and various architectural styles, etc. We'll cover each of these throughout the book too.

**Always be improving yourself.** There is always room for improvement. After reading this book, I encourage you to gradually work your way through the resources and readings I've listed in each chapter. Regularly learn new techniques, vendors, and exchange the tools in your toolbox for better ones.

**Know your industry.** We discussed the brief history of software development, Agile, and the emergence of software craftsmanship. I'll be the first to admit that I had no idea what Agile was about, what the original goals were, and what it meant for us to be *Agile* at work. It's important to know why we do what we do. Keep an eye on emerging trends and updates to the libraries, frameworks, and languages you use to develop software.

**Learn the domain.** Your job is to understand the problem space (what your client wants to accomplish) and create the solution space (software). If you know your client's business,

you'll better understand the requirements; this enables you to ask better questions, understand likely ways that the code may or may not need to change, and contribute to discussions with domain experts. When you know the domain, the code becomes a *metaphor* for what happens in real life. In Domain-Driven Design, we call this the *Ubiquitous Language*, to use the same words from real-life in the business, in the code. If Agile is about making incremental improvements and short feedback loops, we can keep the code simple and avoid excessive designs by projecting what happens in real life to the code.

**Ruthless simplicity.** Build precisely what is needed to get the feature to work. No more. Get a feel for when you're diverging from that, adding more code or unnecessary abstractions, and catch yourself. If you practice TDD, The more code you have to account for, the more ways it can go wrong, and the more tests you need to write to ensure it behaves the way it should. Remember YAGNI — You Aren't Gonna Need It.

**Practice.** If you don't use it, you lose it. As I advance in my career, I frequently find myself doing more writing, speaking, mentoring, and less coding. You need to keep your skills sharp. My favorite way to practice is to work on side-projects based on your interests. Pick something based on your interests, so you stay motivated and driven to improve it as much as possible. I'm into music, so I built a vinyl-trading app. Another option is to contribute to open-source. Pick a library or framework you use frequently, run the tests, read the docs, and take a look at the open issues. Not only is this a great way to practice, but you get to see how others develop software in the real-world.

### To steadily add value...

**Apply the Boy Scout's Rule.** The Boy Scout's rule is to leave the campground cleaner than when we found it. If you come across some confusing, messy, wrong code or code with unused variables, take some time to refactor it right away. It sets an excellent example for others, fostering the idea that this codebase is a place of *cleanliness* and *quality*. Hopefully, others will clue into that and give it the respect it deserves too. Some call this the "Broken Window Theory".

**Refactor, guarded by tests.** If you're going to refactor something to be cleaner, you should first safeguard it with tests. We simply cannot safely change code without the presence of tests. Another good time to add tests and refactor is when we come across a bug. To fix bugs, write the test that proves the bug's existence, then refactor the code so that the test passes, and the bug no longer exists. Over time, you'll harden your app and sleep well knowing that you can change code with an increasing amount of safety.

**Be brave.** Coming from the previous two statements, when you see code you're scared to change because you're scared it might break, and you're scared of what might happen to you if you break it, this is the start of software rot. Everyone else is also scared. Be the brave one. Be brave to surround it with tests and refactor it into something understandable. Tests are your saving grace and will enable you to act like a professional.

**Learn and apply XP.** It's the most extensive, well-thought-out Agile way to deliver a software project. Learn it and challenge **bureaucracy and non-technical management** to apply its technical practices in your day-to-day work. It's the professional thing to do. See "Help them see the value of technical practices" below to learn how to pitch it.

**Delight customers, helping them achieve whatever they want.** Their goal is to make money, save money, and preserve revenue streams. Your goal is to help them do precisely that, by taking their fuzzy requirements, applying Agile best practices (XP), and turning it into the *right* thing.

### Engage in the community...

**Learn from others.** There's only so much you can learn on your own. Read developer blogs, books, watch courses, YouTube videos, and ask questions. Do pair programming. Some developers fear pair programming because the other developer might see them make a mistake. The best advice is to overcome this fear. Exposing yourself to how another person thinks and works can help you either refine or reconstruct the way *you* work.

**Mentor less experienced developers.** Unfortunately, we don't have more software apprenticeships. When you graduate from college, university, or boot camp, you're an apprentice. You have little experience delivering quality software on time. You may have spent time learning a front-end JavaScript library or framework, but the professional practices (TDD, BDD, DDD, etc) are missing. Craftsmen take a note from other trades like plumbing and carve out time to nurture newcomers into the professional way of delivering software products. At Amazon, junior software developers' PRs are subject to many scrutinies, but more experienced engineers leave incredibly helpful comments and feedback (sometimes 100+ comments on a PR). Eventually, junior developers get to the point where their PRs require a lot less feedback, and their code gets merged in 1 or 2 revisions rather than 10.

**Share what you know.** Teaching is learning. I've learned more than I thought possible by sharing what I currently knew on the internet. If you're wrong, someone will eventually correct you. This is important as an industry because we need to preserve what we've learned over time so that others can avoid the same mistakes. Therefore, write blog posts, code, share your wins, failures, ideas, tips — these help developers at different stages in their journey.

**Socialize with others.** I was amazed I could ask for Vaughn Vernon's advice on Twitter and actually get it. I was also surprised I could run polls, asking for what developers think "clean code" means, and receive an influx of opinions to inform my *own* better. Twitter can be a bit of a strange place sometimes, but as a developer, there's a lot of value in being on the platform. You can *network* with others by engaging with their content and ideas, and having technical discussions. Offline, check out meetups and meet other developers using tools, technologies and approaches you're interested in.

### Consider yourself a partner...

**Take responsibility for your own learning.** Your employer is not responsible for your education. Imagine your doctor saying, "yes, I can help cure you, but I'm going to need you to pay for me to read these books as well." Again, this is a reminder that the work we do is a trade, and it's up to you to carve out time to improve.

**Take responsibility for success/failures.** It sucks when things go wrong, but it's noble to take responsibility and take action to fix situations. Taking responsibility for your shortcomings is an excellent way to preserve your relationships and reputation. You will screw up. Trust me. And reputation matters, so that you can land better work and opportunities.

Look at it this way, when things go *right*, you can take the praise for that as well, and that feels good.

**Help them see the value of technical practices.** How do you get your boss to see the value in TDD or refactoring? Don't promote the technical *practices*; promote the *value* instead. People won't change their minds unless they can see the value. So then, what's the value of TDD? Well, you could tell your boss, who is incredibly aggressive in trying to get you to finish the project in as little time as possible, that TDD reduces the time it takes to test the system, results in fewer bugs, which means fewer hours coding. And that means they *save* money. You could also tell them that we could even release software a lot more reliably. They might ask, "what's the cost of this?". And that's a good question to ask because everything comes with a cost. TDD's *cost* is that everyone needs to know how to do it, and that may not be the case. You could offer to learn it and devote an hour to teaching your team how to do something that will save them hundreds of hours fixing bugs.

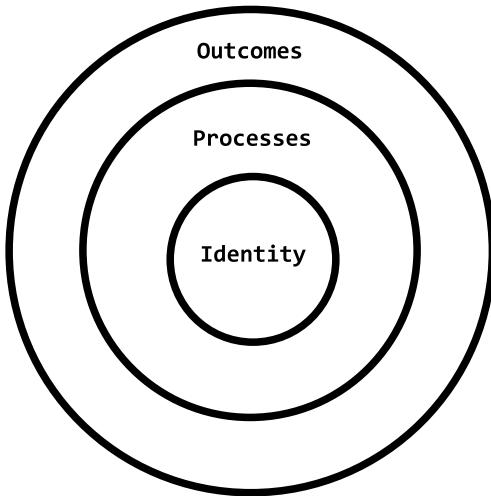
**Don't overwork yourself.** Most people don't do good work when rushed. Most people also don't do good work when they're tired. Personally, as soon as I start to feel brain fatigue, I step away from the computer and go for a walk, exercise, or call it a day. Why? Because I know that whatever code comes out of my hands next is more than likely going to be a liability rather than an asset. You're only human. My tip? Get the most important work done whenever you feel most alert in the day. For me, that's right in the morning. It might be different for you though.

**Provide value, even when it's not coding.** We expect developers in an Agile-era of software development to act a lot more like generalists. So aim to provide the most value possible, even if it doesn't involve coding. For example, you might see an opportunity to improve a process, pitch that. Question requirements and make sure we're building what we need. Spend time understanding the business and how it makes money. Help leaders prioritize the most critical work. Give them *options*. Hopefully, you can see how valuable these things are.

### How to make habits out of craftsmanship principles

People have lofty goals and things that they'd love to turn into habits all the time. For example, I'd love to read a new book every month, exercise, and drink 8 cups of water every day. What's behind making this a reality that I stick to — not just for a few weeks, but for a lifetime?

In James Clear's research on habits in *Atomic Habits*, he argues that the key way to build habits is not *to just try really hard* but instead, it's to start with **identifying who we want to be**.

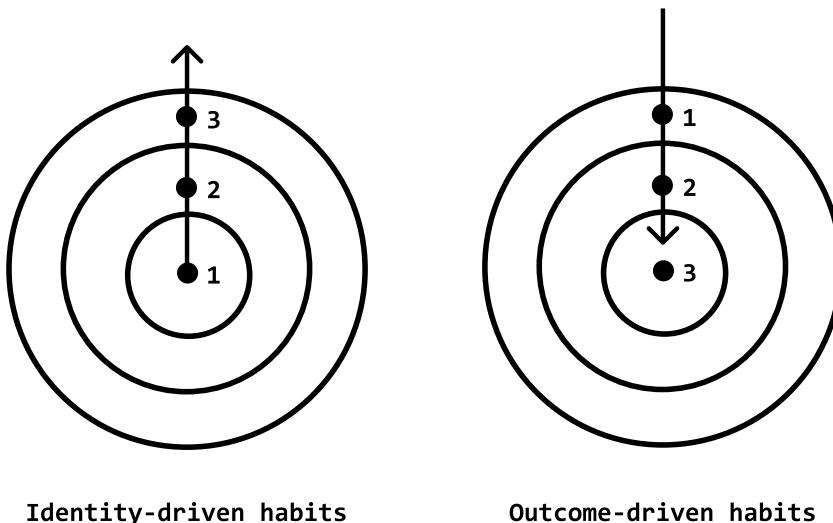


Most people want to **accomplish a goal** or create a set of **outcomes**, so they compose a set of **habits** that they'll follow **until** they meet their goal(s). Only after people accomplish their goal(s) do they decide that they **are the person they want to be**.

eg.: if I write testable, flexible, and maintainable code, then I'm a good software developer, and I'm a software craftsperson

This is what he calls **outcome-driven habits**. The thing about outcome-driven habits is that we only feel motivated to continue performing the habits if we feel like we're getting closer to our goal. That means that any small failure, a step backward instead of a step forward, has a lot of power in our lives. It has the power to wreck our entire system. This is why people sometimes quit their dieting and weight loss routines when they notice that they've plateaued or put weight back on.

Instead, what Clear recommends is the **identity-based approach** to habits.



The identity-based approach to habits is to:

1. Decide on who we want to be
2. Prove that we are that person with small wins, over and over

According to Clear's behavioral science and psychology research, **true behavior change comes from identity change**. If we want to write a book, work backward by seeing ourselves as writers. From there, we ask ourselves "what are the types of values and principles that writers have"? Clear thinking, meeting deadlines, etc. And what types of habits do they have? They probably wake up and journal every morning, right?

Once we do this exercise and we start to *believe* that we are the person we want to be, the easier it is to stick to the habits.

Let's work backward now. What's our goal? To write testable, flexible, and maintainable code? Who is the type of person that does that? That's a software craftsperson. And what do they make a habit of doing?

eg.: I'm a software craftsperson. Software craftspeople write testable, flexible, and maintainable code by [INSERT SOFTWARE CRAFTSMANSHIP PRINCIPLE HERE].

eg.: I'm a software craftsperson. Software craftspeople write testable, flexible, and maintainable code because they **"care about what they do"**.

eg.: I'm a software craftsperson. Software craftspeople write testable, flexible, and maintainable code by **"using Agile technical best practices"**.

The more proud you are about your identity, the easier it will be for you to maintain the habits associated with it. Therefore, if you're proud to be a software craftsperson — the kind of person who walks away from their desk for the day feeling that they've done a great job, it will be much easier for you to stick to mastering challenging techniques like TDD and Domain-Driven Design.

## Summary

- Until now, the biggest problem we face as an industry is not knowing what we should do, it's having the discipline to do it.
- Software craftsmanship is about professionalism in software development.
- Software craftspeople care about well-crafted software, steadily adding value, their community of professionals, and building productive partnerships with customers.
- To turn craftsmanship principles into habits, you must already believe that you *are* a software craftsperson. Focus on putting in the reps and the results will come.

## Exercises

■ Coming soon

## References

### Books

- Apprenticeship Patterns: Guidance For the Aspiring Software Craftsman by Dave Hoover
- Software Craftsmanship: The New Imperative by Pete McBreen
- The Clean Coder by Robert C. Martin
- The Pragmatic Programmer by Andrew Hunt and David Thomas
- The Software Craftsman: Professionalism, Pragmatism, Pride by Sandro Mancuso
- Atomic Habits by James Clear

## 3. A 5000ft View of Software Design

■ The more *unknown unknowns*, the better the chance of accidental complexity. Here, we squash many unknown unknowns in the world of software design through the concept of *layers of design* and a 5000ft view of how it's all connected. The goal of this chapter is to get a high-level understanding of what we need to know to master software design.

### Chapter goals

- Understand the difference between software design and architecture
- Understand the big picture of software design and how various levels of design fit within each other

### 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.

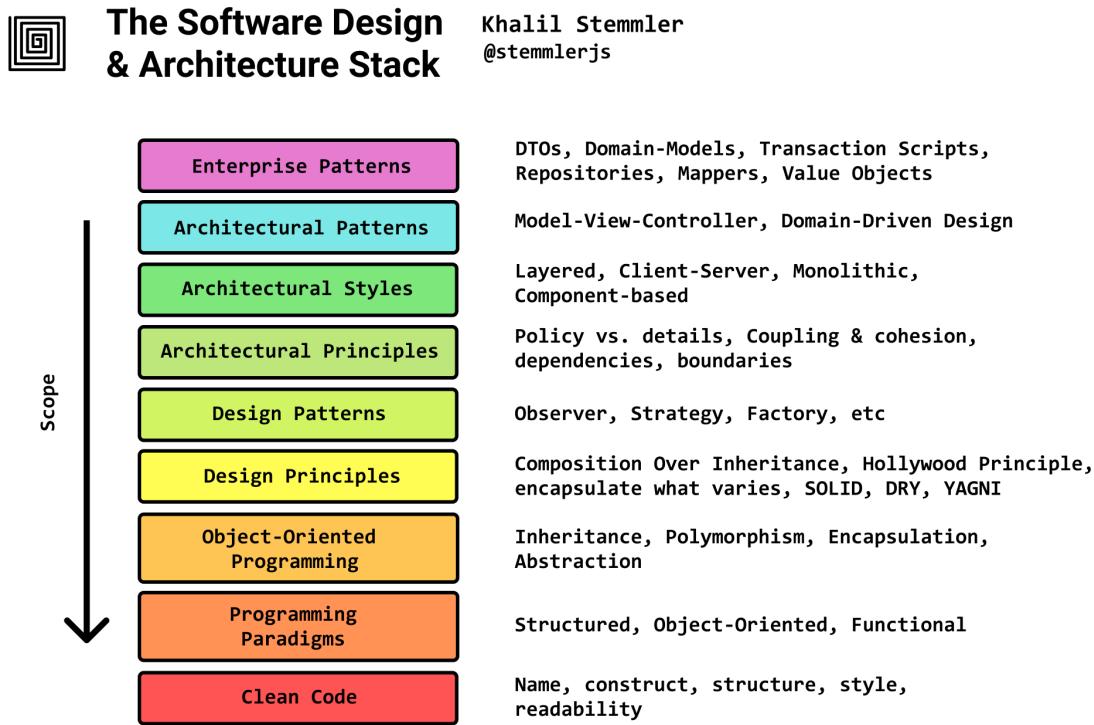
They are the **stack** and the **roadmap**.

### 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 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.

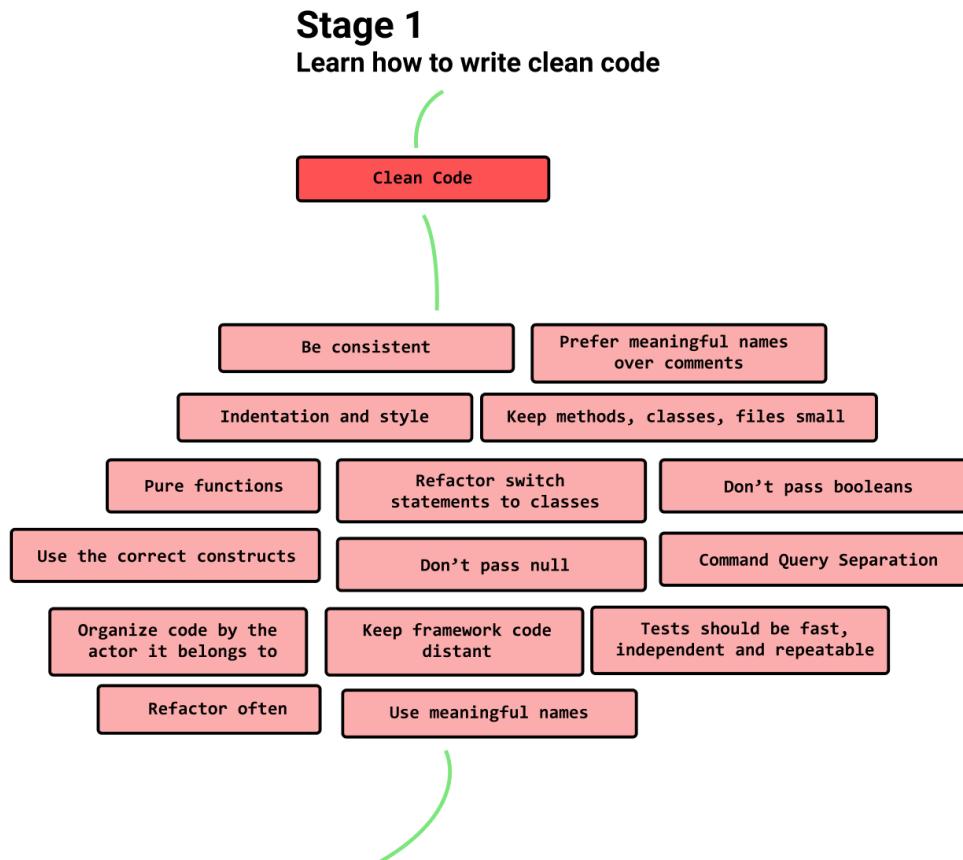
From clean code to Domain-Driven Design concepts, the map is indicative of the path that we're going to take in this book to get you ramped up in the world of software design and

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

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.

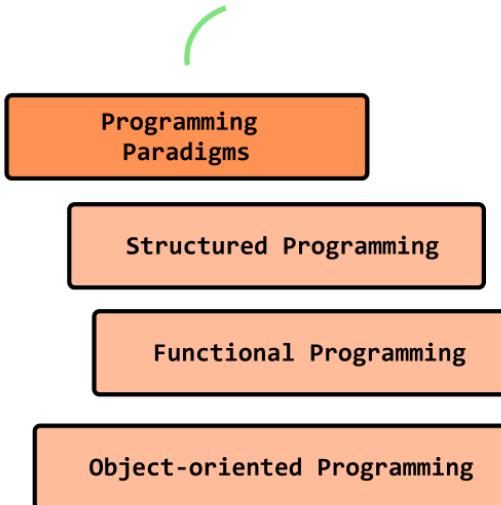
In Part II: Humans & Code, we discuss what clean code is and how to write code that is clean.

## 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.

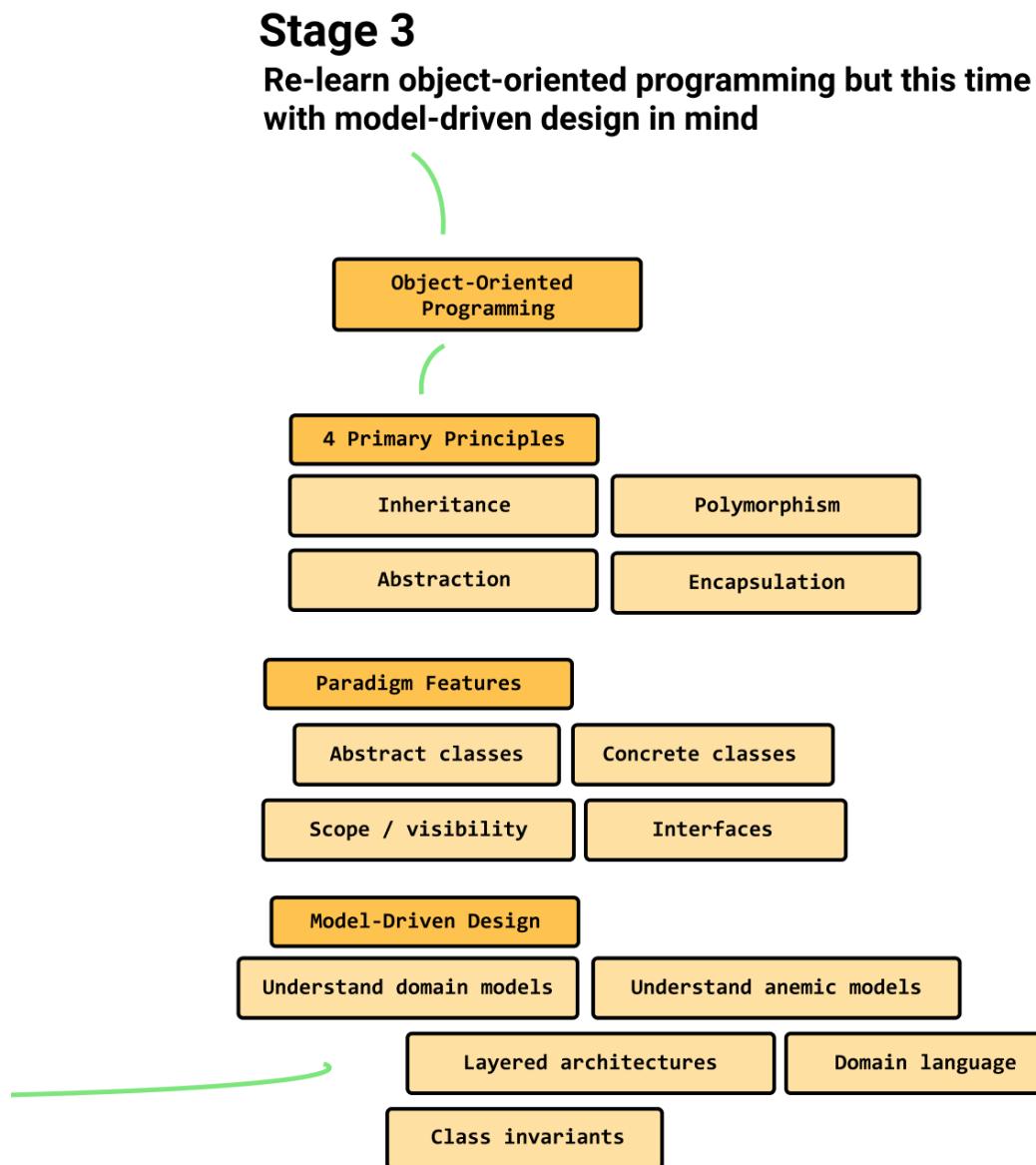
It's one of those scenarios where:

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

In 23. Programming Paradigms, 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 Part V: Object-Oriented Design With Tests, 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?

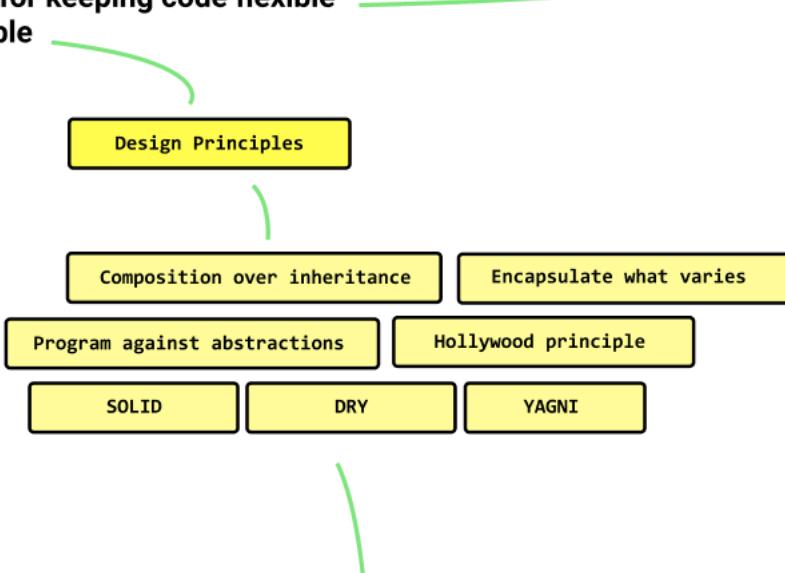
Because if we can create a mental model of a business, we can create the software implementation of the business.

#### 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.

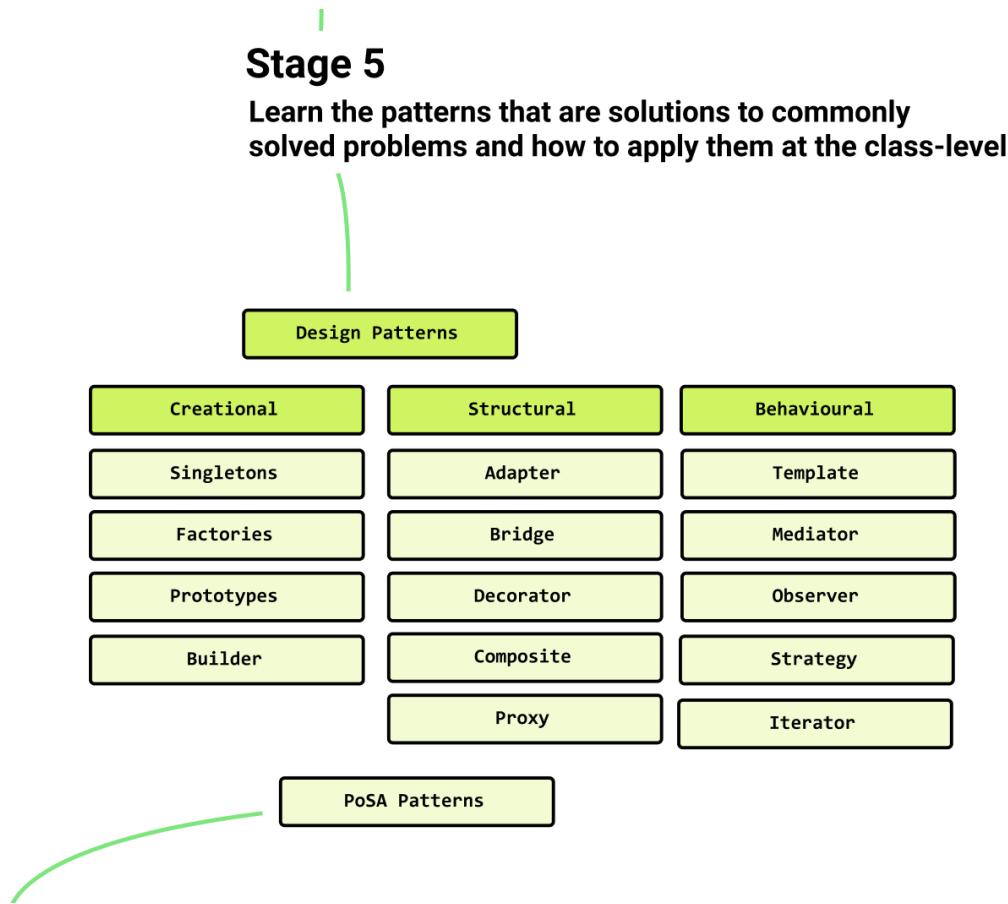
Examples of common design principles we will familiarize ourselves with are:

- 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 Part VII: Design Principles.

## 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 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 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 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** (10. Architectural Patterns) **are just design patterns blown-up in scale to the high-level**, where design patterns are low-level implementations (closer to classes and functions).

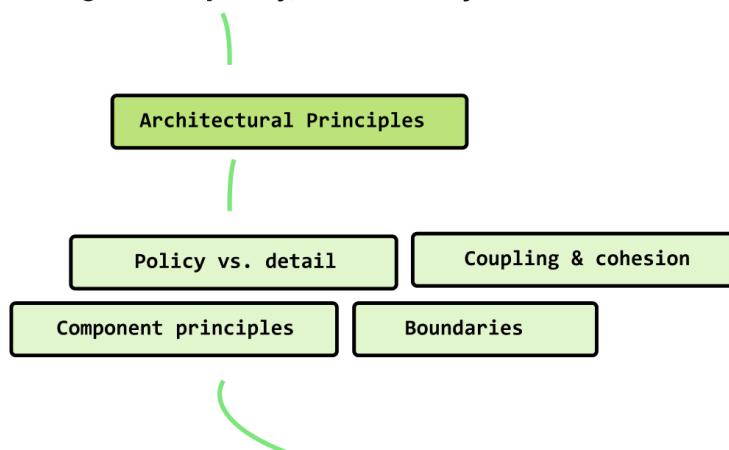
We discuss design patterns in Part VI: Design Patterns.

### Step 6: Architectural Principles

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

## Stage 6

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 8. Architectural Principles, 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:

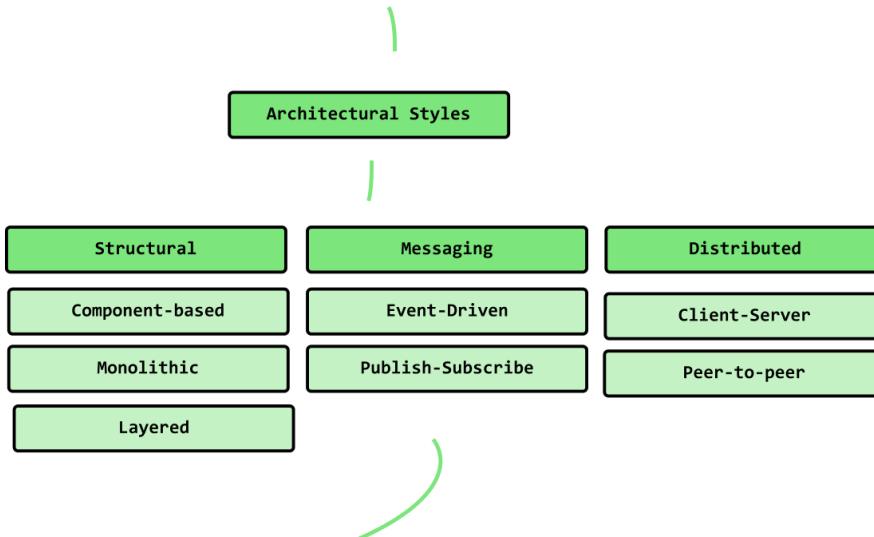
- **Component design principles:** The Stable Abstraction Principle, The Stable Dependency Principle, and The Acyclic Dependency Principle, for how to organize components, their dependencies, when to couple them, and the implications of accidentally creating dependency cycles and relying on unstable 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.



System Quality Attributes (SQAs) are metrics we need to protect to *stack the odds of success* of our application's architecture.

**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

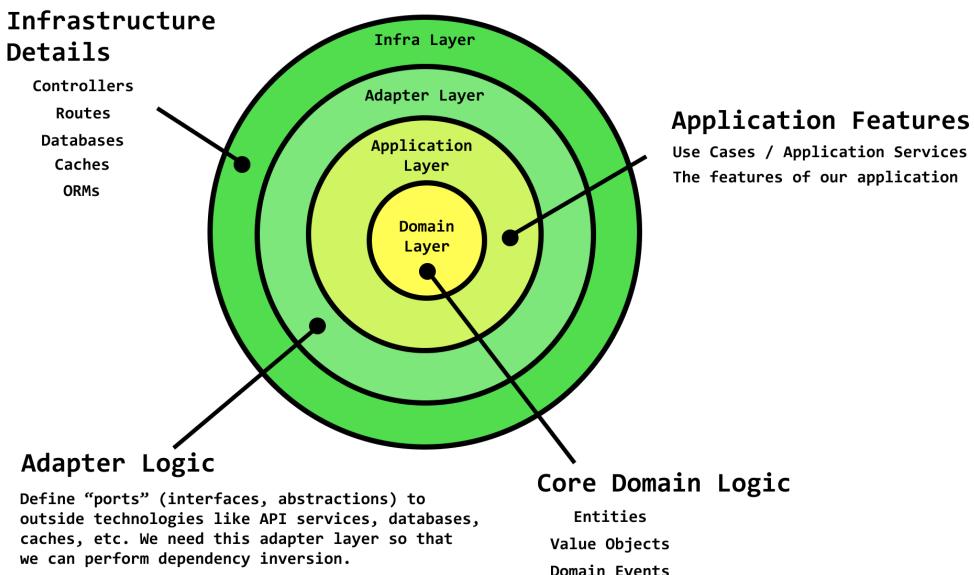
Projects with *varying levels* of components, and wide-ranging functionality are usually looking for - **flexibility** as an SQA. Structural architectural styles make it easier to extend and separate the concerns of complex systems.

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.

- **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.

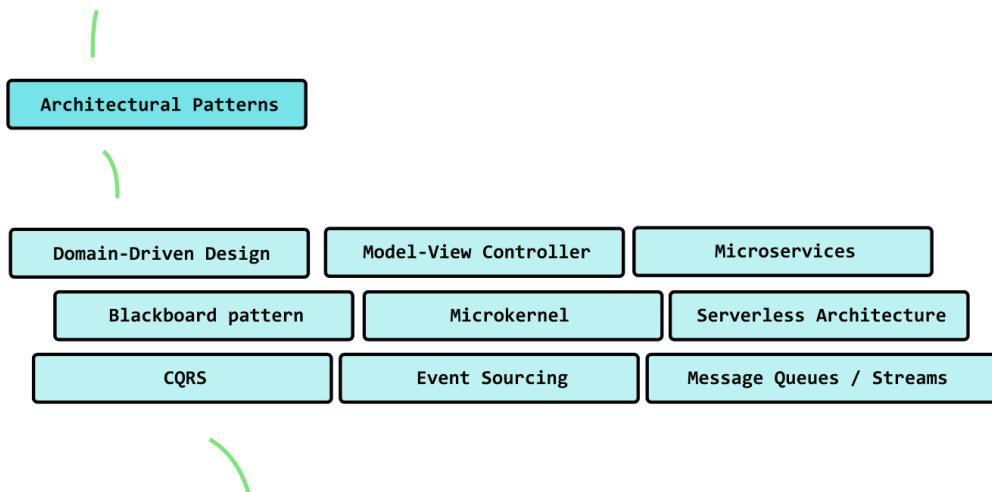
We discuss architectural styles in more detail in 9. Architectural Styles.

## Step 8: Architecture Patterns

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

### Stage 8

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 Part VIII: Architecture Essentials.

### 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 Part IX, we build a real-world app with Domain-Driven Design, the Hexagonal Architecture, and CQRS.

## Summary

- Software design and architecture are fundamentally the same thing — they just have connotations of slightly different levels of design.
- Architecture is about the *skeleton* of our applications and the hard to change things, like the way we've structured it, the way we handle errors, represent features in code, etc. It is the set of patterns that come together to realize our application.
- Much of what has been discovered in the past about software design and architecture is relevant and related in some way. For example, software design patterns, which are traditionally used at the class level, influence new tools, technologies, and architectural patterns at a higher-level of abstraction.

## Exercises

■ Coming soon

## References

### Articles

- 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?

### Books

- Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts by Joseph Ingino

## Part II: Humans & Code

■ Code has two consumers: users and developers. XP's technical \*\*practices will help us become more structured with the way we write software, but how can we tell if our projects are pleasant to work with? We need to \*\*understand how we **understand**. This section of the book introduces the principles of Human-Centered Design and techniques to develop more human-friendly codebases.

### 4. Demystifying Clean Code

■ According to "Simple Design", clean code is code that passes all tests, has no duplication, maximizes clarity, and has fewer elements. To accomplish this, the XP technical practices promote a sort of *emergent* design — that is, to take a gradual approach rather than an up-front one. Emergent design gives us the opportunity to frequently correct bad designs early, but we thrive the most when we can judge a design from both *technical* and *human-centered* perspectives.

## Chapter goals

- Learn what *clean code* means to the general software development community and to the experts
- Discuss how XP technical practices like TDD promote *Emergent Design* as a way to build up structure
- Understand the relationship that structure has with developer experience

### What is clean code anyway?

Clean code is a part of our definition of the goal of software. It is code that “**serves the needs of the users** and can be **cost-effectively changed by developers**”.

Cleanliness is predominantly determined by the property in the second half of that definition — cost-effectively change.

We could say that clean code \*\*\*\*has little amounts of *accidental complexity* like **ripple, cognitive load, poor discoverability, and poor understanding** from i. Complexity & the World of Software Design as possible.

Instead of assuming, we’ll again use the *Aristotelean* approach to mastering software design. To best answer this question, let’s look at what the community thinks and what the experts think clean code is.

### According to the community

Let these act as a sort of *mood board* for what clean code means.



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 the thread here.

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

*Easy to read* could be a comment on naming things. It could also have to do with formatting and style.

*Easy to modify* could be a comment on loose coupling, minimal ripple effects, and it could imply the existence of tests since we’re able to refactor code without breaking it.

To me, *reveals the intended purpose* speaks directly to using the element of Simple Design that says to use the metaphor, but I think it also means that code is *declarative* and *functionally*

*complete*. This is easy to achieve if we practice TDD and write unit and acceptance tests in a way that documents that the customer-driven features are working as *intended*.

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

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. We noted this before, but coupling isn’t one of the first things you think about in the conversation of *clean code*, but it’s not to be excluded. 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.”

Telling the *story of the domain* means that any new developers can learn how the business works by reading the tests.

“I can read it without it being painful. I can change it without it breaking everything.”

Readability! Again! To me, if I have to flip between too many files and folders, and up and down too many layers of abstraction — that’s *pain*. And yes, tests again. Speaking of those...

“Generally speaking, clean code is testable code!”

And in my experience, the easiest way to write 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.”

I’m careful comparing clean code to art, even though 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.

Clean code is as much an art as a plumber installing a toilet, or an auto mechanic performing an oil change is. Clean code is code written **professionally**. The main theme of professionalism? *Taking responsibility*.

“No magic. Simplicity.”

This echoes. Ask yourself if the complexity is related to the *essential* complexity of the problem or *accidentally* based on the way you’ve chosen to solve it.

“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 of your designs.

“Easily replaceable.”

How do you design code to be replaceable? By making it simple, readable, and providing tests.

And the last one,

“Code that I don’t curse its creator”

Yeah- we’ve all been there...

### According to the experts

Alright, now let’s look at how a few of the experts in our industry 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? Extreme Programming practitioners believe that lots of comments are *dirty*. You’ll hear about it soon enough in 9. Comments.

“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.

“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). We are reading (and writing) the book of our company’s solution space, so take your time and write it well.

### Clean coding standards

In the previous section, we sought to know the enemy: complexity.

In the real-world, dealing with customers, ever-changing requirements, and the myriad of approaches we can take to building products using code, perhaps the best *full-picture* approach we've discovered as an industry to tackling that complexity are the set of practices outlined in Extreme Programming (XP).

**One of the main values of XP is to build a *Shared Understanding* of the codebase with your team.** This means:

- A shared *coding standard* (ie: Coding standard)
- A shared *ownership of the codebase* (ie: Collective code ownership)
- A shared aspiration to use the *simplest design* possible (ie: Simple Design)
- A shared *understanding of the domain* and a preference to use the *language of the domain* to inform the names of our variables, methods, functions and classes (ie: System metaphor/Domain-Driven Design)

We'll examine approaches for each of these throughout the book, but in this chapter, let's focus specifically on the ideas behind a good coding standard and simple design. They're going to help us elucidate any confusion we may have about *what clean code is* and how to go about writing it.

### **What is a coding standard?**

By definition, **a coding standard is a collection of rules that pushes code towards a consistent style and approach in a codebase.**

This is where we make decisions about:

- how we'll structure and organize our project
- how we implement new features
- how we'll handle errors
- how we'll name things
- the formatting and style behind files, folders, variables, methods, and so-on
- how code gets committed into production (all code must have accompanying tests, must first be checked for style, etc)
- and anything else that developers may have varying opinions on (comments, file size, etc)

### **Why do we need coding standards?**

Coding standards (also sometimes called *coding conventions*) are helpful on any project, but they're most essential when we're working on a project that *is* or will ever be maintained or developed by more than one developer (and that's most projects with economic value associated).

The big magic word behind why coding standards are so important is **consistency**.

Out of the four ways you can detect complexity (*ripple, cognitive load, poor discoverability, and poor understandability*), consistency alone can improve at least three of the four (except *ripple*, which is by and large a coupling problem).

We, as humans are pattern-matching machines. When we see how something is done once something once, we take a subconscious note of it. When we see it twice, we have good

reason to believe that's the way things are done around these parks. When we see it three times, well — it should signal that it's as good as law.

All in all, consistency:

- Promotes reuse
- Reduces duplication
- Results in cohesive, readable, focused components
- ...and less code

## Simple Design

Humans tend to want to add elements instead of subtracting them to solve problems. This turns programming into something of a puzzle, or a creative activity, with experiments rather than constructs created through disciplined action.

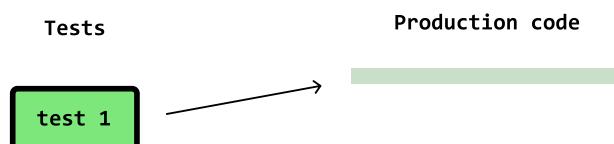
Simple Design is another main idea from XP that **represents the highest level principles for good design**. The four elements of simple design specify that clean code:

1. Runs all tests
2. Contains no duplication
3. Maximizes clarity
4. Has fewer elements

This is probably the **best definition that the experts agree upon** for *what clean code* is and how to achieve it. It's one thing to strive for clarity and readability, but we also need tests to make sure our code works and can be safely changed. And while we may have tests and clarity, we also want to avoid duplication and an unnecessarily large amount of code underneath our tests.

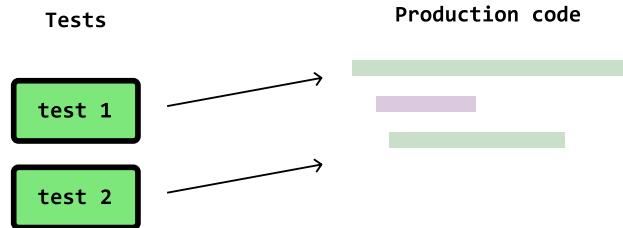
Throughout our journey in this book, we'll learn how to implement simple design by first learning the techniques, rules, and then the principles necessary for it to work. Once learned, here's how this works in practice.

We apply these rules in order.

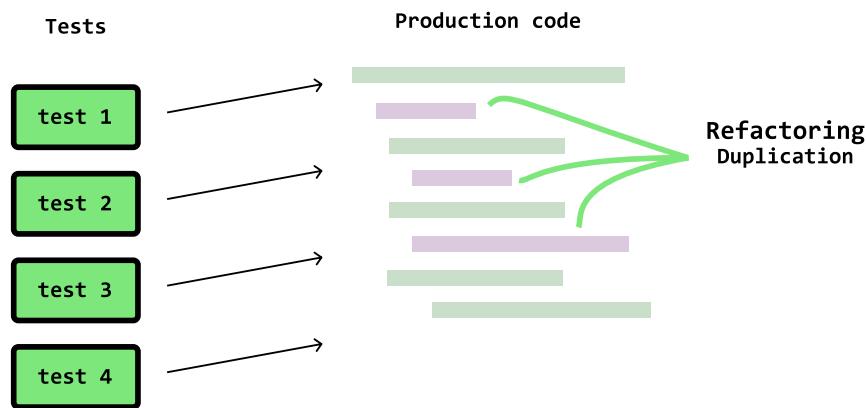


In practicing TDD, we start by writing a failing test, then the minimum amount of code **until the tests pass**. Next, we look at the design for *code smells* and refactor if we find any. If none, we move to the next test.

■ In Part IV: Test-Driven Development Basics, you'll master the TDD basics and it will become the standard way that production code comes to exist.

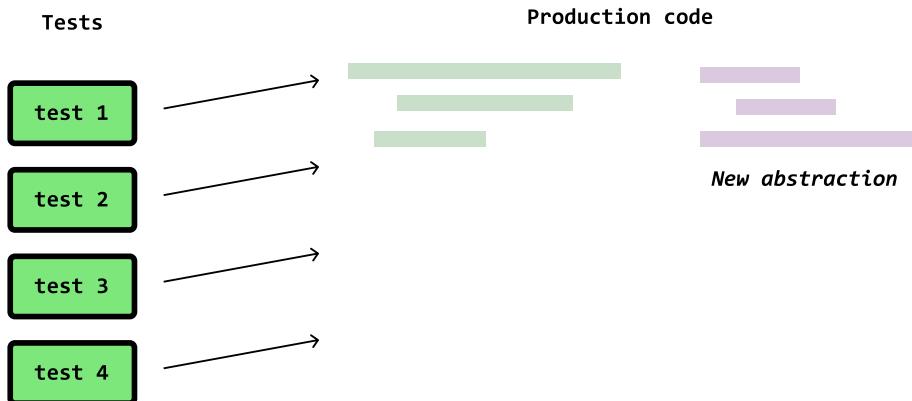


After a few tests, we will inevitably end up with **duplication**.



Duplication, which happens to be one of the first design problems that crops up, is an important phenomenon to prioritize fixing. Within the context of a 28. Test-Driven Development Workflow, we refactor the code, changing the structure — but not the behavior. We do this only when all the tests are already passing.

However, in cleaning up duplication, we introduce new abstractions to express the underlying concept.



Trading duplication for an abstraction moves us to the next step of simple design: **maximizing clarity**. Refactoring in an effective manner, we improve the *cohesion* of our codebase and make code simpler, more expressive, and easier to read. Refactoring poorly, we introduce *coupling* and implement wrong, confusing, or unnecessary abstractions.

The difference between someone who knows how to refactor poorly vs. how to refactor well stems directly from our understanding of object-oriented design, coupling, cohesion, and the effects any particular refactoring has on pushing the design closer to our north star of having **loose coupling & high cohesion**, and avoiding the introduction **tight coupling & low cohesion**.

- In Part V: Object-Oriented Design With Tests and Part VII: Design Principles , we learn various rules, techniques, and principles to call upon within the refactor step of TDD. These learnings help us refactor code in ways that nudge us closer to good coupling & cohesion — not away from it.

Finally, since less is more, if we're doing TDD, we **only introduce new elements** (lines of code, methods, functions, classes) when we absolutely have to. And that's when we encounter duplication, code smells, and the like.

These four elements of simple design are perhaps at the epitome of what we're trying to do with code that we leave for other developers — our future maintainers. To leave clear, tested, minimal, and succinct code.

- We revisit The Four Elements of Simple Design in more detail in Part VII: Design Principles.

## Emergence

Feedback techniques like Pair Programming and TDD promote **emergent design** — an approach to design that means that the design decisions *gradually* emerge.

We do this rather than a lot of massive upfront design decisions.

The major benefit of this is that we get **several opportunities** to correct bad design before it becomes too hard to change. And by doing so, we watch the design *emerge*.

No upfront mandatory UML created by the architect and handed to developers (this is called the *Ivory Tower Architect* and this process is called Big Design Up Front — we generally try to avoid doing this).

### **Structure vs. Developer Experience**

Design is a push and pull of priorities.

In the real world, office furniture manufacturers need to produce items as cost efficiently as possible. Conversely, consumers want good quality furniture, but at a reasonable price. The attributes at odds with each other are **cost vs. quality**.

#### **■ Examples of other important balances to maintain:**

- Note-taking = context vs. compression.
- Security system = security vs. ease of configuration.

Software design is no different. The two forces at play are **structure** and **developer experience**.

Software design is *structure vs developer experience*.

# **Software Design**

=



**Structure**

**vs.**

**Developer  
Experience**

### **Structure**

After a bit of time nursing your designs into existence with TDD, we'll have quite a bit of structure developed.

Structure is about **the decisions you've made about how we're going to approach certain problems in our codebase**. It's things like how we've decided we'll start the application, how we'll enforce authentication and authorization rules, where we'll package things (and what we'll package them together with). This is structure.

We have to be careful with how we implement structure. More structure presents a challenge for developers new to the codebase. Finding more code, more things to be accountable for, and more rules to need to learn — on projects with a lot of structure, the learning curve is said to be high.

A higher learning curve typically means it takes more time to learn your options for completing a task, how to perform the option, and how to know if you're doing things correctly.

This is not a great *developer experience*, at least initially until we figure things out.

## Developer experience

Where UX (user experience) design is about designing applications for end users, **DX (developer experience) design is about designing APIs, tools, languages, frameworks, and codebases for developers**.

In DX, we're most concerned with the top-level user developer goals.

From get up and running:

- How can I run this locally?
- How do I run it in debug mode?

To becoming productive with the codebase:

- What are all the things I can do?
- How easy is it to understand what this code does?
- How quickly can I find the area(s) in the code I need to change to add this new feature?
- What was the learning curve like?

And so on.

Developer Experience is a massive part of why we use some tools and why we don't use others. Along with other factors like social proof (that others are using it) and seeing that it's actively maintained, developer experience is a major *adoption* factor.

For example, think of the differences between each of these two types of technologies that do similar things, but have different developer experiences.

- Apollo Client vs urql
- React vs Angular
- Typed languages vs. untyped ones
- Strictly functional languages (absolutely no state) vs. loosely functional ones (we'll give you state, but we discourage it)

## Developer experience for developer tooling companies

Developer Experience is of utmost importance to companies that build developer tooling and maintain public APIs like GitHub, Stripe, Netlify, and the company that I work at during the time of writing this book — Apollo GraphQL.

Developer Experience is *really* important for these companies. For them, it's important because it's the experience that developers have building on top of their APIs is tied directly to important metrics like revenue.

- eg. If developers enjoy building payments into apps using Stripe, then Stripe has more customers and makes more money through the 1% transaction fee
- eg. If developers succeed with Apollo, then they will feel empowered to build out their data graphs and will eventually need tools like schema, visibility, and graph management — Apollo's cloud platform offers these as paid features

It's easy to see how developer experience is important for these companies, right?

The thing is, it's not really that much different, even for non-developer tooling focused companies. Revenue is still a part of the equation — just indirectly.

## APIs aren't just URLs

Let's get something straight first. What is an API?

A common mental model is that APIs are just URLs that you connect to in order to fetch data or invoke commands to a backend service.

```
// A RESTful API example
fetch('http://movies.com/api/movies/popular')
  .then(response => response.json())
  .then(data => console.log(data));
```

That's a useful model when you're just getting started, but GraphQL and RESTful APIs are just one specific form of APIs (I'd call them HTTP APIs).

An API is any code *intended to be used* by another developer.

This means that public interfaces to abstractions (classes, functions, libraries) are APIs.

Take the popular JavaScript utility library lodash.

```
import * from 'lodash'

_.chunk(['a', 'b', 'c', 'd'], 2);
// => [['a', 'b'], ['c', 'd']]
```

Lodash exposes a large number of utility functions through its public API. For the API designers, careful consideration has to go into the design of the public interface since developers will be relying on it to do their work. Things have to be clear, intuitive, and expose little opportunity for confusion.

## Developer experience is important for all companies

This truth about APIs means that so many of us are also API designers. I would wager that if you're a developer who ever expects your code to be used by another developer (current team member or future maintainers count), that you too, are an API designer.

Like the lodash maintainers, we have to place a careful amount of consideration into the abstractions we create as *infrastructure* in our projects; understanding if their intent is clear, if they expose surface area for misinterpretation, confusion, or can be used in ways we didn't intend them to.

For example, in a front-end application that primarily relies on RESTful APIs, it may be sensible to create a `HTTPService` class to consolidate the way that we ask for data using an HTTP client like Axios. That way, we don't have to keep on creating new instances of Axios for each resource. An initial attempt may look like this:

```
class HTTPService {  
    http: AxiosInstance;  
  
    initialize (baseURL: string) {  
        this.http = axios.create({  
            baseURL  
        });  
    }  
}
```

In this example, we intend to keep an `AxiosInstance` as the `http` instance variable, but to set that up, we need to first call `initialize` and pass in the `baseURL`.

Using this infrastructure, if I wanted to create a `UserService` to get access to an `http` instance, I might merely create a new class, extend the `HTTPService`, and start writing my methods:

```
class UserService extends HTTPService {  
    public getUsers () {  
        return this.http.get<User[]>("/users");  
    }  
}
```

If I did nothing but these two things, I'd notice that all of my requests would fail because the `http` object wasn't initialized.

```
const userService = new UserService();  
userService.getUsers(); // Fail!
```

To use this correctly, I'd have to first call `initialize` on any subclass of `HTTPService`.

```
const userService = new UserService();  
  
// Easy to forget to call  
userService.initialize("http://movies.com/api");  
  
userService.getUsers(); // OK [User, User, User]
```

Not great, right? This abstraction *leaks*. There are things you need to know about the *internals* of the abstraction to use it without error.

To improve this design, we could use the language to impose structural rules. In Object-Oriented Programming, you can enforce constraints on object creation through either static factory methods or the constructor keyword. We'll use the constructor here to force the user to supply the baseURL at the time of object creation so that it can be fully initialized.

```
class HTTPService {  
    http: AxiosInstance;  
  
    // Now the subclass has to provide the baseURL  
    constructor (baseURL: string) {  
        this.http = axios.create({  
            baseURL  
        });  
    }  
}
```

API usage now looks like:

```
const userService = new UserService("http://movies.com/api");  
  
userService.getUsers(); // OK [User, User, User]
```

This structural change creates a rule but it forces the API consumer to go down a happy path, reducing the possibility of confusion.

■ **Object design principle:** Avoid partial object creation — ensure the object is fully created or not created at all.

When you design an abstraction here, or make a decisions on the coding conventions people are going to use there, you're deciding on the APIs that your team is going to rely on. Hopefully they're decisions that have great DX.

■ **Read:** “Developer experience is the next competitive front in enterprise tech” by *Protocol*

### Balancing structure vs. developer experience

The thing is — we need both. We need structure but we also need developer experience. We're looking for the **mean between two extremes**.

“Moral behavior is the mean between two extremes - at one end is excess, at the other deficiency. Find a moderate position between those two extremes, and you will be acting morally.” — Aristotle in *Nicomachean Ethics*

Some of the best technology stack decisions you can make lie right in that sweet spot.

- React **but with TypeScript**
- A typed language **but with the option to relax types**

- A functional language **but with the option to use state if need be (discouraged though)**

### **Structure vs. Developer Experience in Practice: Angular and React**

Angular is a front-end framework that, if you look closely, is thoughtfully crafted to promote the use of object-oriented software design best practices. To become productive with Angular means learning TypeScript, brushing up on your OOP skills, and learning the *Angular-way* to get things done.

For example, in a previous discussion, we noted that Angular has a *correct* way to run code before a route is loaded, and that's a Route Guard.

React (a UI library) on the other hand — is a lot more functional and flexible. Developers are capable of becoming productive with React a lot faster because the API is smaller, there's less to learn, and it contains fewer structural policies that mandate how you can write code within it.

The challenge is, of course, **maintainability**. What if you have 10 developers working on the same project? How would you enforce structure in a library designed to have as positive of a developer experience as possible?

Perhaps if you were to use React and you needed a nice structure/developer experience middle ground, writing your React code with TypeScript could be the way to get there.

### **Structure vs. Developer Experience in Practice: Object-Oriented vs. Functional Programming**

Object-Oriented programming is hard because it's easy to create structure, but hard to make things discoverable, understandable, and delightful to use.

Functional programming is also hard, but that's because it's delightful and easy to get started, but hard to impose structure.

### **Conclusion**

How do we get better at clean code? Well, *practice*. We've got to practice the *structural* side of design, which is going to be your TDD, DDD, and other XP practices. We'll be using design principles, patterns, and architectural patterns to make better structural decisions too. So we've got that covered.

What about the other side? How do we improve the *developer experience* side of design? How to do we learn how to succeed at some common things like organizing files, leaving productive comments, and naming things? How do we learn to how to do harder things like designing intuitive and expressive public interfaces (APIs)?

In the next chapter, we'll learn about human-centered design principles. They provide the baseline philosophy to help us **understand how we understand**. Armed with this capability, we can decide a land realize if someone else would struggle with what we've done or not.

## Summary

- The definition of Simple Design paints the most essential picture of what it means for code to be clean: runs the tests, no duplication, maximizes clarity, has fewer elements.
- XP technical practices promote *emergent design* as a way to curb complexity by gradually adding more features and structure to a program. We periodically contemplate and refactor a design guided by tests.
- The myriad of design patterns, principles, and architectural patterns promote better structural decisions.
- The unhinged addition of features and structure tends to have an adverse effect on developer experience — the human-side of design.

## Exercises

■ Coming soon!

## References

### Articles

- <https://blog.jbrains.ca/permalink/the-four-elements-of-simple-design>
- <http://arlobelshee.com/good-naming-is-a-process-not-a-single-step/>
- <https://www.nature.com/articles/d41586-021-00592-o>
- <https://www.sparknotes.com/philosophy/aristotle/section8/#:~:text=Aristotle defines moral virtue as,than through reasoning and instruction>

## Books

- Badass: Making Users Awesome by Kathy Sierra
- The Design of Everyday Things by Don Norman

## 5. Human-Centered Design For Developers

One thing I love about design principles (like SOLID, YAGNI, or SoC) is that they become *engrained* into your professional way of ensuring a baseline of good structural code quality.

If you've ever run a situation where you weren't sure if you were doing things correctly, design principles sit in the back of your mind like a voice of reason.

Rules are *shortcuts* to principles. In this book, we present a number of rules, but only as an easier way to remember how to implement the underlying principles. As we get better over time, we'll feel a lot more confident breaking the rules. This comes when we know what the implications of breaking them are, and what we need to do to pull the code back into *best practice* zone.

Sometimes you truly *do* need to break the rules. But it's important to first know the rules and their implications before we can decide when to break 'em.

Martin Fowler once said,

“...any fool can write code that a computer can understand. Good programmers write code that *humans* can understand”

Wouldn’t it be amazing if there was a **principled way** for us to write code that could be understood by humans? A way for us to intrinsically understand how to write clean code instead of just following someone else’s coding conventions?

It turns out there *is*.

I’m going to introduce you to the basics of Human-Centered Design and how you can use it to optimize your code to be understood by humans.

We’re going to learn the psychology behind how humans discover and understand what can be done and how things work.

Then we’re going to learn the **seven fundamental principles of design**. Just like how there are *software* design principles, there are *human-centered* design principles. We can use them to optimize the **discoverability** and **understandability** of our code.

Armed with this knowledge, we’ll be able to determine if code is clean and easy to work with, what makes it *challenging* to use, and how to improve it so that we can keep the developer experience of our coworkers and future maintainers high.

*But Khalil, can’t I just read Uncle Bob’s “Clean Code”?*

While I do recommend reading “Clean Code”, this chapter is going to act as a part of the philosophy — the baseline, behind how we reason about our *own coding standards*. It compliments the rest of the structural software design principles we cover in this book.

Lastly, after having established the designer’s philosophy, it will become possible and much easier for us to do two things: design expressive and understandable APIs, and use the **vocabulary** necessary to engage in principled discussions about how to balance code that is **structurally correct** yet still **understandable to humans**.

Let’s get into it.

## Chapter goals

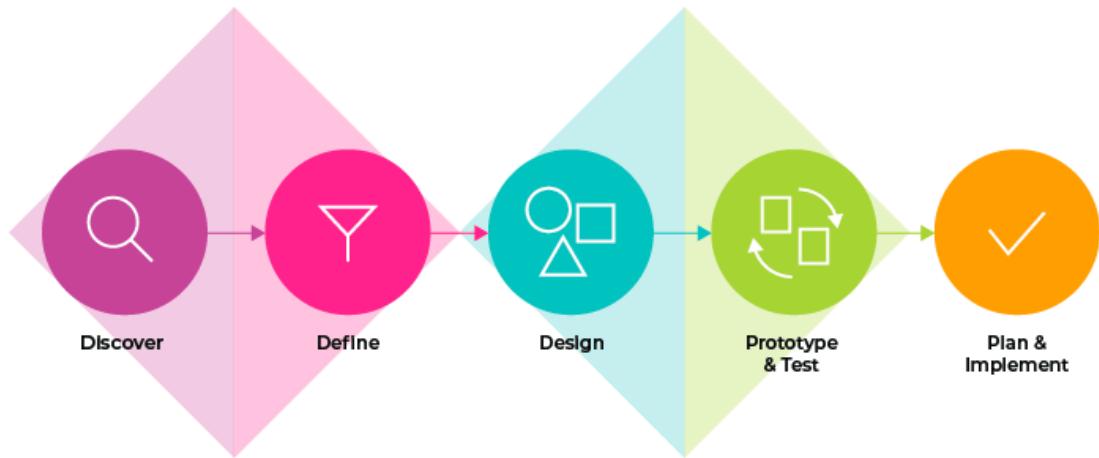
- Learn the basics of Human-Centered Design and understand how we understand how to use things
- Learn how to apply the seven principles of HCD to software development
- Learn how to use affordances, signifiers, constraints, and mapping to optimize for **discoverability**
- Learn how to use feedback and conceptual models to optimize for **understanding**

## Human-Centered Design

### What is it?

HCD is a design philosophy that puts the users’ needs, behavior, characteristics, pain points, and motivations first.

For example, instead of just designing something that works, we brainstorm, test, and implement based on what we know about the humans we’re designing for.



Copyright 2019 | Outwitly Inc. | HCD Process

— Human Centered Design approach (from Outwitly Inc)

### **How is this helpful for us?**

Well, we're in a unique design situation here. Normally, if we were building an app for a particular set of users, like a dating app or something, we'd have to get out there and actually interview the users to learn more about them and find out about their needs.

But since we're trying to figure out how to write code so that it can be easily maintained by our coworkers and future maintainers, this process is a lot easier. We're *also* the users who need clean code in the first place.

Let's think about it. What are the most common use cases as a *developer*, that you need to do your job well?

### **Developer use cases**

- I need to know how to run the app
- I need to know how to run the tests
- I need to understand how and why code is organized the way it is
- I need to understand how the domain is expressed in code and where policy belongs
- I need to understand how to add a new feature and know 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 be able to run through the code in debug mode to test a feature or debug a bug
- I need to be able to improve code over time without breaking anything

This is great.

If you have any other goals that you can think of, feel free to copy this list and add it. These are the things that we're going to be focused on optimizing.

The next thing we need to learn is the psychology behind how humans learn to use things to accomplish their goals.

### How We Figure Things Out — The Psychology of Human Action

How do humans figure out how to use new things? How do we learn what we can do? How do we know that what we just did worked?

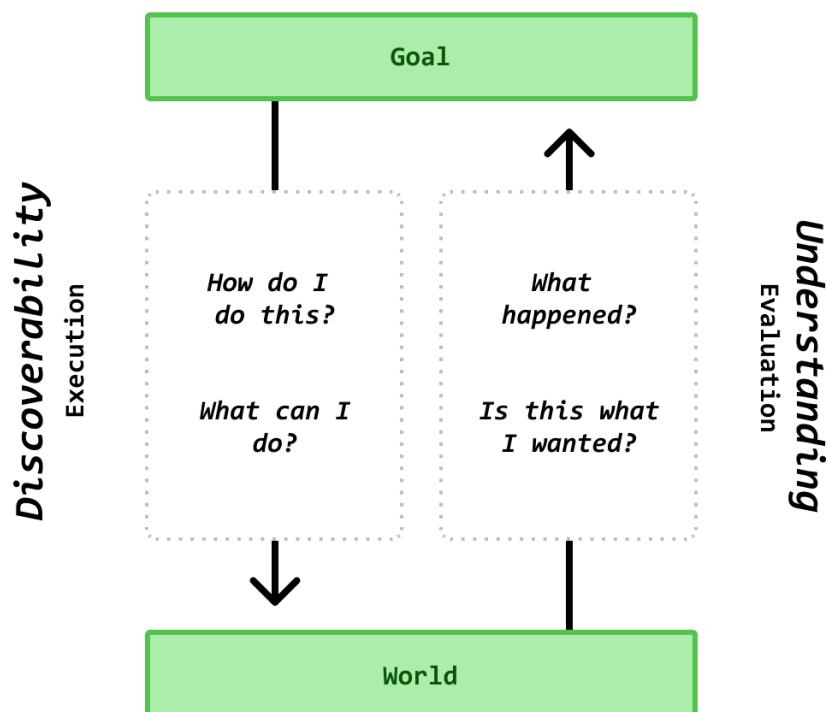
Since we're adopting the role of the designer, our job is to help users (other software developers, in this case) accomplish their goals by optimizing two things: **discoverability** and **understanding**.

“Two of the most important characteristics of good design are **discoverability** and **understanding**.” — Don Norman, author of “The Design of Everyday Things”

### Discoverability & understanding

Let's say you've started a new job and you're looking at a codebase you've never seen before in your life. How quickly can you discover how the project is organized? How long does it take you to understand *what's what*? How long does it take you to become productive?

Whenever a person encounters a new device, they have to go through two swim-lanes to figure out how to use it to accomplish their goals.



On the left-hand side of the diagram above, we have the “**Gulf of Execution**” which is predominantly about *discovering* what can be done. On the right-hand side, we have the “**Gulf of Evaluation**” which is about *understanding* what we’ve done.

If all goes well and we flow right on through this, we end up with a good **conceptual model**, a *true understanding* of what we’re working with. We’re good to go.

But what happens when things don’t go the way we want them to? What happens if the design isn’t great and we get stuck somewhere along these lanes?

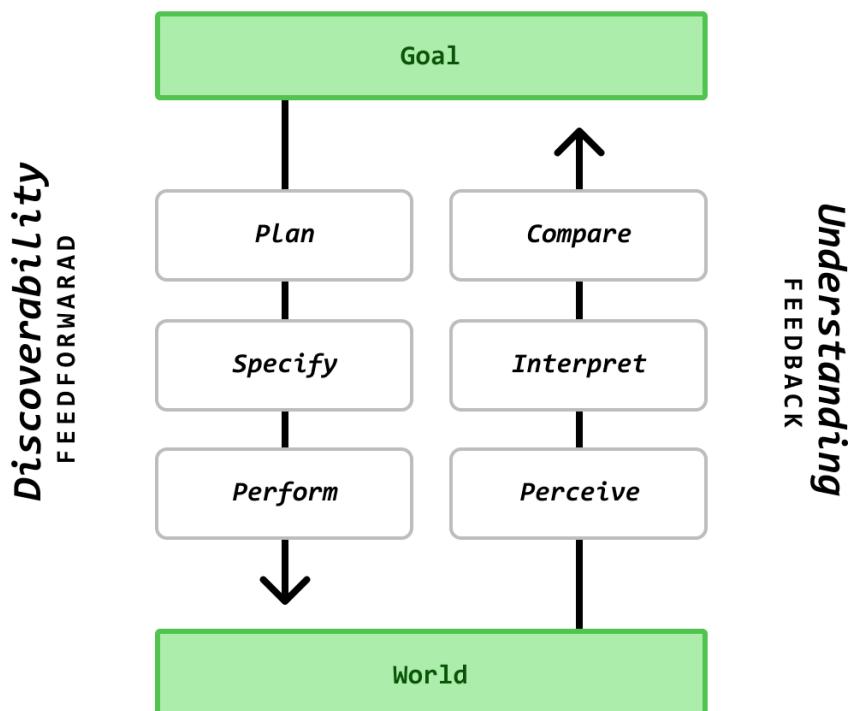
For example, let’s say we just started working on an existing front-end project and we were asked to change the styling on a particular component that we didn’t know the name of — we just knew where it was on the screen. Assume it takes you *way too long* to find that component. Or assume you find it, but the styles don’t seem to be updating when you play with the CSS.

That sounds like a pretty poor experience so far, right?

There are some design deficiencies here to be improved. To do that, we need to go deeper into these two lanes of discoverability and understanding.

### The 7 Stages of Action

The **Seven Stages of Action** is a *general* sequence of psychology that humans follow when they interact with objects in the world to accomplish their goals.



It goes:

1. Goal (form the goal)
2. Plan (the action)
3. Specify (an action sequence)
4. Perform (the action sequence)
5. Perceive (the state of the world)
6. Interpret (the perception)
7. Compare (the outcome with the goal)"

— from “The Design of Everyday Things” by Don Norman.

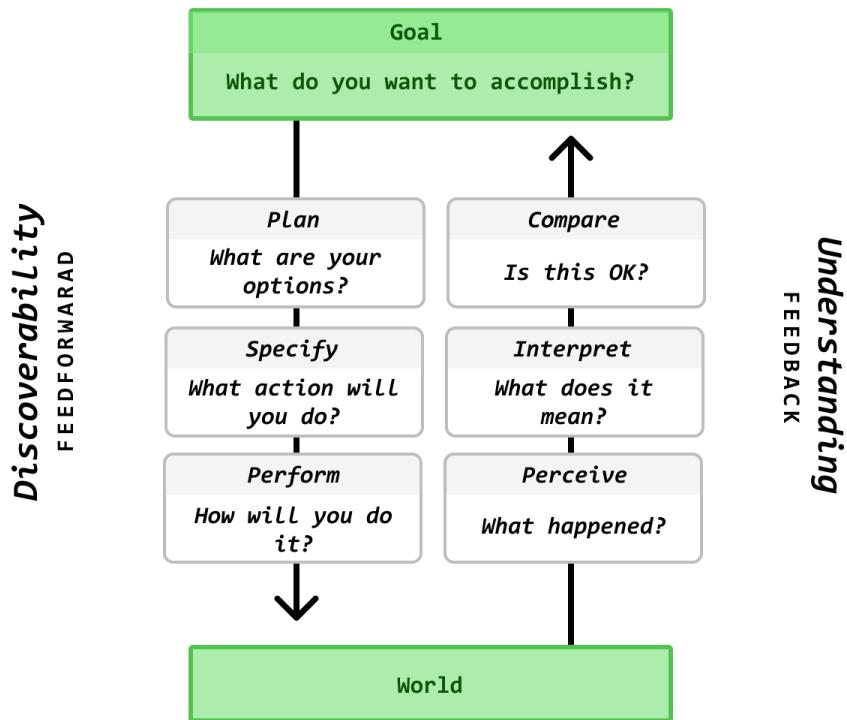
Sometimes, we progress through these steps super quickly. For example, most of us know how to use pens and pencils. For that, it’s pretty much muscle memory and nearly impossible to recognize what stage you’re at.

Other tasks, like adding a new feature to a legacy codebase, aren’t as speedy. In situations like this, when you’re still *discovering* and *understanding* the object (the codebase), you may find yourself spending a noticeable amount of time at each stage.

*Planning, specifying, and performing* are all forms of **feedforward**. We’re utilizing clues to lead us to perform some sort of action.

On the other end, *perceiving, interpreting, and comparing* the result of that action all belong under the **feedback** umbrella. Feedback leads to **true understanding**: the conceptual model.

Now that we have labels \*\*for each of these stages, we can help identify the problems by asking specific questions at each step.



For example, during the *planning* stage of accomplishing some goal, we can ask ourselves (or another person) “what are your options”? At this point, if we notice that the answer to that question feels too hard, it’s probably a sign that the **options aren’t visible, aren’t readily discoverable, or aren’t clear as to what they control**.

Imagine that you’re over at your musician friend’s house for a jam session. They’re playing the electric guitar and they’ve decided they want to try to emulate the guitar sounds at the beginning of The Birthday Party’s 1981 tune, “*The Friend Catcher*”.

“Hey, could you give me some more reverb?”

Your new goal — add reverb to the guitar tone. What the heck is reverb? What’s responsible for that? Looking around the room, you spot an amplifier, a bunch of dirty socks, and something that looks like this:



What is this monstrosity? (It's a pedal board). Perhaps that's what's responsible for this *reverb* of which we speak.

"Quit messin' around, will ya? Turn the reverb up!"

Which of these knobs to turn? Hmm... After looking for 15 seconds, you spot one that says "Studio Reverb" on it. It must be that one.



Oh boy. *Blend, Reverb Time, Dampening/Tone...* What are all these options?

This might seem like a comical real-world example, but it draws incredible parallels for any developer that needs to learn a new library, framework, API, or codebase.

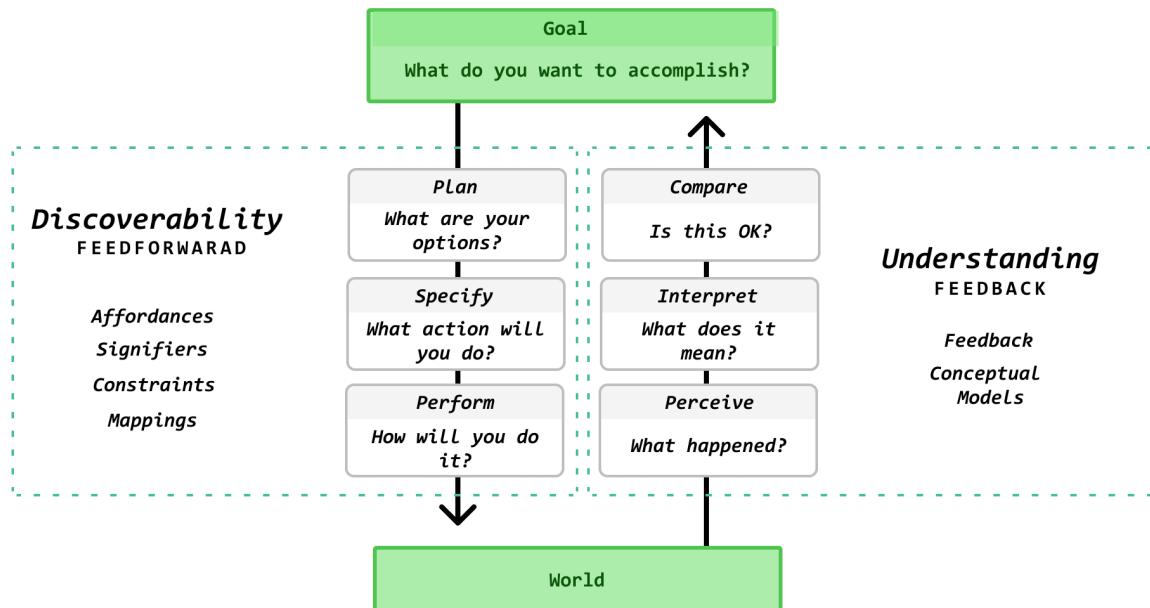
How do we design things so that we struggle less and understand faster? Finally, this leads us to the good part — the **fundamental principles of design**.

### Fundamental Principles of Design

At any point in time, you should be able to answer all of the questions in the seven stages of action. When you're stuck, we can remedy the design issue by stepping a level deeper into the framework and consulting the fundamental principles of design.

■ **The seven fundamental principles of design:** Discoverability, affordances, signifiers, constraints, mappings, feedback, and conceptual models.

Here's a depiction of where they fit into the model.



On the left-hand side, to improve **discoverability**, we optimize the feedforward mechanism with the design principles: **affordances**, **signifiers**, **constraints**, and **mapping**.

And on the right-hand side, to improve **understanding**, we need to optimize **feedback**, which — if done correctly, helps the user build a useful **conceptual model** to work with.

---

The last thing for us to discuss before we learn about each of the fundamental design principles, how they work in real-life, and how we can apply them to software design is the concept of **knowledge** and where it comes from.

### Knowledge in the Head vs. World

Most of us understand what to do when we see a *handle*.



We've all learned that handles are things that can be *gripped*.

When you see a handle on an object you've never come across before, like maybe a new style of car, a weird art piece, hell — even a *ferocious dragon*; when you see a handle, it's hard to stop your brain from thinking "**that's for gripping**, right"?

Of course. *Though, if you come across any sort of dragon on your walk to the grocery store, you might be better off just running the opposite direction.*

Each of the design principles we just mentioned capitalizes on our knowledge. Knowledge is involved in how we discover what we can do and what things mean.

But where does *knowledge* come from anyway?

### **Knowledge in the Head**

The head.

# Knowledge in the Head

 *Fast (like RAM)*

 *Hard at first, but practiced*

 *Muscle memory*

 *Experiential and Learned*

*Invisible*



**Based on conceptual models and constraints (cultural, semantic, logical)**

Knowledge in our *head* can be **fast, efficient, and treated like muscle memory**, but only if we've put in the work to make it so.

Knowledge in the head is stuff that we know. It's the stuff we can do with little effort because we've invested the time and energy into learning.

For example, driving a car has become muscle memory for a lot of people. But if you were to think back to when you first started driving, you might recall your actions were substantially more calculated.

Another example is hobbies. Some people produce music; and while that may be hard for most people, it's actually quite straightforward for music producers.

Also important to note is that if you don't drive, work on your hobby, or code for a long time, you start to lose those skills. Things fall out of your working knowledge.

Knowledge in the head comes from **conceptual models** (understanding) and **constraints** (cultural, semantic, and logical ones).

## Knowledge in the World

The world.

# Knowledge in the World

 *Interpreted*

 *Easy to use at first*

 *Can be reminded*

 *Learning not required*

*Must be visible*



**Based on affordances, signifiers, mappings, and physical constraints**

Knowledge in the *world* is easy to use and interpret, which can also make it super fast. The key thing about knowledge in the world is that it *doesn't have to be learned*.

The best examples of knowledge in the world? Reminders.

For example, how would you remember to collect your lunch from the fridge before you leave for work?

You'd probably want to put something along your path to getting ready, like a sticky note, to remind yourself. In this case, you don't need to *learn* anything, knowledge (that you should remember to collect your lunch) is in the *world*, and as long as you can perceive it, you know what to do.

This is also why most ATMs guide you along what steps to take next. Imagine you had to *learn* how to take out money. Imagine a world where withdrawing money from an ATM required the same amount of tribal knowledge that changing a tire does.

Knowlege in the world is expressed within **affordances, signifiers, mappings** and **physical constraints**.

**How is this helpful for us?**

Why are we talking about this?

As a reminder, it is in the very definition of our goal as software developers to make it easy and cost-efficient for other developers to understand how to work with our code. That is — we are in the business of creating **human-centered codebases**.

There are some activities that you want developers to spend time *learning* and there are some that you want to take absolutely *no time* to learn.

For example, things that you'd like developers to utilize knowledge in the *head* for and spend time *learning*:

- **The domain.** You can't force this. Developers need to spend time and learn the domain to make good design decisions.
- **The architecture of the application.** You want developers to have a solid understanding of how we're building the application. It may take some time to fully understand, but that's a necessary learning, because it would be dangerous for another developer to develop features without an understanding of the architecture that contains them. This also means keeping the architecture as simple as possible.
- **The features within the project, what they do, and how to add new ones.** You want developers to easily *discover* features, and that's trivial if we're using screaming architecture (see Organizing Things). As for *what the features do* and how to add new ones, this takes not only learning the domain, but what technical constructs we've used to realize the features. In an object-oriented context, the simplest way to do this is to write BDD-style acceptance tests that expose the intended behaviour of the features using the language of the domain.

And conversely, here are a few things that you'd like developers to spend zero time on *learning* and you'd like them to merely need to *interpret* and *react* to:

- **Type errors.** By using TypeScript, we force other developers to deal with errors as they occur, at compile time.
- **Failing tests.** How are you supposed to keep track of if you broke something or not? It's not reasonable to expect you to test every feature manually. Instead, we practice TDD, and while we're coding, if we notice that we've done something to put the app into a broken state, we know that we now have to deal with the issue right there at that moment in time.
- **An easy-to-reason-about folder structure.** The names of the files and folders (signifiers) should help guide us towards a feature, component, or module of code that we're looking for. It's not reasonable to ask someone to *memorize* the folder structure. It should be designed in a way that we can rely on the current directory to figure out how to maneuver towards where we want to be.
- **Pre-commit checks.** How do you get developers to remember to run the tests and lint their code before they commit? You don't. You build in reminders instead. We can enforce pre-commit policy with tools like Husky. And if tests fail or code needs to get cleaned up, well — now they're forced to deal with it before they can continue (and that's a physical constraint).

Let's get into the different design principles, shall we?

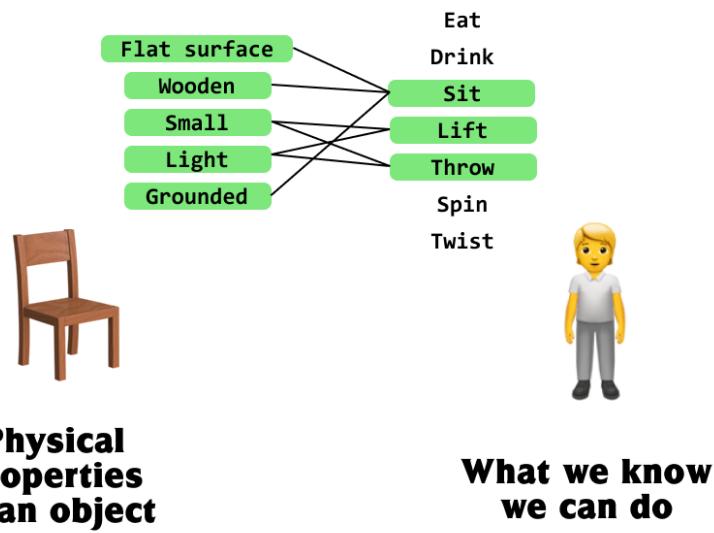
## Affordances

Communicates the intended purpose of an object based on **what we know we can do**, and the hints given to us by **the physical attributes of the object**.

In other words, affordances are visual cues, properties, or attributes of the object that tells you how something works.

# Affordances

**A relationship between what we know we can do AND an object's physical properties**



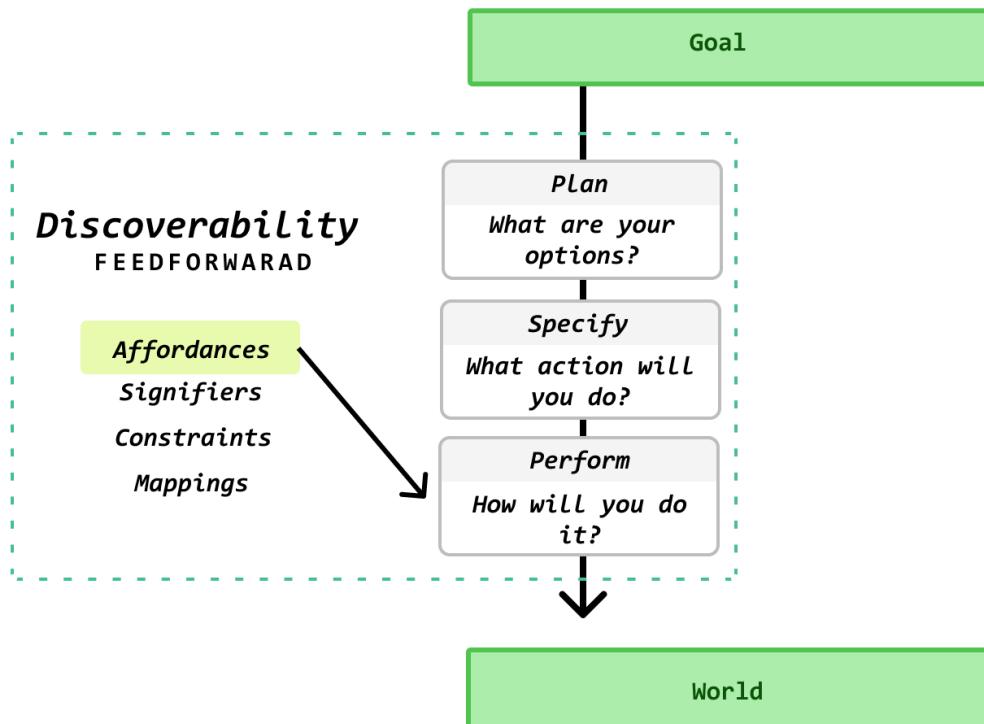
## Real-life examples

**Chairs.** A chair *is for support*. So you might say that a chair *affords* sitting. A *light* chair \*\*might have an affordance for lifting or being carried — one that's *heavy* might **not** afford lifting or being carried.

**Glass.** Glass is transparent, so it has an affordance for seeing through. It also has an anti-affordance, which limits what is possible, and that's to block air-flow and disallow things to pass through it. The challenge with this is that if glass is transparent and we don't notice it's there, we might not notice the anti-affordance. That's how people walk into glass doors. This happens. To remedy this, we sometimes use signifiers (text, images, sounds); they act as clues that help aid along the discovery of affordances.

## Why is this useful?

If we haven't seen something before, even though it's new, if it has physical properties that we're familiar with, then we may be able to quickly **discover what we may be able to do with it**.



For example, I have a typewriter at home.

Though not everyone has used a typewriter, there are *some* familiar aspects such as the *keys*.



Keys are **buttons**, and buttons afford *pushing*. Therefore, you'd likely be able to figure out how to use a typewriter pretty quickly.

There are some *less* familiar aspects of the typewriter, however.

Today, while using traditional keyboards, to get to the next line — to perform a **carriage return** operation — you'd press the ENTER key. There is no such concept of an ENTER key on a typewriter.

When you get to the end of a line on a typewriter, you'll hear a *ding* (feedback) which signals that we need to do something to perform a carriage return to move to the beginning of the next line.

What you *may* notice is that there is a **bar** that sticks out, and at this point, it'll be pretty much right in the middle of your view.

Because we *have* seen bars before, and we know that bars have an affordance for *pushing*, it's likely that we'll try that and see what happens.

We push the bar, and voila — we're on the next line.

Again, if the physical properties are visible and familiar, we *will* know what actions we can perform.

That's what affordances do. Help us maneuver new objects.

## Affordances in programming languages

Things that are possible in one language may not be possible in another.

When programming languages express concepts differently, or are missing features that others we're familiar with *do have*, that reduces the set of options we know of to accomplish our goal.

For example, if someone told you to build a quick in-memory solution to keep track of users that visited a site, recording them by userId, how might we do that?

An efficient solution to this problem might be to use a hash table.

JavaScript developers are familiar with the fact that *objects*, which can be created with curly braces, can be used as hash tables.

```
// Hashtable in JavaScript
const visitors = {};

function saveVisitor (userId) {
    visitors[userId] = userId;
}
```

If we were to move to using Java though, we'd be hard pressed to accomplish this in a similar way.

In Java, we need to use classes. Objects take a much different form in Java. We can't just use curly braces to do this.

```
import java.util.*;

class HashTableDemo {
    public Hashtable<String, String> visitors;

    public void saveVisitor (String userId) {
        this.visitors.put(userId, userId);
    }

    public static void main (String args[]) {
        // creating a hash table
        this.visitors = new Hashtable<String, String>();
    }
}
```

Java is much more explicit and verbose.

While there are convenience methods, and this code may be more intentional, the take-away is that a JavaScript developer may have trouble accomplishing this task in Java if they weren't familiar with the Hashtable class.

This is also a foundational argument behind why I recommend using a typed language (like TypeScript) instead of a non-typed one: the affordances for abstractions (interfaces, abstract classes) are expressive and a part of the language. We need abstraction for the design techniques we use in this book. More on this in [11. Types](#).

## Affordances in design patterns

Some languages have a good affordances for certain design patterns, some have poor affordances, and sometimes, it's completely impossible.

To implement the **observer pattern** in Node.js is trivial. We could just use the `EventEmitter` API.

On the extreme side, to implement the **abstract factory pattern** in JavaScript is nearly impossible to do naturally because JavaScript doesn't have the `abstract` keyword.

## How to do affordances well

Since most of us are coding with the goal of **writing code in a way that makes it easy for future maintainers to understand, improve, add, change, and remove features**, we have to ask ourselves, what are the necessary affordances?

What are the physical properties that will give our code affordances to be *understood, changed, improved, and added onto*?

Well, code isn't really **physical**. It's more or less *invisible*. But the *form* of code is **documentation & reading**.

At a high-level, as far away from the actual code as possible, we *feedforward with*:

- Repo docs that shows examples, how to install, run, run the tests, contribute, and learn more about the project domain and architecture (see [7. Documentation & Repositories](#))
- Folder structure and file/folder names (see [6. Organizing things](#))
- BDD-style tests that explains what the code does (see [Part X: Advanced Test-Driven Development](#))

At the next level down, we're concerned with:

- Formatting, style, file size (see [10. Formatting & Style](#))
- Modular units (see [8. Architectural Principles](#))
- Comments when necessary (see [9. Comments](#))

## Signifiers

When an affordance isn't discoverable, signifiers help. They are a mark or sound, and can communicate intended behavior to a person.

Signifiers come from something called *semiotics* — the study of signs and how meaning is derived from them.

A sign can be thought of as a two-part relationship between a *signifier* and a *signified*.

- Signifier = the physical thing you see, hear, sense, imagine
- Signified = the mental content that the signifier produces

### Real-life examples

There are both **intentional** and **accidental** signifiers.

An *intentional* signifier is when we've consciously added something to help users discover what they can do with an object of some sort.

- e.g: A PUSH or PULL label on a door

Whereas an *accidental* signifier is one that occurs when it is just accidentally created.

- e.g: A trail or a path made visible by people walking along it over a long period of time. This signals that it's a path that can be taken — maybe even a *preferred* path instead of the *intentionally* created one.

One last example I'll use here is the concept of **purple**. Take the English name for the color *purple* as a sign.

If the *written word* “*p.u.r.p.l.e*” is the signifier — or if I was to speak it aloud, the sound “purr-puHL”, that's a signifier for purple as well. Cool. So then what's being conveyed? What is the mental content produced by that? It's the concept of the color purple. That's what's signified.

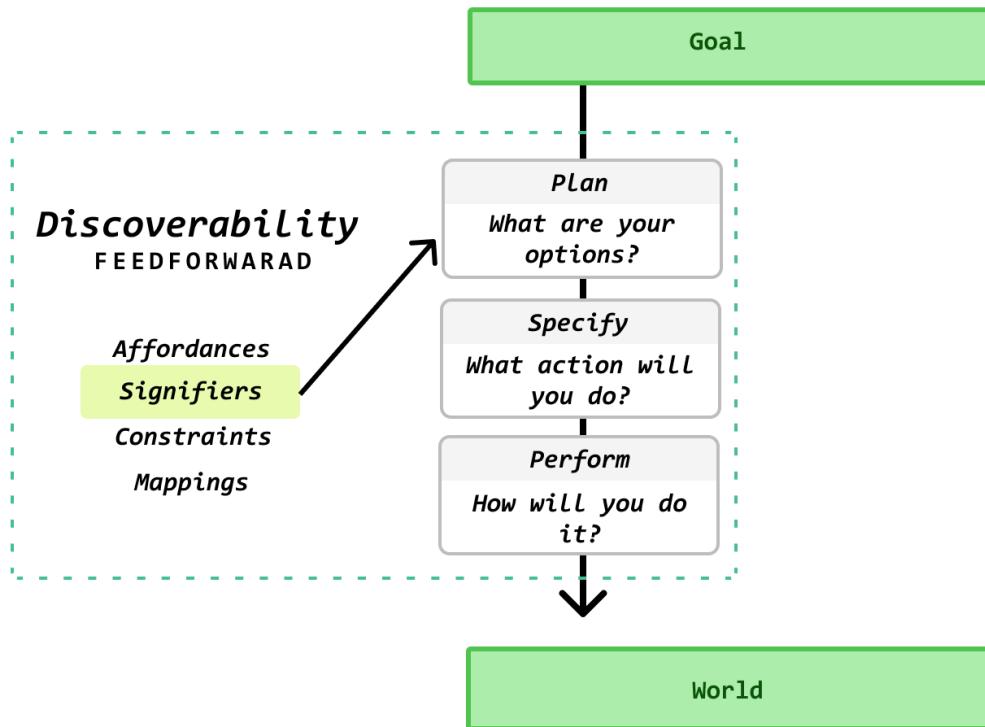
## Sign = Signifier + Signified

The physical thing you see, hear, sense, imagine	The mental content produced
“Purple” or 	The concept, <i>purple</i>
“Controller”	The concept of a class that handles web requests

This is going to be different in your brain, my brain, and everyone else's brain. But you know what? That happens a lot when we design abstractions, doesn't it? We think we're designing something that signifies our ideal with crisp adequacy — only to realize

### Why is this useful?

If affordances signify which actions are possible, signifiers (often just labels), communicate **where the action should take place** and what the options are.



Signifiers are a *cheap* way to enhance the *discoverability* of *affordances* and make sure that the *feedback* is understood clearly by the user.

### Intentional signifiers in software development

- **Design patterns.** There are two ways to recognize a design pattern in code. The first way is to see the code, the structure, and recognize that it's a pattern. For example, if you ever see a private constructor, and a static create method, it's likely we're looking at a **factory method**. The second way to recognize a pattern is to see the pattern name in the actual *name* of the class or function, like StudentController. Both of these examples have the *pattern* as the signifier, whether it's the name or the entire pattern's code itself, but what is *signified* is an understanding of "understanding of **what [the programmer] can (and also, sometimes more importantly, cannot) do** with that code."
- **Comments.** Comments that describe *what* something is, rather than allowing the code to describe *what* it does (we generally discourage these).
- **Tests.** If we do BDD-style tests, we signify the problems that the code solves. This is the most obvious, intentional clue for demonstrating **what the code does**.

### Accidental signifiers in software development

- Dead, commented out code, or todo comments left for too long. These signify that the related code might be a work in progress.
- Too many nested conditionals — signifies to us that this is a place that may contain a lot of application or domain logic.
- Thin controllers — signifies that the application and domain logic is being handled elsewhere and that the architecture may be more robust than a simple MVC app.
- Code smells and anti-patterns. They signify that the code has issues for us to tend to.

## How to use signifiers well

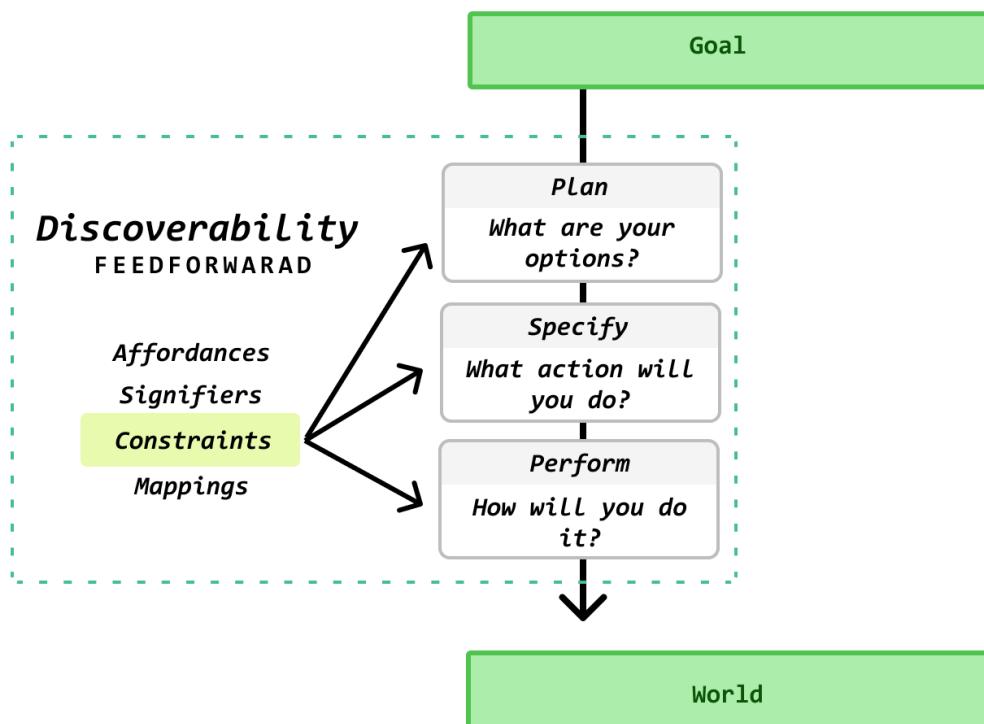
Use good names for files, folders, variables and classes, write tests, make error handling explicit, and utilize design patterns when appropriate.

Next. Constraints.

## Constraints

Constraints limit the set of possible actions we can take in certain situations

Constraints can be physical, cultural, semantic, and logical. Each type of constraint limits the set of possible action and drives us towards things *are appropriate* to do at that time.



## Real-life examples (physical constraints)

Restricts the way in which objects can be moved or manipulated.

- eg.: We can't force puzzle pieces to fit in spots that they physically don't belong.
- eg.: You can nail a *nail* into the wall with a hammer. You can't nail a *hammer* into the wall with a *nail*.

### Real-life examples (cultural constraints)

Constrains a situation by providing a set of allowable actions in social situations.

- eg.: How we behave in an elevator with a stranger. In North America, it can be impolite and uncomfortable to make direct eye contact with a stranger in the elevator.
- eg.: It might seem a bit odd if we opt to offer our bus seat to an athletic person, while neglecting elderly, pregnant, or disabled people.

### Real-life examples (semantic constraints)

Relies upon the meaning of the situation to control the set of possible actions.

- eg.: We wear helmets to ride bikes or skateboard because we are delicate human beings that, if we hit our heads, there's a good possibility of serious injury or death.
- eg.: If somehow we discovered immortality and it was impossible for us to die, helmets would be useless and there would be no reason for us to use 'em.

### Real-life examples (logical constraints)

Relies upon arithmetic, and logical proofs to constrain the set of possible actions.

- eg.: Who took my food from the fridge? If all but *one* of our roommates had valid alibis, it *must* have been that person.

## Constraint examples in software development

- eg.: Static type checking.
- eg.: Value Objects use the Factory Pattern to enforce constraints against creating domain objects.
- eg.: Access modifiers.
- eg.: Language rules. A `const` cannot be redeclared. A `static` method can only be accessed through its `class`, not through an instance of the class. In the Object-Oriented Programming section of the book, we rely on these language rules to enforce policy, reducing the total surface area for ways developers can misuse our solutions.
- eg.: Strictly-typed errors. These force you to deal with potential error states instead of throwing them at you.

## Thin interfaces

In John Osterhout's book, *A Philosophy of Software Design*, one of the recurring ideas is that we should strive to have well-named modules with *thin interfaces*.

It means that anytime we're designing a class and have to decide on which methods should be public vs. private, or when we're designing an interface and deciding what the methods a subclass class should need to implement, we should choose the minimal route.

Humans have limited *working memory* so we should limit the number of options. **Constrain what humans need to know about in order to use the class or module effectively.**

That's important. This idea doesn't just apply to modules. It also applies to class-design, interface-design — essentially, any non-encapsulated, public-facing API. At these times, we should strive to have as little as possible on the public interface and encapsulate away the complexity to class or module itself.

■ **Minimal interface:** Fowler refers to this phenomenon as the idea of the *Minimal Interface*.

### Other ways to use constraints well

In addition to using thin interfaces, there are two other things are going to help us implement constraints in our designs: types and an explicit error handling strategy. We'll discuss them both.

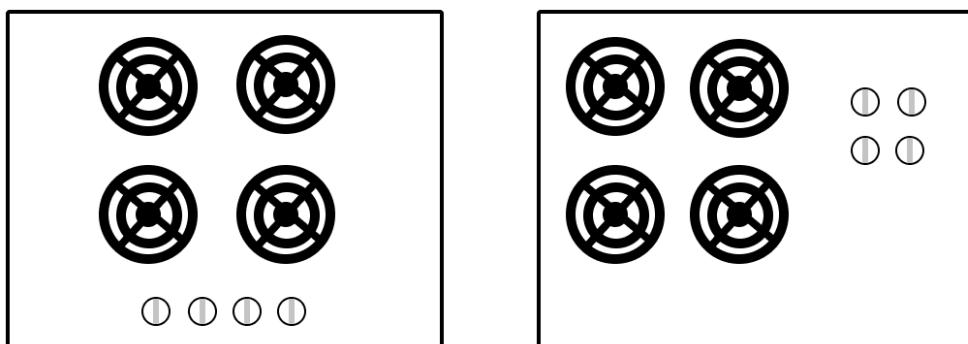
## Mapping

Mapping is the relationship between two sets of things.

Typically, when we're talking about mapping, what we're really concerned with is the **layout of controls with respect to the devices they're hooked up to.**

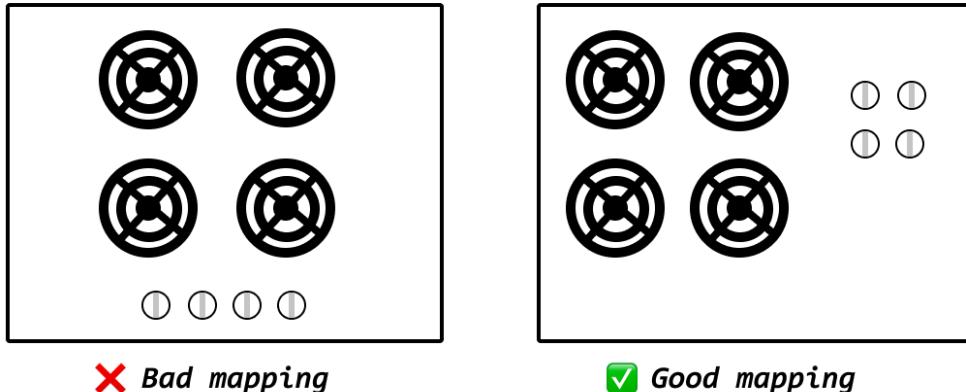
### Real-life examples

**Stove example.** Take a look at the placement of the burners and their controls on both of these stove tops.



In the first example on the left, it's not inherently clear which control maps to which burner. The first control *very well could* map to either the burner on the top-left or the bottom-left. In this case, the stove on the left has poor mapping. If we look to the stove on the right though,

we can actually *spatially* see which control maps to which burner because the layout of the controls mimic the layout of the burners!



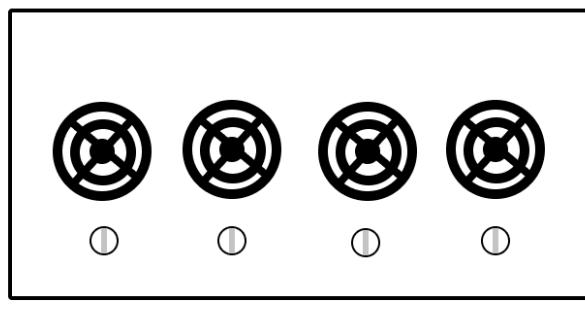
Since the example on the right is a lot easier to **discover** how to use the stove correctly, we'd say the mapping is a lot better.

The best way to implement mapping is to **place the control directly on the item that needs to be controlled**. A lot of the time though, that's not possible. I mean, let's think about the stove top. We can't exactly place the control for the burner *on* the burner itself, can we?

So really, if we had to think about how to do mapping the best, it's:

- 1 — Mount controls directly on the items to be controlled.
- 2 — Mount controls as close to the items to be controlled.
- 3 — *Mount controls in the same spatial arrangement as the items to be controlled.*

■ This is a weird one — but it's probably the best mapping even though it's strangely shaped. I'm not even sure they make stoves like this, especially if it needs to fit a counter-top.

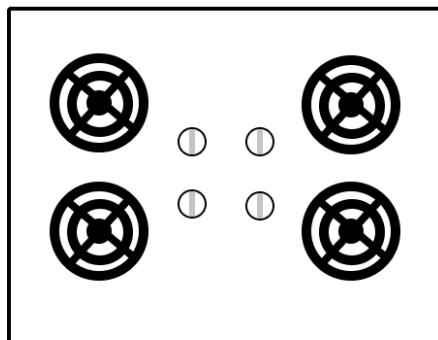


✓ *Better mapping*  
*Oddly shaped, wasted space.*

If you're still awake, you'll notice that we've settled for "*#3 — Mount controls in the same spatial arrangement as the items to be controlled*" in this example." #1 — *Mount controls directly on the items*

*to be controlled*" is impossible, but if we designed it a little differently, we might be able to mount controls closer to the items to be controlled (as per #2), we might be able to improve the design.

Here's an improvement:

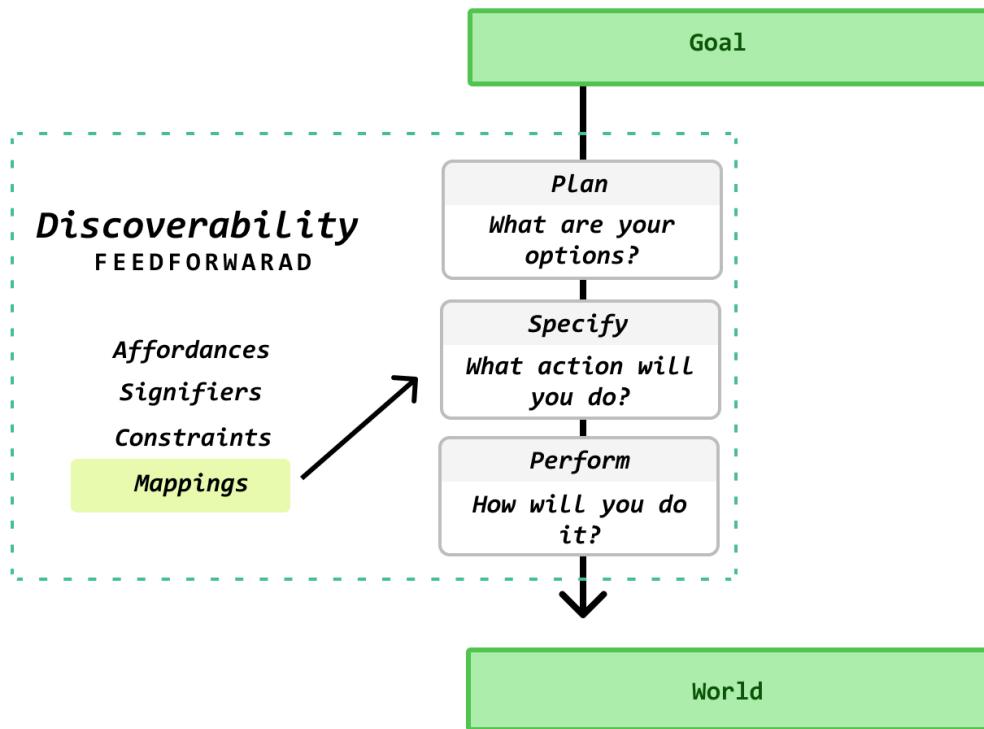


✓ *Better mapping*

Evidently, this is what the stove I have at home looks like.

### Why is this useful?

Good mapping helps us better understand the set of possible actions and *understand* what each control means. This means that it's a lot easier for us to specify what action we want to perform.



When mapping makes sense, we construct a *conceptual model* faster.

### How to do mappings well

To do mapping well, we have to obey two principles: grouping and proximity.

- Grouping — Place *related* controls together.
- Proximity — Place controls *close to the object that they control*.

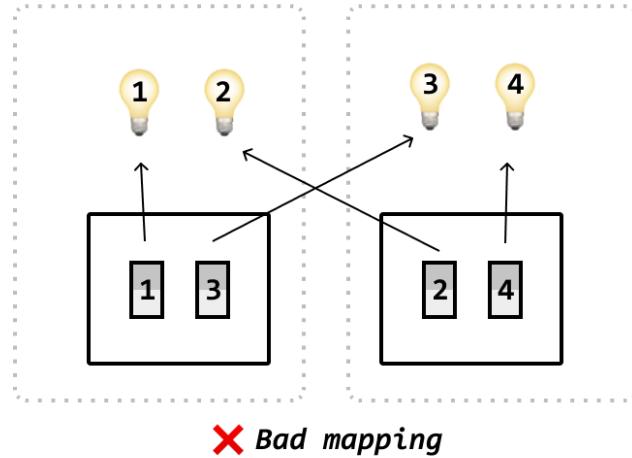
### Grouping

Imagine we were trying to hook up the light panels that control the lighting in our home.

There's a left room and a right room.

The left room has lights 1 and 2 but the panel for the left room controls lights 1 and 3.

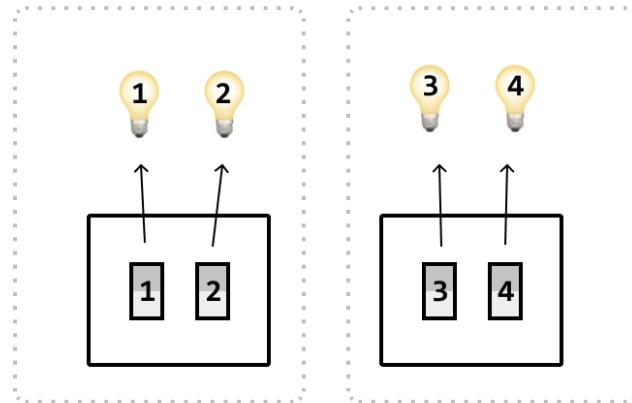
The right room has lights 3 and 4 but the panel for the right room controls the lights 2 and 4.



✗ *Bad mapping*

*The grouping is entirely off, and this would create a ton of confusion.*

It's an easy fix, we ensure the related controls are together — and these controls are related by the **room** that the lights they control are in.

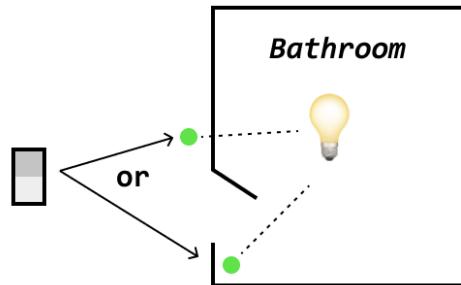


✓ *Good mapping*

## Proximity

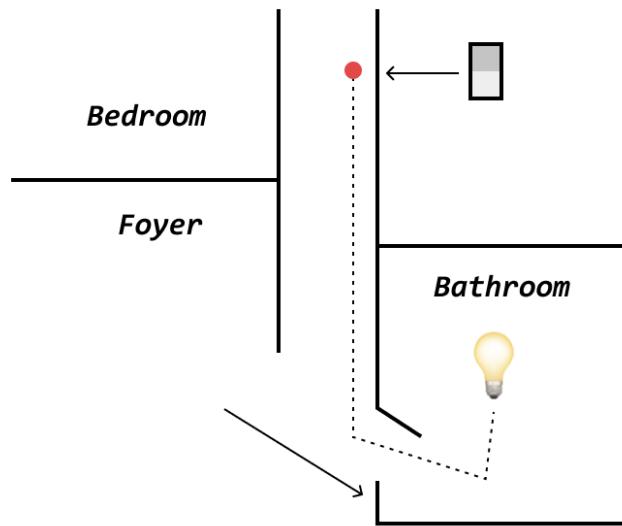
Now consider we're trying to figure out where to put the light switch for a bathroom.

In this first image (below), there are justifiable reasons for having the light switch right outside the door of the bathroom. There is also good reason to place it right *inside* the door to the right as you enter. Both of these are good proximities (and good mapping). You understand what the switch controls because of the attention to proximity.



**Good mapping**

In the following example, with the light switch all the way down the hallway, and with other rooms in proximity, it's less clear which room the switch is meant to control the lighting within. Here, we'd say this is not great proximity, and not great mapping.



**Bad mapping**  
*Too far - context is lost  
 which room is the light for?*

Mapping occurs in programming through grouping and proximity as well.

### Grouping in software development

- Cohesion — do we group related methods, classes, and functions together? For example, in object-oriented programming, common practice is to group state and the operations against that state (methods) into a single cohesive class. While separating

parts of the state like the identifier, the state, and its behaviors may be common architectural things to do in the context of things like game design (see ECS — Entity Component System and *data-oriented design*) for performance reasons, when developing business application software, we're usually OK.

```
class UserState {  
    public name: string;  
    public id: string;  
  
    constructor (name: string, id: string) {  
        this.name = name;  
        this.id = id;  
    }  
}  
  
class User {  
    // State  
    private state: UserState;  
  
    constructor (state: UserState) {  
        this.state = state;  
    }  
  
    // Behaviors  
    getName (): string {  
        return this.state.name;  
    }  
  
    greet (): void {  
        console.log(`Hello, I'm ${this.state.name}`);  
    }  
}
```

- Single Responsibility Principle & boundaries — do we enforce boundaries between things so that they have a single reason for change? Or can a change to the code potentially cause a ripple in another part of the codebase?

## Proximity in software development

- Coupling — if components rely on and are frequently changed together, they should be relatively close to each other for better discoverability and less time spent flipping between files and folders. We explore strategies for this in 6. Organizing things.
- Anemic domain models — do we keep controls that change state as close to the objects (models) as possible? Or do we maintain *slim* models? For example, a UserManager class that does create, update, setPermissions operations on the User model, leaves the User model responsible for nothing. This can lead to duplication and a lack of encapsulation.
- API design — The DOM API has good mapping. It keeps the methods that operate

against objects, physically *on* the objects themselves. *The controls are as close to the item to be controlled as possible.* For example, `document.querySelector` returns an object that contains the methods that may be performed against it.

```
// Great!
const node = document
  .querySelector('a')
  .getAttribute('href')
```

Imagine if you needed to construct a secondary class or wrap the result of `document.querySelector` in another object to call `getAttribute` on the HTML node.

```
const nodeResult = document.querySelector('a');

// This API doesn't exist, but imagine you had to do
// this wrapping everytime you wanted to execute a method
// on a node returned from the DOM.
const node = document.createExecutableNode(nodeResult);

node.getAttribute('href')
```

Some Java APIs are like this, which creates a challenge and adds another level of knowledge necessary for the developer to know **how they may use** what was returned.

*Consequently, this is also the Law of Demeter (Principle of Least Knowledge), ensuring that an object doesn't know too much about the other one (ie: my object doesn't talk to strangers).*

## Feedback

Communicating the results of an action

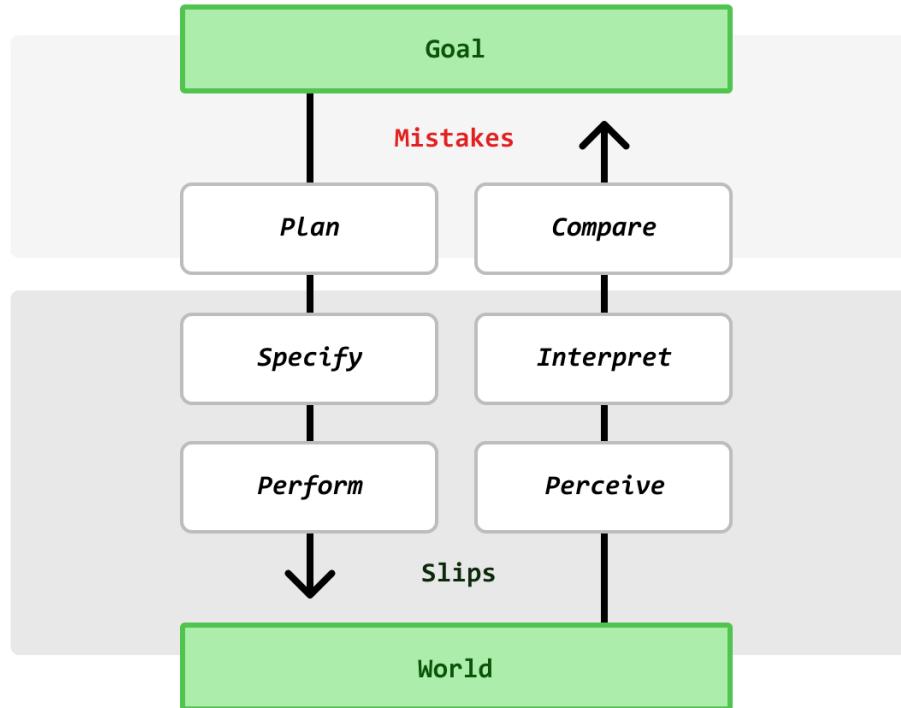
It's expected that feedback will be immediate. Not getting immediate feedback feels weird, and users will try again, expecting to get it immediately.

If users don't get feedback immediately, they might assume that the item they've tried to interact with is broken or slow, even if it's working perfectly in the background.

Success, loading, and failure states are important to report every step of the way. Each action from a user should be confirmed by feedback.

## Types of errors

It's possible that users can also run into errors. There are two kinds of errors: slips and mistakes.



- **Slips:** This is when the goal or plan is **correct**, but the *sequence is off*. In these cases, it's OK — we can point the user in the right sequence of actions.
- **Mistakes:** This is when the goal or plan is **not correct**. This is bad, and you want to avoid this happening. It means that the user is about to go completely astray *or* it means that they *have* gone completely astray, going about accomplishing their goal in an incorrect way.

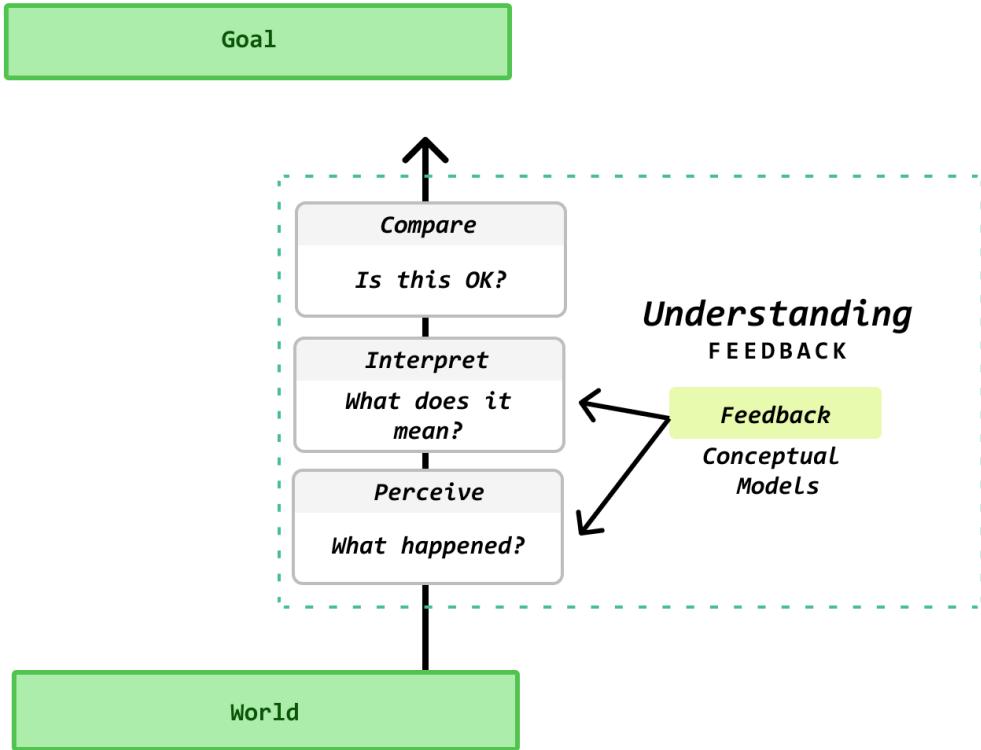
### Real-life examples

- Elevator buttons — they light up (and often make a noise as well) to indicate that the elevator is now moving. When we use the “close” button in an elevator, if the button doesn’t close right after we press it, we might assume it didn’t register, and then proceed to bash it a couple more times.
- Making a purchase on a website — you want to see the loading spinner as soon as possible otherwise you might assume your transaction hasn’t started. This can lead to users double purchasing something if it wasn’t protectively designed against that.

### Why is this useful?

Feedback is the other half of the seven stages of action. It’s what helps to build our conceptual model.

After interacting with something, the next stages are *perceive*, *interpret*, and *compare*.



We want to know that the thing we performed *took*. We want to be able to interpret what happened. Then, we want to know if what happened aligned with our goal.

### Feedback in software development

- The CQS principle. If a **command** is invoked, getting no response is the feedback to signal that *it succeeded*. Getting an error signals that *it failed*. If a **query** is invoked, getting no response is the feedback to signal that *something failed* (or is slow). Getting data back signals that *it succeeded*.
- Command-line utilities. Communicating progress and the current state of a long-running task is important. Think about how useful it is to see your progress when you're npm-installing some dependencies.
- Clicking the *submit* button in a form. Let's say you click *submit* and nothing happens. Do you re-click it? What if a loading spinner appeared right after you clicked it? Would you still feel inclined to re-click right away? Probably not. Immediate feedback is important to helping us discover what state we're currently in at every point in time.
- Autocomplete — this depicts the total set of possible actions, it shows that we're writing a statement using the correct object.
- Compile-time type checking — shows us immediately if what we're doing is legal or not.

## How to do feedback well

Communicate the result **immediately**. As soon as possible.

## Conceptual Models

Conceptual models are the true understanding of how something works.

They are high-level explanations of how something works. They're imperfect, they often skip or gloss over details, but they're good enough *mental models* of things that helps us understand how to use things.

Conceptual models are constructed from experience using something.

All of the feedforward principles and feedback help us to fully construct a mental model of something.

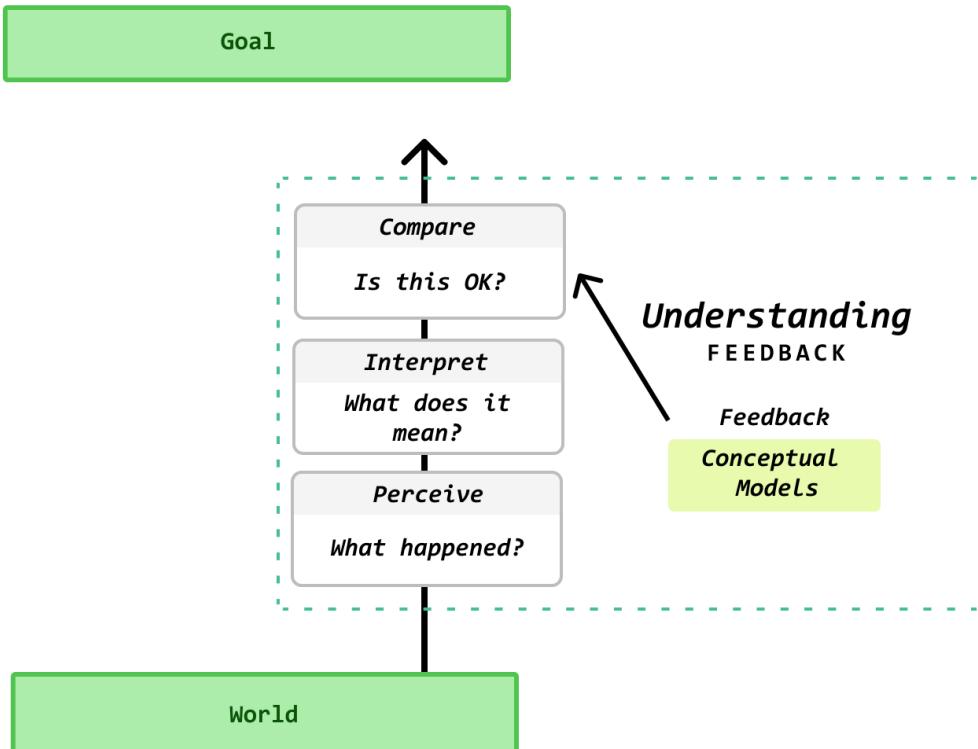
**This is the goal of our designs — we're aiming to build up a conceptual model so that users can more quickly understand how it is to be used to meet our goals.**

## Real-life examples

- Elevators — You might not know everything about the machinery behind elevators, but you generally understand how they work having spent time using them. If you ask the engineer, they'll probably have a much deeper knowledge.
- *The cloud* — Ask your parents where their data gets stored and they might say *the cloud*. That's fair. They don't need to know that your data gets saved in databases somewhere on servers likely running on AWS EC2 containers. It's a *good enough* understanding for them to know how to utilize it.

## Why is this useful?

A good conceptual model allows us to predict the effects of our actions. If we don't have one, we end up needing to just try random things to see what works and what doesn't work. We don't know what to expect.



## Conceptual models in software development

- Design Patterns. Design patterns are signs that point to conceptual models. When you think of the Builder pattern, Factory Pattern, Observer pattern, etc — we have an understanding of how these work. Each design pattern has a name that helps to conceptualize what it means. For example, the Factory Pattern implies that a component acts as the sole place to create an object. The Repository Pattern implies that it has the responsibility of knowing how and where to store and retrieve objects, like a bookkeeper or librarian.
- Encapsulation. Even if you don't fully understand *reactivity in React*, you know that React will re-render when your data changes. That's good enough for you to know. You don't need to know everything about the shadow DOM and what happens underneath the API. That's encapsulated.
- Constructs. Controllers, routes, client-side libraries like React, etc. For nearly everything we want to work with, we have to construct a conceptual model around how it works, and it might not need to be completely in-depth, but it needs to be good enough for us.

## How to do conceptual models well

Use encapsulation to abstract away complexity that we don't want the client to have to think

about.

Optimize all the *feedforward* principles to help create a better conceptual model.

## Testing your code for cleanliness

Ask a friend.

Really. When I studied computer networking, my professor often talk about the projects he worked on. I recall one of his stories from the time he opened the networking closet to check the router configs and was exposed to what he called *barbarian-ism*. Every time he was reminded about the folks who configured those routers, he'd menacingly utter a "*damn those barbarians*".

You're not a barbarian. And you don't want your coworkers to think you are either.

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 *maintainence* 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 a 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 later, when you're long gone from the project.

## **Summary**

We've discussed many things in this chapter. To summarize, we've learned:

### **A philosophy for human-friendly code**

A reminder, software design is **developer experience vs. structure**: two forces that are constantly at odds with each other. We're going to continue to learn about structure throughout the rest of the book (like how to use OOP correctly to create and enforce policy) and developer experience is about making your code easy to understand and work with.

It is unlikely that we'll be able to craft the perfect codebase, but understanding how we understand helps push us towards a practical, pragmatic space.

For example, we can now begin to have nuanced discussions about design, taking both cleanliness and the complexity introduced by additional structure into consideration. Very practically, we should be better off understanding when to make tradeoffs for **structure over**

**developer experience** (like on enterprise applications where learning curves are somewhat unavoidable due to the complexity of the problem domain itself) and alternatively, we when **developer experience** is paramount, like when we're designing an API, library, or framework to be used by clients.

## Exercises

▪ More coming soon...

### Consider the fundamentals of interaction when designing software

We now know that these principles aren't just for UX designers. They were formalized to help communicate how human beings **discover** and **understand** how to use things.

Since designing code for maintainers is an important part of what we do, study these principles and keep them in the back of your mind when coding. They are just as important as the structural software design principles we're going to learn throughout the remainder of the book.

We appreciate the design philosophy because it helps us reason about what makes something *unclean*. Not only that, it helps us follow through with concrete steps we can take to improve the developer experience of our designs rather than following a set of coding conventions because someone else said to do so.

### Learn more about Human-Centered Design

Human-Centered Design is a topic worthy of more research. I think every software developer, regardless of if you code UIs or not, should read *the book* on design. I suggest you spend some time learning more about design starting by reading "The Design of Everyday Things", by Donald Norman.

If you're looking for a way to *quantitatively* measure design effectiveness, check out GOMS. Next time your friend says they can code faster than you in Vim vs. you with your keyboard and mouse, try measuring a task using a GOMS test.

## Resources

### Articles

- The Principles of Least Astonishment and Thin Interfaces in A Philosophy of Software Design

### Books

- The Design of Everyday Things by Don Norman

## 6. Organizing things

▪ By structuring your application's folders and architecture around the use cases, you end up with an expressive and discoverable project structure that makes it easy for you and future

maintainers to quickly understand what the system does. Packaging use cases into cohesive folders promotes feature decoupling and improves your ability to find, add, change, and remove features. This is an approach that can be applied to both back-end and front-end projects.

## Chapter goals

- Learn the symptoms of poor project structure
- Discuss the shortcomings of common approaches to project structure
- Learn the benefits to structuring your project around the use cases of the system
- Learn how to structure front and back-end projects for discoverability, cohesion, and expressiveness

## Symptoms of poor project structure

### Forgetting what the system does

How many times have you worked on a side-project, left it for a few months, and then came back and tried to remember what the heck it does?

Some project structures force you to expend extra energy to remember what the system can do.

### Hard to locate features

“Ah, after you login, I want you to be redirected to the dashboard page instead of going to the latest posts”. How long does it take you to find the *login* feature and all the code associated with it?

### Energy spent flipping back and forth between files

Let’s say you’ve found the *component* that renders the login form. You need to change a couple of different things related to this feature like the React hook, the styling on the page, the popup text that comes down after, etc.

You want a dedicated workspace to work on one particular feature at a time, and you don’t want it to be overwhelming.

This distance of these files in your project influence the amount of cognitive load required to make all the necessary changes.

## Common project structure approaches

Here are some of the most common approaches to project structure that I’ve seen in the wild.

Let’s assume we’re working on a React project for these examples.

## Just evolve it over time

Assuming we're starting from nothing, we're not using TDD, and that we're just building up the application, we start with the following files.

```
src/
  App.js
  index.js
```

Now, over time, we start to craft out new folders and technically separate things.

Eventually, we notice that we need some place to store logic that doesn't really belong in components. So we create a services folder and start putting functions in there.

```
// src/services/users

export const validateEmail = () => { ... }
export const validatePassword = () => { ... }
```

We also realize that we need some statefulness, so we start to create React hooks too.

We watch our folders start to take shape. And it might look something like this:

```
src/
  Register/
    SignupForm.ts
  Todos/
    TodoList.ts
    Todo.ts
  services/
    TodoService.ts
  hooks/
    useTodos.ts
    useRegister.ts
```

With this approach, we don't start with any particular structure in mind — we just let it evolve naturally.

While there's a time and place for this, the major disadvantage of this approach is that it doesn't work very well when there's more than one developer on a project. We may take on a couple of features and the other developer(s) may take on some of their own — how do we decide how we're going to split up the work and how the project should be organized to allow that?

This is a recipe for conflicts. It also means we're constantly using brain-power to ask ourselves, "does this belong here?".

- Will it help us remember what the system does? — No.
- Will it help us locate features? — No.
- Will it reduce the distance of flipping between files when working on features? — No.

## Package by infrastructure/technology/type

As excellent pattern matchers, the next logical approach is to organize files into top-level folder categories based on the type of technical infrastructure they belong to.

```
src/
  components/
    Todos/
    Register/
  config/
    context/
  errors/
  hooks/
  models/
  utils/
  services/
```

While this may seem like a reasonable design idea, it too comes with its own set of disadvantages as well.

The first disadvantage is the amount of cognitive load requires. Features are split across several files and folders and separate from each other. This means that if we've been given the task to *change* or *remove* a feature, we're going to have to have an intimate understanding of how that feature is realized across *several files and folders* — none of which are logically grouped together.

The second disadvantage is that if we were to come back to our project after a year with it no longer fresh in our minds, *we'd have no idea what it does* until we start to peek through the codebase and refresh our memory. The discoverability of the features are remarkably low. That might not be so bad if you were the original author, but consider the case of a new developer trying to learn the codebase for the first time. That would certainly be a much more challenging endeavour.

- Will it help us remember what the system does? — No.
- Will it help us locate features? — No.
- Will it reduce the distance of flipping between files when working on features? — No.

## Package by domain

The last of the three approaches I've most frequently seen is to organize files and folders by the domain that they belong to.

```
src/
  domain/          # Domain-specific
    User/
      Profile/
      Avatar/
  services/
    Auth/
      index.js
  Todos/
```

```
Todo/  
  TodoList/  
    Payment/  
      PaymentForm/  
      CreditCard/  
    components/ # Shared  
      /App  
      /Button  
      /TextInput
```

In the file structure above, the `components` folder contains a list of generic components, and the domain folder groups components by a particular *theme* or domain.\* For example, the `Profile` and `Avatar` components both logically belong to the User domain, so we group them that way.

This is *generally* a good step in the correct direction, but it's still missing one of our discoverability goals.

- Will it help us remember what the system does? — It gives a general idea.
- Will it help us locate features? — Yes, if we understand top-level domains.
- Will it reduce the distance of flipping between files when working on features? — No, that's still an issue.

What do we do here?

Let's go back to the drawing board. Design this from scratch.

## Features (use cases)

We implement features. Correct?

When you sit down to do the work that's been laid out for your sprint, what are we getting paid to develop? Features! Right.

Another word for a **feature** is a *use case*. Use cases can be thought of as user stories, commands (like `CreateUser`, `EditUser`, `DeleteUser`) or queries (like `GetUserById`, `GetAllUsers`, `GetAdminUsers`).

## Features are vertical slices

Every application, front or back, can be divided up into its set of features.

Systems are merely **features/use cases** and the **infrastructure** that supports them.

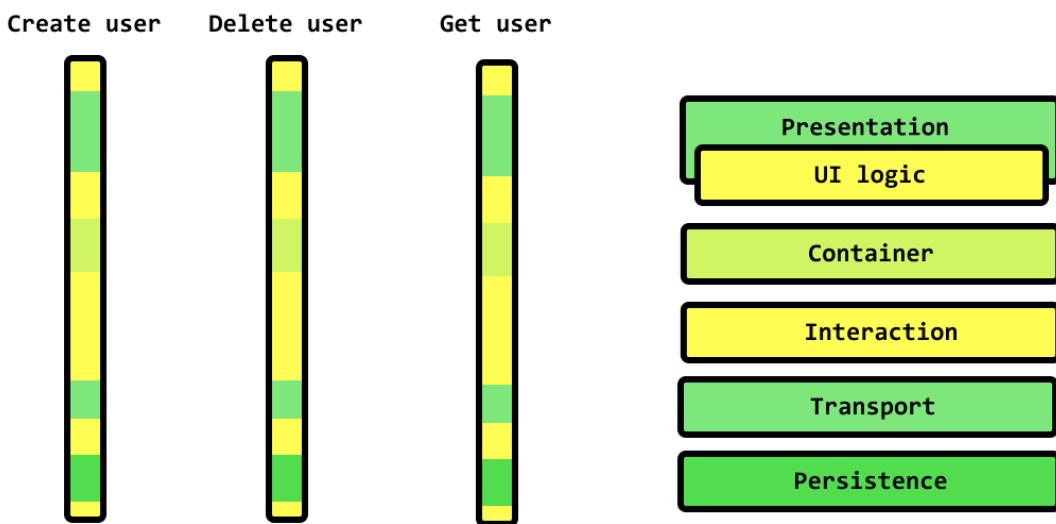
They are *vertical* slices that cut through every layer of our architecture. For example, in a front-end architecture, there are many things we need to concern ourselves with

- Presentation logic, ui logic, interaction logic, transport and persistence logic to name a few.
- Features cut through each of these.

- For a CreateUser use case/feature, you would likely need to create the signup form (presentation), the form-checking logic to ensure that it was filled in (interaction logic), and the API call and caching (transport and persistence).

All of these concerns are necessary and involved with the implementation of a single feature.

## Features are vertical slices of the stack



“Features are vertical slices” — from Client-Side Architecture Basics [2020]

### Features are the entry-point

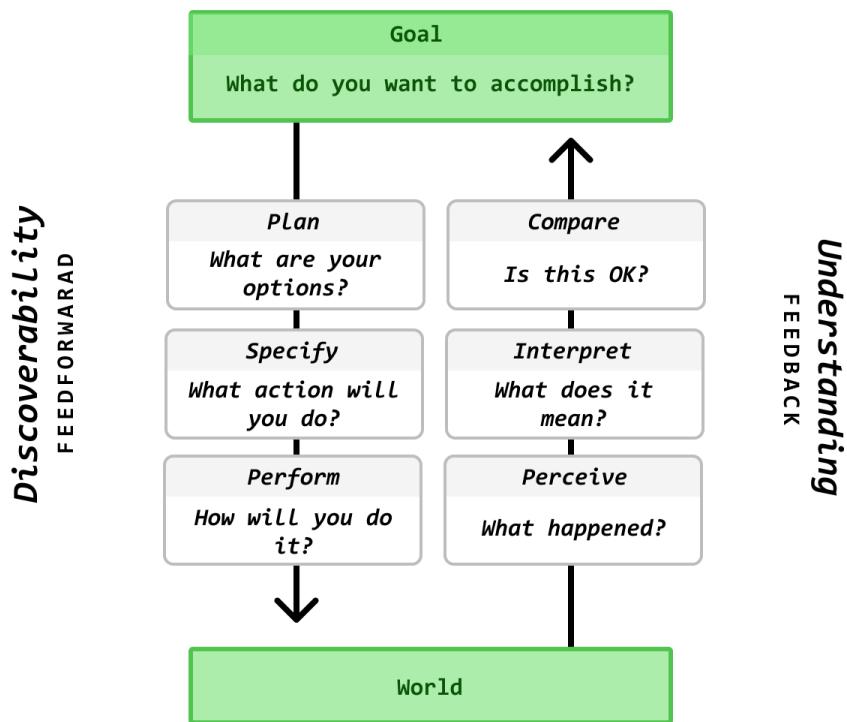
This is primarily how we work. Every sprint, you’re assigned a feature to perform work on. Whether you’re:

- adding a new feature
- changing an existing one
- or removing an old one

We are doing work against *features*.

If that’s the case, then if we want a discoverable, cohesive, and expressive architecture, we had better start with the folder structure — and make it **feature-driven**.

■ **Feature-driven folder-structure:** If our primary goal is to add/change/remove a feature, we should make it easy to see our options for the “Planning” stage of discovery.



Recall the way we discover what to do? We start with the goal and first figure out how to cross the “gulf of execution” (plan, specify, perform). Feature-driven folder structures help us get there faster.

The easier it is to discover where features are and how they’re configured, the faster we’ll be able to determine the next sequence of actions we should take (like creating a new feature folder for a new feature, or changing the style for a component specific to a feature).

### Screaming architecture & feature-driven folder structure

When you open up your text editor and look at your project, what does it *scream* at you?

Does it scream out the infrastructure like “this is a Django application”, “this is a React app”, “this is an Angular” app?

Or does it scream of the domain like “this is a fitness app”, “this is forum software”, or “this is a ticketing system”?

It’s not important for our architecture to scream out the libraries or framework that we’re using for our project. To be **feature-driven**, we want it to scream out the **features and capabilities** of our application.

“Software architectures are structures that support the use cases of the system. Just as the plans for a house or a library **scream** about the use cases of those

**buildings, so should the architecture of a software application scream about the use cases of the application” — Robert C. Martin**

Doing *package by domain* gets us partway there, but it means something more like turning a project that looks like this:

```
src/
  users/                      # Domain
    userController/           # All features in a single controller
  utils/
    date/
  text/
```

Into this:

```
src/
  modules/                   # Top-level modules of our application
    users/                     # Domain
      useCases/                # Use cases / features
        createUser/             # Create user
        editUser/               # Edit user
        deleteUser/             # Delete user feature
        DeleteUserUseCase.ts    # Use case (for delete user)
        DeleteUserErrors.ts     # Errors (for delete user)
        DeleteUserController.ts # Controller (for delete user)
        DeleteUserDTO.ts        # DTO (for delete user)
    /shared                    # Everything else is shared
      /core
      /infra
      /utils
```

In essence,

Great folder structure is centered around the **features/use cases** of the system and makes discovering them (and the supporting **infrastructure**) easy.

Allow me to demonstrate how this works on both frontend and backend projects.

### Feature-driven front-end project structure

Before I can demonstrate how to structure frontend projects, I need to make my argument for **page components**.

#### Pages

Where do *features* live in front-end applications? I believe they live within *container* components (also known as a *page components*).

Most developers don't like to make any distinction between *container* components and *presentational* components, but I think it is exactly this lack of distinction that keeps client-side architecture hard to reason about.

In client-side applications, a page represents the **top-level component** that gets rendered when you land on a **route**.

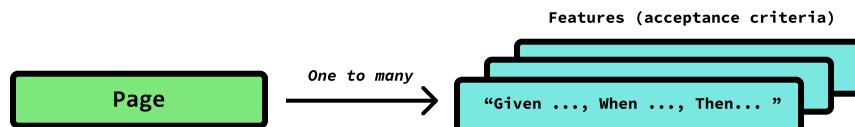
```
import * as React from 'react';
import { BrowserRouter as Router, Route } from "react-router-dom";
import DashboardPage from './pages/dashboard/DashboardPage';
import HomePage from './pages/home/HomePage';
import LoginPage from './pages/login/LoginPage';

function App() {
  return (
    <Router>
      <Route path="/" exact component={HomePage} />
      <Route path="/login" component={LoginPage}/>
      <Route path="/dashboard" component={DashboardPage}/>
    </Router>
  );
}

export default App;
```

The next important thing to note about pages is as follows:

Pages have a 1-to-many relationship with features (or user stories/functional requirements). It's through the pages that we test the features with acceptance tests (in either E2E or integration test style).



Let's think about it. Consider the a “Login” feature. Assume functional requirements can be written as follows:

- Given “I have not yet signed up”, When “I enter my email and password”, Then “I’m presented with a prompt telling me to sign up”
- Given “I have an account”, When “I try to login”, Then “I should be redirected to the dashboard”.

Now let's take that last test and turn it into code. In the context of a React and GraphQL application, an **integration test** for the case of successfully logging in may look like the following.

```

describe('Scenario: Successful login', () => {
  describe('Given I have an account', () => {
    describe('When I try to login', () => {
      test('Then I should be redirected to the dashboard', async () => {

        // Arrange
        const mocks = [
          {
            request: {
              query: LOGIN,
              variables: {
                input: {
                  email: 'khalil@apollographql.com', password: "tacos"
                }
              } as LoginVariables,
            },
            result: {
              data: {
                login: {
                  __typename: 'LoginSuccess',
                  token: "bingo-bango-boom-auth-token",
                },
              } as Login,
            },
          },
        ];
      });
    });
  });
});

const component = RouterTextUtils.renderWithRouter(
  <MockedProvider mocks={mocks} addTypename={true}>
    <LoginPage/>
  </MockedProvider>
);

// Act
const emailInput = await component.getByPlaceholderText(/email/);
fireEvent.change(emailInput, { target: { value: 'khalil@apollographql.com' }});

const passwordInput = await component.getByPlaceholderText(/password/);
fireEvent.change(passwordInput, { target: { value: 'tacos' }})

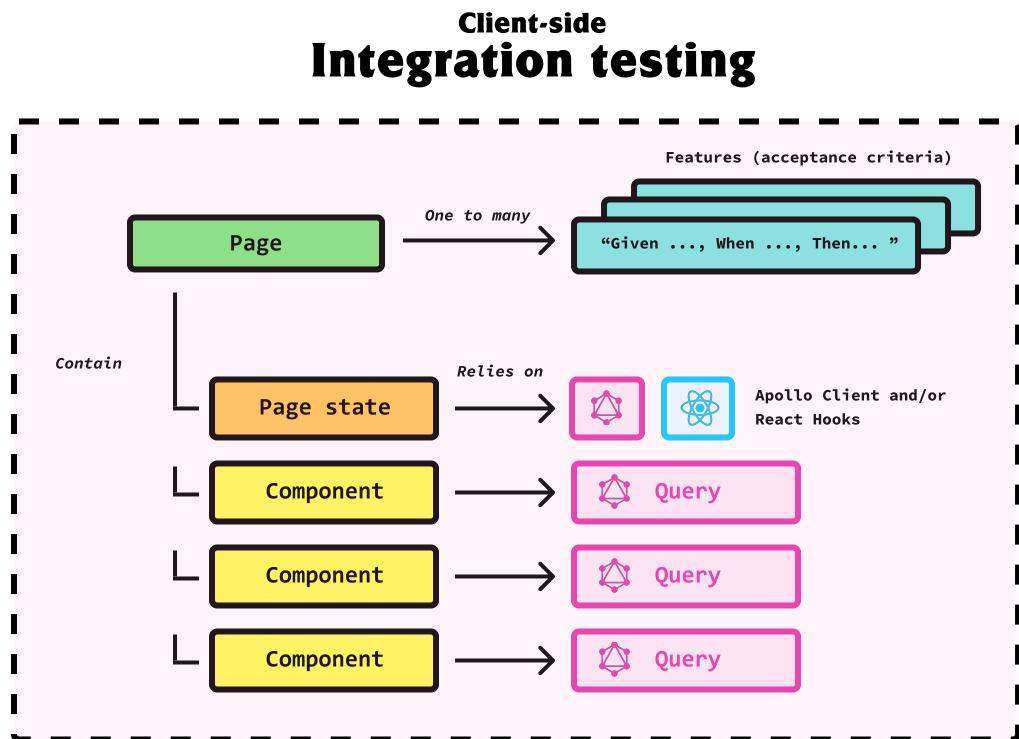
const button = await component.findByRole('button');

button.click();

await act(async() => {
  await waitForResponse()
});

```

Such a test would cover **a lot of ground**. In fact, when we get to Part X: Advanced Test-Driven Development, integration tests are the primary sort of test that we're going to shoot for writing. Why? Because they test the *functional requirements* and cover a heck of a lot of ground at the same time. In a front-end application, they test the **page**, the **state required for that page**, and any **components that were needed to realize the feature(s) on that page**.



Therefore, a page contains features, components, and any shared state (for example, through infrastructure like React Hooks, Redux machinery, or xState state machines) required specifically for that page and the nested components.

Hopefully, you catch my drift.

■ **Client-side architecture basics:** You can read more about client-side architecture basics here.

## Project structure

That being said, the project structure typically starts a little something like this:

```
src/
  pages/                      # All pages
    home/
      dashboard/               # Dashboard page (features here)
        dashboard.page.ts      # Dashboard page component
        dashboard.spec.ts     # Dashboard page test
        useDashboard.ts       # Dashboard page state & operations

  shared/                     # All shared things live here
    components/               # Shared components
    domain/                   # Non-page-specific infrastructure
      posts/
      users/                  # Non-page specific infrastructure for users domain
        dtos/
        hooks/
        mocks/
        models/
        services/

  infra/                     # Generic infrastructure
    graphql/
    router/
  layout/
  utils/
```

And to make the set of possible operations more visible, one may adjust the structure to look more like this:

```
src/
  pages/
    checkout/
      checkout.page.ts
      checkout.spec.ts
  components/
    billingInfo/
    confirmOrder/
    creditCard/
    useCases/
      placeOrder.ts
  setShippingAddress.ts
    setBillingAddress.ts
  index.ts
```

## Feature-driven backend project structure

Structuring backend projects is much of the same, yet more traditional.

### Use cases

Use cases make up the very fabric of what we're implementing on the backend. There's a lot less to debate.

```
src/
  modules/                      # Top-level modules of our application
    users/                        # Domain
      useCases/                  # Use cases / features
        createUser/              # Create user
        editUser/                # Edit user
        deleteUser/              # Delete user feature
        DeleteUserUseCase.ts     # Use case (for delete user)
        DeleteUserErrors.ts      # Errors (for delete user)
        DeleteUserController.ts  # Controller (for delete user)
        DeleteUserDTO.ts         # DTO (for delete user)
    /shared                         # Everything else is shared
      /core
      /infra
      /utils
```

### Shared content

Anything that doesn't belong to a **feature** goes into the **shared** folder.

On the frontend, this includes:

- Configuration files — setup
- Generic components — Button, TextInput, InputField, etc
- Global (shared) state that doesn't belong to a specific page — services, hooks, graphql
- State management configuration — like Apollo Client or Redux (but not including their statefulness that belongs to a specific page — actions, action creators, queries, mutations)

And on the backend, this often includes:

- Configuration files — application setup
- Database infrastructure — database connection
- Caching infrastructure — Redis adapter
- API infrastructure — GraphQL, web server setup

### Benefits of a feature-driven project organization

Why is this such a major improvement? To list a few reasons:

- **The features are the essential complexity** — The features are what we really need. Components, services, hooks, objects — those are implementation details compared to what our
- **Improved discoverability of what we can do** — Since the *features* are the entry-point to how we work, making them a part of our architectures and expressively naming our folders after them helps signify to the reader the possible set of things our system can do. This is helpful for the original author, but it's also especially helpful for developers new to the project.
- **Easily discover where code belongs** — Code now belongs in one of two places: within a specific feature, or within the shared folder as a part of some shared infrastructure. This helps us **consistently** put code in the correct place.
- **Use case folders act as workspaces for features** — We no longer flip around between lots of different files and folders when we need to implement a feature.
- **Constrain future changes feature changes to a single spot** — This means less ripple when new features or changes are introduced. It should also be inherently easier to catch coupling problems since the folders act as boundaries. Notice if you have to flip between feature folders when doing feature work or not. If you do, there may be a coupling problem\*\*\*.
- **Testable architecture** — In Part III: Phronesis, we explore in more detail, the relationship that features have with tests, namely the acceptance tests. But if we've structured our application into *feature folders*, we've made it exceptionally easy for ourselves to test the *features* of your application — not just the React components, but the features \*\*— the things implementation details like components, hooks, and styles are meant to support.

■ **Real-world examples:** Take a look at the folder structures in DDDForum (the app we build later on in this book) here on GitHub.

## Project organization rules

Here are a few rules we can follow for organizing codebases. This is all highly opinionated, however — it's what I've found works best for me, and it checks out with the human-centered design principles.

### Use top-level conventions

Follow the top-level conventions that we know about like:

- `src/` is where the original source files are located, before being compiled into fewer files to `dist/`, `public/` or `build/`.
- `dist/` is where our compiled code goes — if we're using TypeScript to build a backend, the compiled code goes here.
- `public/` is where we host our HTML, CSS, and JavaScript to be read by a web server; assume anything here can be publicly read once it's hosted.

### It belongs to a feature or it's shared

Remember, if it doesn't belong to a particular feature, then it belongs in `/shared`.

## Group files related to a particular feature

To avoid flipping around too much, you'll want to create **cohesive modules of code**. The technique I recommend is to group the files related to a particular feature *together*.

A controversial topic is if tests should be a part of your production code or not or if we should create a separate tests folder at the root of our project and build out a similar file structure.

```
src/
  modules
    users/
      useCases/
        createUser/
          createUser.ts
  shared/
  tests/
    users/
      useCases/
        createUser/
          createUser.spec.ts
```

I disagree with this approach for a number of reasons. The first is that it makes refactoring more complicated and places more cognitive load on the developer to remember to adjust the *tests* folder structure every time the production code folder structure is changed. The larger approach is that it doesn't promote good cohesive modules.

```
src/
  modules
    users/
      useCases/
        createUser/
          createUser.ts
          createUser.spec.ts
```

## Fight the framework

It's not uncommon for frameworks to want to tell you how to organize your code (often using the package by infrastructure technique). Don't let them influence you.

If you know that a feature-driven project structure makes more sense, feel free to tuck the framework code away into /shared/infrastructure/<framework\_name\_here>/.

We have very different goals from frameworks designers. For us — discoverability, for them — getting all the pieces up and working.

## Summary

- We want to strive for a project structure that helps you locate features, reduces the amount of time spent flipping between folders, and practically *screams* the capabilities of the system.

- Features are the elements of which we spend the majority of our time realizing a system. Once all of the features are developed, we're done.
- A feature-driven approach to project structure introduces *workspaces* that make it easier to find the code we need, keep tabs on coupling, reduce cognitive load, test the functional requirements, and scale the codebase over time.
- In a front-end application, features live in pages. Therefore, we structure the project with respect to pages.
- In a back-end application, features *are* use cases. Therefore, we structure the project with respect to use cases.

## References

### Articles

- <https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html>
- <https://www.robinwieruch.de/react-folder-structure>

### Books

### Exercises

■ Coming soon!

## 7. Documentation & Repositories

■ A well-designed repository provides answers to the most common top-level developer goals and acts as the jumping-off point for learning how to work within a codebase.

Design reminds me of when I used to play midfield in soccer. The midfielder needs to keep track of the game, has to be a couple of seconds ahead, and has to think about what's going to be required of them next.

Midfielders often have to survey their surroundings, notice the positioning of your teammates, large gaps, and strategically place themselves in a position to be useful.

Midfielders help strikers set up plays. They predict when the defensive line may need a little bit of support. To me, midfield is highly strategic, and it's about being a team player that knows how to create **leverage**.

A well-designed code repository is very much similar. It creates leverage.

Well-designed repositories account for the most common things that a new developer needs to know to become productive with the codebase. It's our first opportunity to create leverage, having the codebase answer questions for us instead of having to answer them over and over. And what happens when you're gone from the project? Will a future maintainer have questions? You probably don't want them hitting up your cellphone, do you?

In this short chapter, we'll learn what's required to create high-leverage repositories.

## Chapter goals

- Learn the most common high-level goals a developer has on a new project
- Learn why tests can be used as a replacement to traditional API documentation for most enterprise software.
- Discuss how repositories are the first step in a constant endeavour to push complexity downwards.

## User goals and questions to address in a repository

If I started a new job and just got access to the repository where my coworkers have been working, I'd want the repository to answer the following questions as succinctly as possible.

- What is this? — Is it the frontend application? The backend? Some other service? Give your repo a title and a description. The description should describe what the repo is for in a sentence.
- What is it for? — You'd be surprised at how many projects I've been asked to take a look at that I had no idea what they were for. This is helpful because it helps the developer build **affordances** — ah, this can be used for *this problem* that I'm having; got it! Ah, this makes *this other problem* easier, I see! Ah, so I can use *this instead* — sweet!
- How do I get started? — If you can get started in a few commands, make 'em copy-and-pastable
- How do I run the tests? — This should be able to be done in a single command. If it's a little more involved \*\*\*\*and you need to install a bunch of stuff, use a separate page and link to it as an install guide.
- How do I debug it? — If there are special things that need to be done to run in debug mode (like passing in some flags), make that known here. This is definitely a common thing to need to do.
- **Where are the features?** — Lastly, since we primarily work on features, if we're using a feature-driven folder structure, this will help to become productive even faster.

## Push complexity downwards

Just like this book is organized with parts and chapters, you want to use push complexity downwards. In 6. Organizing things, we said that your **feature folders** are the place where we do work. Including a link or making it easy to find from your repository only improves discoverability.

The layers of abstraction follow this hierarchy:

Repository → feature folders → tests → implementation boundary (controller or page) → implementation internals

## Tests as documentation

I've likely said it a few times already, but:

Our tests will act as the primary form of documentation

90% of us are building enterprise applications and web apps to be primarily consumed by paying customers. For us, written API documentation may be more work than it's worth.

For developer tooling teams and teams that maintain public SDKs, libraries, or frameworks (like Apollo or Stripe) — yes documentation is paramount.

For everyone else, the most important thing is to learn the system's use cases, where they are in the code, and how to add, change, and remove them.

## Summary

- Use the repository to answer common questions
- The repository should link to the features, and feature contain tests which demonstrate the functionality — to learn how it works, readers can trace layers of abstractions into the implementation
- Tests act as our primary form of documentation

## Exercises

■ Coming soon!

## Resources

### Articles

- <https://devops.com/dont-make-code-affordances-developers/>

## 8. 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*.

### Naming Principle #1. Consistency & uniqueness

*Each concept should be represented by a single, unique name - via NamingThings.co*

**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 (knowledge in the head).

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.

---

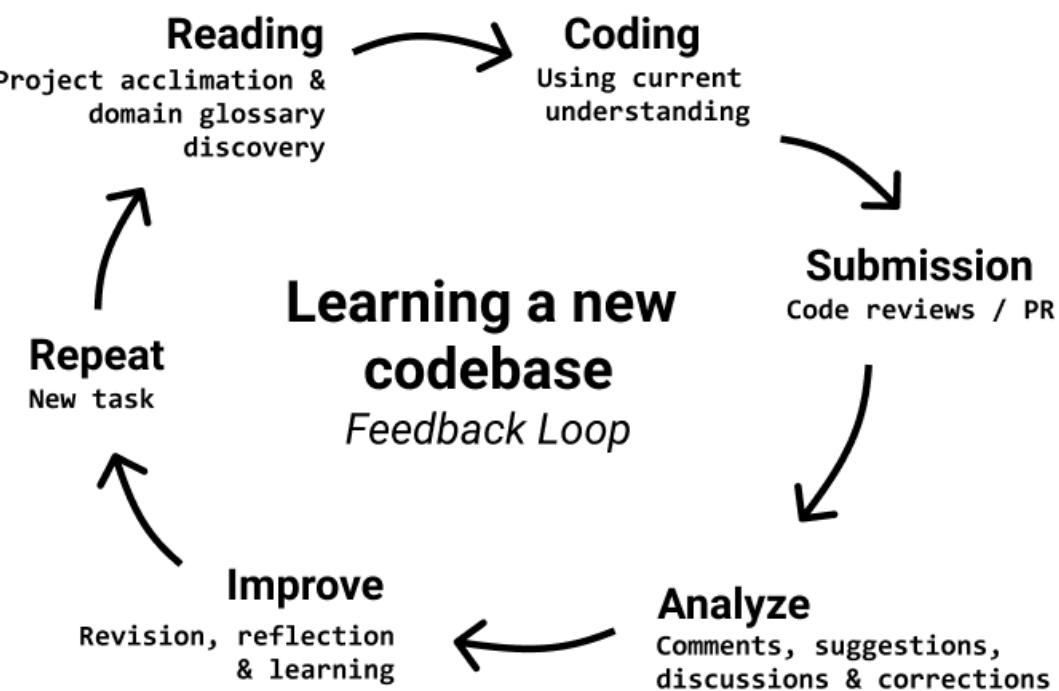
In a new codebase,

- 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*** (a Specificity principle) 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 rules to follow to reinforce 'em.

## **Rule: 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.

## **Rule: 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 shouldListenToNickCave = 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 `isXXXing` or `hasXXXXed` for variables representing states of things (e.g. `isLoading`, `hasCompletedOnboarding`)

The most important thing is that you’re consistent with your choices.

### **Rule: 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`

### **Rule: 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?

### **Rule: 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);
```

### Rule: 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.

## Naming Principle #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  $12 \times 9$ ?

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.

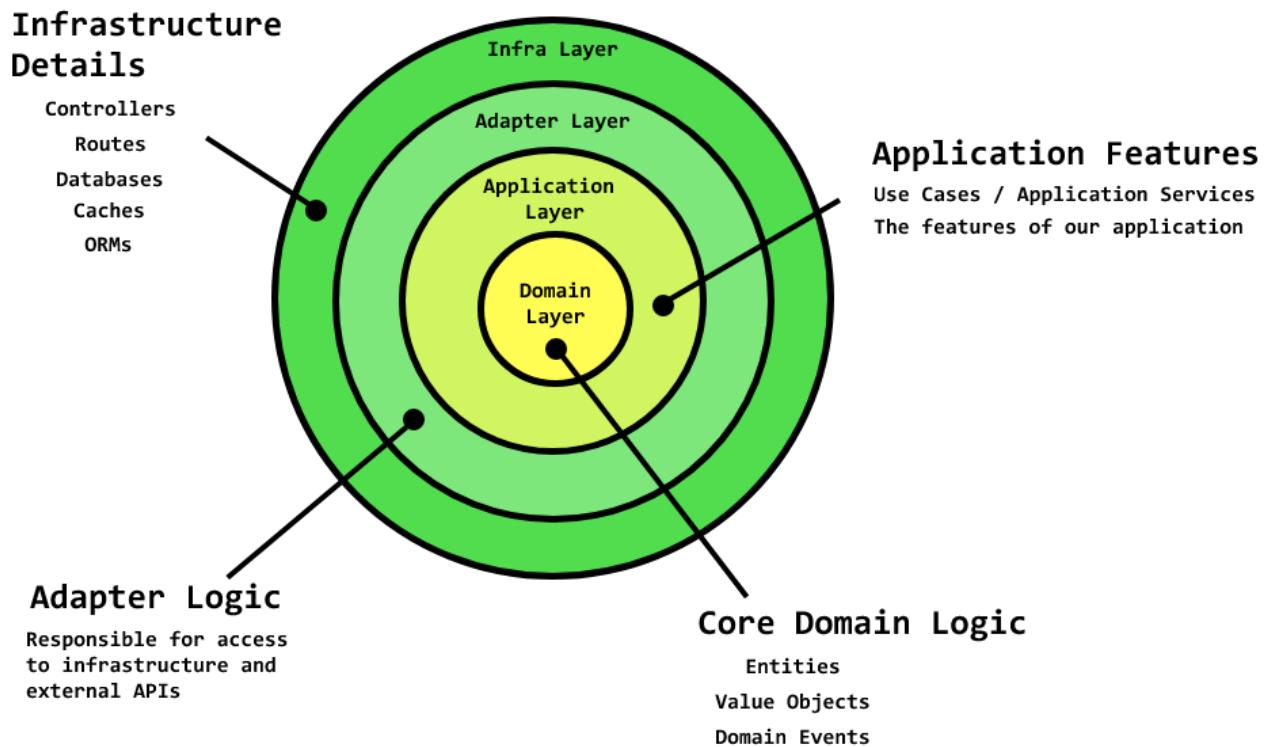
That's called the *domain layer*.

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**.

Again, you'd like to **Avoid tech-y sounding names** (Specificity rule). This includes using things like Processor, Manager, and Helper in your names. They're fluffy. They don't add to helping understand what the class, method, or variable 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 Object.

To solidify my point here, see this list of completely different, yet valid constructs for the same resource type (User):

- User, UserMapper, UserRepository, Username, UserCreated.

That's clicking with you, right? Lovely. Here are a few rules to turn into habits.

### Rule: 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.

### Rule: 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, 10l 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

### Rule: 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.

### Rule: 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 (except perhaps the id of the object that was created), 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> { ... }  
}
```

## **Rule: 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 Astonishment (Surprise).

## **Rule: Use domain concepts to refer to things from the business**

The **main** best practice. Name things as they occur from the business within the *domain* and *application* layers.

## **Rule: 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.

## **Rule: 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.

## **Rule: 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`.

## **Rule: 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<sup>\*\*</sup> 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.

## **Rule: 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)) {  
}
```

### Rule: Avoid irrelevant names

Who knows what this is supposed to mean?

```
marypoppins = (superman + starship) / god;
```

While someone might find this hilarious (or at least think that you need help), you generally want to avoid nonsense.

### Make meaningful distinctions

Distinguish names in such a way that reader knows what the differences offer.

For example, **moneyAmount** is indistinguishable from **money**, whereas **moneyInRupees** is clearly distinguishable from **moneyInDollars**.

### Naming Principle #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. However, naming concrete constructs like *methods*, *functions*, and *classes* is not only over-specification, but \*\*it's a really stinky code smell as well. We can fix this by refactoring to design patterns, namely the Factory or Builder patterns 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.

## 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.

##### Rule: Singular / plural naming

##### Main takeaways

- I. (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.

### Rule: 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();
```

### Rule: 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.

### Rule: 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 UserRepository {  
    // The name is inclusive to describe what may be stored in the variable  
    // using a union type.  
    public getUser (userOrUserId: User | UserId): Promise<User> {  
        ...  
    }  
}
```

### Rule: 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) {  
        ...  
    }  
}
```

### Rule: 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;  
}
```

### Rule: 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:

- **Preferred:** 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.
- The namespace language construct (ie: admittedly, this isn't the easiest thing to cleanly do in languages like TypeScript — but it works very well when building libraries for other developers)

### Rule: 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 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?

### Naming Principle #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

**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.

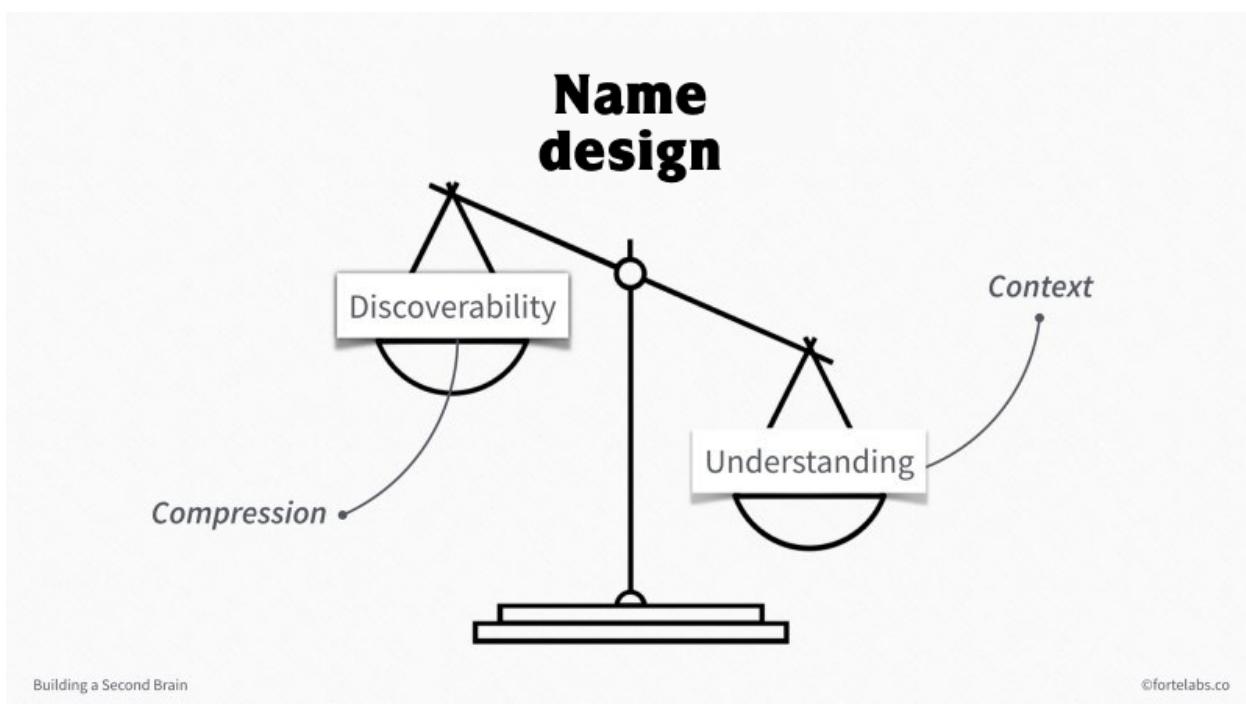
Additionally, going too far **contextually by providing too much information, we can lose the main takeaway.**

---

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.

## **Rule: 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.

## **Rule: 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.

## **Rule: 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.

### **Rule: 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) {}
}
```

### Rule: 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.

### Rule: 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 {
  getFirstName () {
    return this.vmFirstName;
  }
}
```

```

getLastName () {
    return this.vmLastName;
}

getAvatar () {
    return this.vmAvatar;
}

constructor () {
    this.vmFirstName = "";
    this.vmLastName = "";
    this.vmAvatar = null;
}
}

```

*Try this*

```

// Try this
class UserController {
    getFirstName () {
        return this.model.firstName;
    }

    getLastName () {
        return this.model.lastName;
    }

    getAvatar () {
        return this.model.avatar;
    }

    constructor () {
        this.model = new UserViewModel();
    }
}

```

## Naming Principle #5: Search-ability

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.

### **Rule: 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) {}
```

### **Rule: Keep documentation up to date**

Ideally, make updating documentation a part of the release process for new code and features.

### **Naming Principle #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).

### **Rule: Very standard abbreviations don't have to be pronounceable**

Most people understand that ssn stands for **Social Security Number**.

### **Rule: 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.

## **Rule: 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

## **Rule: Don't omit vowels**

For example. This...

```
type timeStamp = Date;
```

is better than...

```
type tmStamp = Date;
```

## **Naming Principle #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?

## **Rule: Don't use temporarily relevant concepts**

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. The maintainer working on those core modules may not find it very funny.

## **Rule: 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.

## **Rule: Don't 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.

## **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.

## **Exercises**

■ Coming soon

## **Resources**

### **Articles**

- NamingThings.co
- 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.

## **Books**

## **9. 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 — Brevity).

```
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, or if there is something contextually important that cannot be said with code, **leave a comment**.

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 peer, 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]
    }

    private shouldAddToFieldsList (fields, field) {
        return this.isAnsweredField(fields, field) || this.isNextField(fields, field)
    }

    ... // and so on

    get formConfig () {
        ...
        .reduce((fields, field) => {

            if (this.isSectionFirstInArray(fields)) {
                fields = this.makeArray(fields);
            }
        })
    }
}

```

```

        if (this.shouldAddToFieldsList(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;
    })
}
}
}

```

■ **Maintain a single layer of abstraction:** The idea here is to have a single layer of abstraction at a time so that readers can descend into complexity if they decide. Give the reader an opportunity to read at the highest layer of abstraction — the table of contents — so to speak. This is the most *expressive* layer of code.

### Example: Adding additional context

In Apollo Client, if you don't declare your primary key fields, it's possible you may lose data when it gets fetched and merged on the client. There's a `warnAboutDataLoss` function that lets you know when you might lose data due to an improper config. This function is useful in development, but it's likely not important in your production builds.

To add additional context, the author left a comment letting readers know that the function would likely be removed from production builds if code gets minified.

```

// Note that this function is unused in production, and thus should be
// pruned by any well-configured minifier.
function warnAboutDataLoss(
  existingRef: Reference,
  incomingObj: StoreObject,
  storeFieldName: string,
  store: NormalizedCache,
) {
  ...
}

```

This is a great example of a comment that adds value by providing context.

### Summary

- Use comments to explain what cannot be *expressively* said using the names of classes, methods and variables.

### Exercises

■ Coming soon!

## Resources

### Books

- Clean Code by Robert C. Martin

## 10. Formatting & Style

■ Tactical ways for us to improve our code by promoting 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.

### Chapter goals

- Learn about the three objective readability truths that matter with respect to formatting and style.
- Learn how to enforce coding conventions using npm packages

### Objective readability truths

A moment ago, I said that formatting is subjective. This is true of anything visual. What constitutes good looking code 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 succinct expression of the main ideas at the *highest layer of abstraction*, readers can lose interest and get fatigued plucking through implementation details.

## 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;
  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: IPostVotesRepo) {
        this.memberRepo = memberRepo;
        this.postRepo = postRepo;
        this.postVotesRepo = postVotesRepo;
        this.postService = postService;
    }
}

```

### *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 need to understand, and less surface area to introduce confusion about what the file **does** (separation of concerns), and what it's **responsible for** (single responsibility).

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 = { /* ... */ }
```

## Consistent whitespace

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

**This section should primarily just be about using layers of abstraction and pushing complexity downwards; that's about it.**

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...
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 `maxPrintWidth` 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

## Summary

- While formatting and style is often a subjective topic, there do exist some objective readability truths: whitespace, consistency, and storytelling.

## Exercises

■ Coming soon!

## Resources

### Articles

- <https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/>

## II. Types

■ Strictly-typed languages let us separate the contract from the implementation. Not only does this open up an entire world of software design and architecture techniques, but they act as a form of **constraints** and **feedback** which can help improve the process of learning a codebase.

### Chapter goals

- Learn the differences between programming language type systems
- Understand how statically-typed languages help us catch mistakes, communicate design intent, and scale codebases.
- Learn the basics of how types work in TypeScript
- Learn when to use types in a few real-world scenarios

### Understanding types

One of the first ways we can categorize a programming language is by the way its *type system* behaves.

## Static vs. dynamic

The two primary differences between static and dynamic languages are:

- Do I manually declare the type of this variable or will the compiler assume it for me?
- When does the compiler do type-checking? At compile time or at run time?

Static languages introduce *types* as an entirely new construct to describe what will be stored in a variable

If you're coming from a dynamic language like JavaScript or Python, you're probably used to declaring variables by merely giving it a name and then assigning it a value.

```
var name = 'Hans';
```

In static languages, there's another piece of work to do. Sometimes it's mandatory. We have to describe the **contract for that variable**. In TypeScript, we tell the compiler what type a variable will contain *beforehand* using a **type annotation**.

```
var name: string = 'Hans';

// or, declare the contract separately

type Name: string;

var name: Name = 'Hans';
```

**More structured**



**Static**

**More flexible**



**Dynamic**

**Type checking**

Types checked immediately  
(compile-time)  
Manually declare types

**Type checking**

Types checked on the fly  
(run-time)  
Automatically assumed types

In most statically typed languages, when we try to access or declare a variable, if it shouldn't exist or if it isn't the type that we **contractually said it should be**, the compiler lets us know *right away*.

Consider the following TypeScript code. We create a contract with the compiler by telling it that the num variable is going to store something with a number type. We do this by using a *type annotation* (more on this momentarily).

```

var num: number
Type 'string' is not assignable to type 'number'. (2322)
Peek Problem (⌞F8) No quick fixes available
var num: number = "2"

```

We say that, yet we break the contract immediately by attempting to assign it a value that actually has a `string` type.

Compare this to how dynamically-typed languages like Python work. In Python, to know we've gone astray, we'd have to wait until we run the code. The compiler doesn't give us any hints.

```

num = 3
print num.hello # "Hello" doesn't exist on the `num` variable, yet the compiler
                # lets us write it anyway.

```

Therefore, running this code spits out an error.

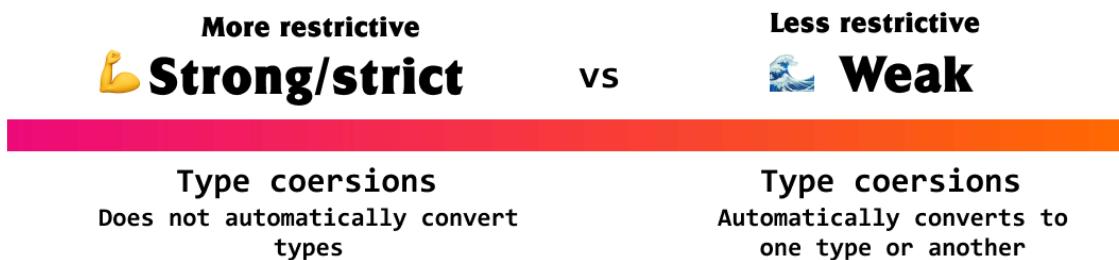
```

$ python app.py
  File "app.py", line 5
    print num.
               ^
SyntaxError: invalid syntax

```

### Strong (or strict) vs. weak

The difference between strong and weak languages is the answer to **how strict are the typing rules?** If we put the compiler in a situation where it needs to *coerce* (meaning *convert* — or fall back to) a resulting type, will it allow that or will it make a stink?



■ **Type coercion:** A type of compiler behavior in weakly-typed languages where the compiler automatically decides on the resulting type based on a scenario where the resulting type could be two or more different types.

Here's a great example. In a **strictly (also called strongly)** typed language like Python or Go, when we try to add "2" to 6, the compiler throws an error. It does this because it doesn't do type coercions.

```
num = "2" + 6
print num
```

Therefore, when we run this — we get an error.

```
$ python app.py
Traceback (most recent call last):
  File "app.py", line 3, in <module>
    num = "2" + 8
TypeError: cannot concatenate 'str' and 'int' objects
```

On the other hand, in **weakly-typed** languages like JavaScript, developers find themselves with the pleasure of dealing with situations like this — where they have to consider which type takes precedence.

```
var age = "2" + 1;
console.log(age); // "21"
```

In this example, the string type takes precedence. Consider the same statement in Perl, but where the integer type takes precedence instead.

```
my $a = "2" + 1;
print $a, "\n"; # 3
```

Strong, weak. Static, dynamic. Kind of confusing and hard to remember, but it's good to know!

If we were to decide on a programming language that gave us the ability to avoid unpredictable states, unnecessary uncertainty, and the compiler doing things to our types auto-*magically*, I'd recommend choosing a **static** and **strictly (strongly) typed** language like TypeScript.

## Why statically-typed languages?

There are a number of great reasons for using a language that has static type capabilities instead of a dynamic one. And most of the reasons have to do with the way they introduce **constraint** possibilities \*\*\*\* and improve **feedback** (see 5. Human-Centered Design For Developers).

## Catch silly mistakes

The most obvious reason I advocate for a static language is because it can help us catch some really simple, silly mistakes that humans are prone to making.

We want the computer to work for us, not against us

I used to have a manager that would double click and paste every name he ever wrote instead of just writing it out again. He developed a new set of programming habits stemming from

his fear of accidentally mistyping a variable or class name.

Statically-abled languages catch typing errors that get the best of every programmer like missing quotes, brackets, spelling errors, and missing arguments or properties.

```
function createMessage (from: string, to: string, text: string): Message {  
    ...  
}  
  
// Missing argument  
const message = createMessage('khalil', 'bill'); // Typescript will catch
```

This is a **feedback** improvement. The compiler is your wingman in static languages. It might not be able to tell you when you're flying low or when you have spinach in your teeth, but it will certainly save you a lot of time debugging things that could have easily been caught.

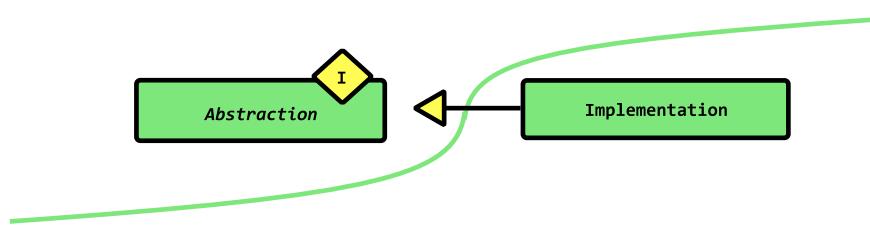
## Abstraction techniques

Coming from JavaScript, one of the biggest shifts in my career was learning how to design things using abstraction constructs like types, interfaces, abstract classes/methods and generics.

These are all features that static languages give us. They enable us to:

- Split the intent from the implementation
- Write the contract, and implement it
- Focus on the “what” (declarative) first, and the “how” (imperative) later’

One has to appreciate the magnitude of this feat. The ability to **separate the abstract from the concrete — the implementation**, is at the very heart of software design techniques. It's foundational to understanding and implementing design principles, patterns, and architectural patterns.



It enables us to write SOLID code, decouple dependencies from each other, use inversion of control to build libraries and frameworks, and create a plugin architecture able to support any number of output adapters and API styles in a clean, safe, and elegant way. Just to list a few.

It is also how we implement *pure functional architectures* structured on the essential complexity, leaving accidental details like statefulness and sequence out of the picture, and to be

merely plugged in later.

This is an important topic in our path to mastering software design. In Part V: Object-Oriented Design With Tests, we'll learn the mechanics of how to use abstraction techniques in the context of object-oriented programming. And in a way where you're not shooting yourself in the foot (like Uncle Ben said — with great power... *you finish the rest*).

## Enforce policy

Everything that we just said of abstractions can do can — of course, still be done without abstraction language features like interfaces. But it's a **lot better** when the language *does* have it, because then we don't have to worry about enforcing it — we can just trust that the compiler will enforce it for us.

For example, if we mandate that, to create a user, you need to pass in both an email and a password — it becomes law.

```
function createUser (email: string, password: string): User {  
    ...  
}  
  
createUser (1, 2); // Error  
createUser ("khalil@khalilstemmler."); // Error
```

This is a rudimentary feature you'd expect from a compiler, right? It's great. This is especially important when there's more code to account for, a growing team, and the ability to have conversations about every single function is no longer humanly possible. It relegates the conversation to the developer and the compiler — where the compiler is the gentle librarian politely asking you "please don't do that".

## Make the implicit, explicit

In the `createUser` example that we just looked at, what's stopping an angsty user (that just wants to see the world burn) from entering two blank strings for their email and password?

```
function createUser (email: string, password: string): User {  
    ...  
}  
  
createUser ("", ""); // Uh-oh, this works
```

Ah, geez.

This is where static languages give us an upper hand. As we learn more about Domain-Driven Design and XP's notion of "using the metaphor", we'll learn how to make the implicit, explicit and to build a layer of *core code* that looks almost like a domain-specific language.

Instead of using primitives like `string` and `number` all over our code, to enforce object-creation rules, we can **wrap primitives in their own domain-specific types**.

```
interface EmailAddressProps {  
    value: string; // Here's where the primitive lives now
```

```

}

class Email {

    getValue (): string {
        return this.props.value;
    }

    // "Private" means you can't create this using the "new" keyword.
    private constructor (props: EmailAddressProps) {
        super(props);
    }

    private static isValidEmail (email: string): boolean {
        ... // Do validation here
    }

    // Only way to create an Email is through the static factory method
    public static create (email: string): EmailAddress {
        const guardResult = Guard.againstNullOrUndefined(email, 'email');

        if (guardResult.isFailure() || !this.isValidEmail(email)) {
            throw new Error("Not a valid email.")
        }

        return new EmailAddress({ value: email })
    }
}

```

Now, to create a user, there's no other way to do that other than by first creating an email using the `Email` factory method.

```

class Email { ... }
class Password { ... }

// Strictly-type these instead of "string-ly" typing them
function createUser (email: Email, password: Password): User {
    ...
}

let email = Email.create("khalil@khalilstemmller.com");
let password = Password.create("verysupersecretpassword");

createUser(email, password); // Good!

```

There's a lot of ways we can make the codebase more expressive and safe when we have the capability to reduce the surface area of what's possible. When we can — *ahem, introduce*

**constraints.**

We'll learn more about these in Part VII: Design Principles and Part VI: Design Patterns.

### Communicate design intent easier

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. Yes, you can do this in JavaScript with closures, but how expressive is that, really?
- 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.

## Non-typed languages don't scale very well

Microsoft, a little company you might have heard of — the ones behind TypeScript, 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 productivity scalability.

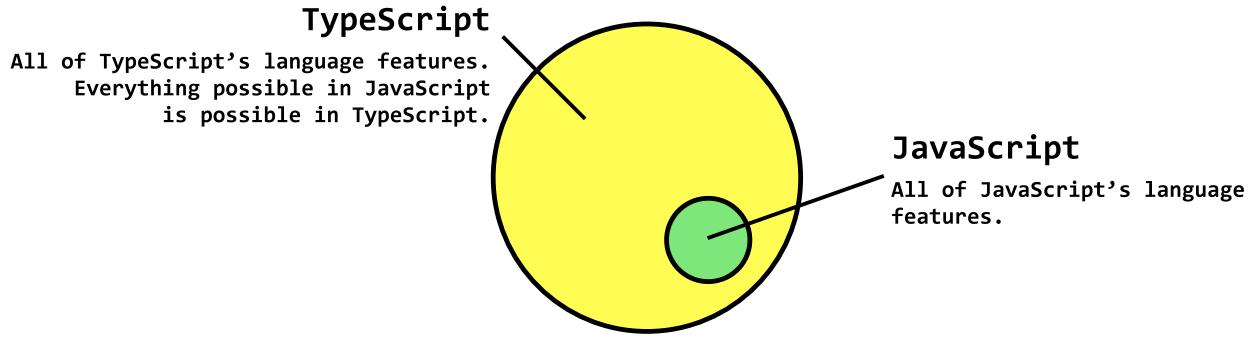
Like most dynamically-typed languages, the **lack of types** in JavaScript can drastically improve **initial productivity levels** on certain projects — but that often comes at a cost later down the road when other productivity-killing factors come into play like team size, code size, essential domain complexity, and the inability to use abstractions to design things properly.

■ **Who uses TypeScript:** Some teams and open-source libraries have shifted off of using JavaScript to build their products and APIs to Typescript instead. Example: Apollo Client, a popular GraphQL client library, is now compiled using TypeScript.

## TypeScript: Our statically-typed language

We've danced around it enough. TypeScript is the language we're using in this book. It's a superset of JavaScript, which means that it can do everything that JavaScript can do (but not vice-versa).

TypeScript is a superset of JavaScript



TypeScript is a superset of JavaScript

### The “type” annotation

For example, usage of the **type annotation** — the way that we contractually define what type a variable will have, is valid TypeScript but not valid JavaScript.

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

Yet, not including a type annotation for a variable in TypeScript is okay.

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

And that's because types are *optional* in TypeScript.

### Types are optional

One of the reasons TypeScript was created was to give JavaScript developers an **optional type system**.

Because types are optional, it means that we can gradually migrate to TypeScript by adding TypeScript to our project, running our code through the compiler, and gradually adding types to things over time.

### Types in TypeScript

There are four primary types of types in TypeScript you'll be dealing with: implicit, explicit, structural (which can be nominal or duck-typed), and ambient.

#### 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

```

var age = 13;
age.|
```

`toExponential`

`toFixed`

`toLocaleString`

`toPrecision`

`toString`

`valueOf`

(method) `Number.toExponential(fractionDigits?: number): string`

Returns a string containing a number represented in exponential notation.

`@param fractionDigits — Number of digits after the decimal point. Must be in the range 0 - 20, inclusive.`

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.

```

var age: number
Type '"old as heck"' is not assignable to type 'number'. ts(2322)
Quick Fix... Peek Problem
age = 'old as heck'
```

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 *dynamically* 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

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.

Traditionally nominal type systems (like Java's) primarily use interfaces and classes to determine type compatibility. Things behave a little different in TypeScript.

For example, take a look at this example where we create a `Guitar` and a `Synth`. Because the `pluckGuitar` function expects a `Guitar`, it makes sense that passing in a instance of a guitar would work. But why does the compiler let me pass in a `Synth`?

```
class Instrument {}

class Guitar extends Instrument {}

class Synth extends Instrument {}

function pluckGuitar (guitar: Guitar): void {
    // Nothing here yet.
}

function storeInstrument (instrument: Instrument): void {
    // Nothing here yet.
}
```

```

const fender = new Guitar();
const juno60 = new Synth();

pluckGuitar(fender);
pluckGuitar(juno60); // This shouldn't let me do this! But it does!

storeInstrument(fender);
storeInstrument(juno60);

```

In languages like Java, the type system requires 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.

If we look at how `Guitar` and `Synth` are constructed, you'll notice that they have the same structural shape. *They aren't built different.*

Because one of TypeScript's design goals was to provide an optional-type system without disrupting productivity for JavaScript developers, TypeScript falls back to **Duck Typing**. If we change the structure of `Guitar` a little bit like this:

```

class Guitar extends Instrument {
    pluck (): void {
        console.log('Boing!')
    }
}

```

And then we look at the `pluckGuitar(synth)` statement, we should now see an error.

The screenshot shows a code editor with the following TypeScript code:

```

function sto const juno60: Synth
| // Nothing
}

const fender
const juno60

pluckGuitar( Peek Problem (XF8) No quick fixes available
pluckGuitar(juno60); // This shouldn't let me do this! But it does!

storeInstrument(fender);
storeInstrument(juno60);

```

A tooltip is displayed over the `pluckGuitar(juno60)` call, showing the error message:

Argument of type 'Synth' is not assignable to parameter of type 'Guitar'.  
Property 'pluck' is missing in type 'Synth' but required in type 'Guitar'. (2345)  
input.tsx(5, 3): 'pluck' is declared here.

## Duck typing

“If it looks like a Duck and it quacks like a Duck... it must be a Duck”.

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

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
```

## Basic 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.

### 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 style in TypeScript

TypeScript lets you write code using the traditional object-oriented style (which is very much different from 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. You can optionally use classes as we're learning how to perform TDD in Part IV: Test-Driven Development Basics. In Part V: Object-Oriented Design With Tests, we'll revisit OO by first learning what all of the different object-oriented concepts are for and how they work. Then we'll improve our TDD workflow by learning how to use objects to effectively model more complex behavior in a safe, minimal, and expressive manner.

## Classes

As you likely know, classes are the primary building-block in object-oriented programming. With discipline, we should create them when we have *state* (data) \*\*and related *behaviour* that — in some way, governs the state.

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);
    }
}

```

To implement this, any time we extend a class, the subclass **needs to invoke 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()`. While this wasn't always the case in earlier versions of TypeScript, it now does follow true.

```
constructor(x: number, y: number, z: number) {  
    this.z = z; // this will not work  
    super(x, y);  
}
```

■ **Try it out yourself:** See what the resulting JavaScript looks like on TypeScript Play-ground.

## 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?
<b>public</b>	yes	yes
<b>protected</b>	no	yes
<b>private</b>	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;
```

```

    }
}

/** 
 * 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.aMethodThat MightNotExist(); /* 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 Separation that specifies that a method should be **either** a command that changes the system in some way but returns no value (except maybe the id of the entity changed), OR a query that returns a value but *causes no side-effects*. We talk more about this principle in Part VII: 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
```

## Summary

- There's a lot of great reasons for using a statically-typed programming language. From a human-centered design perspective, languages where we can explicitly define the type — separating contract from implementation — work well to impose **constraints** when necessary, and help us build true understanding with the immediate **feedback** they afford from compile-time type checking.
- TypeScript gives us the abstraction features we need to implement the design principles, patterns, and approaches that we'll learn about in this book, but it also gives us the flexibility to opt-out of static typing when we don't need it.

## Exercises

■ Coming soon!

## Resources

### Articles

- <https://stackoverflow.com/questions/2690544/what-is-the-difference-between-a-strongly-typed-language-and-a-statically-typed>
- <https://alistapart.com/article/prototypal-object-oriented-programming-using-javascript/>
- [https://en.wikipedia.org/wiki/Comparison\\_of\\_functional\\_programming\\_languages](https://en.wikipedia.org/wiki/Comparison_of_functional_programming_languages)

## I2. Errors and exceptions

■ The key to good error and exception-handling is to make the implicit explicit, and to fail fast, letting clients know when they've gone astray — as soon as possible.

Imagine a world where you could open a *severely rattled* bottle of Pepsi with brazen confidence — and to do so without hesitation or to consider the possibility that it might end up drenching you.

I'm asking you to imagine a world where nothing ever went wrong; where you **only ever had to consider the happy path**.

Let me know when we discover that world — I'd love to go there, because it's certainly not the one we currently live in. Things always go wrong in our programs.

Unfortunately, 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.

Errors have a place in the domain as well, and they deserve to be modelled as domain concepts

"Do I throw an error and let the client figure out how to handle it?"

"Do I return null?"

When we use the `throw` command willy-nilly-like, we disrupt the flow of the program and make it trickier for someone to walk through the code. This statement is somewhat comparable to the criticized `GOTO` command that Djikstra fought hard to relinquish from modern programming languages.

Essentially, we want to avoid the *jerky* behaviour of throwing errors all over the place. But that means we need to get principled about the way that we handle invalid states in our applications.

Let's do just that.

## Chapter goals

- Discover deficiencies in common approaches to error and exception handling
- Understand the difference between errors and exceptions
- Learn how to use unions to model errors as domain concepts
- Learn when to merely throw an exception instead

## Error & exception-handling follies

Two very common (and fragile) approaches to handling errors I've seen are to either (1) return null or (2) log and throw.

### Return null

Consider we had a simple JavaScript function responsible for creating a user.

```
function CreateUser (email, password) {  
  const isEmailValid = validateEmail(email);  
  const isPasswordValid = validatePassword(password);  
  
  if (!isEmailValid) {  
    console.log('Email invalid');  
    return null;  
  }  
  
  if (!isPasswordValid) {  
    console.log('Password invalid');  
    return null;  
  }  
}
```

```
    return new User(email, password);  
}
```

You've seen code like this before. You've also written code like this. I know I have.

The truth is, it's not great. Returning null is lazy because it requires the caller to be aware of the fact that a *failure* to produce a User actually means null. That's not expressive at all. In fact, that's kind of like a bait-and-switch thing to do.

Behaviour like this provokes mistrust (not very human-friendly at all) and clutters code with null-checks everywhere.

A *consumer* of this API would likely be forced to write code like this:

```
const user = CreateUser(email, password);  
  
if (user !== null) {  
    // success  
} else {  
    // fail  
}
```

Ugly? Yes. But that's not the only issue. Even more sinister is the fact that there are at least two separate failure states that could occur here:

- The email could be invalid
- The password could be invalid

Returning null fails to *differentiate* between these two errors. At this point, we've certainly lost any expressiveness of the **domain model**.

How could fix this? How could we return separate error messages for those errors? This may lead us to the log and throw strategy.

## Log and throw

Here's another common (non-optimal) approach to error handling.

```
function CreateUser (email, password) {  
    const isValidEmail = validateEmail(email);  
    const isValidPassword = validatePassword(password);  
  
    if (!isValidEmail) {  
        console.log('Email invalid');  
        return new Error('The email was invalid');  
    }  
  
    if (!isValidPassword) {  
        console.log('Password invalid');  
        return new Error('The password was invalid');  
    }  
}
```

```
    return new User(email, password);
}
```

In this case, we do get back an error message from the Error, but manually throwing errors means having to surround lots of your own code in try-catch blocks.

```
try {
  const user = CreateUser(email, password);
} catch (err) {
  switch (err.message) {
    // Fragile code
    case 'The email was invalid':
      // handle
    case 'The password was invalid':
      // handle
  }
}
```

Not only that, but the errors are not *statically-typed* so the moment the error text changes is the moment we've just turned this text file into a set of bugs that may be very hard to clue into.

## Problems with these approaches

The main issues with these approaches are that they're:

- Not expressive — We're not expressing the errors as domain concepts, and that makes them *fragile* and susceptible to more (accidental) errors \*\*in the future. They both *leak*, meaning that the *consumer* will only know that they need to do error handling **after an error happens**. That's bad. We need to be transparent about what consumers should expect.
- Fragile — The fact that neither of these approaches are statically-typed makes them fragile unless we have tests guarding them. Even if we did, those tests would likely hinge on implementation details (like the error text) and that's no good.
- Hard for the consumer to use — Neither one of these approaches presents a good API for the consumer \*\*to use.

## Understanding errors & exceptions

Let's back up for a second. We've been talking about *errors* and we've mentioned the word *exception* as well. Errors are exceptions are entirely different things. Let's get clear on their differences.

### Errors

Errors are **expected** reasons why an operation failed. These are errors that are a part of the domain (or application layer) that belongs to *us*. This is code that *we own*.

For example, if we were building login and signup features in an application, the `LoginFailed` and `AccountAlreadyCreated` errors would be expected errors because they're a part of the *essential* application complexity.

These types of errors are domain concepts, and we should model them with as much significance as we'd model `User` and `Email` concepts in an identity and access management domain.

Use the language to model errors as types

## Exceptions

Exceptions, on the other hand, are **unexpected** reasons why an operation failed. These are errors that are outside of *our hands*. It has to do with code or context that we *don't own*.

To present the example of **code that we don't own**, consider using a third party API, trying to connect to a database, or making a RESTful request to an HTTP server from a front-end application. We'd *like* these operations to all work, but ultimately, we're not the ones in control of it they are going to work or not.

And when we're working with **context that we don't own**, for example — like when we're building scripts that expect a certain number of arguments to be passed in at the command line, what do we do when the user hasn't passed them in? We have to fail and throw an error since there's no reason for us to continue execution.

Exceptions are a part of the *accidental complexity*. It's what happens when we get into the *real world* and we have to deal with I/O, real-life users, and messy real-life constraints like bandwidth, network connectivity, and so on.

Throw exceptions when there's no reason to continue execution, and surround execution of code you don't own in a try/catch

## A philosophy for error handling

Now that we know the difference between errors and exceptions, let's define a philosophy for handling *errors* first.

## Features & use cases

We've previously discussed why you should be feature (or use-case) driven. The natural way that we split up all of the things we need to do in an application is to think of it as a collection of features (or operations, user stories, use cases, commands or queries).

## One happy path, multiple sad paths

Every feature has **one happy path** and typically **multiple sad paths**.

For example, if we were implementing the `CreateUser` feature (use case), what's the happy path? The happy path is `CreateUserSuccess`, and it might just return the `id` of the user that was created.

What are the sad paths for `CreateUser`? I can think of a number of them.

- Domain errors
  - UserAlreadyExists
  - EmailInvalid
  - PasswordDoesntMeetCriteria
  - UsernameTaken
- Exceptions
  - Database could blow up
  - An API we rely on could go down
  - Some other uncaught error

It's only a matter of time until these sad paths are realized once your program is up and running. One could even argue that it's a lot easier for these sad paths to occur than it is for the happy path.

And if you were building a web application, you'd want a good way to structure your HTTP responses — for example, if the request was successful, you'd send a 201 – Created back to the client. If the client did something wrong, you'd want to send a 400-x error code back. And if it was *our fault*, you'd send a 500-x error.

How do we structure this kind of information to make life easier for the caller? We need some way to model this relationship between good and bad paths.

### Aggregate errors with unions

In some statically-typed languages, a *union type* can be used to express a type that could be *one of* multiple values. As we saw in II. Types, unions behave as a **conditional OR**.

Unions are the perfect tool for modelling both happy and sad paths in a single response object.

```
type Result = Success | Fail;
```

### Building error-handling infrastructure

Continuing with the CreateUser example, let's build some error-handling infrastructure.

Here we have a createUser function that takes in an email and a password and gives us back some result.

```
type CreateUserRequest = {
  email: string;
  password: string;
}

type CreateUserResult = {
  /* Todo */
}

function createUser (request: CreateUserRequest): CreateUserResult {
  ...
}
```

```
const userResult = createUser({  
  email: 'khalil',  
  password: 'bingo-bango-boom'  
});
```

## What do we want in our error-handling API?

What could happen here? We could *either* get a success or a failure response.

And what do we want in terms of a result object? We want an easy way to be able to tell if the request succeeded or failed.

If it succeeded, it should be possible to get success value.

And if it failed, we should be able to tell *which* failure it was that occurred, and possibly get an error message or the original error that was created so that we can handle it.

```
const userResult = createUser({  
  email: 'khalil',  
  password: 'bingo-bango-boom'  
});  
  
if (userResult.isSuccess()) {  
  const value = userResult.value; // And we want this to be typed!  
}  
  
if (userResult.isFailure()) {  
  // We want to somehow know which failure occurred and any  
  // additional message or data for that failure.  
}
```

## Introducing the Either type

Ah, *either*. That's the ticket. We need to model some infrastructure that gives us these capabilities. Let's take a look at the following TypeScript code from Bruno Vegreville.

```
type Either<S, F> = Success<S, F> | Failure<S, F>;  
  
export class Success<S, F> {  
  readonly value: S;  
  
  constructor(value: S) {  
    this.value = value;  
  }  
  
  isSuccess(): this is Success<S, F> {  
    return true;  
  }  
}
```

```

    isFailure(): this is Failure<S, F> {
      return false;
    }
  }

  class Failure<S, F> {
    readonly value: F;

    constructor(value: F) {
      this.value = value;
    }

    isSuccess(): this is Success<S, F> {
      return false;
    }

    isFailure(): this is Failure<S, F> {
      return true;
    }
  }

  export const success = <S, F>(l: S): Either<S, F> => {
    return new Success(l);
};

  export const failure = <S, F>(a: F): Either<S, F> => {
    return new Failure<S, F>(a);
};

```

Here, we make use of TypeScript's *generics* and *type inference*. We've got two classes, Success and Failure, which we will use to model our Either result as a union. But we also wanted the ability to extract the value and determine if we're working with a success or failure object. That's why we've made two classes, each which share the same interface, but have different behaviours — depending on which state the result is in (success or failure).

Finally, we present two shorthand methods `success` and `failure`, which act as a *factory functions*. They make it easy to create a valid `Either<S, F>` result, whether it be a success or a failure.

## Modelling the response type

Alright, so let's start modelling both sides of this response type.

The success side is easy. If we successfully create the user, let's just return the id of the user that was created.

```

type CreateUserSuccess = {
  id: string;
}

```

---

That works!

Now the failure side. We can start by creating some sort of way to define these types, perhaps as a `DomainError`. A domain error will require a message of some sort to describe the error.

```
type DomainError = {  
  message: string;  
}
```

Next, let's list all the errors as `DomainError` types.

```
type UserAlreadyExists = DomainError;  
type EmailInvalid = DomainError;  
type UserAlreadyExists = DomainError;  
type PasswordDoesntMeetCriteria = DomainError;  
type UsernameTaken = DomainError;
```

Great. Now putting this altogether, our `CreateUserResult`, modelled as an `Either` type, looks something like this.

```
type DomainError = {  
  message: string;  
}  
  
type UserAlreadyExists = DomainError;  
type EmailInvalid = DomainError;  
type UserAlreadyExists = DomainError;  
type PasswordDoesntMeetCriteria = DomainError;  
type UsernameTaken = DomainError;  
  
type CreateUserSuccess = {  
  id: string;  
}  
  
type CreateUserResult = Either<  
  // Success  
  CreateUserSuccess,  
  // Failures  
  UserAlreadyExists |  
  EmailInvalid |  
  PasswordDoesntMeetCriteria |  
  UsernameTaken  
>
```

Awesome! Now let's hook it up to our function.

## Hooking it up

First, we set the return type on our `createUser` function to the new `CreateUserResult` type.

```
function createUser (request: CreateUserRequest): CreateUserResult {  
    ...  
}
```

Then, we'll add some dummy methods for demonstration purposes. These represent the business and application logic that will be responsible for raising errors.

```
function createUser (request: CreateUserRequest): CreateUserResult {  
    // Dummy methods  
    const isEmailValid = () => { return true };  
    const userAlreadyExists = () => { return false };  
    const passwordMatchesCriteria = () => { return true };  
    const isUsernameTaken = () => { return false };  
  
    ...  
}
```

As soon as we check to see if we should raise the first error — if the email is valid, we should be able to spot a potential design flaw (or merely room for improvement).

```
function createUser (request: CreateUserRequest): CreateUserResult {  
    // Dummy methods  
    const isEmailValid = () => { return true };  
    const userAlreadyExists = () => { return false };  
    const passwordMatchesCriteria = () => { return true };  
    const isUsernameTaken = () => { return false };  
  
    // Performing the first check  
    if (!isEmailValid()) {  
        return failure({  
            message: 'I have to write the message here?'  
            as EmailInvalid);  
    }  
  
    ...  
}
```

It appears that having built our errors types as *types*, it means that we're going to have to write all of our error message construction in the use cases themselves. To me, this breaks encapsulation — because error message construction should be much closer to the error itself, rather than in the function that raises the error.

Not only that, but we have to manually cast the error type using the `as` keyword, **which isn't an easy thing to remember not to do**.

For this reason, we'll change the definition by modelling the domain errors as classes instead.

## Encapsulating error message construction in classes

Here

```
class EmailInvalid implements DomainError {  
    public message: string;  
  
    constructor (email: string) {  
        this.message = `The email ${email} is invalid.`  
    }  
}
```

and to create this

```
function createUser (request: CreateUserRequest): CreateUserResult {  
    // Dummy methods  
    const isEmailValid = () => { return true };  
    const userAlreadyExists = () => { return false };  
    const passwordMatchesCriteria = () => { return true };  
    const isUsernameTaken = () => { return false };  
  
    // Performing the first check  
    if (!isEmailValid()) {  
        return failure(new EmailInvalid(request.email));  
    }  
  
    ...  
}
```

We can go ahead and build out the rest

```
class EmailInvalid implements DomainError {  
    public message: string;  
  
    constructor (email: string) {  
        this.message = `The email ${email} is invalid.`  
    }  
}  
  
class UserAlreadyExists implements DomainError {  
    public message: string;  
  
    constructor (username: string) {  
        this.message = `The username ${username} is already taken.`  
    }  
}  
  
class PasswordDoesntMeetCriteria implements DomainError {  
    public message: string;
```

```

constructor (password: string) {
  this.message = `A password must be ..., ${password} is not valid.`
}
}

class UsernameTaken implements DomainError {
  public message: string;

  constructor (username: string) {
    this.message = `The ${username} is already in use.`
  }
}

// We'll even turn this into a class, to make it more consistent
class CreateUserSuccess {
  id: string;

  constructor (id: string) {
    this.id = id
  }
}

```

And then finish this up.

```

function createUser (request: CreateUserRequest): CreateUserResult {
  // Dummy methods
  const isEmailValid = () => { return true };
  const userAlreadyExists = () => { return false };
  const passwordMatchesCriteria = () => { return true };
  const isUsernameTaken = () => { return false };

  // Run through all possible error states
  if (!isEmailValid()) {
    return failure(new EmailInvalid(request.email));
  }

  if (!passwordMatchesCriteria()) {
    return failure(new PasswordDoesntMeetCriteria(request.password))
  }

  if (userAlreadyExists()) {
    return failure(new UserAlreadyExists(request.email))
  }

  if (isUsernameTaken()) {
    return failure(new UserAlreadyExists(request.email));
  }
}

```

```

// Create user, save it to the db, then get the id
// Let's assume this is synchronous
const user = db.User.save({ email, password })

// Finally
return success(new CreateUserSuccess(user.userId))
}

```

And because TypeScript falls back to structural typing, we shouldn't have to change the `CreateUserResult` response type at all.

### Usage

```

const userResult = createUser({
  email: 'khalil',
  password: 'bingo-bango-boom'
});

if (userResult.isSuccess()) {
  userResult.value.id
}

if (userResult.isFailure()) {
  let error = userResult.value;

  // Handle errors in the switch
  switch (error.constructor) {
    case EmailInvalid:
    case PasswordDoesntMeetCriteria:
    case UserAlreadyExists:
    case UsernameTaken:
    default:
  }
}

```

## A philosophy for exception handling

With respect to *exception handling*, there are two questions that we need answers to:

- How do we deal with code that might throw exceptions?
- And when should we throw our own exceptions?

### Dealing with other people's code

This all depends on context, but generally speaking, when we're dealing with other people's code that might throw exceptions, I have two principles:

- **Wrap I/O in a try/catch block**
- **Turn exceptions into meaningful errors**

In the previous example, what happens if we tried to save the user to the database and it failed? What if we lost database connectivity? What if some other database issue happened that we didn't expect? How do we handle that?

```
function createUser (request: CreateUserRequest): CreateUserResult {  
    ...  
  
    // Create user, save it to the db, then get the id  
    const user = db.User.save({ email, password })  
  
    // Finally  
    return success(new CreateUserSuccess(user.userId))  
}
```

We still want to handle this, because if our `createUser` function were to unexpectedly have to deal with an uncaught exception thrown up into its face, we'd still want to provide some context to the calling code as to what the heck just happened.

### Wrap I/O code in a try/catch block

Therefore, let us wrap the potentially offending code in a try-catch block.

```
function createUser (request: CreateUserRequest): CreateUserResult {  
    ...  
  
    let user;  
  
    try {  
        // We don't own this code, so we'll wrap it in a try-catch  
        user = db.User.save({ email, password })  
    } catch (err) {  
        // What do we do here?  
    }  
  
    // Finally  
    return success(new CreateUserSuccess(user.userId))  
}
```

The question here is, how do we handle this? Do we just *throw* an exception here?

Nah. We can get smart about this. Consider if a controller were calling `createUser` — we'd want to report back what happened, wouldn't we? This is essentially equivalent of a 500-status error.

### Turn exceptions into meaningful errors

One approach is to ideate an entirely separate type of error (to distinguish from the domain errors — errors that arise with respect to the essential complexity of a feature). We'll call that an `ApplicationError` for example.

```
interface ApplicationError {
  message: string;
  error?: any;
}
```

With this, we can wrap the error returned from the exception, and we can provide some more meaningful context as to what happened.

```
class DatabaseError implements ApplicationError {
  message: string;
  error?: any;

  constructor (error: string) {
    this.message = "A database error occurred";
    this.error = error;
  }
}
```

Then, in our `createUser` function, we'll wrap the database's exception with an error that *we own*.

```
function createUser (request: CreateUserRequest): CreateUserResult {
  ...

  let user;

  try {
    // We don't own this code, so we'll wrap it in a try-catch
    user = db.User.save({ email, password })
  } catch (err) {
    return failure(new DatabaseError(err))
  }

  // Finally
  return success(new CreateUserSuccess(user.userId))
}
```

For the client, the *consumer*, the specifics of what kind of application exception was thrown (database, caching, etc) doesn't matter. It may matter to *us* for reporting or logging, and we can reach into the object to figure out what kind of error it is, but from a security standpoint, it's better to withhold that information from the API *consumer*, at least within the context of web application architecture.

So our approach is to just generalize it.

Adjusting our `CreateUserResult`, we merely add a new error type to represent the database error and any other kind of wrapped exception.

```
type CreateUserResult = Either<
  CreateUserSuccess,
```

```
UserAlreadyExists |  
EmailInvalid |  
PasswordDoesntMeetCriteria |  
UsernameTaken |  
ApplicationError // Any exception-based errors  
>
```

## Deciding when to throw exceptions

If we're writing code that others are going to rely on, the principle is to **fail fast**.

For example, when you're building a GraphQL API with Apollo Server, if there's a syntactic error your GraphQL schema, the server won't even let you start up until you fix the error in the schema. It throws an exception right away.

## Throw exceptions when a consumer should be forced to fix the problem

Consider these scenarios:

- A client forgets to enter the third argument of a command line script
- A client forgets to pass in their auth token in a payments SDK that you're building
- A client calls a URL that doesn't exist in a RESTful API
- A client doesn't implement an abstract method on a class that they've subclassed

I believe that these are scenarios where you'd be better off throwing an exception and getting the client to remedy their wrong as soon as possible. We can't continue with anything unless the client can properly give us everything we've asked for from them.

These notes are opinion, of course, and they're a part of my own error and exception-handling philosophy, but you'll want to decide on what makes sense for you and stay consistent with it.

## Handling nothingness

This is a common enough scenario, so it's good that we mention it.

What about when we ask for something and it's not found? For example, if I was to create a service that enabled consumers to query for a user by id, what should I do if it's not found?

We saw that a common approach was to return null. And honestly, that's not great because null acts like a monkey wrench getting thrown in the spokes of your bike. If we're at least **explicitly signalling null an option in the function signature**, then it gives the consumer a chance to decide how to handle it upfront. And that's better at least.

Again, we can do this type of thing with unions.

```
class UserService {  
    async getUserById (id: string): Promise<User | Nothing> {  
        ...  
    }  
}
```

What does null really mean anyway? We can improve the expressiveness by wrapping null with an actual type.

```
type Nothing = null | undefined | '';
```

So instead,

```
class UserService {
    async getUserById (): Promise<User | Nothing> {
        const userOrNothing = await this.db.User.findById({ where: { id } });
        return userOrNothing
            ? userOrNothing
            : '' as Nothing;
    }
}
```

And now the client has a chance to deal with it because we've made the implicit explicit.

## Summary

- The main idea behind this chapter is to make the implicit explicit. When we use a statically-typed language, this is a lot easier to do than in dynamically-typed ones.
- Traditional error and exception-handling techniques like logging and throwing or returning null **introduces the element of surprise** and merely puts off the work of writing error-handling code to the consumer, and that's a party-fowl in the realm of designing good APIs that humans enjoy using.
- Aggregating errors into a single statically-typed object expresses the domain better and improves the **discoverability** of how to use an API properly.
- When dealing with other people's code, we should wrap I/O in try-catch blocks and optionally wrap exceptions in more meaningful errors should they be important in the context of our own applications.
- When creating APIs for others, we should fail fast, throwing exceptions as soon as possible in order to improve the element of **feedback**, and **constrain** users from getting too far ahead of themselves and not knowing where they've gone astray.
- Use these strategies to decide on your own error and exception-handling philosophy, but make your approach consistent throughout the entirety of your application.

## Exercises

■ Coming soon!

## Resources

### Articles

- <https://khalilstemmler.com/articles/enterprise-typescript-nodejs/functional-error-handling/>
- <https://khalilstemmler.com/articles/enterprise-typescript-nodejs/handling-errors-result-class/>

- <https://khalilstemmler.com/articles/typescript-domain-driven-design/make-illegal-states-unrepresentable/>
- <https://stackoverflow.com/questions/45712106/why-are-promises-monads>
- <https://softwareengineering.stackexchange.com/questions/322842/are-promises-functional>
- <https://medium.com/inato/expressive-error-handling-in-typescript-and-benefits-for-domain-driven-design-70726eo6ic86>
- 200 OK! Error Handling in GraphQL

## Books

- A Philosophy of Software Design by John Ousterhout

## Part III: Phronesis

■ Phronesis means *wisdom* or mastery. This section synthesizes the techniques used by expert developers (the *phronimos*) to develop software in the real world. In the context of an XP project, from discovery to planning to deployment, what follows are the feature-driven techniques upon which professional software projects are crafted from.

### 13. Features (use-cases) are the key

■ A feature-first (or use-case-driven) approach to software development is, arguably, the best way to write testable, flexible, and maintainable code. To focus on the features (the use cases) is to focus on the essential complexity. Such an intentional approach to software development improves virtually every aspect of our work: from planning to architecture, testing, design, and beyond.

The way our industry's organized, for newer developers, it's almost as if we enter real software development projects flying by the seat of our pants. It's as if we're learning to fly a plane that's already ten thousand feet into the air. We sometimes treat greenfield projects like experiments. Now, don't get me wrong, experiments are good, but when there's a customer behind us paying good money to see their project come to fruition, I don't think it's ethical to be running experiments. If there's anything we know to be true about experiments, it's that they may very well fail. I don't know about you, but nothing else gives me imposter syndrome quite like not knowing if I'm carrying out my work similarly to the way a professional would.

I've been there many times. I was there when I joined a consultancy as a freshly graduated over-confident junior developer. I wanted to prove my worth at my first gig. Without a mentor or much knowledge of principles and software development processes, my outlook on coding was that "coding is just typing". If the user stories looked like they worked, I was golden. And if there were bugs, I'd just add more code until there weren't. The way I learned I messed up was getting called back into the office to re-fix bugs or re-implement features *after embarrassing customer disappointments*. Yikes!

I also had the same nervous, terrible gut feeling when I dived into my first contract gig, gave the customer a fixed price, built out what I believed worked, and then proceeded to write

untested bugs in text files. When the angry client called me and told me that code didn't work, I had to bite the bullet, come back, and right my wrong with several hours of free work to fix those bugs.

As modern software craftspeople, there's a lot to learn from planning, dealing with customers, making sure that your code will work, and feeling confident about it.

There a lot of different ways to develop software, yes — but to produce testable, flexible, and maintainable code *all while under time and cost constraints* is something of a feat. If someone said they knew how to do that, I'd pay close attention. This is evidently what drew me to XP.

We briefly discussed Agile and the concept of XP: Extreme Programming in Part I: Up to Speed. To remind you, XP is a mixture of business and technical practices. It is a relatively standard approach that claims it can consistently lead us to well-crafted code and happy customers. XP is our starting point for obtaining the knowledge and confidence to do the right thing.

Know the right thing and be doing the right thing, regardless of how you feel about it.

After several months of research, conversations with developers online, and reading books written by our *phronimos* (expert) developers, I've compiled a plan. What you'll find in the following pages of Part III is a condensed version of the most effective principles, practices, and patterns used by professional software developers. This section of the book introduces us, at a high level, to all of the other topics we aim to master throughout the rest of the book.

In Part III, we'll learn how the *phronimos* software developers work from start to finish. Specifically, you'll learn a philosophy for software design which helps us answer the following questions:

- How do we scope a project?
- How do we kick off a project?
- What's the best way to learn the domain?
- What's the best way to extract the requirements from the customer?
- How do we give good estimates?
- How do we keep the customer happy (so that they keep paying us)?
- How do we balance power between the customer and us? What should the customer have a say over, and what should we have the final say over?
- Just how involved should the customer be throughout the project?
- What happens if we can't meet the deadlines?
- How do we develop the initial project structure (the architecture)?
- How do we decide what constitutes good architecture?
- Should we account for the time it takes to get things set up?
- When does actual *design* happen?
- How do we prevent bugs and regressions in the codebase?
- How do we effectively test the requirements?
- Should we use mocks? Why or why not?
- What do we do about non-functional requirements? How do we discover them? How do we test them?
- How do we keep our code flexible and maintainable?
- How do we know when we're done?

Regardless of if you're a freelance developer, you work at a consultancy, a startup, or you're building your own startup company, I think you'll find something of value here.

---

In this first chapter of Phronesis, I want to start by introducing you to an outlook to software design that permeates all of the principles and practices. It's something that I carry in the back of my mind in discussions and while I'm coding. To me, it's obvious and elegant. It's an outlook critical to helping us succeed in serving both the needs of the customer and ourselves as developers. You may need to open your mind to a paradigm shift.

In this chapter, I'm going to ask you to be feature/use-case driven.

## Chapter goals

In this chapter, we will:

- Understand the shortcomings of the less strategic code-first approaches to software design
- Learn why features/use cases are the best way to capture the essential complexity of the problem at hand
- Learn about the anatomy and lifecycle of a feature/use case in an XP project
- Discover how we'll build sources of feedback into our feature/use case-driven approach to software design and how it addresses the needs of both the customer and future developers

## Code-first approaches to software development

If you'll recall, the goal of software design, as paid professionals, is:

To build products that **serve the needs of the customer** and can be **cost-effectively changed by developers**.

In Part II: Humans & Code, we learned about discoverability, understanding, HCD principles, and everyday developer use cases like locating features, understanding how things are configured, and adding/changing/removing features. We can now describe, from an HCD standpoint, why we believe our designs are good. We can analyze design decisions and determine if they help or hurt our ability to carry out common developer *use cases*. If we learn nothing else from here on, we're at least in a better position to succeed at the developer-oriented aspect of the goal of software design.

That's great, but that's not enough. There's still an entire customer side to the goal of software design we've neglected up until now. Allow me to illustrate how common code-first approaches to software development often miss the mark.

## Tactical programming

In 1. Complexity & the World of Software Design, we talked about **tactical programming** and **strategic programming**.

For a long time, I started all my projects using the *brute force-like* tactical approach. I'd sit down, figure out what I needed to build, and write code until it did what it needed to do.

When I was done, I'd run the code and manually walk through it to make sure it worked. If it did, then I was done. If it didn't, I'd spend a varying amounts of time with the debugging tools, hacking away until it finally worked.

Eventually, I realized that this kind of approach doesn't scale. I had no tests (because it was hard to write them after I had finished building out all the functionality) and very little design (since it was me, mostly just *adding* code). As projects grew larger and larger, it became increasingly more likely for things to go wrong.

I also later realized that the tactical approach to design was pretty dang hard to estimate. It always felt like I was on a never-ending marathon. I was going and going. The tactical approach wasn't such a big problem when I was working on personal projects. However, when I found myself working in a professional capacity, I saw how unideal it was. I soon started to think a little more consciously about how I was developing software.

## Imperative programming

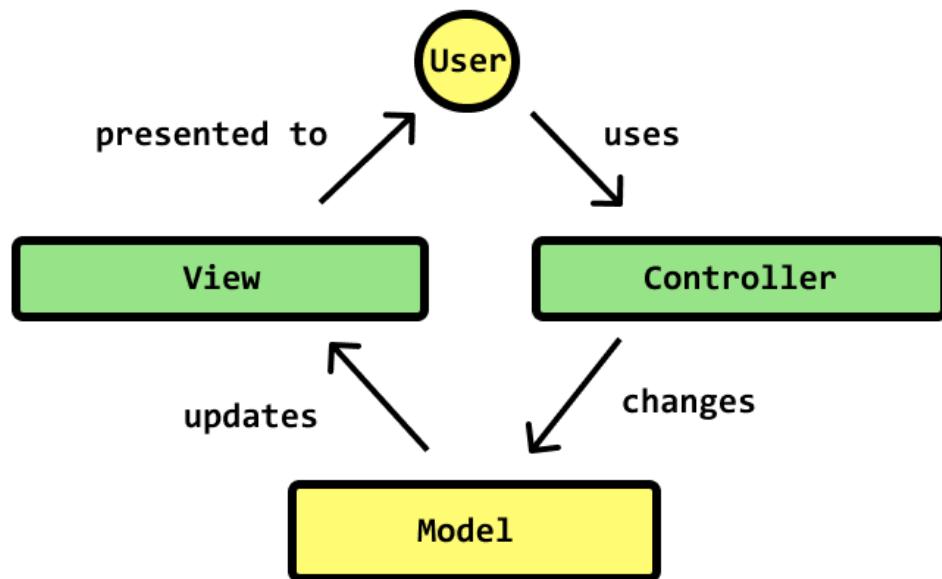
As early-career developers, we tend to think of development in an *imperative* sense: we conceptualize an application as the sum of its parts. The imperative way to approach building a web app is to single out **one of the technologies required to complete the project** and code that up first. Repeat that until everything works together. In a web-based MVC architecture, the components involved are:

Database + RESTful API (backend) + Front-end application

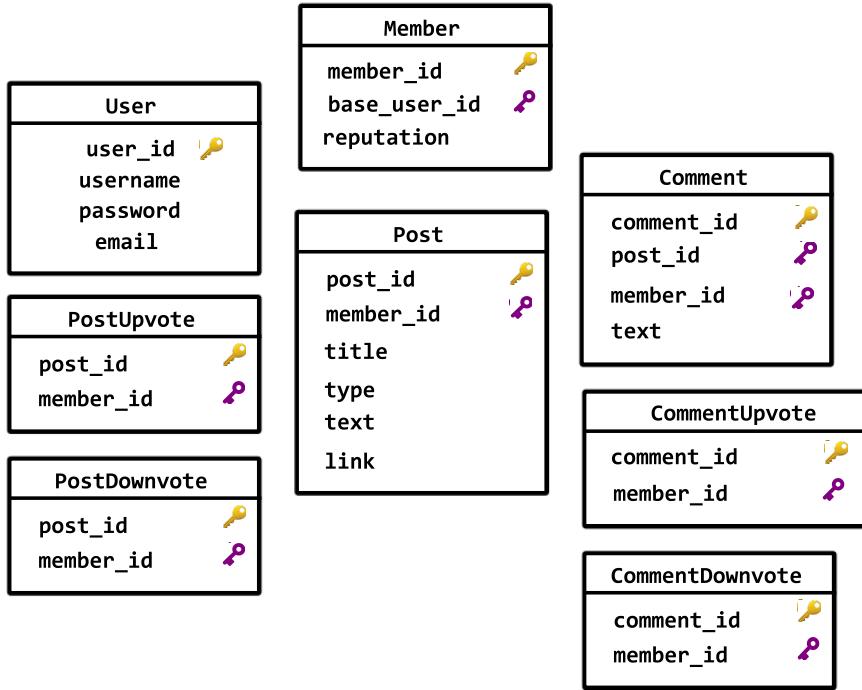
Based on my definition of imperative design, it's not uncommon to start either database-first, UI-first, or API-first.

# **Model View Controller**

**used to separate concerns between client & server**



Taking a **database-first** approach, I might start by drawing out all of the tables I'm sure I'll need, defining their relationships, and adding all their columns.



Database-first design approach. Start with the database schema.

A **UI-first** approach would mean creating all of the system's wireframes, sculpting out the components in the view, and then discovering each necessary API call based on what the UI component data requirements. In this approach, the fields and relationships present in the UI dictate the shape of the database.

Lastly, **API-first** design involves starting with identifying and enumerating all of the API calls up-front. Knowing what we need to build, I'd take guesses at the API calls that would accomplish that, think about the screens and the mixture of data required, and then eventually tally up all the operations that change state (commands) and ask for data (queries).

As we will learn much about in this part of the book, all three approaches are flawed, especially from a Domain-Driven Design point of view. However, if I had to pick one of the three as the *most correct* approach, I'd choose API-first design. Why?

Well, if we start from the database, it's very likely that we'll end up creating types, fields, classes, and tables that may not be essential.

UI-first design is great for discovering a *lot* of what we need to put in the API, but if we're to ascertain all of the features primarily based on what we can see, we'll miss a lot of capabilities (background processes, server-side logic, and so on) because the UI is not the single source of truth.

API-first design, however, is somehow the best because we focus on the essential complexity

first. Not UIs or database representations. We document the essential complexity — the features — as a contract that sits between the UI and the backend application. In a way, that's all an API is. A contract. An agreement.

■ **Feature/use case:** A feature is an operation that can be either a command (changes state) or a query (fetches data).

As you'll learn later on in the book (see Contracts), starting with the contract is advantageous for a number of reasons. We can exercise the *Dependency Inversion Principle* and decouple the client and the backend from each other, making it easier to maintain stability and flexibility since the client and backend don't depend on each other directly.

So if API-first design is the best approach, what's wrong with it? What's missing exactly? To answer that, we have to recap the anatomy of a feature.

## Data, behavior, and namespaces

Scott Bellware once told me that “feature-development is simple: it’s just data, behavior, and namespaces.” I believe this to be true.

Take the following RESTful API calls.

```
# Users-related API calls
/api/users/create
/api/users/edit
/api/users/verifyAccount

# Forum-related API calls
/api/forum/post/create
/api/forum/post/edit
/api/forum/post/comment/create
```

How have we utilized all three of those aspects here?

*Namespaces* have to do with the logical grouping of features. We can see that that's at play here because we've grouped the User calls and the Forum calls together.

Next is the *data* aspect of a feature. What are the various types of data we should be concerned with? Well, there's:

- Input data for **command-like operations** like POST and DELETE or query parameters and variables for **query-like operations** where we need to fetch some resource by an id.
- **Return data.** Sometimes commands return data (even if it's just the id of what's changed or it's to signal that something went wrong), but more obviously, **query-like operations** like GET return data.

That's not really shown here in a RESTful API definition. How do we encode all these different types of data in the API as a part of a contract? GraphQL does a better job at that. Check out the following GraphQL snippet for example.

```

type PostNotFoundError = { ... }
type MemberNotFoundError = { ... }
type AlreadyUpvotedError = { ... }
type UpvotePostSuccess = { ... }

union UpvotePostResult = PostNotFoundError |
    MemberNotFoundError |
    AlreadyUpvotedError |
    UpvotePostSuccess

type Mutation {
    upvotePost(postId: PostId, upvotedBy: MemberId): UpvotePostResult
}

```

We can document the `upvotePost` mutation by saying that it takes in a `PostId` and a `MemberId` and returns an `UpvotePostResult`. With `UpvotePostResult`, we're doing what we've practiced in 12. Errors and exceptions and use a union type to aggregate the various ways that the command can fail in addition to the one way it can succeed. This is good so far.

What's next? What's missing? Behavior, right?

Ahh, here's the tricky question. How exactly do we *contractualize* behavior in an API? How do we document success states? Failure states? Side-effects that may take place (eg, saving to a database, sending an email, charging a customer, etc)?

With REST or GraphQL, we're not able to document side-effects or business logic. Remember how we said that the leading causes of complexity has to do with state and sequence? **The way in which we organize and perform state changes what determines quite a fair amount of accidental complexity.**

Contractualizing behavior is important. It means figuring out and documenting what state needs to exist, how it can change, and what we should expect to occur after it has. It means doing this *ahead of time*, before we turn it into code. Just like we have *interfaces* at the class level, we'll need something to act as a contract at the API or service level (nudge, nudge — see 22. Acceptance Tests).

So, with API-first design, we can describe data and namespaces, but we can't contractualize behavior. That's a shame.

API-first, or rather *API-only design*, is not a complete or safe approach. Yet, many newer developers swear by API-first design, auto-generate CRUD RESTful APIs, and face a number of challenges dealing with behavior much later down the road.

## Drawbacks of an API-first approach

Without a complete understanding of the requirements for each feature and how they fit together, we'll encounter a lot of challenges. Here are a few.

- **Missed requirements:** There are two types of requirements — functional and non-functional. The first means that the system *should do* something. For example, “after

I sign up, the system should send me an account verification email.” Non-functional requirements are more about what the system *should be*. A non-functional requirement might specify that “this page needs to be able to handle a lot of traffic spikes throughout the day.” In that case, we’re speaking about software quality attributes like scalability and resilience. Like most code-first approaches, **failing to do proper discovery, planning, and testing, it’s more than likely that we’ll fail to fully discover all of the requirements — functional and non-functional**. You could say that **code-first approaches to software design often fail to uncover the essential complexity**.

- **Anemic domain model with lots of conditionals and duplication:** There are fundamentally two ways to organize business logic: using the *Transaction Script* pattern or using a *Domain Model*. We cover how the *Domain Model* pattern works in 24. An Object-Oriented Architecture. While you may not be familiar with the term *Transaction Script*, I’m confident that you’ve written code that conforms to its definition. The *Transaction Script* pattern “organizes business logic by procedures where each procedure handles a single request from the presentation” [Fowler]. Therefore, for a request to /api/jobs/new, you could expect to see something like:

```
//debugger
export class EmployerJobsControllers {
    ...
    handleCreateNewJob (req) {
        var _reqHelper = this.reqHelper
        var _Employer = this.Employer
        var _Job = this.Job

        var title = req.body.title
        var jobType = Number(req.body.job_type)
        var paid = Number(req.body.paid)
        // ... more detailed pulled from the request body

        // Check that required fields have been included
        ensureRequiredFieldsPresent()

        // Check that id's are of type 'Number'
        .then(checkIntegers)

        // If questions are included, ensure that they are
        // filled out in full.
        .then(checkQuestions)

        // Getting closer to save to the db
        .then((questions) => {

            /* Ensure that the user's profile has been completed.
             * This is business logic so that we don't let them
             * create new jobs without having completed their
```

```

* profile.
*/
_reqHelper.isProfileCompleted(userId)
  .then((isProfileComplete) => {

    if (!isProfileComplete) {

      return res.status(405).json({
        success: false,
        err: `Please complete your profile first before
              creating job postings.`
      })
    }

  } else {

    /* We need to know what employer Id to bind
     * to this new job for the posted_by column.
    */

    _Employer.findOne({
      where: {
        employer_base_id: userId
      }
    })
    .then((employer) => {

      var employerId = employer.dataValues.employer_id

      _Job.methods.createNewJob({
        title: title,
        posted_by: employerId,
        type: jobType,
        paid: paid,
        start_date: startDate,
        // More details
        ...
      }, questions)

      .then((jobCreateResult) => {

        return res.json({
          success: true,
          job: jobCreateResult.job,
        }

        /* Sequelize can't get the id for each

```

```
* bulk created row. So the id for each of
* will be shown as null.
* Front end apps will need to call for this
* by querying the appropriate /api/job url.
*/



questions: jobCreateResult.questions !== undefined
            ? jobCreateResult.questions
            : []
        })
    })
    .catch((err) => {
        console.log(err)
        res.status(500).json({
            error: err.toString()
        })
    })
})
})
}
})
})
})
.res.json({
    err: err.toString()
})
})
})
})
}

.catch((err) => {
    console.log(err)
    res.status(400).json({
        error: err.toString()
    })
})
}

function checkQuestions() {
    ...
}

function ensureRequiredFieldsPresent () {
    ...
}

function checkIntegers() {
    ...
}
```

```
}
```

Notice how this method seems to be a little-bit scatter-brained? We'd call that a lack of cohesion. In the context of MVC, developers unfamiliar with testing make very little distinction to keep different forms of logic separate. Business logic, persistence logic, application logic, and so on — or more succinctly, *core code* and *infrastructure code* — all live in controllers and “services” (see 24. An Object-Oriented Architecture).

It can seem productive to move logic to services, but if not done properly, we'll just find ourselves with an **anemic domain model** (read here). Anemic domain models are rather fallible, often turning to mush and resulting in something hard to maintain.

The messiness of the anemic domain model is amplified when new business rules arise. The most common scenario that this happens in is when we find ourselves needing to slot in code that occurs “after”, “before”, or “if” something has happened. For example, consider the various number of things that need to happen *after a job is posted* on a job board application.

```
function afterJobPosted (jobId: string) {  
    // Charge the customer.  
    // Add the job to our queue of new jobs for the week.  
    // Trigger a gatsby build to rebuild our Gatsbyjs site.  
    // Post the job to social.  
    // Do a bunch of other stuff.  
    // Send an email to the customer letting them know their job is active.  
  
    // and more...  
    // and more.....  
}
```

As we go on to truly understand the nature of features and use cases in 16. Learning the Domain, you'll find that there's a *temporal* aspect to feature design. By *temporal*, I'm relating to the notion of *time* and the fact that features tend to chain together and occur in a sequence. This phenomenon is also why Event-Driven architectures are so effective at representing real-world processes.

Overall, the *Transaction Script* pattern is the simplest, most *procedural* way to write code. For quick, one-off projects or trivial CRUD applications with little to no business logic and few sequential features, it's the most practical approach. However, for applications with strict testing requirements and more complexity, the *Transaction Script* transforms into an anti-pattern.

- **We are much more likely to introduce accidental complexity, poor design, and poor architecture:** This is a philosophical problem. How can we do the simplest thing possible if we're not starting with a complete picture of the *essential* complexity? They say that architecture is something we develop in order to support the features we need to build. But if we're not crystal clear about the features and what we need to build to support them, then our architecture will surely be lacking. This affects testability as well. One of the first things we do in a project is design the architecture so that it can support the tests we need to write (see 26. The Walking Skeleton ). Bad architec-

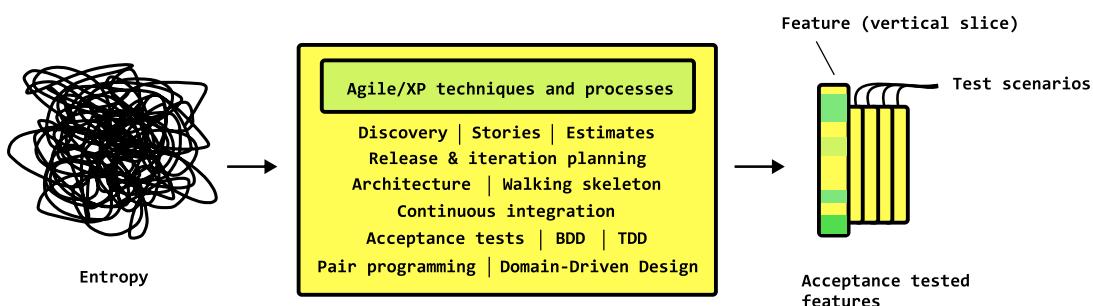
ture negatively affects our testing strategy. Therefore, the downstream problem that arises is not understanding the essential complexity, which leads to bad architecture, which leads to bad testing capabilities, which leads to bad design, which leads to bugs and regressions.

Hopefully, you see how important it is to get the **behavior** part right. Alright. Here's how we avoid these mistakes.

### A feature/use-case driven philosophy for software design

Features represent the essential complexity of the problem at hand. Therefore, my philosophy for software design orbits entirely around features as the **atomic units of value**. Our success rides on our ability to untangle, understand, and implement features in a consistent and reliable fashion.

Recognize that we start with complete and utter entropy, and that we need to humble ourselves to learn the domain and the wrangle the features we need to build from within it. We should also respect the fact that it is *extremely likely* we'll implement features incorrectly if we don't have **real customer involvement** in their specification and design. This is where a number of Agile/XP techniques and processes come to our aid. They are the way we nurse customer-specified, fully tested features into existence.



Some of the ways our feature/use-case driven approach to software design helps us **serve the needs of the customer** is by understanding the scope, estimated cost, and effort required to develop functionality. This lets customers plan out releases so that they can get their business, product, and marketing strategies in order. A feature/use-case driven approach to software design also gives customers the power to change their minds at any time. Whether it's to add new functionality, remove existing functionality, or completely abandon the work that's being done for something else, we can confidently communicate the consequences and adjust the plan. That's the real power, a flexible plan.

From the developer side, we **serve the needs of future maintainers** by ensuring that any code that enters production, enters under the context of a failing *acceptance test* — a test automated by us but specified *by the customer* using a technique called BDD (behavior-driven development). A feature is complete when its acceptance tests (and associated unit, integration, and E2E tests) pass. This makes it easy for future maintainers to locate features,

understand their behaviour, and conduct the rest of their activities with confidence.

## Feature = use case

Throughout the remainder of the book, you'll hear me use the word *feature* and *use case*. I'll flip between them, so just understand that they're both the same thing. And while both refer to the same concept, a *use case* is probably the more technically correct term to use (see Alistair Cockburn's "Writing Effective Use Cases" [2000]). Additionally, in Part IX, we rely heavily on the Use Case pattern to realize these atomic units of value in code form.

Essentially, I want to get you to be hyped-focused on the use cases. That means sniffing use cases out of conversations with customers, understanding when we're writing pointless code (not backed by a use case — just adding accidental complexity), and figuring out how to write use cases as acceptance tests.

Use cases are so important because they transform software development from being a creative, lofty, *I'm-not-sure-when-I'm-quite-done* kind of thing into a definite, formal, focused practice which can be mastered.

## The anatomy of a feature

Data, behavior, and name-spacing is still the best way to think about features. Here's the complete picture.

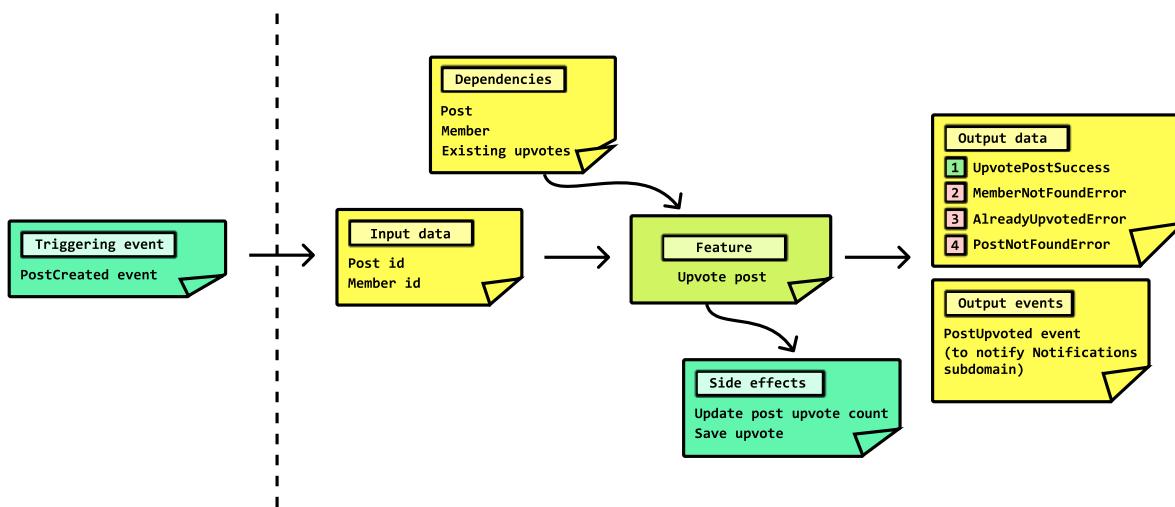


Image inspired by Scott Wlaschin's idea to start by focusing on the inputs and outputs from his excellent book, "Domain-Modeling Made Functional", which is highly-recommended reading.

## Data

There are **four types of data** that we need to concern ourselves with when discovering, designing, and implementing features: *input data*, *dependencies*, *output data* and *output events*.

- **Input data:** Most of us are familiar with input data. It is what is *necessary* and *sufficient* to execute the feature. This means that for an upvotePost use case, depending on the programming paradigm, architecture, and approach, instead of passing in a Post and a Member, we could pass in thepostId and memberId instead. Yes, we'll likely need the Post and the Member objects to execute business logic, but we offload the work of retrieving those objects from within the use case itself, most often through the use of the R\_repository pattern\_. That transitions nicely to the next type of data.
- **Dependencies:** Most features need a little bit of help to perform their work. To keep the behavior in our feature/use cases as cohesive as possible, we rely on the help of *other objects (or functions) with specific capabilities* to do things like fetching and persisting objects to a database, checking for the existence or non-existence of an object, or connecting to an external API like Stripe, the Google Maps API, and so on.
- **Output data:** As we've discussed, there are a number of ways that a feature can succeed or fail. It's important to make the success and failure states explicit so that the API is complete, expressive, and so that clients are well-positioned to deal with the result.

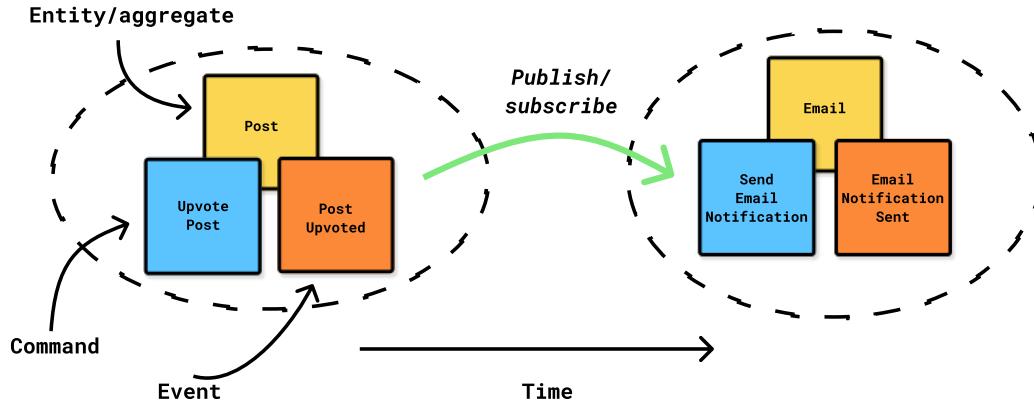
Apply the techniques from 12. Errors and exceptions, modelling the output result using the Either or Result pattern.

```
type UpvotePostResult = Either<
    // Success
    UpvotePostSuccess,
    // Failures
    MemberNotFound |
    AlreadyUpvoted |
    PostNotFound
>
```

- **Output events:** This is new to us because we haven't discussed this yet, but the key to **decoupling features, keeping them cohesive, yet chaining them together** is to use events.

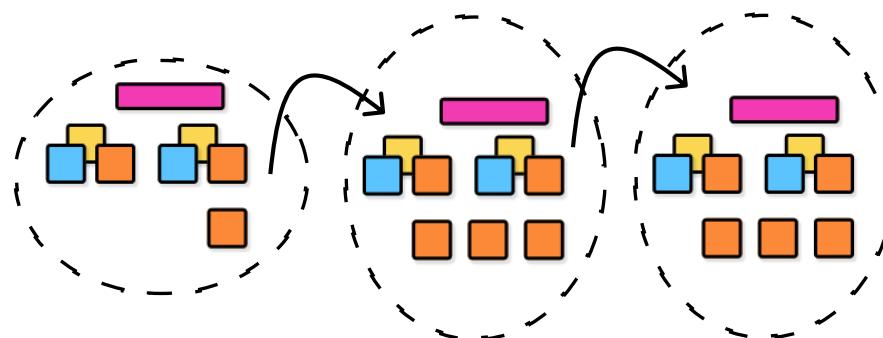
Features/use-cases that change state are called commands. Commands are written in imperative, present-tense style like we're giving an instruction to somebody: upvotePost, deleteComment, createAccount. Alternatively, we write events in past tense style like postUpvoted, commentDeleted, and accountCreated.

When a command succeeds, we open up the possibility to chaining features by publishing a Domain Event. Utilizing a piece of our architecture dedicated to notifying other interested modules, subdomains, or even micro-services that the event occurred (typically a queue, a bus, or event-listener), we can chain functionality in a clean, decoupled way.



Events are the better solution to setting up “before” and “after” chains. This is the temporal aspect behind the essential complexity I referred to earlier. We can think of systems (real-life and computerized) as a collection of *features* that can be organized by chronological events to depict how we maneuver through them. For example, the *state machine* for a washing machine can go from OFF to ON and then to WASH, but it can’t go from OFF to WASH to ON. That doesn’t make any sense.

When we’re asked to digitize an annoying real-life workflow for someone, what we’re really being asked to do consolidate a number of events (separated by use cases) that naturally occur within the real-world.



Output events are all about enabling another area of our code to do something in response to published events. Example: “After JobPosted, execute the postToSocialMedia use case”.

In 16. Learning the Domain, we discuss these ideas in more detail. We also learn techniques to uncover all of the commands and events in a domain before we write our first lines of code.

## Behaviour

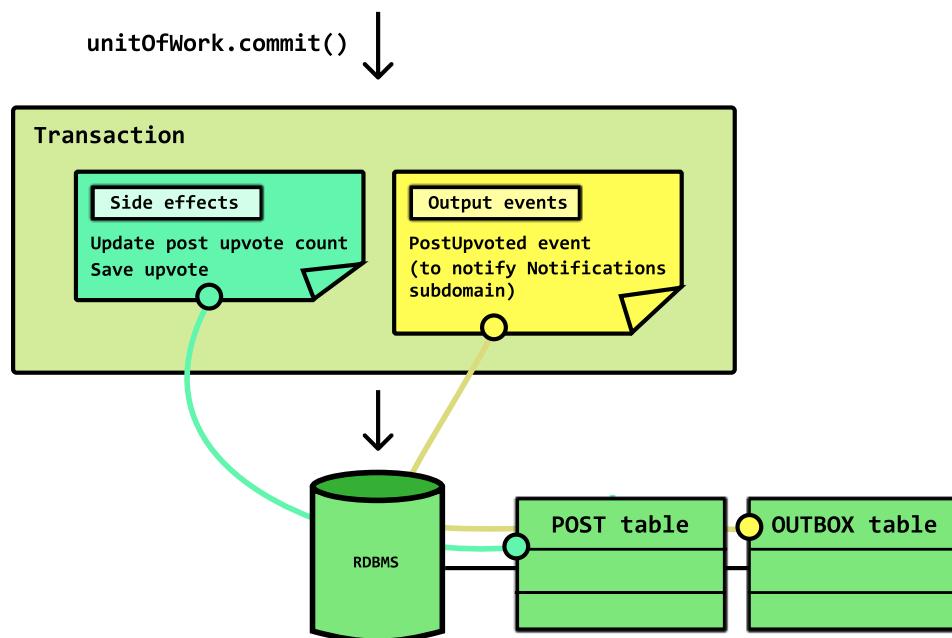
There are two aspects to behaviour that command-like features have: side-effects and triggering events.

- **Side-effects:** Command-like features change the system in some way, like adding, removing, or updating rows in a database. A lot of commands actually add, change, or remove multiple rows at the same time. One big challenge in web programming is figuring out how to *rollback* operations if one part of the request fails. This can get messy.

A principle we learn from functional programming is to push side-effects to the boundaries of an operation. Make all of the changes at the end of the request. In the context of a web request, it means utilizing some sort of transactional object capable of letting us either commit all the changes atomically in a single go, or to do nothing. The Unit of Work pattern, while guilty for adding complexity, is helpful if we want to thread a transaction through all of the changes that need to happen at the same time. When we're ready, we can call `commit` once we're sure that we should save everything to a database.

This comes in handy, especially when we're dealing with *Domain Events*. We need to make sure that *Domain Events* get saved at the *same time* as the application database changes do, because if we were to successfully send a *Domain Event* down a message queue but notice that the database transaction failed, we're in trouble.

Later on in the book, we'll learn more about these challenges and how to overcome them. The Transactional Outbox pattern (Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS) specifically, is *key* to helping us persist both the domain event and the data at the same time.



How the Unit of Work and Transactional Outbox patterns help us persist state changes

and domain events at the same time.

- **Triggering events:** Use cases are invoked by either an actor (be it a user, a server, an automation) or through a subscription to an event. To decouple logic across separate modules, subdomains and microservices, we subscribe to domain events. Not every feature has a triggering domain event. But for the ones that do, we need to ensure that our architecture supports the ability to publish and subscribe to them.

### Name-spacing

Lastly, the most straightforward: name-spacing. We need to concern ourselves with the *Subdomain* that features belong to and the name we give them.

- **Subdomain:** We explore the concept of subdomains in 16. Learning the Domain and in Part IX, but generally speaking, you can look at any application and decompose it into a number of subdomains. In DDD, a *Subdomain* is a **smaller piece of the entire problem space**.

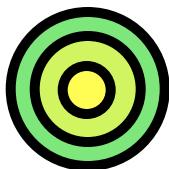
For example, if a company wanted to build vinyl trading application, more often than not, there would be more to the story than *just* trading vinyl. In addition to the trading aspect (Trading), the company also has to account for several other concerns: identity and access management (Users), cataloging 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 subdomain is responsible for a certain set of features within the business. Some features are more appropriate to live within the context of the Billing subdomain (like making payments and creating paid subscriptions) while others make more sense to be handed off to the Users one (like verifying an account or resetting a password).

- **Name:** Lastly, the name. There's not much else to say here other than by separating features into their appropriate subdomain, we prevent namespace confusions and collisions.

## The lifecycle of a feature

Now that we have a base understanding of what a feature is (data, behavior, name-space), it's time to discuss the different shapes a features takes throughout the lifecycle of a project: from initial conversations with the customer to implemented, tested, and deployed to production.

### Discovery

At the start of the project, we have to scope out — at a large scale — what the customer wants and how long we think that'll take. This doesn't have to be accurate, but it should get us started. We rely on past experiences to make this an estimate that's not completely unreasonable.

Next, we have to learn the domain. The DDD community recommends rallying up customers, domain experts, developers, and anyone who can help elucidate the way the business works. With everyone sitting together in one room or together online, we use Event Storming (or Modelling) to build knowledge and learn the domain.

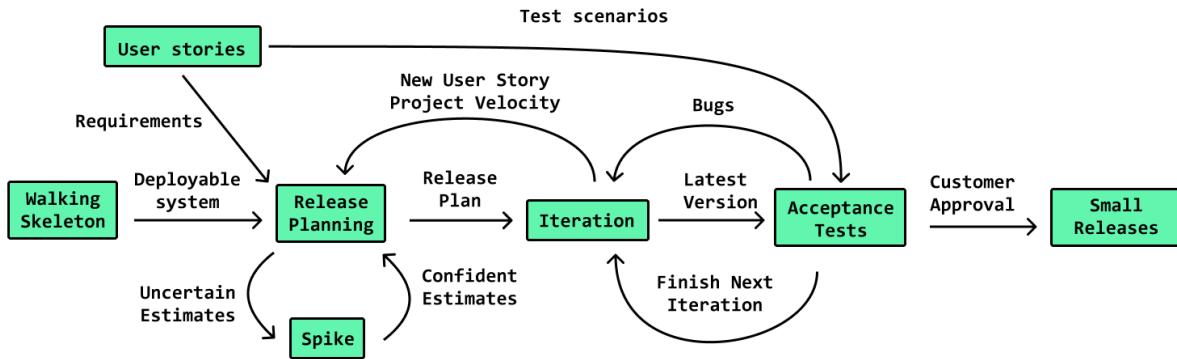
At the end of this, we'll have identified the features. At this stage, we refer to the features as (user) stories. We'll also have identified some of the key problem areas that we'll need to spend more time investigating. Maybe we've never used the CQRS before, but based on the non-functional requirements, we realize that it's something that we should investigate. When we're uncertain about an estimate, we do something called *spiking* here. It gives us time to research and come up with a proper estimate.

At this point, we don't know too much about the details of the features yet (data, behavior, namespace), but we at least identified a number of them (as stories) and have a common language to discuss and plan them with customers.

- In 15. Customers, we discuss the necessity of having an on-site customer. We also discuss the driving vs. steering relationship we strive to maintain.
- In 16. Learning the Domain, we learn about Domain-Driven Design and use Event Storming/Modelling to uncover stories and a high-level picture of the domain.
- In 17. Stories, we learn good story writing principles, how to turn functional requirements into stories, and how to deal with non-functional ones.

### Planning

Now that we've identified the stories, customers put together a release plan — a plan that specifies the major releases that makes sense from a business perspective. It's important to note that this is again, an *estimate* and something that's not set in stone. We communicate this with the customer.



- In 19. Release Planning, we walk through the process of allocating user stories to releases and iterations.
- 20. Iteration Planning, we learn how to break stories down into development tasks that can be divvied up to developers on the team.

## Estimation

Customers decide which stories they would like to be implemented in iterations — a cycle of work that generally lasts one to two weeks. This is only made possible by our estimates.

Based on our previous experiences building features, we estimate features using the expected amount of effort we believe they'll take. We split stories that are too big, we spike stories that we're uncertain about, and we join small stories.

When we plan our iterations, we make sure to only take on stories that add up to less than or equal to the current velocity we're completing stories at. We measure our velocity each week and re-adjust the iteration and release plan accordingly.

- In 18. Estimates & Story Points, we discuss effective estimation techniques.
- In 21. Understanding a Story , we learn how to interview a domain expert to uncover and document the essential complexity for a feature (data, behavior, namespaces) before we turn it into code.

## Architecture

Before we go about building our first feature, we'll need the architecture to support it. To decide on the correct architecture for our project, we look to both the functional and non-functional requirements.

The goal is to develop a deployable system that works end-to-end. A common technique is to perform a zero-functionality iteration where we spend the first two weeks building out what is known as a *walking skeleton* — a minimal implementation of the system that performs a small end-to-end function and proves that all of the main architectural components work together.

Before we write our first acceptance test, we should have a broad-stroke idea of the architecture, an understanding of our testing strategy, and all the scripts necessary to build, test, and deploy to a production-like environment in an automated way.

- In 23. Programming Paradigms, we learn about the three main programming paradigms and discuss the correct mental model for Object-Oriented Programming.
- In 24. An Object-Oriented Architecture, we learn how to apply the separation of concerns principle to compose a layered architecture which opens the door for various testing strategies.
- In 25. Testing Strategies, we learn how to take advantage of a layered architecture to write end-to-end, integration, and unit tests effectively.
- In 26. The Walking Skeleton , we learn how to build the initial testing architecture leading up to the first acceptance test.

## Acceptance tests

At this stage, features transform from user stories into acceptance tests. With a test architecture built out, we're ready to implement a number of features within regular two week iterations.

To write acceptance tests, we work with the customer and get them to specify scenarios that define what completion looks like. Using BDD, the customer can write acceptance criteria that is readable by business folk.

We spend this time with the customer to make sure that we really understand a feature from the inside-out. We should have a clear idea of the data, behaviour and namespace at this point.

Using TDD, we turn this specification into automated tests and code it until it passes.

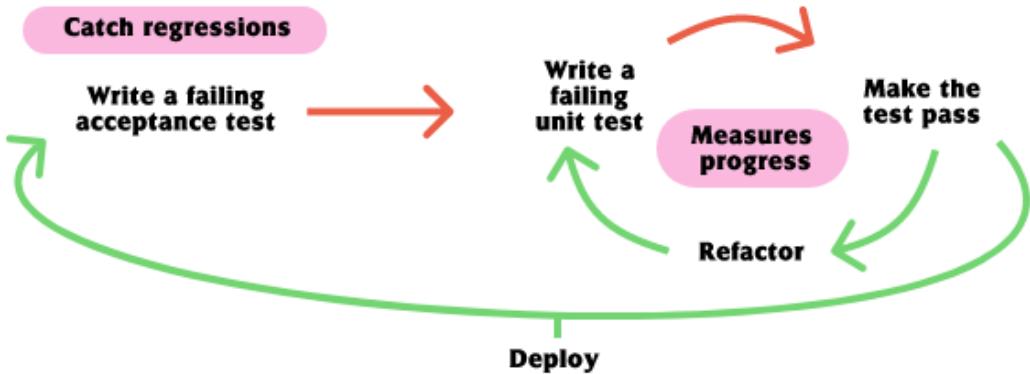
- In 22. Acceptance Tests, we learn how to write BDD-style acceptance tests.

## Design & implementation

Acceptance tests can be written as either end-to-end or unit tests or both. End-to-end tests cover more ground, but unit tests run faster.

Regardless of which approach we choose to write our acceptance tests, **to realize the essential complexity of a feature, we perform a technique called Double Loop TDD**. This means we maintain an *outer* acceptance test loop and an *inner* unit test loop. When we code at two different levels like this, we maintain a **single level of abstraction at a time** (see Part VII: Design Principles) which makes the highest layer of code declarative and easy to read, even by non-technical people.

To make the outer loop test pass, we create objects that do the work we're prescribing. As we go about creating these objects, we introduce new surface area for things to go wrong. So we take special care to — in a test-driven way — create new objects paired with smaller *inner loop* unit tests.



The outer acceptance test loop prevents regressions and the inner unit test loops measure our progress towards completing a feature.

Once the inner and outer loop tests are passing, we're done working on that core application functionality. We likely still need to test the way it integrates with our infrastructure (through integration tests), but we can rest assured knowing that the essential complexity has been realized in code. Congratulations.

- In 28. Test-Driven Development Workflow, we walk through the acceptance testing workflow using a testing strategy capable of dealing with two types of code: core and infrastructure code.
- In 27. Pair Programming, we cover various pair programming switching techniques as well as a few ways to make pair programming sessions enjoyable and effective.

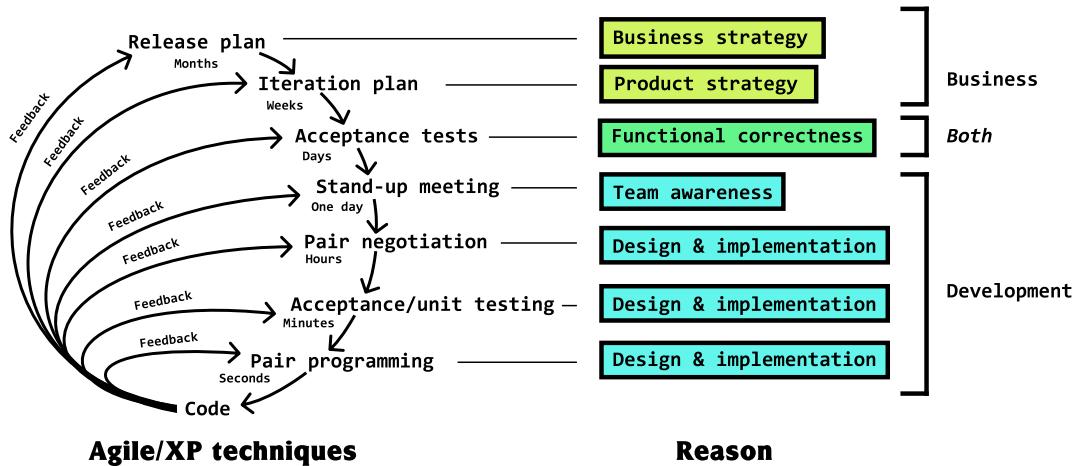
## How to solve our goals in software design

What you've just read is fundamentally, the way that the phronimos software developers produce software within the context of a professional software project. Allow me to summarize how this approach works to serve the needs of customers and maintainers.

**For the customer:** We serve the needs of the customers by using Agile/XP processes that empower them to adjust their business and product strategy when change occurs. And it will. We also use techniques that ensure what gets built is functionally correct. That's one side of what we're trying to learn how to do well — serve the needs of the customer.

# How to Serve the needs of the customer

Consistently build the right thing in a timely fashion

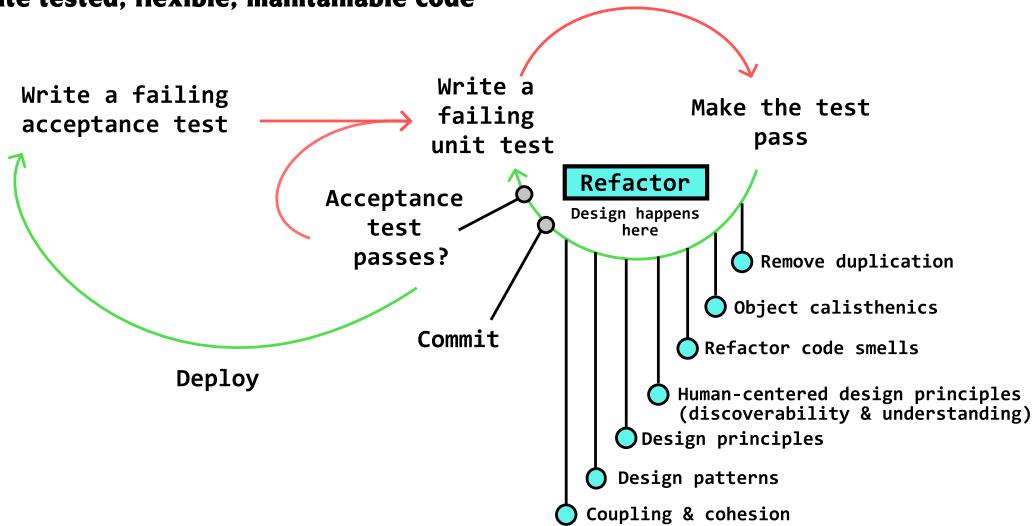


**For the maintainers:** With respect to serving the needs of the developers, we have a consistent, repeatable, and simplified model that describes how we can go about writing tested, flexible, maintainable code.

When we get to the actual *implementation* phase, we focus on making failing tests pass with the simplest possible solution first, and then apply everything we know about software design within the *refactor* phase. Take note of the various techniques, principles, and practices listed in the illustration below.

# How to Serve the needs of developers

Write tested, flexible, maintainable code



Throughout the rest of this book, beyond Part III: Phronesis, we'll learn how to master these techniques, principles, and practices.

## Building sources of feedback (mistake proofing)

The reason I love this approach so much is because if we do it correctly, it's fundamentally mistake-proof. Look back to the drawing of the XP circle of life and how many feedback loops there are. It's designed specifically to account for human error. The circle of life shares similarities with *poka-yoke*, a Japanese term formalized as a part of the Toyota Production System that means "inadvertent error prevention".

There are three symptoms for bugs:

1. When the requirements are correct, **but the code doesn't work**
2. When the requirements are incorrect or incomplete, **and the code works correctly — but this doesn't matter**
3. When the requirements are correct, **but the developers misunderstood and the code does something else**

To prevent the first symptom of bugs, we ensure that all code that enters passes through tests. That's easily solved. We just have to stay disciplined.

The second symptom happens when there's a disconnect between who the customer really is. In XP, we should be working with a domain expert — preferably the person who needs the product and will be using it. This doesn't always happen. If you have a product manager, they become the person who gives you the requirements. If the PM fails to properly

understand the problem, then we're only going to be able to produce what's about as good as what we get. This is why it's important to get the customer to assist in writing the acceptance tests.

Finally, for the third symptom of bugs — this is what happens when we don't interact with customers, learn the domain, and work to build a common language of the domain (also known as the *ubiquitous language* — see 16. Learning the Domain). We want to, as much as possible, avoid playing the game of telephone. If we follow the rules of XP, we should be in good shape.

■ **Principle:** Remember, all features must pass through communication, feedback loops, and tests to exist. That's the key to preventing bugs.

## Questions

### Isn't this just XP?

- For the most part, yes. But if you look into Feature-Driven Development, Lean software development, Rapid application development, and other Agile methodologies and frameworks, you'll notice that there's a common thread — and it's that we push a feature from one end of the process to the other.

### Do I have to do everything here?

- In my experience, very few companies practice all of these Agile techniques to a tee. Some have different names for things. Some do pair programming on certain days of the week or times in the day. And that's OK. If we're on the same page about the fact that *more communication channels* means we're more likely to misunderstand a feature, that *it's better to discover what's going wrong upstream (earlier) rather than downstream*, and that *automated tests are how we prevent regressions*, then it's up to you and your team to decide on the right amount of these techniques and practices. Principles first. Understand the why, and the how-to's will come.

## Summary

- Tactical and imperative approaches to software design tend to introduce a lot of accidental complexity. Why? Because we haven't taken the time to understand the essential complexity first. Building out functionality without fully understanding it is like learning how to fly an airplane while it's already in the air.
- Since features represent the essential complexity of the problem at hand, adopt a feature/use-case-driven mindset to software development.
- In real-world projects, infrastructural concerns like libraries, frameworks, and libraries matter, but they're merely the platform upon which we develop features: the atomic units of value.
- A feature is comprised of three main parts: data (input & output data, dependencies, and output events), behavior (side-effects and triggering events), and a namespace (subdomain).
- A feature goes by many names and takes on many shapes throughout the lifecycle of an XP project. The initial form of a feature is an undocumented customer pain-point

or laborious real-world workflow. The final form of a feature is a passing automated acceptance test which runs in a production-like environment.

- The way to reduce bugs in feature development is to build multiple sources of feedback. We use various techniques like Event Storming, acceptance testing, pair programming, unit testing, automated builds, and more.
- The business aspect of real-world feature development is to do so reliably, consistently, and in a timely fashion. In this section, we'll learn about the XP practices that helps us uncover, plan, estimate, and reliably develop features with respect to budgets, deadlines, expectations, and human error. We need both business practices and technical practices. XP is the way we'll accomplish this.
- This section of the book acts as the conduit to all the remaining sections in the book. View Phronesis as a high-level overview of the principles, practices, and patterns used by the phronimos developers — the experts — in writing testable, flexible, maintainable code. In subsequent parts of the book, we seek to master the topics explored here.

## References

### Books

- Domain Modelling Made Functional by Scott Wlaschin
- Writing Effective Use Cases by Alistair Cockburn
- Planning Extreme Programming by Kent Beck and Martin Fowler

### Articles

- [https://en.wikipedia.org/wiki/Feature-driven\\_development](https://en.wikipedia.org/wiki/Feature-driven_development)
- <https://en.wikipedia.org/wiki/Poka-yoke>

## 14. Planning

■ In XP, we use planning techniques that lets the customer prioritize and coordinate activities, and gives us the power to recalibrate when things inevitably get off track.

Like many preadolescent boys, I was obsessed with super-heroes. Spider-Man, specifically. The best way to describe my obsession is *drawing-spider-man-pictures-on-the-walls-with-invisible-markers*-level obsessed.

In the 2004 film, Spider-Man 2, there's a scene where Spider-Man stops a speeding train full of passengers from flying off the rails.

His valiant efforts are successful, of course — but at what cost? Well, his suit rips (tailor bills off the roof), his mask flies off and everyone sees his face (similar embarrassment to your demo not working in front of customers), but most importantly — one can only imagine the excruciating cost to his body. How would you feel holding back a speeding train using nothing but your two hands (I also seriously doubt Spider-Man's super-hero-benefits package covers Swedish massages).

Jokes aside, there's just no room for super-heroism in software development. I mean, we are in search of ways to expend relatively consistent amounts of effort to accomplish quality

work — and to do it on a time schedule. There should be no swooping in at the last minute with energy drinks and all-nighters to meet deadlines.

Evidently, it seems to be that the notion of time, deadlines, rather — is what harbours the most fear within developers working on *seemingly* Agile projects. To many of us, deadlines are the gamma ray — the kryptonite — or the death star of software projects.

When we fear deadlines, we feel as if we should cut corners and make excuses for not doing the right thing. “I didn’t have time to write tests. I’m just trying to meet the deadlines.” It’s important to note that when corners are cut, someone — whether it’s sooner or later down the line — ends up paying for it (like future maintainers). And that’s a direct violation of what we’ve deemed to be in good virtue for writing code.

The key to overcoming the deadline problem is to acknowledge that deadlines can change. This starts by shifting the way we think about the relationship we have with customers. Remember from 2. Software Craftsmanship, it’s “**Not only customer collaboration**, but also productive partnerships”. We need to change the thought process and culture of seeing the customer as a super-villain working against us — as the one that sets the ticking time bomb that we sometimes equate to be the deadline.

Instead, we need to see the customer as someone that we can work together with. Someone we perform **continuous planning** with. Someone that we respect enough to let them know when things are going a little bit slower, and that we’re going to need more time. And that goes the other direction too.

Collaboration in planning is indispensable. Because if there’s anything that can be considered a constant in software development, it’s change. From both ends: us and the customer. Let’s learn how to work together.

## Chapter goals

In this chapter, we’ll discuss:

- Why unacknowledged fear is the main reason software projects fail
- The customer and developer bills of rights so that we know what to reasonably expect from each other
- How planning helps us prioritize, coordinate, and recalibrate when things inevitably get off track
- The difference between the initial and the continuous plan and how to use the initial one to kickstart a project and the continuous one to give better estimates, keep progress flowing, and give both us and the customer the ability to adapt to change

## The most common reason projects fail

According to a trio of phronimos trio developers (Beck, Fowler, and Martin), projects don’t fail for the reasons many people think they do. It has less to do with skill, tools, or resources.

I know we tend to say that *deadlines* are the reason why projects fail, but let’s consider another possibility. What if the customer knew ahead of time that it didn’t look like we were going to be able to deliver the latest release by some certain date? What if we held back on that feeling of needing to rush cut corners, add unnecessary technical dept, and instead,

just let them know honestly and transparently, “hey, so it looks like we’ve only been able to get about 70% of the features done so far — it doesn’t look like we’re going to be able to complete this on time. We’ll need another two to three weeks at this pace”.

The reason most projects fail is because of unacknowledged **fear**.

Customers are scared of a number of things. They’re scared that:

- They’ll be stuck working with developers who lack professionalism, who could care less about the business, and that if things don’t work out, the developers won’t care to right the situation
- They’re not going to get what they asked for
- They’re going to pay way too much money for something way too little
- They won’t ever see a plan that makes sense and assures them that they are well taken care of
- The plans that they *do* get from us are going to be ridiculous, over-confident, and that we’re not going to be able to deliver on what we say we will
- They will be completely in the dark as to our progress so far
- They’ll be stuck with their earliest decisions and that they won’t be able to pivot on the plan when the business demands it
- That no one will be honest with them

And as developers, we’re scared that:

- We will get tasks that we don’t know how to complete
- We’ll face challenges that we won’t know how to solve
- We’ll be asked to do or build things that we can’t make sense of
- We’re not smart enough
- We’ll make bad technical decisions that will hurt our ability to make progress
- We’ll be bossed around and told what to do without the opportunity to voice our own opinions
- We won’t get clear requirements
- We’ll have to sacrifice doing the right things in order to meet deadlines
- We’ll have to figure out how to solve hard problems without access to a domain expert for help
- We’ll run out of time

These are clearly valid fears for both parties. Unfortunately, instead of *addressing* those fears, the common approach both parties take to remedy these very real concerns is to put up walls to protect ourselves. Both customers and developers lie, exaggerate, refrain from telling the honest and transparent truth, institute processes, rules, mandatory <sup>\*\*</sup>documents, and refuse to share important information — such as knowing that we probably won’t meet our goals at the pace we’re going — with a sole focus on removing ourselves from fallibility.

When you take your car for a tire change, you expect the wheels to be on tight, to get your car back without any scuffs, and for it to be done promptly. It goes both ways — the automotive technician has expectations for *you* as a client too. They expect you to show up on time, give them your keys, and give them clear answers on the services that you’d like provided. Just like any other trade, **customers and developers need to establish what we should expect from each other**.

## **Bill of rights**

“When our fears are acknowledged and our rights are accepted, then we can be courageous. We can set goals that are hard to reach and collaborate to meet those goals.” — Fowler, Beck, Martin from Planning Extreme Programming

### **For customers**

In any software project, here's what customers have a right to:

- Customers have a right to know what the plan is, to know how long it will take to accomplish milestones, and to know how much it will **cost** to do so.
- Customers have the right to choose the stories that will yield the most value each week.
- Customers have the right to be informed of schedule changes in time to be able to choose how to reduce the scope of the work to the original date.
- Customers have a right to see the current progress in a running system at any point in time, where progress is measured by automated acceptance tests specified by them.
- Customers have the right to cancel the project at any time and be left with a useful working system reflecting the total investment to date.
- Customers have the right to change their mind, swap out functionality and rearrange priorities without having to pay unreasonable costs.

### **For developers**

And with respect to us, our rights specify that:

- We have the right to know what tasks need to be done, and the priority associated to each one.
- We have the right to carry out technical practices that will yield quality work, always, at all times.
- We have the right to ask for and receive help from peers, managers, and customers.
- We have the right to make and update our own estimates.
- We have the right to accept our responsibilities instead of having them assigned to us.

## **Stay on script**

The bill of rights for customers and developers makes life a whole lot easier. Knowing what to expect of each other means we just have to focus on staying on script.

For example, if a customer tells you that they want to swap out a story, that's perfectly valid with respect to the script. And we'll know that. As a part of the XP script, we go on to help them re-work the new story into the iteration if possible, regardless of how excited we were to get to the story that we're swapping in for the new one.

## **Why plan?**

We are constantly, continuously, and collaboratively planning in an XP project.

Planning allows us to do three things: prioritize, coordinate, and recalibrate.

## Prioritize — Do the most important stories first

In 19. Release Planning, customer arranges stories into releases that coincide with business objectives. For example, it may make sense to focus on getting an MVP out so that we can at least see money trickling in, then schedule another release with more features, capabilities, and so on.

According to the XP script, **customer decide the most important stories to do first.**

## Coordinate — Keep everyone synced up and in the know

Most of the time, there's more than *just* a development team at a company. There's marketing, sales, and so on.

Let's say that we're working on building a feature that we think will attract a significant amount of users (and profit). From the perspective of the marketing team, they'll want to know when the new release drops so that they can come up with a marketing campaign around it.

Or perhaps from the perspective of the sales team, knowing what we've got planned down the pipe helps so that they can start looking for leads on customers that this new set of features is going to benefit the most.

### Example Release plan

Story	Release	Iteration
Login (Google)	MVP	1
Make offer	MVP	1
List vinyl	MVP	1
Accept trade	MVP	1
Decline trade	MVP	2
Get offers	MVP	2
Search vinyl	MVP	2
Get recommendations	Recommended	1
Update profile	Recommended	1
...	...	...

An example release plan. Other departments sometimes organize their activities around this as well.

The release plan isn't the only technique we use to do coordination. Nearly each of the twelve XP practices is focused on coordination the work we do in some way or another.

For example:

- The practice of *pair programming* focuses on coordinating code reviews so that we get instant feedback.
- The practice of using the *metaphor* (Domain-Driven Design — see 16. Learning the Domain) is coordinating the structure and language that we use throughout the project so that we're all on the same page, learn the domain faster, and so that we're more likely to build the right thing.
- The practice of *coding standards* is coordinating how we achieve consistency in our designs.

Essentially, coordination is necessary when we're working with other people or other teams. And we almost always are.

### Recalibrate — Get back on track when we get off track

It's incredibly likely that we'll get off track. Things will come up, we'll go a little slower one week, or we'll have to switch out stories, etc. Knowing what to do when we inevitably run into roadblocks or unintended stages is something only a plan can prepare us for.

## How planning works

There are two distinct stages to planning in an XP project. There is **the initial plan** and there's **continuous planning**, which is every plan subsequent to the initial one.

Why the distinction? Because the first plan is incredibly inaccurate and flaky due to us having no history to base our estimates upon. Developing the first plan starts with scoping a project.

### I — Scoping a project

One of the first things we'll have to do on any project is scope it. Perhaps you haven't even landed the project yet. Perhaps the client is shopping around and they want to know how long it'll take and how much it will cost if we take it on. Or maybe you work at a consultancy and you're ramping up to do a new project for an existing client.

In any case, this is the kind of activity that we shouldn't dedicate too much time to. The goal of scoping is understand what it is we have to do, see that it makes sense, and then kickstart it with an estimate so that we can at least get the ball rolling and create better estimates later during **continuous planning**.

Therefore, we need an **initial plan**.

In essence, here's what we use to scope out a project:

- **Epics** — Identify the big stories to build
- **Prices** — Provide very rough estimates of how long you think it'll take
- **Budgets** — Consider the number of people you have to work with you on the project

- **Constraints** — Do we have a domain expert that we can reach out to when we need to ask questions about the business? Will they be accessible throughout the entirety of the project?

To start, we'll ask the customer questions like "what is the goal of the app?" and "what would you be able to accomplish using it?". Here, some of the **functional requirements** emerge for the first time. These can sometimes be *big* stories like "hotel booking", "do user profile", or "create content". We don't *love* big stories because they're trickier to estimate. For the initial plan, we can deal with it, but later on, we'll work on breaking these down in 17. Stories.

In identifying the **non-functional requirements** — the technical, cross-cutting concerns that don't necessarily have anything to do with features, but influence the overall architectural styles and patterns we need to employ — we'd ask slightly more pragmatic questions like: "How many users do you expect to be using this at a time?", "What's your busiest time of year?", "Are there any other technical concerns that you think are important for us to know about? Audits? Compliance?", "Who are your users? What are their speed/reliability requirements?", "Does it really need to be real-time?"

The process of scoping a project and developing an initial plan is very much just applying the process we apply in **continuous planning**, but at a much broader scope. Think in terms of *aggregated* features — a collection (some call these *epics*), and don't go too into details. If you're thinking in terms of days or weeks, you've gone too far. This should usually be *months*.

For example, you may identify the login, signup, and email verification features. We can just say "auth" and assume it'll take a month. Where does "a month" come from? Well, either it comes from you having done that before, you knowing someone who has done that before and asking them, or it's a complete guess.

This entire process of scoping the project shouldn't take any longer than one to two hours.

At the end, you should have a table like this:

## Scoped stories

Broad-stroke stories (epics)

Story	Estimate
Submit & review a study spot	2 months
Profiles	2 months
Social sign in	1 month
Neighborhood study spots timeline	3 months
Friends timeline	1 month
Spot discussions and suggestions	3 months
Search spots	1 month
Recommended spots	4 months
Favourite spots	1 month
Sponsored spots	2 months

The Fowler, Beck, Martin trio recommend that when you scope this initial plan, you simplify your estimates by assuming that you're going to build the infrastructure as you go. We may very well need to spend a week or two building out 26. The Walking Skeleton , but we put that thought aside for now.

### 2 — Negotiating the first release plan

At this point, it's not uncommon for customers to *absolutely hate* what we've just done. Scoping the entire project including everything that the customer asked for, I counted 21 months (or approximately 2 years).

Customer: "How the hell will it take you 2 years? That's ridiculous. I'll find someone else that can built it faster".

When the customer gives you every possible conceivable thing that they'd like to see in the product, we scope out how long all of that will take. It's just the honest truth. Because I'm a developer too, I know that in my gut, I'd like to say "y'know, maybe if we try really hard, we can cut this down to 2 months". But that's not helping anyone. We're just lying to ourselves. We're not going to be able to work any faster. There's a limited amount of energy that we have every day to think critically and continuing to work past that threshold is a liability, so we shouldn't do that. Always assume we work a *Sustainable Pace* of at max 40 hours a week.

If we can't work faster, how do we remedy this frustration?

The good thing is that it's unlikely that the customer will need *all of the features* they've listed

\*\*right away. So we work together and negotiate.

In XP, **the customer makes business decisions and we make technical ones**. It's our responsibility to help the customer find that sweet spot — to squeeze the most amount of value realistically possible — for a first release.

Developer: "What's the most critical functionality that will bring the most value the earliest?"

Customer: "Hmm. Well, the main idea is that people should be able to find places to study in the city. The sooner we get that done, the better."

Developer: "Alright, we can certainly prioritize that for the first release. To submit and search for spots is about 3 months of work. How does that sound for an MVP?"

Customer: "Okay, not bad. So, for the MVP, we'll just focus on getting people in on the initial launch so that we can get feedback. I guess we could cut the timeline feature, followers, and friends for now. And we may not need realtime chat now that I think about it. I think the most important thing — the most valuable thing — is that people can submit spots so that others can start reviewing them and finding value. With that covered, we can focus on getting the traffic up. We'll coordinate with marketing. They'll start a blog and perhaps some other initiatives to start building the audience for the launch. After the MVP, in the next release, we'll focus on building out the sponsored spots feature — you know, for co-working spaces, incubators, innovation hubs — part of our business strategy is to get them to pay for advertising. But we'll need a critical mass of traffic first."

Very, very nice. If we can manage to identify a way to make the customer happy and mitigate the scenario where we grossly overwork ourselves, that's ideal. Just think of any of your favourite services — do you think they built them all in one go? After all, that's what Agile is all about, remember? **Providing value incrementally instead of all at once**.

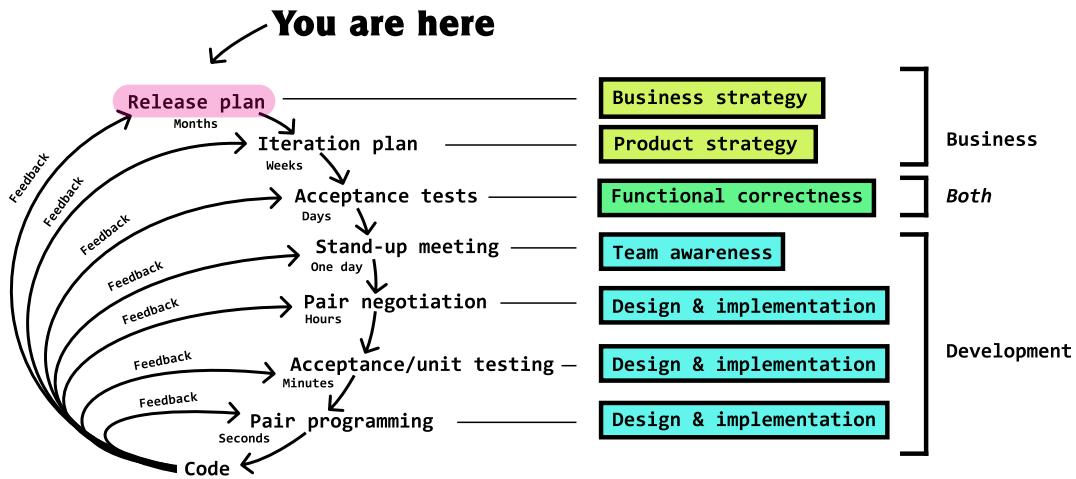
After some further discussion, we compose the very first *Release Plan* with dates on it.

## Initial **Release plan**

Release	Date
MVP	Feb 15th
Sponsored spots	Apr 30th
Friends and spot discussions	Aug 15th
Public timeline (explore spots)	Nov 30th
...	...

And there we have it: the first plan. This is all subject to change, of course — but we now

have something to work against.



### 3 — Executing the first plan

We discuss the release planning process in more detail in 19. Release Planning. However, based on our first broad-stroke release plan, we:

- Estimate the stories in more detail, giving each story a grade based on something called *story points*. Story points are a number measuring the ideal effort it will take to complete the story. You can use whatever range of numbers you like, but the Fibonacci numbers work well for reasons we'll discuss (see 18. Estimates & Story Points)
- Divide the stories into smaller tasks in the iteration (see 20. Iteration Planning )
- Ensure we *deeply understand* the features we're about to implement (see 21. Understanding a Story )
- Work synchronously or async with the customer to get acceptance tests (see 22. Acceptance Tests)
- And often, we take the first iteration to get our project set up and build out our initial infrastructure only so much so that we can write our first acceptance test (see 26. The Walking Skeleton )
- Finally, we'll start coding out the features using acceptance and unit tests (see 28. Test-Driven Development Workflow)

In executing this first plan, it is *highly likely* that you will feel like you're not able to get all of the stories in the sprint completed. You'll remember very early on that the estimates we made are, well — *estimates*.

“Agile doesn’t work! This is trash!”

No, it’s okay. Your first plan is going to be wildly inaccurate. This is especially amplified if you don’t have any past experience to back it up as well.

Luckily, this is expected. The first plan is like breaking in a new pair of shoes. In the future, we can’t be just making random estimates that we’re not sure if we’re going to be able to

meet. Is there a better way?

Absolutely, there is a better way. It's to track our *velocity*.

## 4 — Measuring velocity

The most effective way to create better estimates is to keep track of your **actuals**. If we assign stories estimated story points, actuals are the **actual values** for the story points that were completed.

For example, if you estimated that a story was a 3 (up to two days) but it actually took you 3 and a half days, you would record the actual as a 5.

In doing this, we start to get a better sense of how effective we actually are at completing our work.

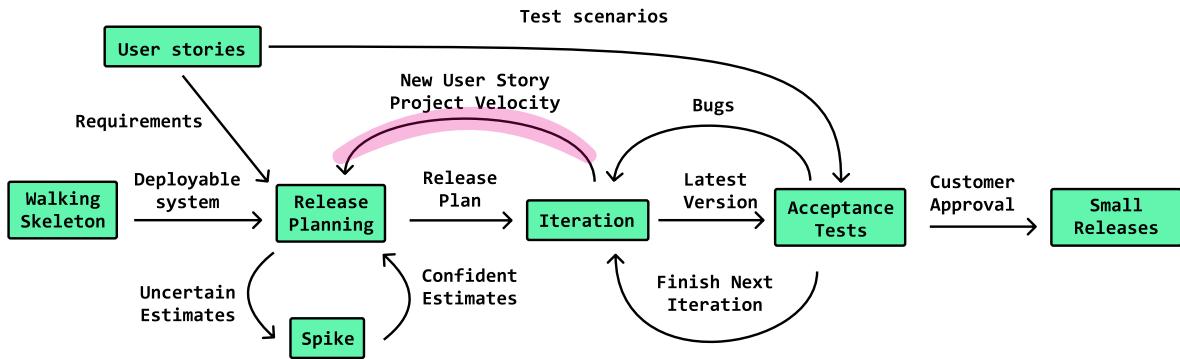
Velocity then, is the cumulative amount of story points (tasks) you were able to complete in a sprint. So, if we thought that our velocity was 20, meaning that we could get 20 story points done in a week, but by the end of the sprint, we only got 16 of them done, then we'd have to document that our velocity is *not* 20 — it's 16. And it's cumulative, so that means if you're working on a team of 4 developers and *together*, y'all can complete up to 16 story points as a team, that's the velocity going forward.

■ **Yesterday's Weather:** Fowler calls this technique *Yesterday's Weather* — using the previous iteration's velocity to determine the current one.

When we're planning out our sprints, customers can add stories that add up to but don't overflow our current velocity.

Yesterday's weather gives us a chance to adjust the plans before we realize we're in trouble. At the *mid-point* of every sprint, we check to see if velocity has gone up, down, or stayed the same.

- **When velocity goes up:** We can add more to our plate for that sprint — it looks like we're going to be able to get some more stuff done this sprint. Ask the customer what we should do.
- **When velocity goes down:** We remove something from our plate and say we'll get to it next sprint. Why? Because the goal is to — at the end of every sprint — have some acceptance tested, deployable code that provides value. If velocity declines dramatically two sprints in a row (which is very likely early on), we not only have to update the iteration plan, but the release plan as well since it's likely that we won't be able to meet the dates if we continue at the current pace.



We adjust the release plan when the velocity for the iteration declines dramatically two sprints in a row.

## Why this approach works

This is the very nature of the **continuous plan**.

In every adjustment to the plan, we remove the looming fear of failure to meet deadlines. It relies on us readily communicating ahead of time when we're going to need to adjust timelines based on velocity changes. Developers join projects, get moved to other projects, leave companies. Customers change their minds, come up with more features, pluck features from sprints, add them, and so on.

“The primary benefit of Agile is that you learn how screwed you are in time to do something about it.” — Robert Martin

Agile is about feedback loops and communication. If it sounds like I’m repeating myself, I am. I sincerely hope that knowing that there’s an entire community of software professionals out there planning projects, working with customers, and managing expectations this way reduces a lot of the anxiety common with developers unfamiliar with how good things can actually be.

## Summary

- Fear is the main reason why software projects fail — not skill, deadlines or external factors. When customers and developers are fearful, we put up walls — all to protect our egos and prevent hard, honest conversations. Admittedly, without a clear picture of what it means to act virtuously in these situations, the easiest thing to do is to attempt to protect ourselves by just avoiding them overall, never solving the underlying issues.
- To overcome fear, we need a code to work by. Customers and developers. The bill of rights are inalienable truths that help us understand what is expected from each other. They acknowledge both parties’ fears and give us the opportunity to focus on what we can do to mitigate bad situations.

- Planning lets the customer prioritize important work, coordinate activities, and helps us recalibrate when things inevitably get off track.
- We compose an initial plan to scope and kick off a project, and we keep track of our velocity using *yesterday's weather* because it lets us improve estimates and make adjustments to the continuous plan.

## Exercises

■ Coming soon!

## References

### Articles

- Yesterday's Weather — Agilenutshell.com
- Yesterday's Weather — Martin Fowler

### Books

- Extreme Programming Explained: Embrace Change by Kent Beck
- Planning Extreme Programming by Kent Beck, Mike Hendrickson, and Martin Fowler

### Papers

- The Agile Coordination Processes by Manuel Mazzara and Alberto Sillitti

## I5. Customers

■ The role of the customer is far too important to leave to ambiguity. We need to a concrete customer to become a part of the team to help us learn the domain, prioritize work, and build the right thing.

Do you remember playing “telephone” as a kid? It’s a little game where you tell a friend a sentence, then they’re supposed to tell another person who then goes on to tell *another* person, and so on. At the end, when the message finally makes its way back to the original sender, you can almost always count on pure hilarity to ensue next.

How the heck did “I love my mother” turn into “flush the puppy water”?

This game exposes a very critical flaw in human communication. It’s lossy. The more people a message has to pass through, the less likely it is to be accurate.

In software projects, building the right thing is critical and we can’t afford lossy-ness.

You can reasonably expect that if you’re getting your requirements for what to build through anyone other than the customer, we’ve created a telephone chain. If this is the case, and if we don’t have access to the right people to ask questions, well — then who knows if we’ll ever build the right thing.

“It’s not stakeholder knowledge but developers’ ignorance that gets deployed into production.” — Alberto Brandolini

In XP, we talk about the customer a *lot*.

Some say that you shouldn't even bother if you can't find one. Well, who is the customer, really? Could it be the designers? Sometimes we get the screens we need to build from them, right? Is it the engineering manager? The project manager?

Spending time around developers, I tend to hear complaints about customers and hear stories about how un-involved the customer can be in projects. Many developers:

- Believe that don't need to work with the customer beyond getting the initial up-front requirements; that they can merely hack something together
- Get frustrated when the requirements change, pointing the blame of drastic slowdowns on the customer
- Build ultimately unclean, unscalable, anemic software (meaning — there is very little that one could learn about the domain by reading the codebase) because they don't fully understand the domain

If you're starting to understand how XP works, you'll agree that there are some major mindsets to correct here if we're going to be victorious in our efforts to develop something meaningful, and to do that consistently and reliably.

Let's discuss.

## Chapter goals

In this chapter, we learn about:

- The ideal relationship between customers and developers
- Who the customer really is and their role in a software project
- How to locate a customer

## Driving vs. steering

In both the original *Extreme Programming Explained* and *Planning Extreme Programming* books, it was written that software development is analogous to driving a car.

How so? Well, when you're driving, you don't just set your GPS and hit the gas. When you're driving, you're constantly aware of your surroundings, you apply the discipline of stopping at lights and looking both ways, signalling, occasionally taking detours when there's construction or potential slowdowns ahead, and you sometimes stop and pick things up along the way before resuming the path towards your original destination. Ultimately, the execution aspect of an Agile software development project is similar. It's a process that takes discipline and attention because **we're going to need to make frequent small little adjustments**.

They say that we're the driver and the customer is the one navigating — giving us directions.

One way we drive the customer is by knowing what techniques we're going to use to get us where we're going (like TDD or acceptance testing), and it's completely irrelevant for the customer to know about it. Passengers don't need to know which gear you're in.

One way the customer navigates is by deciding to completely re-work the features in the next few releases or add a new requirement, like "*after you complete your profile for the first time*,

*you should be redirected to the dashboard".* Completely fair. Don't get mad that this new requirement is coming up *now*, just after you finished building the profile functionality. That's to be expected.

We succeed by taking them on their journey, continuously adjusting the plan and direction, and by sticking to the principles and practices we'll be studying in more detail.

## Who is the customer?

Let's make this clear.

### The person who makes business decisions

We say that **customers make business decisions, developers make technical ones**. This means that customers are responsible for:

- Scoping
- Prioritizing
- Dates
- Functional and non-functional requirements (both of which we can acceptance test)

And developers are responsible for:

- Estimates
- Design & development

### A domain expert

In the coming chapters where we delve into the main ideas of domain-driven design, you'll hear about the idea of a domain expert more frequently.

A domain expert is someone who ideally **works in the domain** and **has a good understanding of how things work**. Sometimes you'll encounter someone who has one or the other but not both.

### Must be available for questions

The customer (or domain expert) is a person that we can ask for clarification on the real-world workflows we're digitizing, the problems we're solving, and can help us learn the domain.

We expect this person to be readily available to help pick stories, set priorities, write acceptance tests (or at least sign off on them), give immediate feedback, and let us know if we're moving in the correct direction.

### Responsible for the success or the failure of the project

The customer is someone who accepts full responsibility for if the project thrives or dies.

Perhaps you're seeing how essential the role of the customer is. As we mentioned in the previous chapter, potential customers may attempt to distance themselves from us with

requirements docs, broken communication, and processes — all to protect themselves from having to be the person whose door gets knocked on if the project flops.

Nah. We need someone to step up and be the leader.

If you believe in garbage in-garbage out, then we need a customer to iron out the **business strategy**, **product strategy**, and get the **requirements** in order. They need to know that what we spend our time building *actually matters*.

### **Often an entire team**

So you're saying the customer needs to make all the business decisions, take responsibility for the outcome of the entire project, and they have to answer questions too? Holy smokes. That's a lot, no?

That is a lot for a single person, yes. But there's a reason why the XP practice is called *Whole Team* and not *On-Site Customer* anymore.

The customer isn't necessarily one single person. Instead, the idea is that the entire development team (be it engineering managers, designers, testers, architects, developers, technical writers and such) be as close as possible (physically) to the business roles that all ultimately represent the customer.

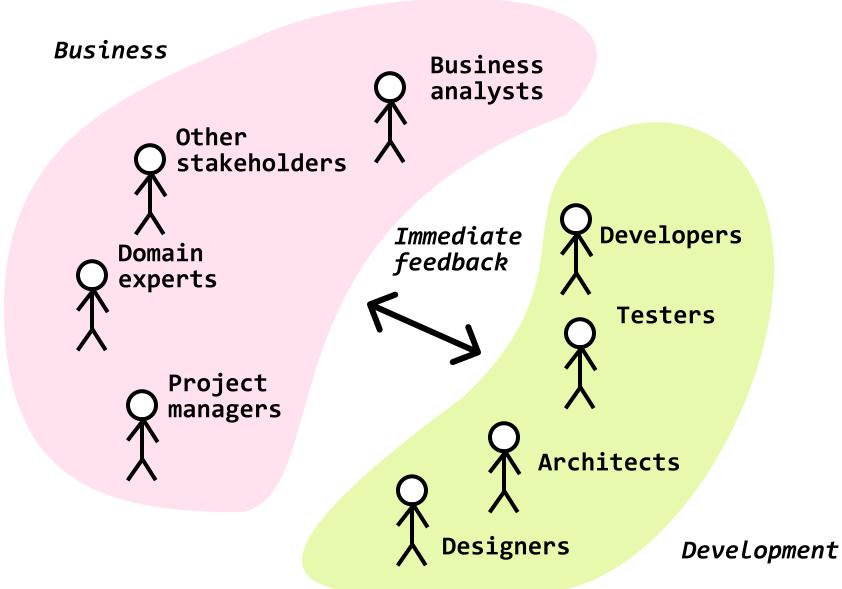
The customer can be a combination of any number of:

- domain experts
- business analysts
- product managers
- a user representative
- perhaps the person paying for the project
- one or more of your customer's enterprise customers
- employees from adjacent departments that would benefit from the product
- and other stakeholders

This team is the select roster of people who will inherit the business responsibilities attached to and responsible for the success or failure of the project.

## Whole team

Immediate feedback between customer and development



Ideally, the phronimos developers seem to agree that *Whole Team* works best when everyone is co-located in the same physical vicinity. That isn't to say that this can't be done remotely, it's just to say — well, think about those times when you got to stand in front of a whiteboard and spitball some solutions with your peers. That kind of synergy is a lot harder to achieve over a screen. Although, I still do believe it can be done.

### Locating a customer

In summary, you need someone (or a team of these people) who tick off all of these boxes: understands the domain, is on board with small frequent releases, can prioritize the stories, and accepts responsibility for the project.

For a lot of us, this is the project manager or a *requirements jockey*. In that case, they're the customer. Isn't that a telephone chain? Yup. Is that good? Eh, it's alright. It would be better if we had a real customer or domain expert to work with as well — to add to the team. But it's 100x better than nothing.

The *designer* is not the customer. Designers design screens for us, and that's great, but they'll often miss details because they don't hold the entirety of the essential complexity, and they most certainly cannot prioritize stories. We can ask them questions about the screens, but don't count on them being able to answer what we should build first or what the functional requirements are. You need a legitimate customer for that.

### What if you can't find a customer?

You need one. As Kent Beck says,

"No customer at all, or a 'proxy' for a real customer, leads to waste as you develop features that aren't used, specify tests that don't reflect the real acceptance criteria, and [you] lose the chance to build real relationships between the people with the most diverse perspectives of the project."

We also know this to be true.

As we discussed in 13. Features (use-cases) are the key, we can easily avoid the scenarios where **the requirements are correct and the code doesn't work** by simply ensuring we write tests. But without a customer, we can't avoid the situations like when **the requirements are correct and the code does something else** or when **the requirements are incorrect (and in that case, it doesn't matter what the code does because the product is useless)**.

We talked a lot about the customer, but in closing, remember this:

"If you get lost driving, it isn't the car's fault, it's the driver's." — Fowler, Beck  
(Planning Extreme Programming)

## Summary

- In XP, we drive and the customer steers/gives us directions. The continuous planning techniques from the previous chapter help to continually make small adjustments — safely changing directions without running off the road. Similarly to how a passenger doesn't need to know which gear we're in or when we're going to signal, we don't need to debate which technical practices we're going to use to do our best work (TDD, refactoring, acceptance tests, and so on). The customer's sole responsibility is to focus on making sure that we're going in the correct direction.
- The customer is someone (or a group of people) that understands the domain, is on board with small frequent releases, can prioritize the stories, and accepts responsibility for the outcome of the project.

## Exercises

■ Coming soon!

## References

### Articles

- <http://www.extremeprogramming.org/rules/customer.html>
- <https://www.boost.co.nz/blog/2020/03/extreme-programming-customer>

## Papers

- One XP Experience: Introducing Agile (XP) Software Development into a Culture that is Willing but not Ready — *great case study on teams from IBM using XP and their learnings*

## Books

- Clean Agile by Robert C. Martin

- Planning Extreme Programming by Kent Beck and Martin Fowler
- Extreme Programming Explained: Embrace Change by Kent Beck

## 16. Learning the Domain

To write high-quality code, we need to understand the domain at a high-level and use the domain language within our code. Domain-Driven Design presents a number of patterns, principles, and practices to learn the domain, build a shared mental model, and tackle complex problems by structuring code so that it utilizes the ubiquitous language (or “the common language”) of the domain.

I believe everybody has some sort of specific knowledge. Whether it’s your dad who has been running his own construction business for fifteen years, your neighbour who restores old photographs, or your best friend who just graduated with a major in biochemistry. Everybody is a master of *some* domain, regardless of it’s related to work or just something like they like to do for fun like collecting model trains.

I see software developers as a form of modern day magicians. If we can learn the domain, we strategically locate software within it that brings people together, solves problems, and automates tasks.

You’ll be happy to know that there are a set of principles, patterns, and practices that can help us fast-track our understanding of any domain, regardless of how complex or boring it may initially seem. With it, we can use our new-found knowledge to not only build useful software, but to do it in sustainable way.

The approach we’re about to discuss can help you learn how to turn domain experts’ specialized knowledge into highly practical, valuable applications capable of doing complex things like trading stocks, automating workflows, and honestly, pretty much anything else you can think of that people regularly have to do by hand.

The set of techniques we’ll cover at a high-level in this chapter all fall under my personal favourite topic in software design: Domain-Driven Design.

### Chapter goals

In this chapter, we’re going to:

- Learn how domain-driven design helps us tame complex domains and build rich, maintainable software that embodies the very heart of the domain within the code itself.
- Discover how to build a shared mental model of the domain and discover the features to use as stories in the next stage by conducting an Event Storming session with domain experts and developers.

### Where are we so far?

First, let’s take a moment to ground — to remember where we are.

**Features are the key.** They are the atomic elements of value. They take on different shapes throughout the duration of a project (from being undiscovered, to being stories, estimated

stories, planned stories, and then working code covered by acceptance tests). We've also said that features are comprised of data, behaviour, and namespace that they belong to.

**Customers (domain experts) inherently understand the essential complexity. Because of this, it is through them that we should be discovering the features in an Agile project.** Ideally, we'd like to have people on our team that work in the domain and are going to be using the software that we create. That's not always the case — there are many different contexts and scenarios in which software gets created, but fundamentally, we need someone that understands what we're doing, that is taking responsibility for the success or failure of the project, and can point us down the correct path. With that, we're in good shape to move forward. If not, you may be opening yourself up to world of potential headaches.

## Domain-Driven Design

Domain-Driven Design is a set of principles, patterns, and practices that help us **learn the domain, build a shared mental model, and tackle complex problems** by structuring code so that it utilizes the **ubiquitous language (or “the common language”)** of the domain.

### A shared model

In this chapter, we're going to specifically focus on the task of building a shared model. What do I mean by that — a shared model?

A *model* is an imprecise, good enough, simplified, conceptually understandable idea of something else more complex. The utility of a model comes from the fact that it is grossly simplified. Models include what is absolutely necessary and sufficient for us to understand, generally speaking, how they work. In 5. Human-Centered Design For Developers, we talked about conceptual models and used the example of “the cloud” — as in where user data ends up. For most non-technical people, it's a lot easier to understand that your data is saved in a “cloud” — meaning, some remote location that they can't actually see — than it is to understand that it's saved in a database installed in a server rack in a warehouse somewhere. Models *condense complexity* into something easier to understand and discuss. Look how easy it is for non-technical people to talk about new requirements:

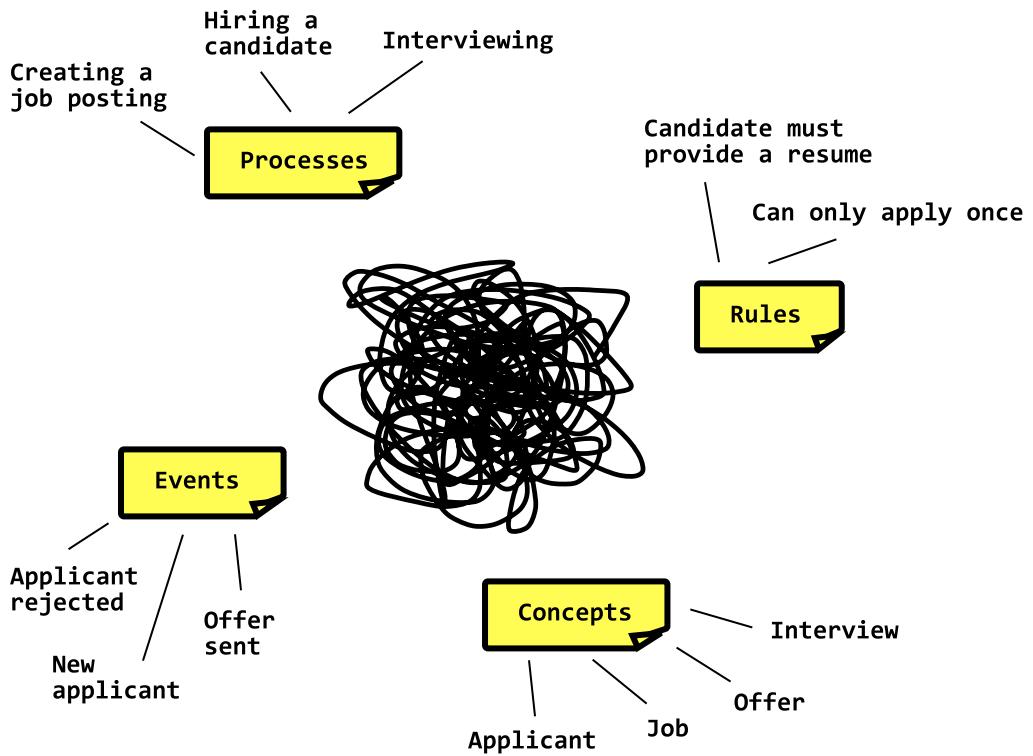
“When a new photo **is uploaded to the cloud**, the app should notify everyone who has access to the folder”.

In DDD, we focus on something called the *domain model*, which is an imprecise, good enough conceptual model of the entire domain.

### The domain

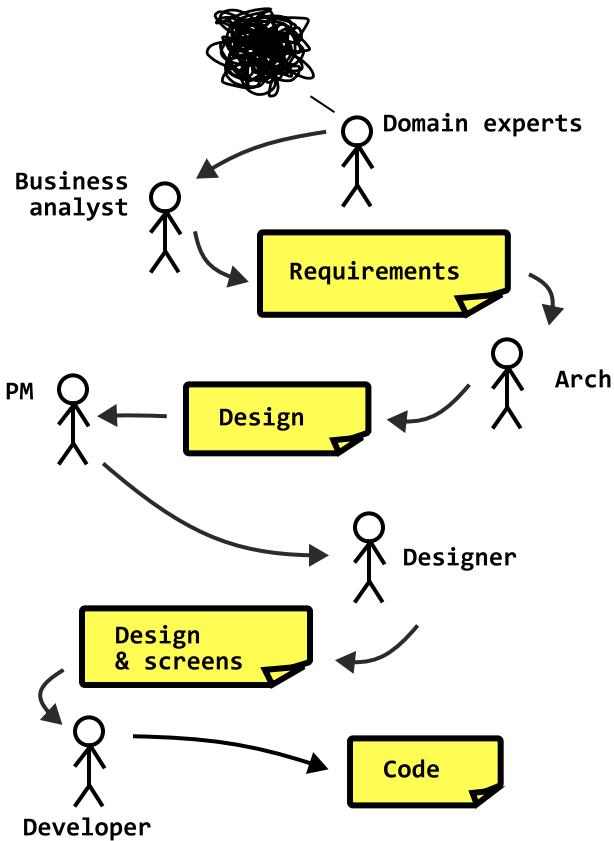
What's a *domain*, I can hear you ask?

While we can generically say that a *domain* is an entire subject area of specific knowledge (like how the domain of movies includes the concepts of cameras, genres, directors, scripts, actresses, etc), as problem solvers, we are most interested in the *problem space*. In DDD, the *problem space* is the entire set of relevant concepts, processes, rules, and events that occur throughout the organization that we've been asked to build software for.



### Learning the domain without a shared model

Let's again consider the unfortunate telephone requirements chain scenario discussed in 15. Customers. It starts with the domain experts. They know the *problem space*. From there, perhaps a business analyst or some other customer-like person gets the requirements from them through conversation, and then starts a chain of documents — maneuvering through teams and roles until it eventually gets to its terminal point — you: the developer.



Sometimes the requirements documents can be really good and detailed. I mean, if you think about it, in a way, everyone gets to touch a part of the requirements before it gets written and deployed to production, so there's that. But because we're not working together on creating a shared model, your interpretation of a job and or a customer could be completely different than that of the customer. The lossy-ness of a broken feedback loop rears its ugly head again.

Sometimes, developers invent terminology that no one really understands. The lack of a shared model only turns up the dial on the pain felt by the remaining developers when the original one leaves the company.

Some developer later down the road: “What the hell is a DE002Factory? What is that even supposed to mean?”

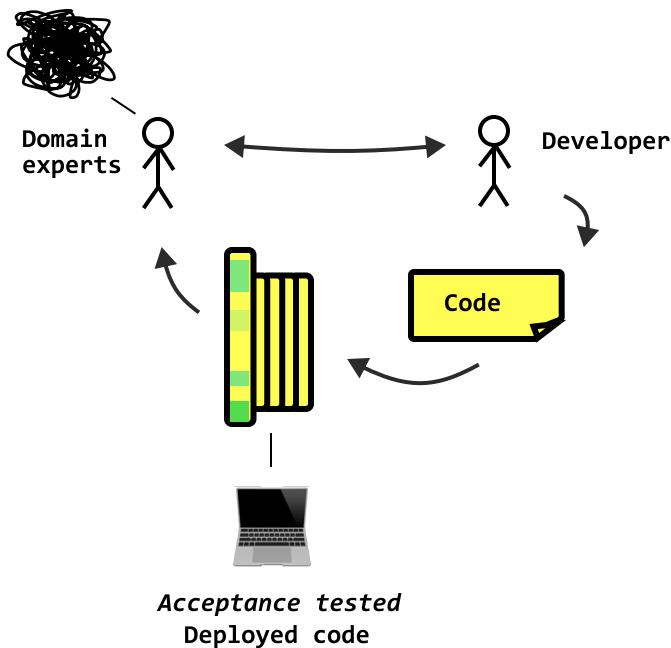
A lack of a shared model and vocabulary can even get to a point where it's downright offensive to the customer. In Robert Martin's *Clean Agile*, he shares an anecdote:

“In the late '80s, I worked on a project that measured the quality of T1 communication networks. We downloaded error counts from the endpoints of each T1 line. Those error counts were collected into 30-minute time slices. We considered those slices to be raw data that needed to be cooked. What cooks slices? A toaster. And thus began the bread metaphor. We had slices, loaves, crumbs, etc.”

He goes onto describe how this silly conceptual model which had nothing to do with *slices of*

*data* actually happened to work pretty well for the programmers, but whenever they were discussing with the customers, it seemed to upset them. To the customer, it appeared as if they were trivializing the work — making a joke out of it. As discussed in 8. Naming things, some austerity is necessary here.

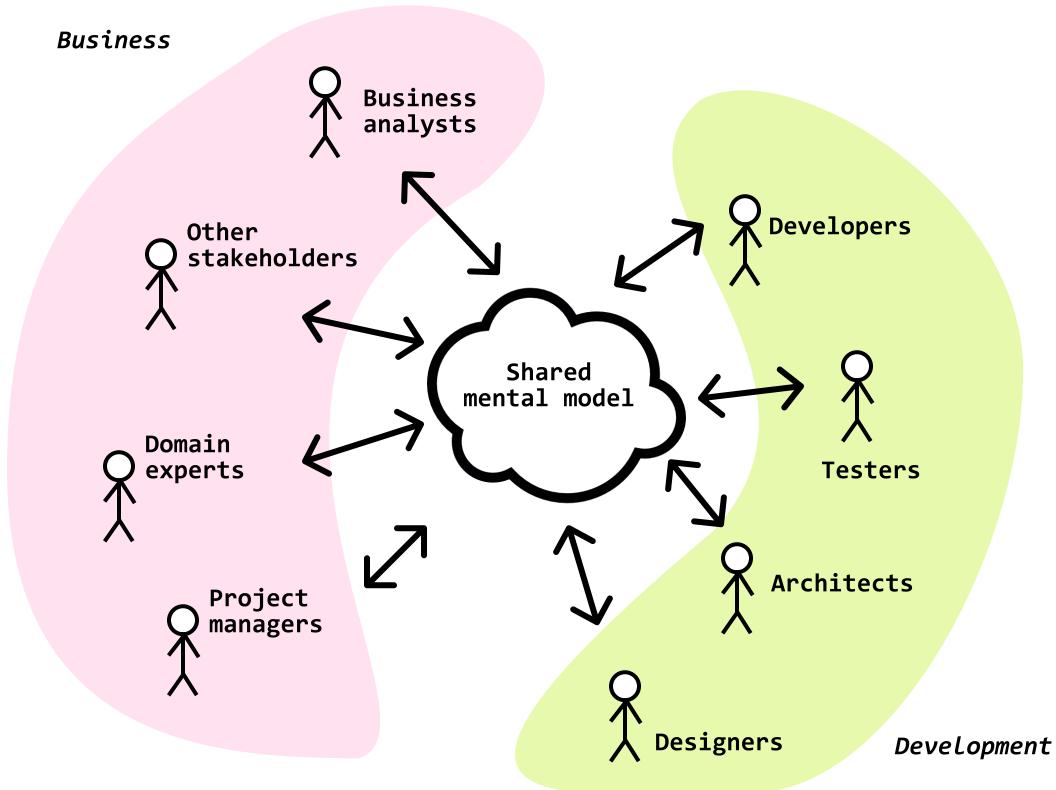
A better approach altogether would be to cut out all the middle-men to work with the domain experts directly.



This is better. We get to actually have conversations with the people that understand the essential complexity much better than we do.

There is, however, one key downside to this approach. There's only *one translator*: the development team. With only a single translator, we'll find that there's a lot of pressure on developers in interpreting the domain expert's ideas *accurately* so that future developers can make sense of the domain as well. While I know we'll try our best, let's be real. Us developers have a pretty *technical* way of looking at things.

The best technique is the one where the entire team is involved. Again I'm referring to the practice of *Whole Team*. Instead of it being just the developers and domain experts translating and documenting ideas, let it be the entire team — that may also involve members from other departments too.



As you'll learn, sometimes domain concepts mean different things to different teams. For example, the idea of an *Order* or a *Product* might mean something completely different to the team that handles shipping vs. the team responsible for the ecommerce website.

### Benefits of a shared model

There are a few reasons why this approach is highly recommended.

- *More clarity and less time spent on low-value work* — If you want to spend less time going back and forth trying to understand the requirements, this is key. A shared model will give us a broad-stroke view of the entire business, the processes that flow between teams, and what we should really be spending time on.
- *Solve problems faster* — Think of yourself as the vessel between the domain experts and the code that solves those problems. Domain experts know what their problems are, we know how to solve 'em. The closer the code and the person having the problem are, the more likely we'll be able to sync up and build it faster (and better).
- *Perennial and maintainable* — It's unlikely for an entire business model to change overnight. If we're able to turn the code into a model of the practical parts of the real-world, we'll find ourselves in a position to implement new functionality a lot faster than ever. We'll write less bugs, make it easier for new developers to ramp up, and we'll have built something timeless and valuable.

### Patterns, principles, practices

I mentioned before that DDD is a set of patterns, principles, and practices. Sometimes you'll hear people referring to DDD as something you *do* in your codebase. Maybe you've heard someone say "I built this using DDD". That's a little awkward. DDD isn't necessarily something you've *done*. When I hear this, I think what developers are referring to is the fact that they've implemented some of the important technical patterns DDD recommends.

The **patterns** of importance are:

- Aggregates, commands, queries, events, repositories, a layered architecture (also known as hex/ports/clean/onion), value objects, entities, event handlers, application services, and domain services.

If that means nothing to you right now, that's completely OK. We'll get to all of these patterns in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS.

Next we have the **principles**. These are higher-level. They show us how to mitigate common pitfalls what the best practices approaches look like. For example, with respect to the task of building out a shared model, the community outlines four principles:

1. Focus on events and processes rather than data structures.
2. Divide the problem domain into smaller subdomains.
3. Create a model of each subdomain, but focus on the core one first.
4. Develop a common (ubiquitous) language that is shared and used within the code and everyone involved in the project

Now, principles are great, but we need something concrete to show us how to implement principles. This is where we get the **practices**. To build a shared model using those four principles, we have:

- Event Storming / Event Modelling and
- Context Mapping

Let's start about *Event Storming* and see how it can be used to build a shared model.

## **Building a shared model with Event storming**

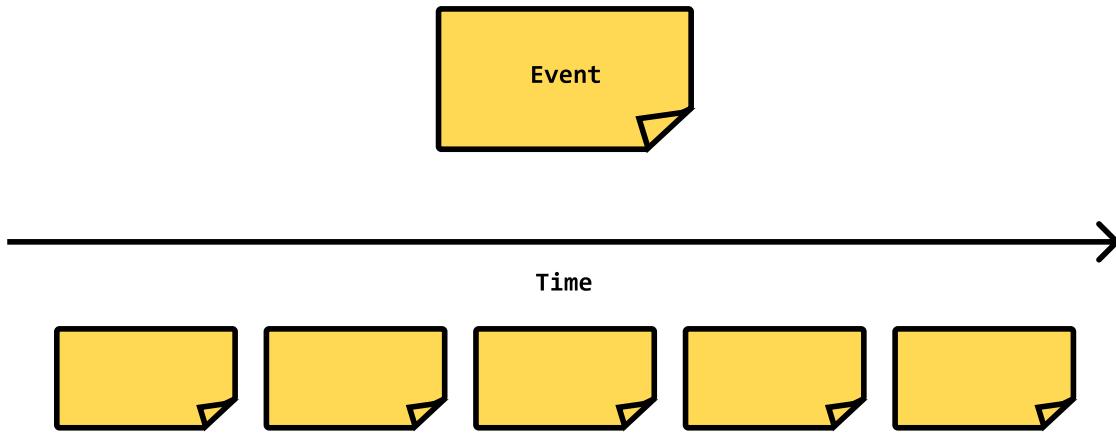
Once upon a time, a developer named Alberto Brandolini found himself short on time for a traditional UML use case design session. You know, those stickman diagrams? Well, instead of showing up empty-handed, he improvised a technique we now call *Event Storming*.



We use lots of sticky notes when we do Event Storming sessions.

Event Storming is an interactive design process or workshop which makes use of sticky notes, markers, and a whiteboard to engage both developers and business-folk into quickly and cheaply learning the business and creating a shared understanding of the entire *problem domain*.

Here's how it works. You gather up a number of people who are particularly knowledgeable about the domain, people who have questions, stakeholders, developers, etc. We find a room that has ample wall or whiteboard space. If we're using walls, we cover it with paper so that people can put down their sticky notes. Then chronologically, from left to right, we map out the *story* of the business using sticky notes and sharpies.



This, in turn, should inspire other participants to jot down their events and processes that before or after other events as well. Towards the end of the session, we should have a clear picture — a timeline — of how data flows from one end of the enterprise to the other side, and the teams it passes through. Because that's all that really happens, data flows.

We'll jump through how this works with an example in this chapter, but you can also read [Event Storming](#) for a complete walkthrough.

### **White Label: An example domain**

To learn this process, we'll imagine a fictional (yet realistic) business to demonstrate learning the domain, identifying the stories, estimating them, planning them, and implementing them with acceptance tests.

Assume that we land a gig to build an application for a local record store. The store owner, Nick, tells us:

"We're a record store specializing almost exclusively in rare vinyl. We've been doing pretty well these days. Between our online and the brick-and-mortar store, business has been steady. Recently, one of our employees at the store has been running a little trading initiative. The way it works is this: customers come into the shop, take a look at our rare requests board then write down their phone number and vinyl they're looking for. We've noticed that it's one of the first things that customers regularly check when they come back. Our rare requests board has been something of a neighbourhood phenomenon recently. Customers have been telling us how much they love the community feeling of trading vinyl. So we came up with the idea to replace the rare request board with an online vinyl trading platform where people can trade records and connect with each other. Think [paperbackswap.com](#) but with Vinyl instead of books. Traders should be able to list the vinyl they want to trade, request vinyl they're looking for, and trade with other members!"

Sounds pretty nifty, right?

## I — Plot the domain events

Assume we have everyone in a room. We kick things off by asking the *Event Storming* participants to start by plotting some domain events.

You: "Who wants to start? Somebody put down a business event!"

Veronica: "I'll start. I'm Veronica, I started the vinyl trading community here at the store."

You: "Nice to meet you, Veronica. Can you tell me what one of the first things that happens is?"

Veronica: "Well, vinyl traders go to the rare requests board, write down the records they're looking for, and leave their number."

Nick: "We're supposed to write these in past tense, right? What would we call that?"

Veronica: "How about *Vinyl Requested*? That's the idea, right?"

You: "Perfect, yes. Let's write that one down."



Great, that's the first sticky on the board. Now would be a good time to ask what happens after that.

You: "What's next?"

James: "Hey, I work as a clerk here and I've actually used the board and done a few trades myself. If I see a record or even just a general category of records — I mean, I've seen someone say they're were looking for like, *San Francisco grindcore* — you just send a text to the person letting them know you'd be down to trade."

Veronica: "Do you normally trade records for records?"

James: "Yeah, sometimes — I'll ask them what kind of stuff they have and see if they have something they'd be interested in parting with. But often, I'll just make them a monetary offer. Like, cash or something."

You: "That's interesting. Do we want people to be meeting up in person and exchanging money in person or do we want this to be a solely online kind of thing?"

Nick: "Online. Ideally, we'd like folks to be able to ship items to each other. Kate might be able to share some insights as to how that would work."

Kate: "Hey, yeah. So, I work at the shipping department. We mostly focus on shipping orders to customers who buy things from the ecommerce store, but if people are going to be

shipping things to each other, maybe there's a way we could integrate the new app with our system to let traders print shipping and tracking labels?"

■ **Building new and novel systems vs. automating existing processes:** You'll find that you're either going to be building a completely novel system where there previously was nothing of the like (think applications like Instagram, Reddit, Facebook) or you'll find yourself to be automating existing processes (like food delivery, document-signing, or booking a dentist appointment). When you're building new systems, the event storming process is a lot more about *what could happen*. It's a little more imaginative. This is where UX researchers come in handy. When you're automating existing processes, the design is more focused on *what does happen*.

With new systems design, domain experts, designers, and other stakeholders may have an idea in their heads or on wireframes about what they think the application should do. **Because there's nothing in the real-world to really base these new systems off of, the essential complexity is something the domain experts invent.** In this case, event storming sessions are extremely helpful to identify missing requirements and spot trouble areas in their invention. Quite often, these trouble areas go unfound until much later into the design process. For example, "How do we punish buyers that send incorrect or damaged vinyl? Can we ultimately prevent that from happening overall?" Event storming helps figure out where you'll need to spitball ideas, get creative, and see how things would work within the timeline. For a large number of developers that build side-projects or work at startups, this is the box we're in.

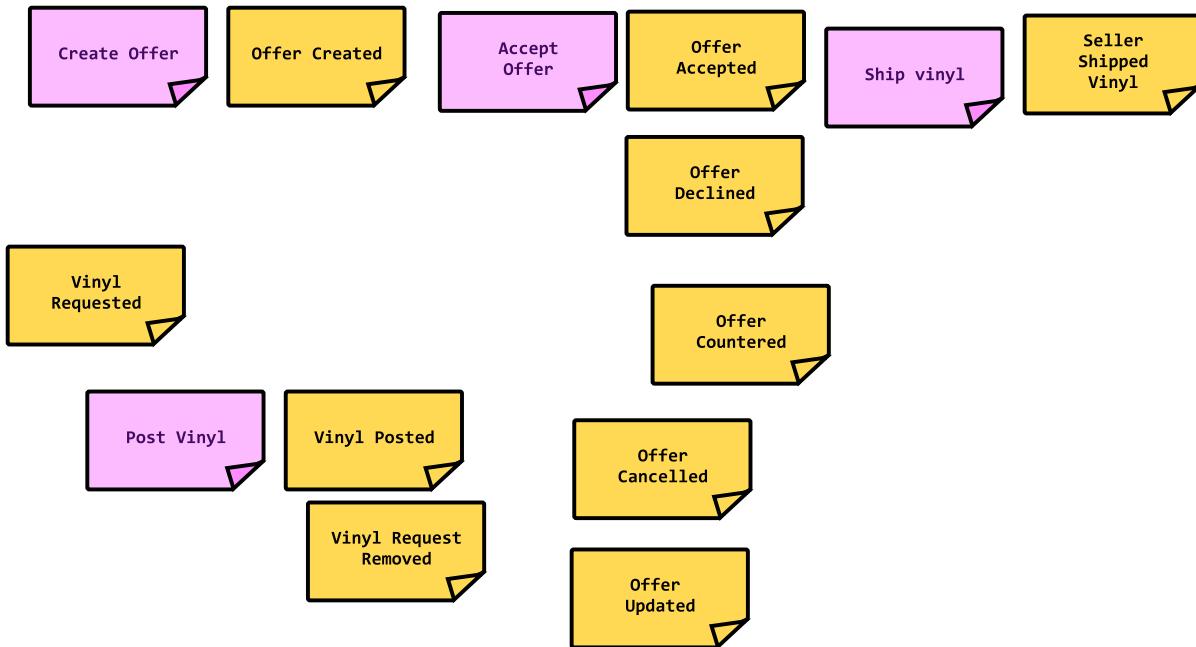
When we computerize existing laborious, repetitive, or inefficient real-world processes done primarily on paper or by hand, we lean a lot more on the experiences of people who are already doing the work and focus on merely learning what they do. We don't really need to pull a system out of thin air and invent some new essential complexity. It's already there. Our focus is making sure we're just uncovering the essential complexity as it is. The value we bring always varies, but common across most is the ability to help people work more efficiently, save time, and do their work better. This is the kind of work that a lot of folks who work on computer systems in government, law, manufacturing, and trades tend to do.

White Label is, in a way, a mixture of the two. We *could* merely mimic the way that the trading community works in the real-world by facilitating real-world meet-ups, but Nick (the project owner) has a different idea. He'd like it to be a completely online experience. That means that we're going to be inventing a fair amount of new processes.

We continue putting events down on the board until we feel like we can't think of anything useful to write down anymore. At this point, the board may look something like the following:

- Trader Registered
- Vinyl Requested
- Vinyl Posted
- Vinyl Request Removed
- Offer Created
- Offer Accepted
- Offer Declined

- Offer Countered
  - Offer Updated
  - Offer Cancelled
  - Shipping Label Created
  - Customer Charged
  - Seller Shipped Vinyl



You'll notice that some of the events don't actually sound like events at all "Create Offer" and "Ship Vinyl". These are *commands* (and the focus of the next step). If these come out while you're coming up with the events, go ahead and write them down next to the appropriate originating event with purple stickies.

**Sort the events chronologically**

To see the full picture of how this system works (or could work), organize the events chronologically from left to right. This is why we need quite a bit of wall or whiteboard space. If there are a number of events that could alternatively occur at the same time (*Offer Accepted*, *Offer Declined*, *Offer Countered*, etc), place those vertically to denote that they are all possible next step events.

## Push events to the edges

Once it seems like everyone has run out of steam on coming up with events, query into if there are any more events that happen before the first event or after the last one. You may find that this could trickles into other teams' workflows.

For example, you might say “what happens after *Seller Shipped Vinyl*”?

Nick: "You know how Etsy lets you track where your orders are? How do we do something like that?"

Kate: "It's not too hard. If we give sellers the ability to print out shipping labels, we could certainly integrate with the app so that traders could see how far along their orders are. So then, in that case, let me place a few more relevant events."

Kate then creates *Transit Status Updated* and *Vinyl Delivered* events.



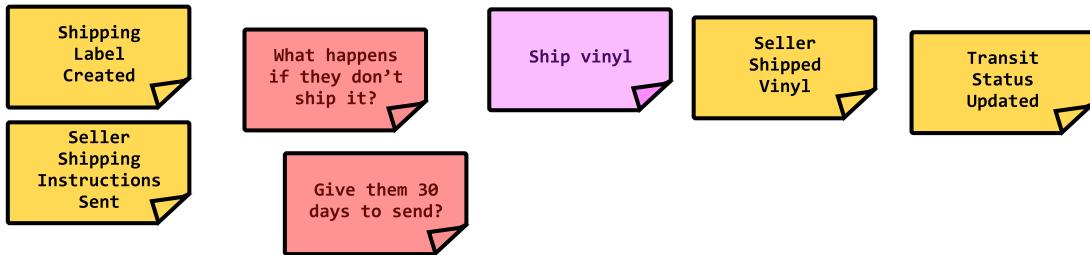
And thus triggers more great conversation:

Nick: "That's great. And you know how we were talking about how we're going to make sure that people send out the correct items? What if, after traders receive their records, we asked them to rate the quality of the items they received?"

Veronica: "Fantastic idea. Those ratings can be used to compute the reputation for the trader so that if someone has a bad experience. Then other traders know not to deal with them anymore. This wasn't really too much of an issue when people were doing it in person, because you could always just test out the records in person by putting them on a player and looking for scratches, so this makes sense to me."



Thankfully, our team is pretty clever at coming up with ideas on the spot, but if you find yourself stuck, it's best to put down a red sticky to note that as a problem area to come back to later on.

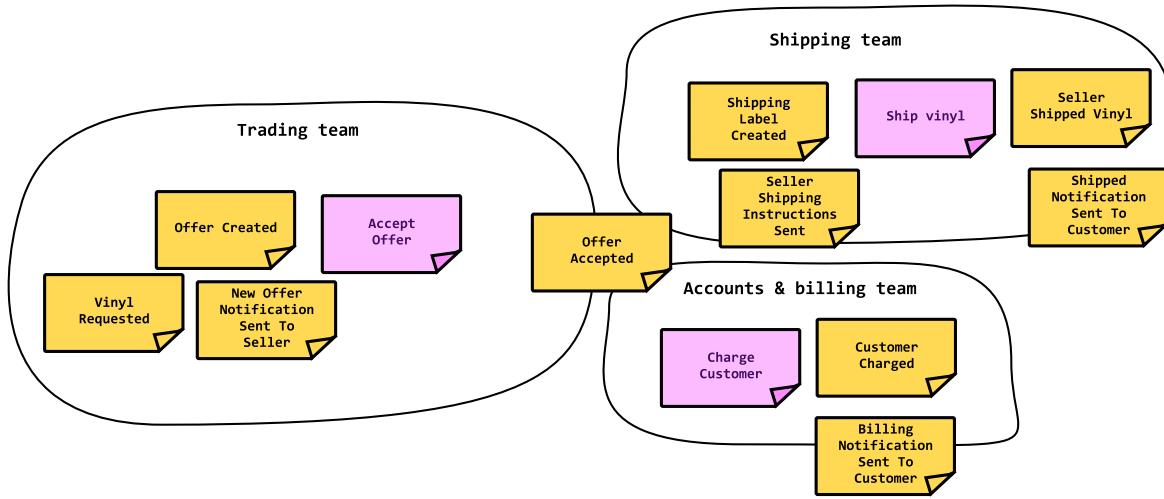


## Benefits to modelling with events

Hopefully you're starting to see how effective this strategy is.

- **Feasibility** — We get an honest look at what the customer would like to build and locate technical and semantic problems. For example, it's hard to expect users to do things that they have no reason to do (like log in and say that they've received an item after they've received it). People can challenge ideas, propose different strategies, and ultimately, improve the strength of our product designs. On new and novel software, we come up with something that works. On automation projects, we make sure we have a clear picture of the essential complexity.
- **Capture missing requirements** — Towards the start of a project, it's pretty common for requirements to be fuzzy. That's why we're doing this! When we lay everything out on the wall or whiteboard, it's a lot easier to see when requirements are missing. Before I discovered this approach to learning the domain, I used to always forget to account for all the "notifications" and "emails" I needed to ensure got sent out in response to events. With everything on the board, it's very hard to miss these things now. Traditional approaches to requirements gathering only give you pieces of the story (use cases, class design, API calls, GraphQL schemas, what have you). It's those *after this-do that* requirements that always seem to crop up at the last minute and wreak the most havoc. They tend to be clumsily worked into a codebase after they're found and damage the coupling and cohesion of the code (especially in non-event driven architectures).
- **Auditing & reporting** — Events act as the history of everything that happened in the system. A common non-functional requirement for enterprise software is to persist this history of events for auditing, reporting, or analysis reasons. Running this exercise with a good mix of experts and team members throughout the company should give us a good picture of the events so that we don't miss any. For reporting, we'll also have to think about the UIs that we need to create (we'll discuss this further on in this chapter).
- **Collaboratively understand how teams connect** — My favourite aspect about this entire process of using events is that everybody gets to see how their work is related. For example, once an *Offer* is accepted, we create work for the *Shipping team*. At that point, *Shipping* should print a shipping label and send instructions to the seller. That same domain event, *Offer Accepted*, also has the potential to create work for the *Accounts & Billing Team*, who may need to charge the traders if there was a monetary aspect in-

volved in the trade.



"I notice you're talking a lot about teams. What if those teams don't exist yet? What if I'm the first developer on the project just building out a monolith? Does all this stuff still apply?"

It's a good idea to start building out projects as *modular monoliths*: that is, really well structured monolithic codebases. If we can identify all of domain events and the *namespaces* they belong to (we call them *subdomains* in DDD terminology), we can — from very early on — structure out the moats of functionality in our codebase. That way, when we finally *do* have to scale out to teams, our monolithic codebase will be designed to be split up.

## 2 — Identify commands

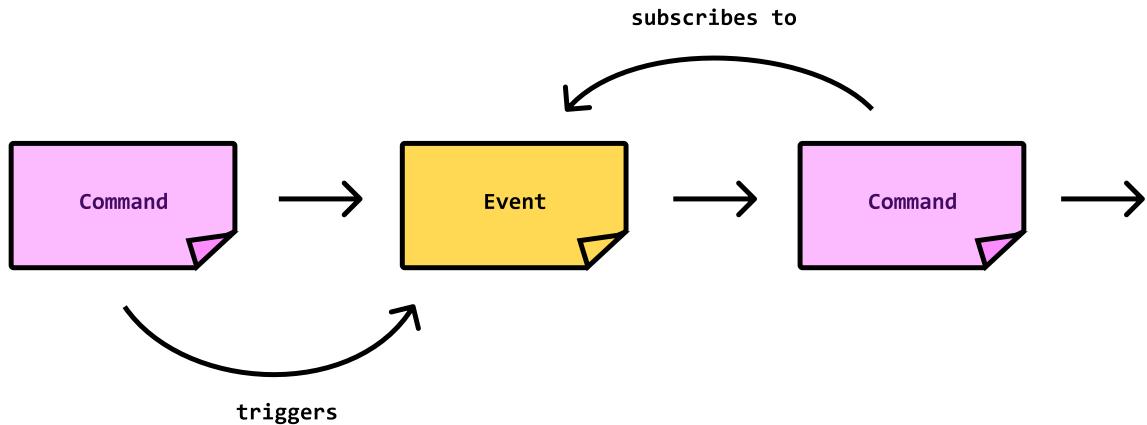
Where did all of these events come from? Well, you know this by now. Most of the time, they come from *commands*. Commands are always written in the imperative “do this” style as if you’re telling somebody or something to do something for you.

For example:

- Accepting an offer would be written as *Accept Offer*
- Declining one would be written as *Decline Offer*

Why are commands relevant? **The commands are our use cases (features).** Commands account for half of our application’s functionality (since queries are the other half).

Now, if we think back to what we’ve learned in 13. Features (use-cases) are the key, commands create events, which in turn — when subscribed to — kick off other commands. This is how decoupled systems flow.



**■ System-generated events:** Sometimes events can be generated by the system. For example, consider the automatic SubscriptionExpired event that may originate in paid online services. Or perhaps the idea we proposed earlier of a sort of VinylWasntShipped event created after 30 days of noticing that the VinylShipped event wasn't created after the OfferAccepted event happened. At that point, the system could automatically punish the trader by reducing reputation points or refunding a trader. *Do you see how powerful this is? Much more powerful than just designing a CRUD API, right?* So, hone into the idea of a timeline. Because much of how we understand the world and its systems has to do with time.

At this point in the session, you should go through the entire wall or board and match each event that has an originating command with the command.

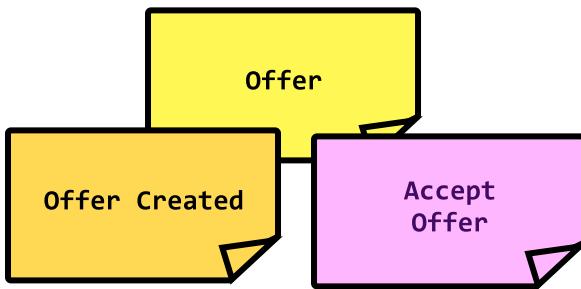
While we're doing this exercise, we'll run into a lot of unhappy paths based on commands that failed. And they *can* fail for reasons valid to the domain (not considering technical reasons). For example, the ListVinyl command could fail if the validation rules don't pass or we invent some *rules* that state if you're not listing it for a fair and reasonable trading price, we should fail the command. Leave these sad paths for now. We'll cover them in more detail when we sit down to understand a story and turn them into acceptance tests (see 21. Understanding a Story and 22. Acceptance Tests ).

See how many command-event pairs you can find and match them up on the board.

### 3 — Identify aggregates (optional)

After you've labelled as many command-event pairs as possible, the next step is an optional one. It's to identify the *Aggregate* most closely related to them.

Aggregates are another DDD pattern. It is an object that represents the domain concept upon which we perform the command *against*. Aggregates encapsulate the rules (also known as invariants) and decide whether a particular command should succeed or fail. The *aggregate* sits in-between the command-event pair.



There are advantages and drawbacks to using aggregates.

Some of the advantages:

- You define a common way *domain events* get created (the aggregate creates domain events)
- You can put common logic in aggregates and re-use it across commands
- Identify relevant entities early on and work on the common language.

Some of the disadvantages:

- It can be challenging to design aggregates and get the boundaries correct
- One aggregate for many different commands seems data may be unnecessarily coupled

What should we do? I've noticed that the DDD community is a little bit divided on this topic. Some say that you don't need aggregates while some swear by it. I believe this very much has to do with if you're using the functional (Scott Wlashin-style) or the object-oriented (Vaughn Vernon, Eric Evans-style) approach to domain modelling.

There are a number of different approaches we can take that range from:

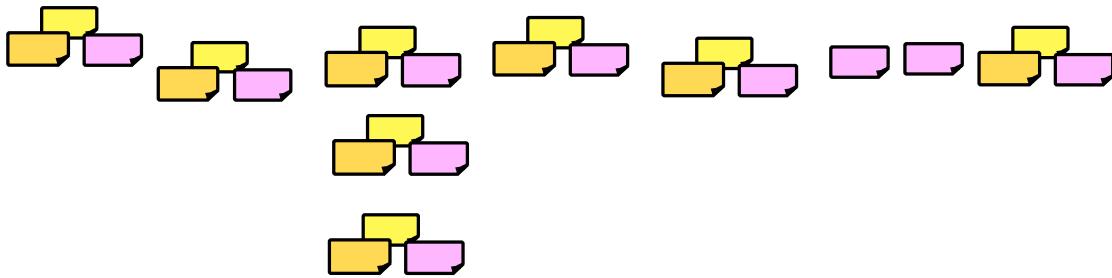
- one single object-oriented aggregate for all commands — like just `Offer`
- the middle of the road FDD-style aggregates where we have a new aggregate for each command based solely on the data necessary — think `OfferForCreateOffer`, `OfferForDeclineOffer`, `OfferForAcceptOffer`, etc
- and the functional approach which completely forgoes the use of a formal aggregate and focuses more on the input data instead

For reasons that we'll discuss in 23. Programming Paradigms, we focus mostly on the first option. However, we still attempt to make use of FP design principles in our OO-style implementation.

I also defining the aggregate in *Event Storming* because we get to start to build some common terminology at this point. If it's not coming up because the name of the aggregate isn't obvious, don't sweat it. We'll always be able to discuss more later.

## 4 — Decomposing to subdomains

With an entire board full of commands, events (and possibly aggregates), we should now have a pretty decent grasp of what goes on within the business, and the various processes associated with what we need to build.



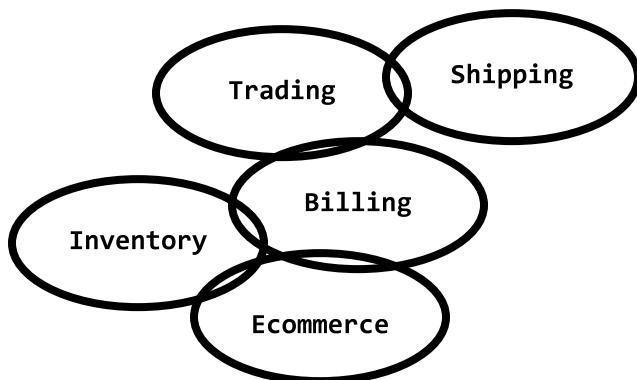
The next principle for us to follow is to “divide the problem domain into smaller subdomains”.

Why split things up? Well, we can’t build the entire company from scratch. And also, we need to make sure that commands-event pairs are co-located to the correct team. Since we’re supposed to be building out the vinyl-trading aspect, details about shipping and fulfilment probably aren’t our concern right now.

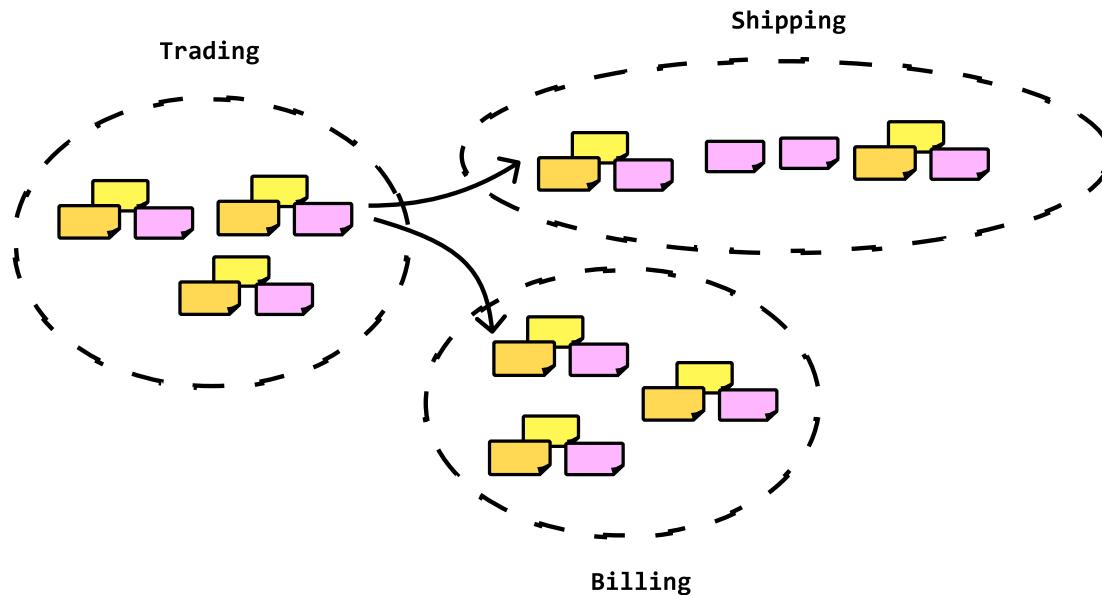
### Subdomains

We used the word *subdomain* a few times. What is that, really?

If we think of the entire problem space, all of these command-event pairs and teams involved at White Label, we could say that White Label is the entire *domain*. When we break it down, decompose it, we have the actual teams that make the whole thing function (or just modules if we’re thinking monolithically). Those teams (or modules) represent the *subdomains* of White Label: decomposed logical slices of the entire problem domain.



At this stage, you'd draw a circle around each of the subdomains containing the command-event pairs and decide on a name for each that makes sense.



It's important to note that subdomains often overlap. We can think about why this happens by mirroring it to the real world, because in reality, many team members from one department often need to know a little bit about how things work in other departments to do their own work properly.

But more concretely at a technical level, to perform a command, we may actually need to rely on commands from entirely separate subdomains. For example, let's consider some of the pseudocode needed to perform *Accept Offer*, a use case from the trading subdomain.

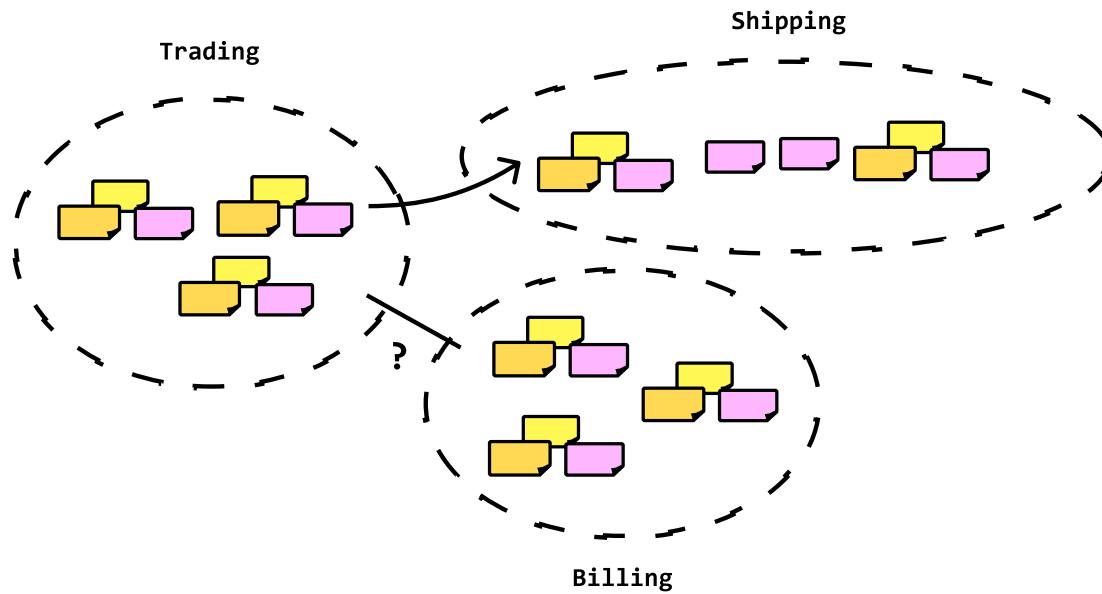
```
// modules/trading/useCases/acceptOffer.ts
```

```
class AcceptOffer {  
    ...  
    public async execute () {  
        // 1. Get offer  
        // 2. Update offer state to accepted  
        // 3. Charge customer  
        // 4. Save new offer state  
        // 5. Save offer accepted domain event  
    }  
}
```

Notice that step #3 involves charging the customer? We want to charge the customer and make sure that their credit card works before we continue on and make the entire transaction succeed, right? Well, how are we supposed to do that if *Charge Customer* is a command from an entirely separate subdomain?

This demonstrates the coupling reality that often happens when designing subdomains. It's extremely rare that subdomains will be completely decoupled from each other. They often need to rely on each other to make things happen.

This also means we really shouldn't be drawing a line from the *Offer Accepted* domain event in the trading subdomain to the *Charge Customer* command in the billing one because that's not the reality of the relationship at all.



I wanted to show you this example here right now because it's contextually important, but typically — you don't need to be diving this deep just yet and drawing pseudocode out in an *Event Storming* session. *Event Storming* is more high-level. Just getting the events-command pairs out and seeing the relationships between subdomains. This is more of a task when you sit down to understand a use case in detail (as demonstrated in 21. Understanding a Story ).

Alright, so where are we? We know there's a relationship of some sort between the two subdomains, but it's not one of subscribing to domain events. It's more like the trading subdomain relies on things from billing. What can we do about this? Well, we be intentional about the relationships between subdomains using something called *Context Mapping*.

### Context maps — documenting the relationships between subdomains

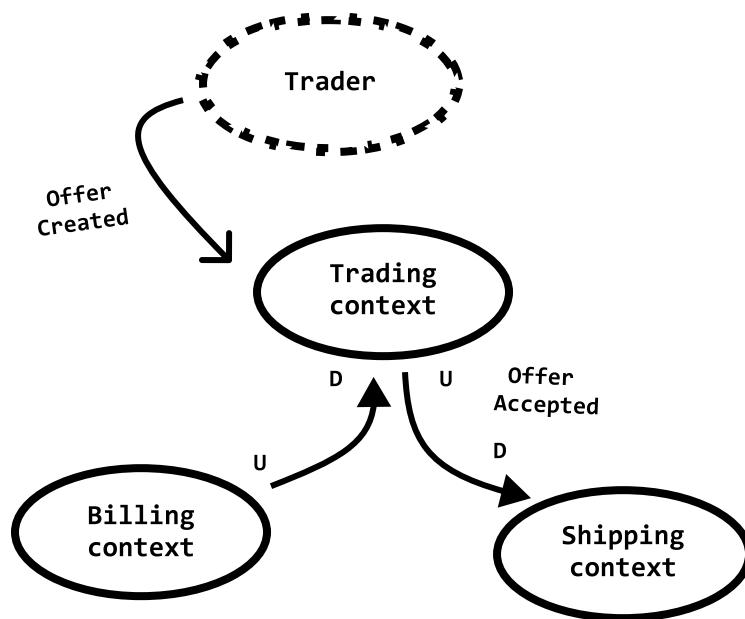
One of the most common questions I get from readers is “how do you rely on commands from other subdomains”? The answer is to make the relationship explicit. This is also one of the most complex aspects of Domain-Driven Design. Even the original *blue book* written by Eric Evans in 2000 was pretty vague about how it works.

Way before we even get into any code, you can use something called *Context Maps* to illustrate the relationships between subdomains at a high-level. By documenting the upstream-downstream or partnership relationship that subdomains have between subdomains, once we sit down to actually write code, we know how we'll need to expose certain functionality from our subdomain modules. Not only does this fend off a lot of confusion when we get

to implementation-time, but if we draw it out, it helps newcomers understand how the system works without having to dive into all the details.

In our particular example, we'd say that the relationship:

- From trading to shipping is *Customer-Supplier* where trading is the upstream supplier and shipping is the downstream customer
- From billing to trading is *Customer-Supplier* where billing is the upstream supplier and trading is the downstream customer



■ **Context map cheat sheet:** This is a more complex DDD topic and there are a myriad of different relationship types between subdomains (ex: Published Language, Anticorruption Layer, Consumer-Supplier, Conformist, etc). You learn about the various context mapping relationships by checking out the Context Map Cheat Sheet here.

### Core, generic, supporting domains

The third principle is to “create a model of each subdomain, but focus on the core one first”.

There's a lot of debate about what the *core subdomain* is. Some say it's the most valuable subdomain — the one that is truly the family jewels. Personally, I believe it is the one that we're getting paid to build. In our example with White Label, that'd be the *trading* \*\*subdomain in which our trading application will rely on.

As Scott Wlaschin writes, sometimes it isn't all that simple:

“An e-commerce business might find that having items in stock and ready to ship is critical to customer satisfaction, in which case *inventory management* might become a core domain, just as important to the success of the business as an easy-to-use website.”

Additionally, we can often sometimes pull subdomains from off the shelf. For example, if

you didn't want to build out your accounts and billing subdomains, you could offload that work to Autho and some other paid service that handles all of your billing for you.

This paves way to the notion that there are **core**, **generic** and **supporting** subdomains. The core one is the one we should focus on, the generic ones are the ones that act like utility subdomains, and the supporting ones are ones that merely support an existing subdomain.

In Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, we implement a core domain and show how it connects to generic and supporting ones.

## 5 — Utilize the ubiquitous language everywhere

And lastly, the final principle we really need at this point is to “develop a common (ubiquitous) language that is shared and used within the code and everyone involved in the project”. This is a long-term task. To do this:

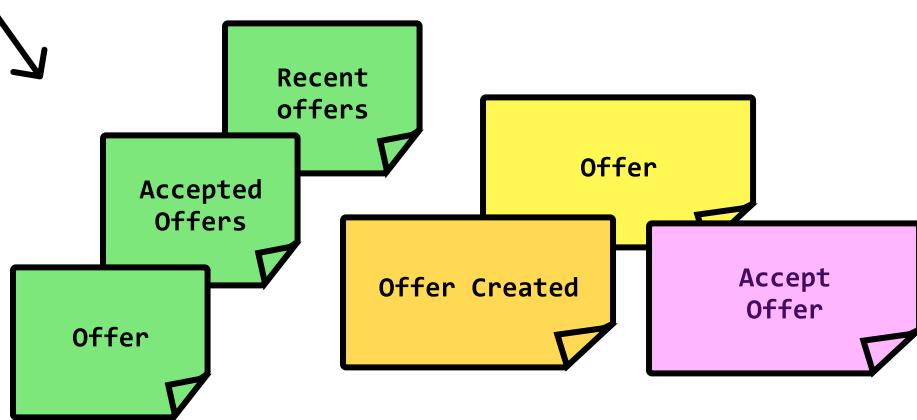
- **Use concepts that occur in the real world, in the code** — If the domain expert calls something a “Vinyl” then we should have something to represent Vinyl in our code as well.
- **Don’t use concepts that don’t occur in the real world, in the code** — Sometimes you do have to use technical words and concepts, but generally, when we’re implementing DDD patterns, we use an architecture that contains a layer of code called a *domain layer*. It’s code that is clear and non-corrupted by things like “factories”, “observers”, “managers”, and other technical concepts — *unless of course, you’re building a code-related domain and it very much has to do with these things*.
- **Can’t cover all domains — each one has their own** — Understand that each domain is going to have their own \*\*dialect. Remember that. A *product* in one domain probably means something completely different in another one. Be very careful of trying to conflate terminology across domains. Actually, just don’t do it.

### What about the queries?

Most of us are building web applications with a view involved, right? At which point in the Event Storming session do we stop to identify the views that we need?

Technically, we can do this anytime after we’ve figured out the commands and the events.

`View models/queries`



If we've identified all the commands and queries, we've identified all the necessary functionality. This is what we go on to estimate, plan, test, and release.

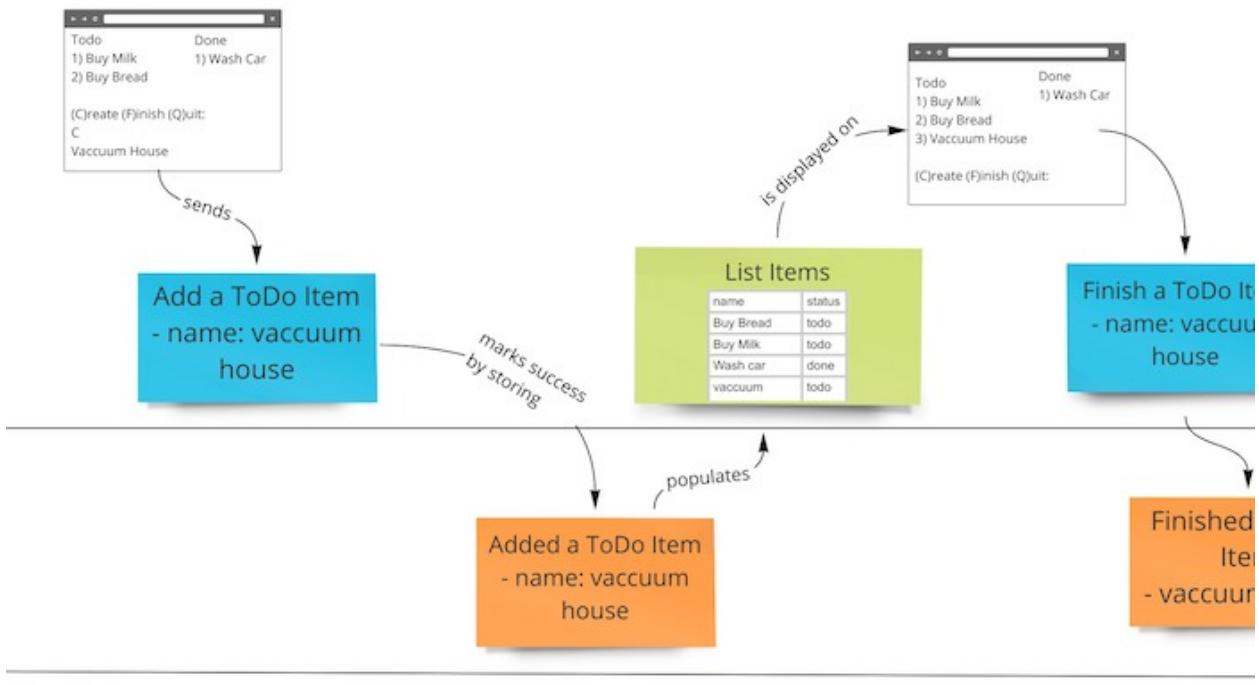
However, in my experience, it's a little bit hard to come up with the views in *Event Storming*. It just doesn't quite feel natural. It's hard to figure out what views you'll need without seeing at least a rough sketch of what the UI would look like and how things *could* work.

There is another practice however, which is very similar to Event Storming. It makes up for this deficiency and more. It's called *Event Modelling*.

## Event Modelling

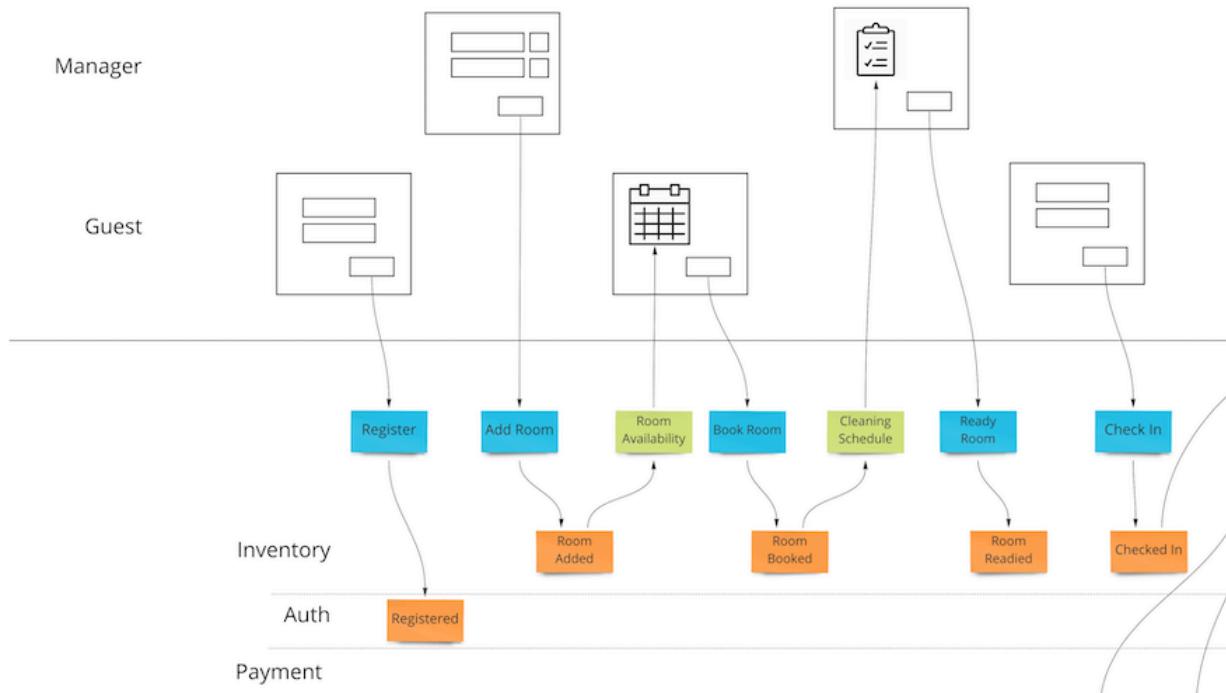
There are two problems with Event Storming, I find. A) what we just talked about — finding the views, and b) often, when designing out the events on a timeline, you'll find that the chronological sequence of events can sometimes flip back and forth between subdomains. For example, if we had to refund a *trader* for something, the path may go from *Trading* to *Shipping* to *Billing* to *Trading* again. That's not really a problem, but drawing it out on a board can be pretty challenging.

*Event Modeling* is the latest approach which takes into consideration all these pain points. Very similar to *Event Storming*, we focus on all of the piece of the puzzle chronologically, planning a system before we code it.



Event modelling is combines the best of Event Storming and architectural principles to give us a clear view of information systems. Image courtesy of [eventmodeling.org](http://eventmodeling.org).

However, while it shares its similarities with *Event Storming*, what makes *Event Modelling* special is that it makes use of rough **UI drawings** and **swimlanes**.



Snippet of an Event Modeling session done on a Hotel Reservation domain. Image courtesy of [eventmodeling.org](http://eventmodeling.org).

The philosophy here is that rough sketches help you come up with potential queries for the view, and swimlanes let you document the jumps between subdomains where it would be virtually impossible with *Event Storming*. This is a way to maintain the sequential aspect of the design (from left to right) and keep things looking clean.

The ability to document the **Roles** of the users and perform some rough drawings of the UI that contains the buttons which issue **commands** is a nice touch. In my opinion, this is the most clear and complete approach to a necessary and sufficient systems design I've yet to come across. It combines 8. Architectural Principles like Conway's Law, the best parts of *Event Storming* and some of the traditional aspects of classic use case design into something wildly useful.

I know a lot of developers still prefer *Event Storming* and that's why it's good to learn it first, but I personally prefer *Event Modelling*. I highly recommend you look into it at EventModeling.org.

## Summary

### Domain-Driven Design concepts

In this chapter, you learned a good chunk of topics in Domain-Driven Design and how we can use them to learn the domain a lot faster. We learned that:

- A *domain* is a set of concepts, processes, and knowledge related to the problem we're attempting to solve, or as Wlaschin says, "that which a *domain expert*" is expert in.
- A *domain model* is a simplification of the domain that enables us to build implement a practical *solution space* for the problems we're trying to solve.
- The *ubiquitous language* is the set of shared terminology that is used by team members and within our code.
- A *context map* is a high-level diagram that shows the subdomains and the relationships between them.
- A *domain event* is a past-tense record of something that happened. It sometimes triggers commands.
- *Commands* are requests for some behavior to happen. They can be triggered by a person or an event. When a command is successful, we change the state of the system and save one or more *domain events*.

## Wrapping up

- Domain-Driven Design is a set of principles, patterns, and practices that help us learn the domain, build a shared mental model, and tackle complex problems by structuring code so that it utilizes the ubiquitous language ("the common language") of the domain.
- The four most important principles to implementing domain-driven design are to:
  - 1 — Focus on events and processes rather than data.
  - 2 — Partition the problem domain into smaller subdomains.
  - 3 — Create a model of each subdomain in the solution.
  - 4 — Develop an "everywhere language" that can be shared and utilized between everyone involved in the project.

- To kick off learning, we discover all of the *Domain Event* with Event Storming: a practice which involves gathering up a group of \*\*developers and domain experts from across the organization and asking them to plot out their business events and workflows.
- At the end of an Event Storming or Event Modelling session, you and everyone involved in the exercise should have a high-level understanding of the domain.
- We learn about some of the interactions between subdomains by seeing how events, commands (and views) chronologically pass through them. We saw how to use a *context map* to illustrate the relationships.
- It's likely that we'll have missed some details or nuances. It's also likely that we'll have a number of questions on trouble spots as well, but we'll iron those out when we spend time 21. Understanding a Story before we implement the feature.
- The next step in our process is to pick up the commands and queries (use cases) and perform the story-writing process with them.

## References

### Articles

- <https://medium.com/nick-tune-tech-strategy-blog/domains-subdomain-problem-solution-space-in-ddd-clearly-defined-e0b49c7b586c>
- <https://khalilstemmler.com/articles/graphql/ddd/schema-design/>
- <https://vladikk.com/2018/01/21/bounded-contexts-vs-microservices/>

### Books

- Domain Modelling Made Functional by Scott Wlaschin
- Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans
- Implementing Domain-Driven Design by Vaughn Vernon
- Domain Driven Design: Distilled by Vaughn Vernon
- Patterns, Principles, and Practices of Domain-Driven Design by Scott Millett and Nick Tune

## I7. Stories

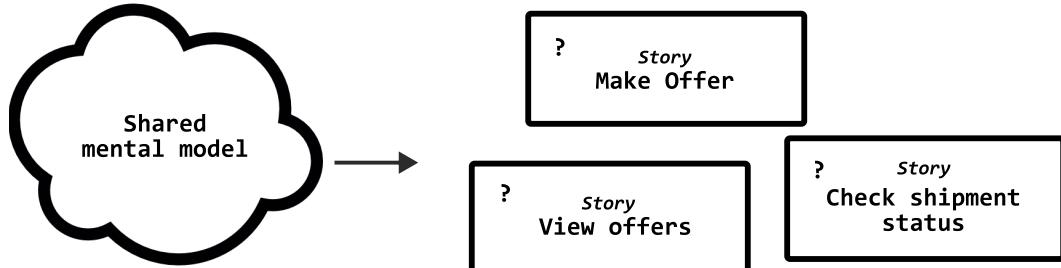
■ Stories are an early part of the evolution of a feature. We use stories as temporary placeholders for communicating, estimating, and planning features (both functional and non-functional).

In the last chapter, we learned how to use Event Storming or Event Modelling to create a shared mental model of the domain.

At the end of either of these discovery-like workshops, we will have identified the vast majority of the business events, commands, and queries involved with creating the system we've been tasked to build. This puts us in an incredibly good place for the next steps: estimating and planning the features. How so? Because the commands and queries are the *functional requirements*.

Each command (like *Make Offer* or *Ship Vinyl*) represent valuable, atomic units of functionality. On the UI side, each view contains any number of queries (like *Check Shipment Status* and

*View Offers*) which can also be communicated as atomic chunks of valuable functionality.



*Communication* is the key word here. We need an artifact to effectively communicate the hazy, non-detailed idea of a feature that we have come to realize as being a part of the essential complexity. We call this artifact a *story*.

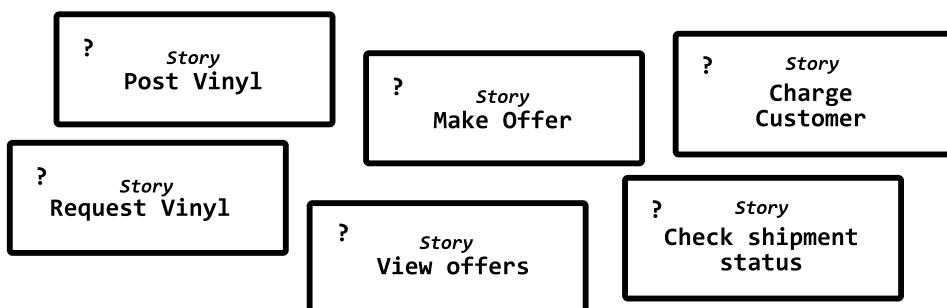
## Chapter goals

In this chapter, we'll learn about:

- Stories and the dual-utility for both customers and developers
- The principles of writing good stories
- How you can write non-functional requirements as stories too

## User stories

A story is temporary placeholder for communicating, estimating, and planning features — and features are units of functionality that are valuable to the customer.



Since our entire goal in this endeavour of software development relies on consistently de-

livering value to the customer, we need those chunks of value to be readily visible so that we agree on what it is we're going to build. User stories are the way that we track progress. They're like a contract. Once we turn a user story into acceptance tested, deployed code, we're good. We can report that the work is done and tell the customer to go check it out on the staging or production server.

## Developer responsibilities

From the perspective of a developer, we help customers identify stories, estimate them, and then implement them.

*Event storming/modelling* is a great way for us to collaboratively extract stories from the customer's minds in a succinct way, but there's a fair amount of developers who also just sit down, ask questions, and try to understand what it is the customer wants too. Either works.

The most challenging aspect to either approach is **rejecting detail**, and that's what *Event Storming/Modelling* does well — keeping things high-level. If you're taking a more person-to-person approach to requirements gathering, you don't want to be the person sitting there copiously copying every detail that the customer gives you into a story. Too much detail makes stories un-estimable and evidently, un-plannable. Leave those details for later. We're merely focused on identifying the features at this point. Not all of the data and behaviour involved in making it work.

In estimating stories, the phronimos developers will generally advise you to use index cards if possible, just because it's a lot more pleasant and easy to pick up a card, write a few words on it, and then assign it a story point. But nowadays, with the quality of online collaboration tools, you could just as easily run a good story-writing and estimation session with any kind of Kanban software. It's also advised that when you write the story, you write the estimate at the same time. Trust your initial instinct — whatever your gut is telling you. This keeps things rolling. We will learn how to develop the knack for this in 18. Estimates & Story Points. Upon further examination, if you find you need to split, merge, or spike a story later, that can also be done.

## Customer responsibilities

In XP, customers write stories. This means taking a backseat role in *Event Storming/Modelling* as much as possible and letting the domain experts do most of the talking and placing stickies on the board. We, the developers, should not be coming up with stories.

You: "Hey wouldn't it be cool if we wrote the cache in a way that we could —"

No, don't do that. Don't be like that kid in elementary school who asks the teacher "aren't you going to assign us weekend homework?" right before the bell rings on a Friday afternoon. If it's important, we'll hear about it.

With stories estimated, the customer needs to sort these into iterations leading up to a release. And before we get to the halfway point in an iteration, we'll need to make sure we have the acceptance tests for the stories we're building specified.

## INVEST: Story-writing principles

There's a nice little acronym for how to write good stories. It's called INVEST: stories must be independent, negotiable, valuable, estimable, small, and testable.

### I — Stories must be independent

This principle is about dependencies. Sometimes it may feel like have to implement functionality in a particular order. That is more often than not, not true. If we wanted to, we could implement *Logout* before we implement *Login*. Or perhaps,

Developer: "How can I implement *Make Offer* feature \*\*if *Create Account* hasn't already been done?"

*With care* is how. When we build out our acceptance tests, we shouldn't be running tests through several other existing stories in order to execute the one under test. If we structure our acceptance tests properly (assembling the pre-conditions — the state of the world necessary to exercise our acceptance test), all stories *can* be written independently.

From the business side, it's more likely that there will be dependencies here. For example, if we're working towards an initial MVP, having the *Create Account* and *Login* features in the same release that has *Make Offer* makes sense.

### N — Stories must be negotiable

It's important to recognize that anything we write down at this stage, from the user story, to the estimate, all the way up until but before we have the acceptance test is negotiable. Just like an *estimate* isn't contractual, neither is a user story.

User stories are merely to capture the essence of what is to be created. Some user stories, especially when roughly ideated by the customer, may be unreasonably large, complex, or just borderline impossible for us to *story-ize*.

Here are a few examples of stories that just don't make sense.

- "Solves the problem"
- "Show the user a popup in a way that makes them feel nice"

The result of what gets written on the card comes from a negotiation between us and the customer (or a proxy for the customer like the Product Owner/Manager). Sometimes we may need to ask questions.

**The magic question:** Here's a trick to weed out stories that don't make sense and aren't testable. Ask **the magic question**:

"How will I know when I've done that?"

■ **Sh\*t user stories:** For fun, you can find a whole slew of badly written user stories here.

### V — Stories must provide value

All stories must have some sort of business value otherwise they shouldn't be done.

Messing around with databases, styling, layout, class design, and so on are technical concerns. Those aren't stories. Neither are fixing bugs or refactoring.

We have to be careful here though. Functional and non-functional requirements are both valuable. One is a sort of external, customer or business-facing value, and the other is a more *internal* sort of value.

As a rule of thumb, you should be able to write all stories (functional and non-functional) as *user-oriented* stories. The “as a [role], I want [feature], so that [value]” story format helps, especially that last part — the “so that [value]”. We'll learn more about this momentarily.

## E — Stories must be estimable

Stories that are too large are hard to estimate. In that case, we split them into smaller stories. Stories that don't have a clear observable result or value to them are also hard to estimate. In that case, we **ask the magic question again**.

“How will I know when I've done that?”

Other times, stories may rely on us to use some technology we've never learned before. Maybe we have a non-functional requirement as a story that relies on us using an architectural pattern or approach that we're not familiar with. Maybe there are performance considerations we have to address, but we have no idea how to do that.

In these cases where we're uncertain because we need to do some research, we *spike* a story (18. Estimates & Story Points), which ultimately means we allocate time for us to do that research.

## S — Prefer short stories

The shorter the story, the better right? Generally, yes. On average, we expect user stories to take between 2-4 ideal days to complete. If, when reading a story, you notice that there are several commands and queries nested within it, consider splitting it.

For example, depending on the application, *Do Profile* could be broken into *Edit Profile*, *Upload Picture*, *Get Profile Details*, *Get Public Profile*, and more.

## T — Stories must be testable

Lastly, and very importantly, stories need to be testable. If the business can't articulate how we'll be able to prove that the story has been completed, then we need to rework it (use the magic question again). For something to be testable, there needs to be a precondition (GIVEN), some action (WHEN), and a postcondition that we can use to ascertain the intended result (THEN).

If the business can't specify how we *could* test it, we'll never \*\*be able to write the tests for it.

## Story-writing formats

There are ultimately two recommended formats for writing stories: the *command/query* format or the “as a, I want, so that” format.

## Command/query format

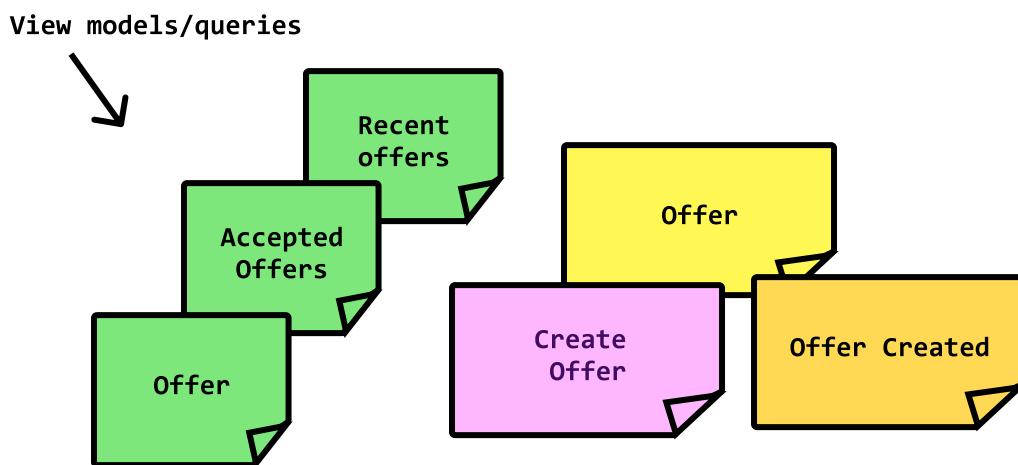
Turning commands and queries into stories is relatively straightforward. You can merely take them from an *Event Storming/Modelling* session and use those as stories.

For examples, here are some commands that we've found:

- Create Offer
- Accept Offer
- Post Vinyl

These are all valid commands *and stories*.

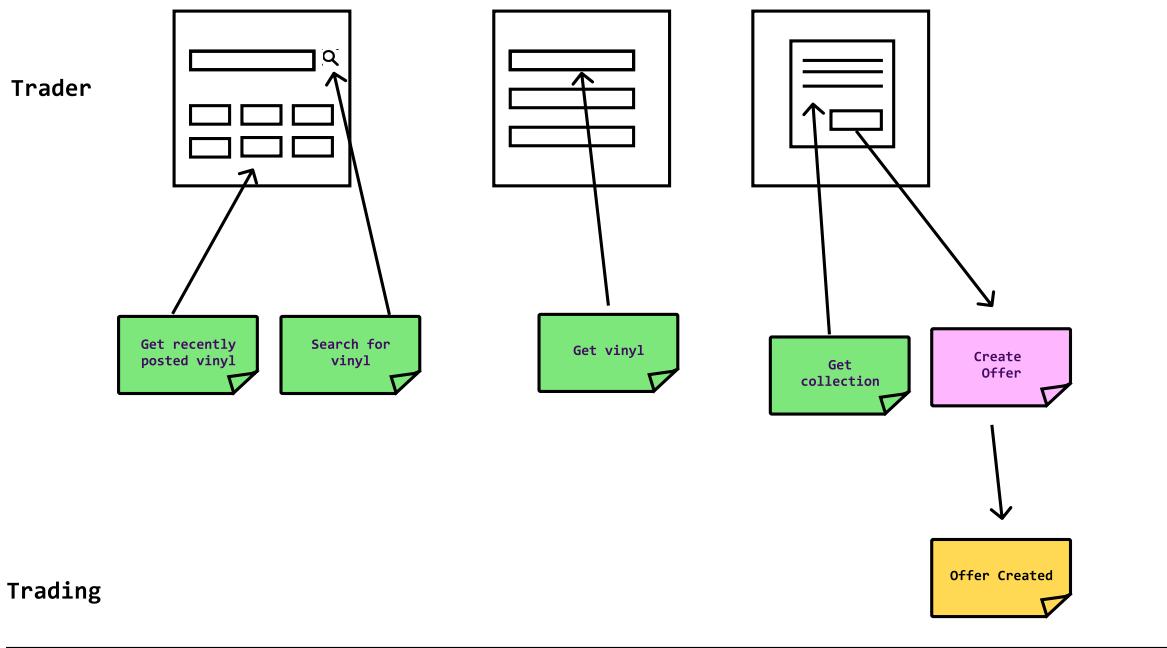
Queries come from reporting requirements and whatever is necessary to enable a user to click through the UI to actually even invoke the commands in the first place.



As we mentioned in the previous chapter, *Event Storming* is an alright way to gather the queries, but I really do prefer *Event Modelling* as a way to ensure we don't miss any important queries, and to reduce us creating queries we don't need. We may not actually need the ability to view *Accepted Offers*. It's hard to tell.

With respect to the story-writing process, the biggest benefit I believe we get from *Event Modelling* is the ability to ascertain which *page* queries belong to. If you'll recall from 6. Organizing things, features (commands and queries) live in pages.

For example, look at all of the queries and pages we tally up in the process of getting to the screen where we can finally invoke the *Offer Created* command.



A home screen, a results page, and a screen with a modal displaying an offer. Amazing, right? With a rough sketch, we've identified four separate queries on at least three different screens.

*Get Recently Posted Vinyl* could easily be its own story because it seems to adhere to INVEST pretty well.

*Search For Vinyl* could also be its own story. It'll \*\*require work on at least two separate screens though: the main page with the search bar and the results page.

■ **Screen vs. page:** I've been using the term *screen* and *page* interchangeably, but some developers actually like to think of a page as the root element and a *screen* as one of various different states that a page can have. The school of thought here is that a *page* has a one-to-many relationship with *screens*.

### As a [role], I want [feature], so that [value]

*Search For Vinyl* is an OK-ish name for a story, but there are details that are missing.

Another format is the “as a [role], I want [feature], so that [value]”. If we were to re-write the *Search For Vinyl* story using it, it'd look like:

“As a *trader*, I want to *search for vinyl* from both the main page and the results page so that I *can find rare vinyl that I'd like to make an offer for*.”

This is about as long as stories should ever be — a few words to a sentence.

## Writing stories for non-functional requirements

In case you forgot, non-functional requirements are requirements that are *still valuable*, but they're more to do with things like performance, security, reliability, and other cross-cutting concerns.

For example, if we were asked to display the total number of records stored in a database, we could say that's a *functional requirement*. Now, as for how up-to-date and current that number needs to be? That's a *non-functional requirement* (consistency). This would certainly affect the way we design our system.

Let's say that you've never had to build an architecture to support those kinds of software quality attributes before. How can we turn non-functional requirements into user stories?

The same way we did before with the “as a [role], I want [feature], so that [value]” format. Focus on the business value.

Let's say we were building an application that monitored stocks. A functional requirement could be:

*As a day trader, I want to be notified when the price of a stock on my watchlist goes up over 5% so that I can decide whether I should make a trade or not.*

That's still functional. Making it non-functional:

*As a day trader, I want to be notified when the price of a stock on my watchlist goes up over 5% **within 1 to 10 seconds of it occurring** so that I can decide whether I should make a trade or not.*

Words like *quickly* or *as soon as possible* are subjective — they mean something different to every single person. But “*within 1 to 10 seconds*” — well, that's something that lives in the objective.

Here are some more examples:

- As a *customer*, I want to be able to run your application on all versions of Windows from *Windows 95 on*. **This is better than it being able to work on most machines.**
- As a *user*, I want the site to be available 99.999 percent of the time I try to access it, so that I don't get frustrated and find another site to use. **This is better than the website is always up.**
- As someone who speaks German, I might want to run your software someday. **This is better than it has accessibility built in.**

Sometimes the value is clear and we don't need to include it. The key to this approach is to focus on what the real value for the user actually is and write it that way. This often means putting in a quantitative value instead of a qualitative one.

## Summary

- So far, the evolution of a feature has gone from *Essential complexity* → *Shared mental model* → *Un-estimated story*. Not bad.
- A story is temporary placeholder for communicating, estimating, and planning features — units of functionality.

- Developers help customers identify stories and estimate them. Once stories are planned and have acceptance tests specified, we turn them into tested, integrated code. This is our primary goal and it's how we deliver value in an XP project.
- Customer responsibilities are to ideate stories, plan their releases, and write their acceptance tests.
- Stories can be written as just the command or query. They can also be written using the “as a, I want, so that” format.
- Good stories implement the INVEST principle, meaning that they are independent, negotiable, valuable, estimable, small, and testable.
- Non-functional requirements can be expressed as stories too; they are most effectively communicated by using the “as a, I want, so that” format.

## Exercises

■ Coming soon!

## References

### Articles

- <https://www.mountaingoatsoftware.com/blog/non-functional-requirements-as-user-stories>
- <https://agileforall.com/when-in-doubt-ask-how-will-i-know-ive-done-that-2/>
- <https://agileforall.com/new-to-agile-invest-in-good-user-stories/>
- [https://en.wikipedia.org/wiki/Non-functional\\_requirement](https://en.wikipedia.org/wiki/Non-functional_requirement)

## Books

- Clean Agile by Robert C. Martin
- Extreme Programming Explained by Kent Beck
- Planning Extreme Programming by Kent Beck and Martin Fowler

## I8. Estimates & Story Points

■ We use estimates to approximate the amount of time it will take to implement a story in *ideal time*.

What do performing stand-up comedy, starting a business, and playing a round of golf for the first time have in common? You have absolutely no idea how it is going to go. Yet, so many people go and do 'em anyways. They just jump straight into the void.

Estimates are very similar. When we first start out, we have a zero batting average — no data. This, combined with the fact that it can sometimes feel like if we give a bad estimate, we'll be personally judged for it, is what can make having to provide estimates feel like a developer's nightmare.

You don't have to feel scared giving estimates. And even if you're bad at them (everybody is at first), with a noble amount of discipline, you can drastically improve them.

In fact, I have a friend who is currently killing it as a freelance developer. He's gotten so good at estimating projects that he no longer charges hourly. He charges by *value*. If a project is worth \$10,000 and it will only take him \$5000-worth of his time, he'll charge the \$10K as a flat rate and get it done on time. This wisdom only comes through practical experience. Phronesis.

It's like when a lady noticed Picasso having lunch and asked him if he could draw her a quick sketch. Picasso politely agreed, promptly created a drawing, and handed back the napkin — but not before asking for a million Francs. The lady was shocked:

"How can you ask for so much? It took you five minutes to draw this!"

"No", Picasso replied, "It took me 40 years to draw this in five minutes."

## Chapter goals

In this chapter, we'll learn:

- How and why we provide estimates based on ideal time or effort
- How to estimate stories that are too small, too large, or carry uncertainty to them
- How the stories we estimate fit into releases and iterations
- Effective estimation techniques you can perform by yourself or with a team
- How to improve your estimates over time

## About estimates

Our job is to estimate the user stories so that customers can organize them into releases and iterations. Even more so, the Agile value of estimates is so that the **customer can start arranging the stories in order of what will bring the most value soonest**.

Typically, the best time to estimate stories is when we're discovering the stories. If you're using cards, write a number on it. If you're using a spreadsheet, slap a value in there.

Estimation is a technical task. That means developers do it. If a business-person is signing you up for a task and telling you how long you have to complete it, they are inadvertently estimating the task for you and you should tell them if it can or cannot be done.

Estimation is stressful because business tends to think that estimates are commitments. They're not commitments. **Professionals never commit to anything they're uncertain of. And because we're professionals, we are not committing to estimates.** We're just *estimating* how long we think they'll take to deliver, hypothetically.

## Ideal time (effort)

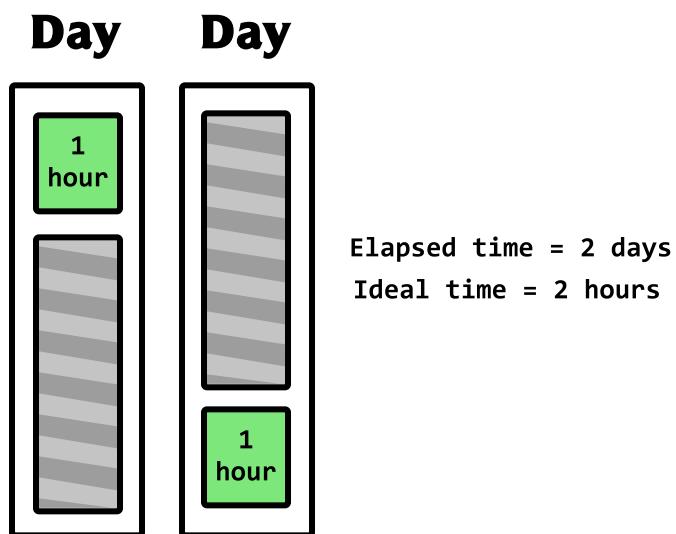
We're estimating **ideal time or effort**. Ideal time can be thought of as *time gone uninterrupted without any distractions*. It refers to the time we spent fully-committed to the task and productive, and doesn't include time spent taking meetings, helping other developers, or getting pulled onto other commitments.

It's important to make the distinction because distractions like these are common and likely to occur. To simplify our estimates, we don't account for the entire range of possible distractions, natural disasters, or big-foot sightings.

Make estimates based on the assumption that we're sitting there at our desk plugging away at the story. This means that if you've estimated something that should take you one day of ideal time (and let's be honest about the number of hours we *actually* work in a day — it's only going to help make our estimates accurate) \*\*but we don't complete it in that one day (due to distractions), then we pick it up tomorrow.

Yes, the *ideal time* and the *elapsed time* could be very different! That's something we monitor separately (actuals). But when we make our estimates, we're making them based on *ideal time*, not *elapsed time*.

So if you only really get one hour of ideal time to code each day, two days of actual time is two hours of ideal time. See the diagram below.



## Story points

When it comes down to the actual unit of measurement that you use, a lot of people like to use something called *story points*.

Story points are a numerical value that maps to the amount of effort required to implement stories. That effort is *ideal time*. When you're estimating stories, you'll use one of the numerical values to *grade* the story.

Story points are related to *velocity* in the sense that the amount of story points you're able to complete in an iteration is your current velocity. Your velocity is the bucket for future iterations. Customers can add stories that don't overflow the bucket.

So then, which numbers should you use for your story points? In all actuality, it doesn't matter as long as you're consistent. For example, a lot of people like to use the Fibonacci sequence as story points (1, 2, 3, 5, 8, 13, 21). Hey, that works.

## How to make estimates using the recommended story points table

I mentioned that story points map to *ideal time*, whether it be days or weeks, when you assign a story some story points, you're saying *roughly* how long you think it'll take.

One of the most common story point mappings is with the Fibonacci sequence. The values are as follows:

- Story point = 1 (we know what to do, no dependencies, or less than 2 hours of work)
- Story point = 2 (we mostly know what to do, almost no dependencies, or about a half day of work)
- Story point = 3 (we know some of what we need to do, some dependencies, or less than 2 days of work)
- Story point = 5 (we know very little about what we need to do, a fair amount of dependencies, or 2 to 4 days of work)
- Story point = 13 (we don't know what we need to do, we don't even know the dependencies yet, or should take longer than a week)

What do we know?	Everything	Most things	Some things	Few things	Nothing	Nothing
Dependencies?	None	Nearly none	Some	Fair amount	Many	Unknown
Effort?	< 2 hr	Half day	< 2 days	2 to 4 days	~ week	> week
	<b>1</b>	<b>2</b>	<b>3</b>	<b>5</b>	<b>8</b>	<b>13</b>

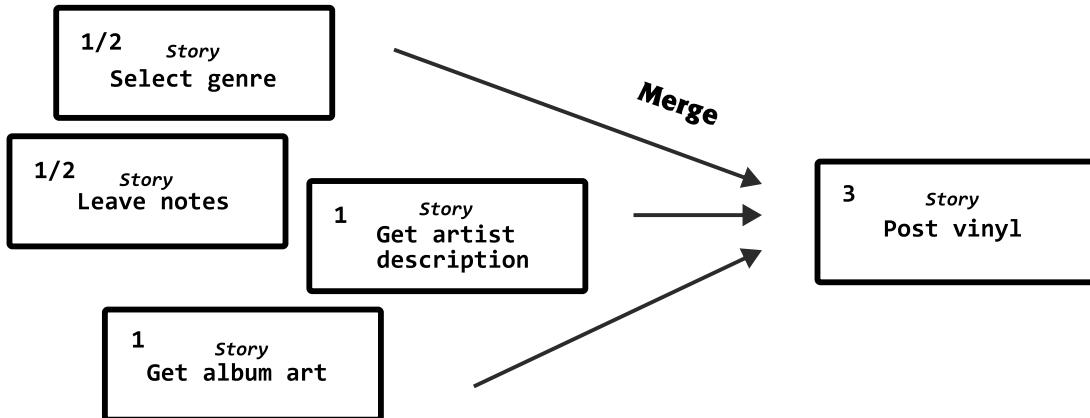
You'll sometimes find yourself in a position where you have to make a decision between one or two options because the story doesn't perfectly match up with the values. For example, if I know **everything about the task** but it has **one dependency**, I could rank that either a 1 or a 2. It's generally safer to estimate up than estimate down.

## Refactoring stories

If you're having trouble estimating a story, there are a few common reasons: it's either too small, too big, contains a non-functional requirement, or there's uncertainty involved. For each of these, you'll either have to merge, split, or spike it.

### Merging — when stories can be consolidated

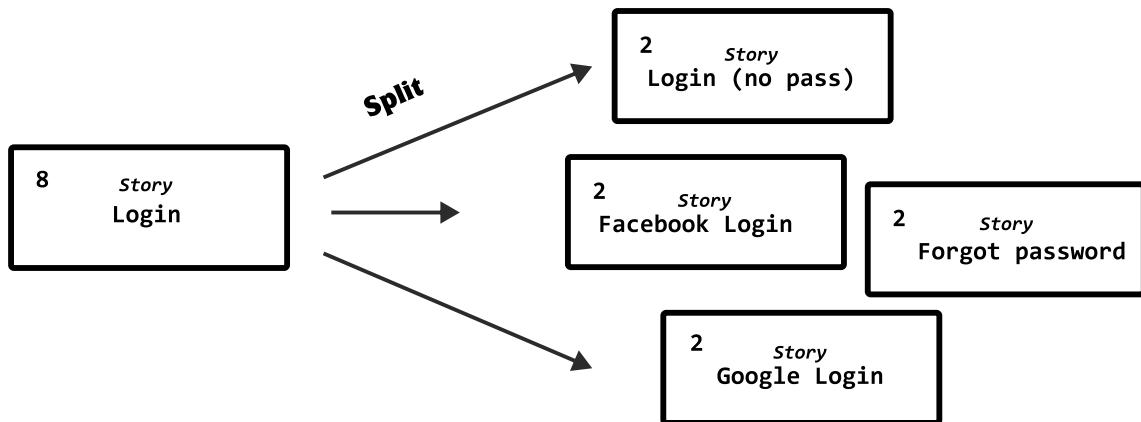
When there are a number of stories that are too small, cohesive, and it makes more sense for the work to be aggregated into a single story, we merge it. Of course, you add all the points up when you do this.



### Splitting — when a story is too big

When a story is too large (this could be anywhere from 8 and above), it makes sense to split it. In this case, you split the story into multiple ones and use your best judgement to assign the points correctly.

For example, consider “Login” as a story. We could split this into “Login with No Password”, “Allow Multiple Login Attempts Before Locking Account”, “Facebook Login”, “Google Login”, “Forgot Password”, etc.



Note that pretty much any story can be split. Splitting it isn't the hard part. Maintaining INVEST is.

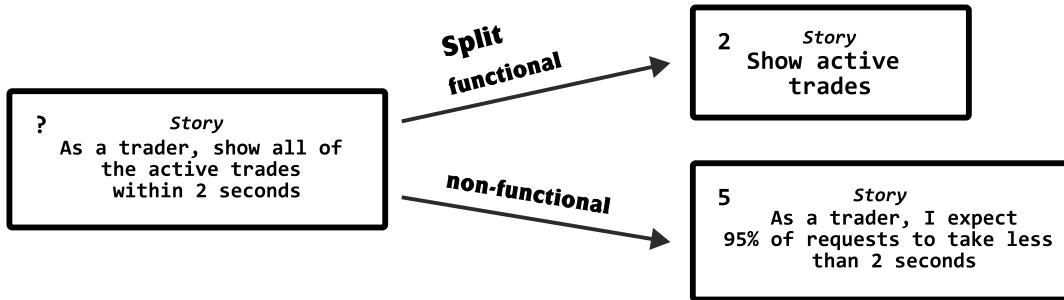
### Splitting — when a story contains functional and non-functional requirements

Here's a more interesting example. Sometimes, you'll get stories containing the conventional **functional requirement** but come mixed together with a **non-functional** one. For example:

"As a trader, it needs to show all of the active trades in less than 2 seconds so that I can respond quickly."

The functional part of this is the "showing of active trades" while the non-functional part of this is "doing it in less than 2 seconds".

As we know from 17. Stories, non-functional requirements can be written as stories too. Be sure to catch these and split them into their own stories when you see them.



## Spike — when there's uncertainty

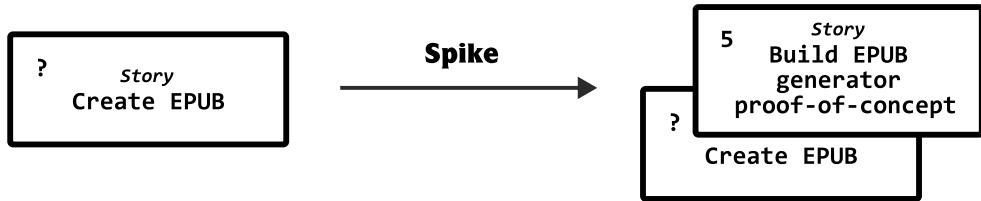
When we're uncertain about a story because there's a tough technical or design problem, we spike it.

A spike is a kind of *meta-story* in a way. Some may even say it's a story for estimating a story. It works by writing the minimum amount of code that cuts through all the layers of the system (like a spike or slice) and proves that we can do what we need to do. A spike is a proof-of-concept or experiment. We intend to throw it away after we figure out how to do what we've been asked to do.

For example, if we need to create EPUB files on demand using a library we've never used before, there are a number of technical challenges there that we're going to have to figure out. Or what if we're being asked to implement a non-functional requirement (like performance) and we're not exactly sure what kind of architecture would support it. Maybe we know that we need to use a cache but we've never done that before. What do we do?

In any scenario where we obviously need to do some research, we spike the story by assigning it an estimate on how long we think it'll take for us to run the experiment and come up with an *actual* estimate.

If we're using cards, we staple the spiked estimate overtop of the original estimate. And in a future *Iteration Planning Meeting*, if the customer decides they want to play the original story, they can't because of the spike. They have to play that instead. After the work on the spike is done, then we can do the work on the original story.



## Team estimation techniques

If you're the only developer, the estimation story points table from above should be good enough. If you're working with others though, you'll need to come to a consensus on your story estimates. Here are a few techniques you can use with your team.

### Flying fingers

First, everyone discusses the story and the possible challenges. Give everyone a moment to think about the estimate individually. Then simultaneously, everyone uses their fingers to demonstrate the number of story points they feel is necessary for the story.

If the result isn't completely unanimous, then we can take the average of what was shown.

### Planning poker

In planning poker, we do the exact same thing as flying fingers, except we use cards in a deck.

Each person selects a card from their deck and we reveal them all at the same time.

If everyone agrees on the same story points estimate, great. We're done. But again, if someone disagrees, we have to do something about that.

### Ways to handle non-unanimous estimates

In flying fingers, planning poker, or whatever different form of this game you can think of, it poses the question of what we do if someone doesn't agree. There are a few different options.

- Take the average (or discuss really wild estimations).
- Take the *lowest* estimate — if the team gets burned, then you'll learn to not make extremely low estimates like that anymore.
- Force the high and low estimators to discuss and then repeat the exercise until you come to a consensus.

## How to improve estimates

The only way to improve your estimates is to make estimates and track your actuals. There's no other way.

To track how long your work has actually taken, I recommend a tool called RescueTime. It records the amount of time you spend using programs (like VsCode and Terminal) every day.

Personally, I record my actuals in a database using Notion. That way, in the future, if I need to estimate a story, I can look to what I've done in the past and base my estimates off of that.

Sometimes you'll have to use approximations like "that looks like it's going to be *two Logins* and we know how long *Login* usually takes". This is made ready only by historical data.

## When to update estimates

Update estimates when the velocity changes dramatically. For example, if someone leaves the company, goes on vacation or whatever — it may be a good idea to reconsider the estimates, especially if you're mid-sprint.

## Summary

- Customers need us to estimate user stories so that they can plan iterations and releases with them.
- We estimate stories based on *ideal time* or *effort*, a number — typically the Fibonacci sequence — which measures effort and roughly maps to days or weeks.
- The numerical value we assigned to a story is called *story points*. The number of story points we can complete in an iteration is called the velocity. Each iteration can only be assigned stories whose points add up to but doesn't go over the max velocity for the team.
- When estimating stories, we merge stories that can be consolidated, split ones that are too big, and spike stories when there's uncertainty.
- When estimating with a team, use a technique like flying fingers or planning poker.
- Estimates are inaccurate at the start of a project, when we haven't done something similar in the past, and when we don't track our actuals. As long as we record our actuals and adjust our velocity based on Yesterday's Weather, estimates should improve over time.

## References

### Articles

- <https://8thlight.com/blog/bjorn-johnson/2016/11/29/making-large-projects-manageable.html>

### Books

- Planning Extreme Programming by Kent Beck and Martin Fowler
- Extreme Programming Explained by Kent Beck
- Clean Agile by Robert C. Martin

## Videos

- Agile Estimation by Mike Cohn

## 19. Release Planning

■ Release planning is the process of allocating user stories to releases and iterations. It is driven by the customer and navigated by the programmers.

One afternoon, I'm finishing up lunch at a restaurant with a couple of friends.

Chris: "Oh man, I'm so full. I'll just pack this up for later".

Nick looks at Chris' plate.

Nick: "Naw, man. Did you really come all this way, fill up on the rice, and *not* even touch your steak?! You barely even touched it. What's wrong with you?"

Chris: "What? Relax, man!"

Just like Nick, customers want to eat the steak first. That is, we have to try to give them the most valuable features as soon as possible. Release planning is an activity in which we can help them do just that.

### Chapter goals

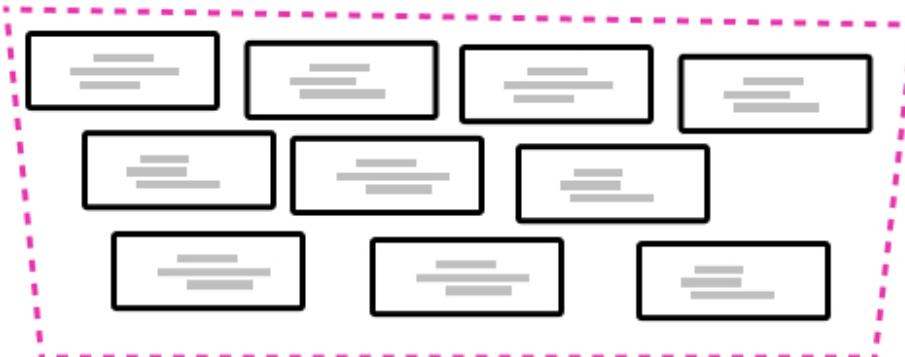
In this chapter, we'll discuss:

- Release plans and the division of responsibility between the customer and the developer in release planning meetings
- How to balance release plans by value and by technical risk
- Factors that cause the release plan to change
- How to factor in developing the infrastructure for the stories

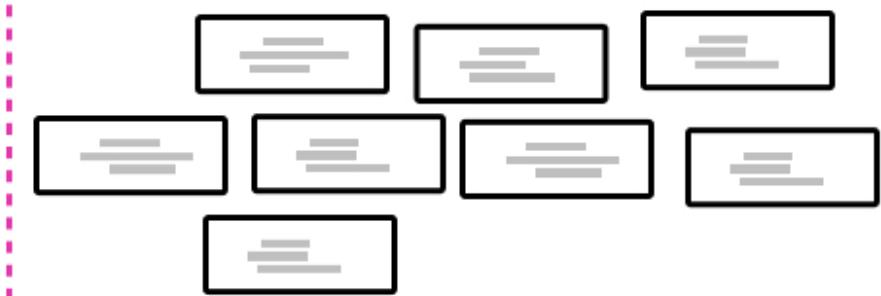
### What is release planning?

Release planning is a technique where the customers choose several months worth of stories from our estimates with the intention of scheduling a public release or two.

## MVP



## Recommended vinyl



The customer and developer collaborate on the release plan, with the customer driving and the developer navigating. Since customers handle business decisions, they decide the **time** and **scope**. Developers merely provide the **estimates** and let the customer know when there are technical risks afoot.

### For the customer

For the customer, the release plan is excellent because it helps them synchronize the work to be done in other departments, stay ahead of competition, perhaps deliver a minimal first release or two which enables the company to at least start making *some* money, gather early feedback on the product, and get a fundraising story going. There are lots of other business reasons too, but they're out of the scope of this book. Regardless, it's a good idea to know what's going on with the business and their short and long-term goals.

### For developers

For us, release plans are good because **we get a chance to make the customer feel heard**. If we know how much they want a particular feature to be developed as soon as possible, we can work with them to re-arrange stories so that they can get it out there and into the wild. This keeps the money flowing in for us and lets them know we are genuinely trying our best to **consistently deliver the most value we can as soon as possible**.

## Ordering stories

There are two ways to order stories: by **business value** and by **technical risk**. Business value always comes first because the customer drives this process, but there are the cases where we might find ourselves in a pretty deep puddle and nervous about some of the technical risk involved.

For example, if you're building a mobile app that's supposed to remotely connect to and control a slow-cooker but you've never done any IoT work and you've never used any of the vendor's tools or SDKs — well, that's a lot of uncertainty to leave for later, right? Yeah, it is. While the customer may want the first release to merely focus on getting people to sign up, register their devices, and purchase cookbooks, we know that the eventual ability to control the appliance is what is *really important*. And also evidently *really hard*.

**Your responsibility is to merely address the technical risk.** Bring it up with the customer, make sure they know about it, and leave it up to them to decide what they want us to do about it. What can be done? In 18. Estimates & Story Points, we learned about *spiking* stories. That's an XP technique to reduce risk.

A good middle ground here is this: when you're planning out the releases and their accompanying iterations, play some of the *spiked* stories along with the value-bringing functionality stories to ease the burden of having to leave all the risk to be figured out way later on in the future.

## Iteration size

Release plans are composed of a number of iterations. While iterations can be anywhere from 1 to 3 weeks, the phronimos developers generally find that shorter iterations are better. 2 week iterations seem to be the sweet spot for a lot of teams.

The longer the iteration, the more chance you have for things to get off track.

The shorter the iteration, the more overhead and replanning at the beginning and ends of the cycles you have to account for.

## Example release plan

You can store your release plan however you like, though a spreadsheet works best. Here's an example of a release plan showing at least two different releases.

**Example**  
**Release plan**

Story	Release	Iteration
Login (Google)	MVP	1
Make offer	MVP	1
List vinyl	MVP	1
Accept trade	MVP	1
Decline trade	MVP	2
Get offers	MVP	2
Search vinyl	MVP	2
Get recommendations	Recommended	1
Update profile	Recommended	1
...	...	...

And just like we did in 14. Planning with the *initial plan*, for the continuous plan, we'll make sure to save the release dates as well.

**Example**  
**Release plan dates**

Release	Date
MVP	Feb 15th
Recommended vinyl	Apr 30th
Friends and vinyl discussions	Aug 15th
Public timeline (see trades)	Nov 30th
...	...

### Plan stability

Is the release plan stable? Heck no. The only thing that we can be absolutely certain about with respect to the plan is that development will not be the way we want it to.

## **Events that effect the plan**

Every time the customer changes their minds about the requirements and the priority, this changes the plan.

Every time developers learn something new about the story, this changes the plan.

Every time the developers' velocity changes, this changes the plan.

Every time the customer wants to add or remove a story, this changes the plan.

Therefore, it's important to consider the release plan to be something of a *snapshot* of what we're *currently* shooting for. Everyone involved needs to accept the fact that the plan will change. It's as sure as night and day.

## **Regular rebuilds**

It's a good idea to rebuild the plan regularly, just like a tire change or spring cleaning. If you notice that you fail to complete a number of stories at the end of the sprint, coordinate some time to re-estimate them. You may have learned something new that could make the stories take longer or shorter to complete.

Fowler recommends rebuilding the release plan every two to four iterations.

## **Dealing with bugs**

Even though XP helps us drastically cut down on the number of defects in our code, there will still be a few.

First and foremost, Kent Beck and Martin Fowler advise us not to let emotions get the best of us and point fingers — no one can write code with absolutely zero bugs, so treat it as a kind of *code tax*. We should also level set with customers, letting them know that despite our best efforts, bugs are bound to happen eventually.

Secondly, the way we address bugs is not by dropping everything we're currently doing and fixing them. Time spent fixing bugs is time away from developing new functionality. And not *all* bugs need to be fixed right away.

Therefore, it's up to the customer to *explicitly* decide when we address the current defects. When we encounter bugs, we log them, estimate them, and *ideally*, turn them into stories (which we later prove been fixed by first using a failing test and making it pass). With bugs as stories, the customer schedules which iteration we'll spend working on it. Either now or later.

This rule doesn't apply for life-threatening bugs or major security issues. Whenever public safety and security is at risk, I believe we have a duty to the people to remedy those issues as soon as possible.

## **Planning infrastructure**

Since features (which are evidently *stories*) are the entry-point to us actually writing production code, when exactly do we work on the non functional stuff like infrastructure? How

exactly are we supposed to develop stories if there isn't a GraphQL API, a database, or our major components set up to test and deploy it?

The idea is that we develop the infrastructure *while* we're building out the functionality. That way, we only build the infrastructure that is absolutely necessary. We start this process with a *walking skeleton*.

## **Walking the skeleton**

In 26. The Walking Skeleton , we discuss how you can kick off a project by developing a *walking skeleton* — the simplest possible architecture that performs an end-to-end function, runs our tests, and can be deployed.

The walking skeleton is the platform upon which we can then begin to pick up functional requirements, write their failing tests, and then deploy them when they pass.

## **Iteration zero: Zero-functionality iteration**

There's a chance you've never used some of the infrastructure you know you'll need to dive into. Never seen Elasticsearch? Curious about all these AWS services? Wondering how to set up server-side caching with Redis?

If you've never used some of the tools before, the *zero-functionality iteration* is a really good idea.

By devoting the first iteration to getting infrastructural tasks completed on the *walking skeleton*, you set yourself up for the functionality-rich iterations where you can merely focus on the stories. For example, you may need to:

- Develop the testing framework
- Get the automated builds working
- Set up install, testing, and deployment scripts
- Configure Docker containers

A zero-functionality iteration helps avoid the situation where we commit to some functionality for the first iteration but never get around to actually building it because we were so focused on getting things to talk to each other.

So often, developers don't know how to test their code. This is an opportunity to sit down and think about an architecture supports our testing strategy before we get into the weeds with code and testing becomes impossible.

## **Pizza party iteration (or mob programming)**

One last suggestion. Let's say you have your plan ready for the first iteration. Now, depending on how many people are on your project, it's going to be pretty challenging to split out the work into separate parts so that others can begin developing functionality.

In *Planning Extreme Programming*, the phronimos developers wrote about their successes spending a couple of days programming with everyone in the room or looking at the same computer for a while. With one person at the keyboard, they write the first test cases, make them pass, then evolve the design. After a little while, you'll have enough pieces that can

be split into work for other developers to work on independently. If you're familiar with the idea of *Mob Programming*, this is pretty much the same thing.

The basic concept of mob programming is simple: the entire team works as a team together on one task at the time. That is: one team – one (active) keyboard – one screen (projector of course).

— Marcus Hammarberg, *Mob programming – Full Team, Full Throttle*

Great way to get started.

## Summary

- Release planning is an exercise where we work with the customer to help them plan out which stories to implement for the current release and which to implement later based on the story estimates.
- Release planning helps business (marketing, sales, etc) synchronize themselves with development and puts us in a position to make the customer feel heard and build trust should we be able to deliver the most valuable thing each iteration.
- Customers will want to order stories based on business value while developers will want to order them by technical risk so that we can address uncertainty early so that we have time to deal with it. The solution is to make the technical risks known and let the customer decide *or* find a middle ground where they allow us to splice in spiked stories earlier.
- The most common iteration size is two weeks.
- Despite our best efforts, the plan is not and will never be stable. Everyone needs to accept constant change.
- It's recommended to build the infrastructure as we develop the functionality. If you're not familiar with your tools, the phronimos developers recommend a zero-functionality iteration in which you develop a *walking skeleton* — the simplest possible architecture that performs an end-to-end function, runs tests, and can be deployed.

## Exercises

■ Coming soon!

## References

### Books

- Planning Extreme Programming by Kent Beck and Martin Fowler

## 20. Iteration Planning

■ The iteration plan breaks stories down into development tasks that can be divvied up to developers on the team.

Years ago, when I was working on the failed startup company I mentioned in the introduction, I used to work with a developer named Julio. Julio was hardcore. With a love for the command-line and a passionate distaste for GUIs, when I think about it, I don't think we

ever saw him use a mouse. Such a smart guy. He even wrote his own terminal application to browse Facebook.

Because stories are full-stack — back and front — we need a way to play to the individual strengths and weaknesses of the developers on our team. While CSS may have been the bane of Julio's existence, he was practically the God of infrastructure automation, so we knew better than to ask him to code out React components. We also knew that Julio building our entire deployment pipeline and environments on AWS would be like breathing for him. We didn't even have to ask him. He signed up for it.

If release planning is for the customer, iteration planning is for developers. It's how we let our teammates do what they do best.

## Chapter goals

In this short chapter, we're going to learn:

- What iteration plans are and what to do in an iteration planning meeting

## Iteration Planning Meeting

This activity is called the *Iteration Planning Meeting*. It goes like this:

- Spend a few hours understanding the stories
- Break them down into smaller developer-focused tasks that can be signed up for
- Developers choose which tasks they want and estimate them using their own individual velocity
- Ensure we have a way to track our progress throughout the iteration

By the end of it, we'll have some artifact — like a chart, Kanban board, or spreadsheet — that represents who is doing which tasks for which stories and how long we expect they'll take to complete.

Let's briefly discuss each step.

## Understanding a story

We need to understand a story before we can draw out all of the technical tasks for it. For this reason, we'll need the help of a customer or a domain expert. They can either:

- Explain the story
- Put something in writing for us
- Write the acceptance tests for the story

You're going to need the acceptance tests anyway **because a story is not completed until it has passing acceptance tests**, but the phronimos developers agree that it's pretty uncommon to get the customers to have the acceptance tests ready towards the beginning of an iteration, unfortunately. So typically, we do the first or second options.

In the following chapter, 21. Understanding a Story , we'll learn how to understand a story and find the essence of what needs to be built by interviewing a domain expert.

## **Listing the tasks**

### **Technical tasks**

The beauty of the iteration plan is that the tasks don't need to be understood by customers. This is for us. For any one story, there are a number of technical tasks that need to be done. Perhaps experiment with a few different charting JavaScript libraries, refactor some ugly code that needs to change to make the feature work, build the backend use case, create the components, hook up the pages, write some scripts, etc.

Overall, we want to keep these technical tasks small. Preferably, a half, 1 or 2 story points if possible (that's anywhere from half an hour to half a day). Smaller tasks are easier to juggle around and split between each other.

### **Signing up for tasks**

When people can sign up for the tasks within areas that they specialize in or would like to get better at, everyone is more motivated and does better work.

If you're doing pair programming (which if you're doing XP properly, you should be at least sometimes), then it's important to make your estimates based on how long you think it'll take you in a pair.

"This should only take a few hours if Peggy helps me with it".

Regardless, each developer should be monitoring their own velocity using *Yesterday's Weather*.

While some devs may have a velocity of 2 (story points) per day and others may be at 5 or 6, it's okay. The phronimos developers say that it is far more important that everyone provides *accurate* estimates than trying to be overly productive just to push through a large number of tasks. That's not saying productivity isn't good, it's just the advice is that we should place a lot of importance on our ability to predict what we sign up for and execute on it accurately. Keep working on this muscle.

### **Keeping track of the iteration plan**

In order to know if we're going off the tracks or not, someone (or some tool) needs to keep track of the iteration plan. Nowadays, people use all kinds of tools to keep track of the plan. You can use Notion, Jira, Asana, and so on.

A good reason for keeping track of the plan is so that we can let the team know when someone has over or under-committed. If we can catch this early, we can juggle some tasks without having to adjust the overall velocity for the team (and also likely the release plan too). And that's fantastic.

You probably already know this, but the Agile/XP way to keep everyone up to date on how things are going is to sync up regularly. Lots of us are familiar with short stand-up meetings. I've never been a huge fan of stand-up meetings (Robert Martin thinks you should just leave when they're no longer useful), but it should be good to note that there are some pretty good Slack bots and tools that can be used to asynchronously perform stand-up meetings too.

This works particularly well with distributed teams and it's what several engineering teams at Apollo do.

## Example iteration plan

Let's take a look at what an example iteration plan might look like for an iteration in White-Label, the vinyl trading application. Let's assume the following:

- We're a few iterations in (perhaps the second or third one)
- There are three developers on the team (KS — *that's me*, MS, SB)
- MS is most comfortable working on the frontend
- Each developer has a velocity of roughly seven

Here's what that may look like:

### Story: Approve trade

- Create the unit of work object (*generic & reusable*) — KS 2
- Build the approve trade use case (with integration test) — SB 3
- Trades GraphQL API types and resolvers — MS 1
- Trades repository interface — KS 1
- Trade MySQL repository adapter with contract tests — SB 2
- UI for current trade screen — MS 3
- UI for Approve trade (with end to end test) — MS 3
- Trade approved event handler — KS 2

### Story: Offer countered ...

### Story: Offer declined ...

### Other:

- Clean up the UI — KS 2
- Investigate JavaScript charting libraries — SB 2

## Summary

- If the release plan is for the customers, the iteration plan is for developers.
- During an *Iteration Planning Meeting*, developers break the stories for the iteration down into developer-focused tasks, sign up for them, and estimate them based on their own individual velocity.
- Stories may involve a number of front-end, back-end, database, deployment, and scripting tasks — by breaking the stories down into smaller parts, developers can share work, shuffle them around, and specialize in a particular part of the stack.

## Exercises

■ Coming soon!

## Resources

## Articles

- <https://www.scaledagileframework.com/iteration-planning>

## Books

- Planning Extreme Programming by Kent Beck and Martin Fowler

## 21. Understanding a Story

We interview the customer or domain expert to uncover the entirety of the essential complexity for a feature. With pseudocode, we turn our vague understanding into a detailed one that can be broken into smaller parts, understood by the customer, and eventually acceptance tested.

It's true. Every customer is out there to make your life a little bit more miserable by tossing in additional requirements *just as you thought you were done implementing a feature*.

In case you couldn't tell, that was sarcasm. But that doesn't change the fact that this still does seem to happen. "The requirements are always changing!" I can hear the complaining in already.

Here's a perspective-shifting question: what if the requirements that tend to crop up towards the tail end of implementing a feature were actually always there? What if they *never changed*? What if it was just that we never discovered them?

Of course, both scenarios happen. We sometimes miss requirements and they sometimes get tacked on later once the business uncovers new information.

However, at this stage of our feature-development journey, expect your current understanding of a feature to be pretty vague. We should not touch code yet. If we've done Event Modelling or Event Storming, all we know is the command/event pair and the subdomain (or *bounded context* — we discuss this later, as it pertains to how we physically package the application). At this stage, we don't yet know the details. The complete details, if you'll recall from 13. Features (use-cases) are the key, are the data, behavior, and namespace — and we're missing the first two.

Now is the time to uncover as much of the inherent essential complexity that exists as possible. There may be a lot of it. That's okay. Our goal at this step is to merely shine a light on it.

When it comes to complexity: elucidate the essential, avoid the accidental

In this chapter, we'll discuss how to learn and document the essence of a story in a way that the customer can understand and check, that can be almost seamlessly translated to code, and that is void of unnecessary technical details.

After this, we'll feel confident enough to break the story into smaller parts and get to building out the feature.

## Chapter goals

Specifically, in this chapter, we will:

- Learn good principles for interviewing a domain expert

- Learn how to document the essential complexity of a feature with pseudocode

## Interviewing a domain expert

In 20. Iteration Planning, we discussed a few different ways to learn the details of a story.

The most common approach I've seen is the where someone writes down the requirements and hands it to the developers. It's usually a ticket — a story with written details — created in a project management tool like Jira or Asana. In this scenario, the story details are written by someone known to XP practitioners as the *requirements jockey* (be it the project manager, the engineering manager, or some other representative for the customer) and contain a good amount of detail and resources including mockups of the screens and a number of scenarios that need to work.

Regardless of the approach we use, what's important is that we get time to interview the domain expert and ask them questions to clarify. Preferably face-to-face or over a video call. Also remember that the customer/domain expert could be one person but it could also be a number of people (see 15. Customers).

**"But domain experts are busy!"**

Yes, but since stories are such small slices of functionality, interviews can be very quick. It should take anywhere from 15 to 30 minutes, and rarely longer than an hour.

## Rules for a good interview

To conduct a good interview, there are a few rules the DDD community recommends.

- The first is to **listen and ask questions more than you speak**. We're here to learn about the story from the domain expert's point of view because that's ultimately what goes into production.
- The second is to **avoid jumping to technical conclusions**. Rejecting technical conclusions means doing what we talked about in 13. Features (use-cases) are the key, forgetting the fact that you normally start building features and projects database-first, API-first, or class-first. When we think this way, we end up corrupting the domain with unnecessary technical details and make things harder for ourselves in the long-run. Domain-driven design is persistence ignorant, technology ignorant, and as we'll discuss in an upcoming chapter (23. Programming Paradigms), paradigm-ignorant. We'll learn how to document the essence purely.
- The last is to **avoid the use of technical language**. Domain experts don't know what floats or observables are. Remember who you're talking to and try to speak in their language, the ubiquitous one.

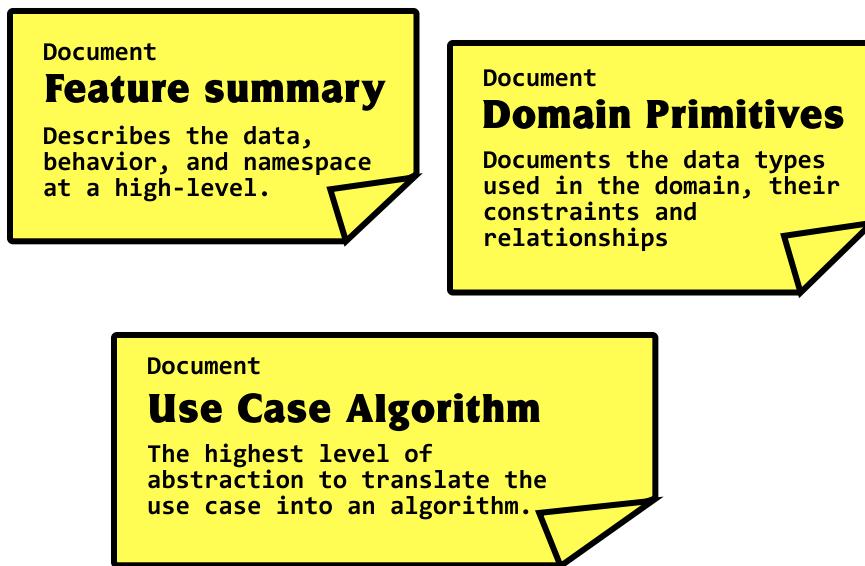
## Documenting with pseudocode

As we interview the domain expert, we'll want to document what we're learning, but not using traditional methods like UML, use-case diagrams and other formal documents that are expensive, hard to change, understand, and work with.

Instead, we'll be using a lightweight text-based language I learned by Scott Wlaschin, one of our phronimos developers. Scott's method is to use one of the first types of code we ever

learned how to write: pseudocode. Not only will this approach result in something that is understandable to the customer, but it'll also force us to recognize previously undiscovered nuances and remedy some of the problem areas that came up in our *Event Storming/Modelling* session.

We'll make use of three pseudocode artifacts: a feature summary, domain primitives, and the use case algorithm.



## Feature Summary

There are three pseudocode artifacts that give us the understanding we need. The first is a high-level *Feature Summary*. It looks like this:

```
# Feature summary
```

```
Feature: <Name of feature>
```

```
Namespace:
```

```
    Subdomain: <Name of subdomain>
```

```
Data:
```

```
    Input data:
```

```
        <All input data>
```

```
Dependencies:
```

```
        <All dependent operations or services>
```

```
Output (events and results):
```

```
    Success:
```

```

        <Success event>

    Failure:
        <All failure results>

Behavior:
Triggered by:
    <Triggering event or the role of the actor who triggers it>

Side-effects:
    <All side effects>

```

If we think of features as functions with inputs and outputs, this tells us everything about the data, behavior, and namespace from the outside.

## Domain Primitives

The second pseudocode doc about the *Domain Primitives*. Here, we document the various domain-specific data types, their constraints, variants, and relationships to each other. To document parts that are required, we can use AND. To specify that something is optional, we can use OR. Simplicity is key here. I also like to describe the constraints using basic English.

Here's an example from DDDForum, the domain used in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS:

```

# Feature Data Types

data Text = string greater than 10 but less than 3000
data Link = a valid web URL which is also not from any domains on the
blacklist
data TextOrLink = Text OR Link
data Upvote = Post AND Member
data Downvote = Post AND Member
data Post =
    TextOrLink AND
    Slug AND
    list of Upvotes AND
    list of Downvotes

```

We'll build the domain primitives up while we listen to the domain expert discuss the concepts.

## Feature Workflow

The third template is the **Feature Workflow Template**. It looks like the following:

```

Feature: <Feature name>
Input: <Any input data>
Output:
    <Output event OR error>

```

```
// Step 1  
// Step 2  
// Step 3
```

This is the highest level of abstraction for the feature, translates roughly Application Service methods or Use Cases (see Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS). It documents the inputs and outputs declaratively describes the solution space by abstracting the complexity to sub-steps.

For example, here's what a sub-step looks like. You can use the same template to model them as well:

```
Substep: <Substep name>  
    Input: <Any input data>  
    Output: <Any output data>  
    Dependencies:  
        <All dependent operations or services>  
  
    // Substep 1  
    // Substep 2  
    // Substep 3
```

There should be no explicit mention of objects, classes, functions or anything technical of the sort. None of that. Just the essential complexity. That's it. The pseudocode here should really resemble what actual code could look like. The fun part is going to be translating the pseudocode into code that looks like it.

Now let's learn how to build this up by interviewing the domain expert.

## Step 1 — Get a high-level understanding of the entire workflow

Let's assume that we're working on White Label again. The use case we'll focus on is the *Make Offer* one. Assume that in the room with us, we've got Nick (store owner) and Veronica (a domain expert and partner at the company very familiar with what it is that we need to build).

### Input and output data

To get started, we just want to get a high-level understanding of how the *Make Offer* use case works from input to output. A great way to kick things off is to ask the domain expert(s) how the use case gets kicked off.

You: So how do you imagine this use case works?

Veronica: Traders should be able to browse the website and find someone's collection of records. As you look through, you can select vinyl that you're interested in and then make an offer.

You jot down some notes.

You: Okay cool, happens next?

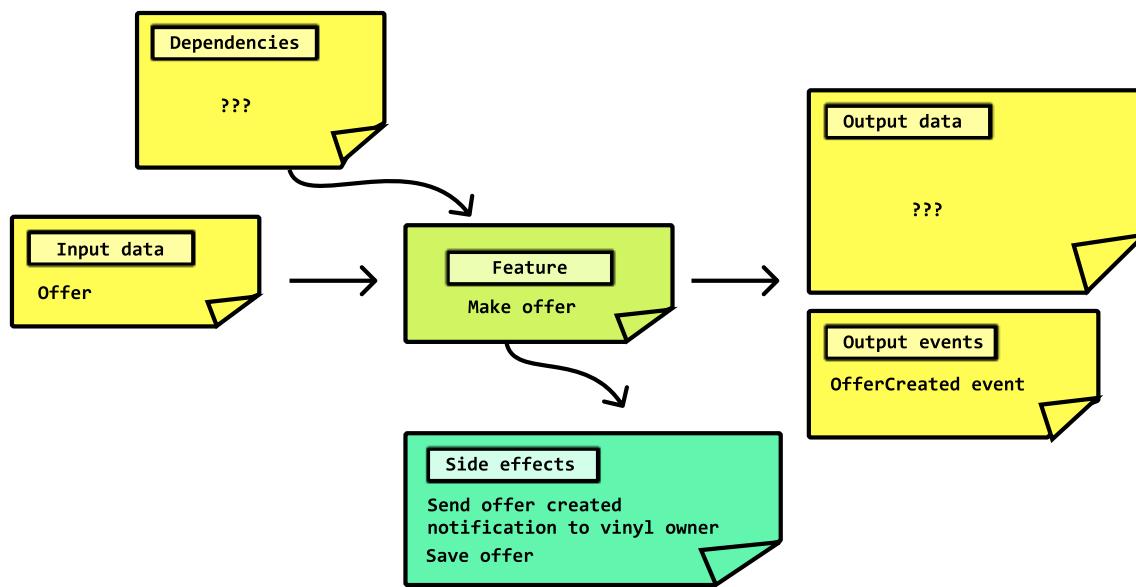
Veronica: Well, after you save an offer, the system should let the vinyl owner know that someone is interested in buying records from their collection. It should also send a notification. From that notification, vinyl owners should be able to log in and accept, decline, or counter-offer their offers.

Well, look at that. What have we learned? If we listened, we'd have written down that the *Offer* is the input data — we don't know exactly what constitutes an *Offer* just yet, but we know that it's the input. That's good. What else? The offer gets created and that should create an event, the *OfferCreated* event. We probably already knew that if we did the *Event Storming/Modelling* workshop, and it's also just easy to recognize that since it's the past-tense version of the use case name.

Most interestingly, there are two important side-effects that take place:

- sending a notification to the vinyl owner
- and saving the offer

If we were to draw this out on a *Make Offer* feature diagram, it might look like this so far:



Taking a look at this, it looks like we're missing the *Dependencies* and the *Output data*. We haven't explored *Dependencies* too much yet, but we already know that *Output data* is about either the success or failure results.

A great way to identify both is to move the conversation towards what could make the feature fail. Let's do that now.

### Success states, failure states and dependencies

You continue the conversation:

You: How could this use case fail? Is there anything that needs to be checked?

Veronica: Oh, well the trader could have insufficient credits.

You: Credits?

Veronica: Yeah, traders get credits for sending in their vinyl to us.

Perhaps you weren't aware of this.

You: How does that work?

Veronica: To make trades on the app, you use credits. When shipping receives a record that someone has mailed in, they send it over to the inventory team to catalog it. The inventory team handles all the data entry stuff but we want it to integrate with the trading application so that it rewards traders with credits that they can use towards trades for other vinyl on the website.

We've just learned about the existence of credits and an Inventory subdomain. Credits seem important to the story here, but we're not exactly sure how the Inventory subdomain comes into play. We'll jot it down for now.

You: So just to make sure I understand, if I want to make a trade, I have to have credits. And the only way for me to get credits is to trade in my vinyl?

Nick: That's one way, but there's another way too. We did some research and had a discussion on this. Just like in real-life, if you're not interested in trading some of your own vinyl, you can trade with money instead. That is, you can actually purchase credits directly from the website.

You: Okay, so if I put in an offer, I have to make sure that I have enough credits first, and if I don't have enough credits, we should let the user know.

Veronica: Yes, but we also decided that we also wanted to allow traders to optionally *automatically purchase* the credits should the owner of the vinyl decide that they want to go through with the trade.

You: You mean when the trade is accepted?

Veronica: Yes, exactly.

You: OK, that's cool. How does that work? Is that like, a setting in your profile, or is that something that you'd handle specifically for every trade?

Veronica: Yes. Definitely something we want traders to be able to set when they're making an offer. Sometimes they'll come across things that they really don't want to miss.

Nick: Yeah, exactly. Here's why we want to do that. Assume you only have 5 credits. Now, you make two offers — and both offers each cost 5 credits. Let's say the first offer gets accepted while you're away, but what about the second one? Well, now you don't have enough credits on to fund the trade.

You: Right, so that'd be an error.

**Right here, we realize that we'll need some sort of dependency that's capable of checking that the trader has sufficient credits to complete the trade.** Otherwise, we'll have to return an error (InsufficientCredits error).

Nick: Yes. That's why we want traders to also have the option to automatically purchase credits if the *Offer* gets accepted. This will rely on having a credit card linked.

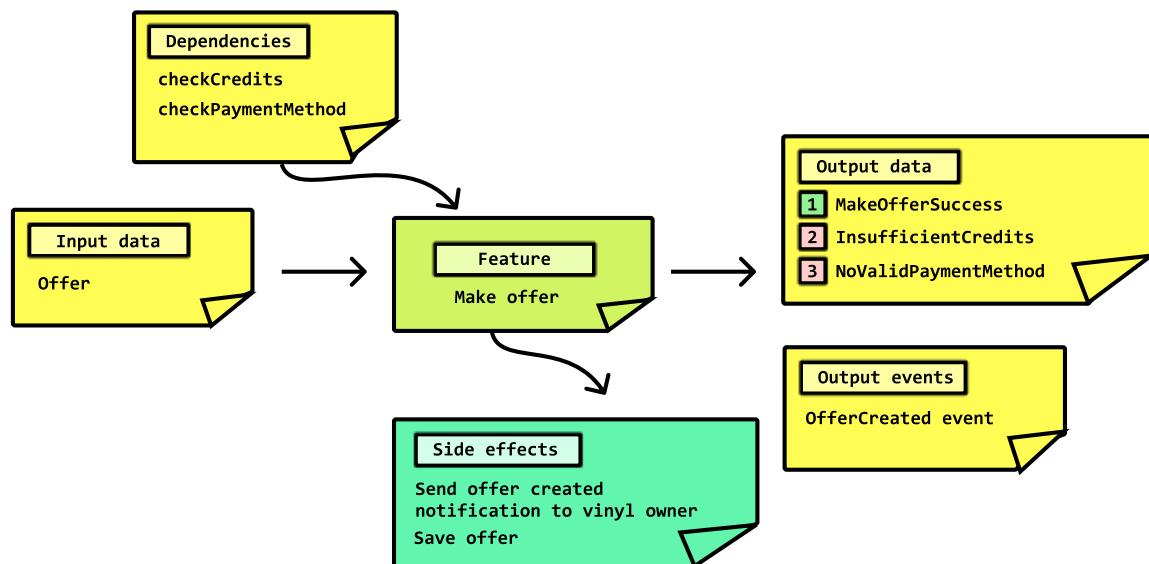
You: And what if you don't have a credit card linked? Or what if there are insufficient funds on your card?

Veronica: Well then the trade will most certainly fail for that reason, but we'd have to let them know what happened via email. Like, "hey, we tried to purchase this item for you but there seems to be a problem with your payment method".

In acceptance tests, we test by example. A good test in this scenario may be one where the trader has zero credits and doesn't have a credit card on file, but selects the option to automatically purchase credits. As we now know, that should fail with something we may call a `NoValidPaymentMethodExists`.

You: Okay, that makes sense. So that's what happens when the offer is accepted. Coming back to the *Make Offer* use case, when you make an *Offer*, it sounds like there are two ways you can place an *Offer*. Either you use credits or you use money (which evidently buys the credits necessary to complete the trade).

Alright, let's take a moment to reflect on what we've learned here. We've identified a couple of dependencies and some of the possible ways that the feature could fail. Here's another look at the updated diagram.



Fantastic. Now let's start documenting what we have so far in our lightweight *Feature Summary*.

## # Feature summary

Feature: "Make offer"

Namespace:

Subdomain: Trading

```

Data:
  Input data:
    An offer

Dependencies:
  Check credits (credits service)
  Check payment method (payments service)
  Check vinyl (inventory service)

Output (events and results):
  Success:
    "Offer placed" event

  Failure:
    Insufficient credits result
    No valid payment method result

Behavior:
  Triggered by:
    "A request made by a trader"

  Side-effects:
    Offer saved
    Email notification sent to vinyl owner

```

## Probe edge cases with hypothetical scenarios

Those two failure states — having insufficient credits one and an invalid payment method scenario — it's very likely that there are more ways for the request to fail than that. We'd like to continue to identify as many of these as possible. At least the semi-obvious ones.

Looking at where we are so far, I might hone into my devil's advocate skills and ask:

- “What happens if I make an offer on multiple records at the same time? Can I do that?”
- “Do they all have to be from the same owner?”
- “How can we make an offer on records owned by multiple owners?”
- “Is it possible for vinyl to be unavailable?”

From here, we figure out that *yes, you can make an offer on multiple records*, we discover that *no, they all have to be from the same vinyl owner*, and that *yes, it is possible for vinyl to be unavailable*.

We learn two more ways the request could fail:

- MixedVinylOwnersInOffer
- VinylNotAvailable

And we utilize some more dependencies to do that:

- CheckVinylOwner
- CheckIfVinylAvailableForTrade

It's not completely likely that we'll get all of the edge cases right away, but as we continue to through the steps and we see what the code could look like in the following steps, it'll inspire more questions and more hypothetical scenarios and edge cases. The funny thing about edge cases is that it's just a matter of time until they rear their head in production. That's why these hypothetical scenarios actually serve as really good *Acceptance Tests* to protect against later on.

## Step 2 — Document the domain primitives (data types)

Once we're feeling good about the different edge cases, we can pause and start modelling some of the domain primitives we've started to learn about. Let's take a look at what we've learned so far.

```
data Offer =
    AND OfferMethod
    AND list of Vinyl

data OfferMethod =
    CreditsForTrade
    OR AutomaticallyPurchaseCredits

data AutomaticallyPurchaseCredits = boolean

data Vinyl = ??           // not sure yet
data CreditsForTrade = ?? // a number of some sort
    (not sure about the constraints yet)
```

Take special notice that we haven't done anything fancy here. No classes. No inheritance. Just an attempt to model the domain in a structured fashion.

There are two great advantages to this approach. The first is that it can easily be checked by domain experts early on so that we get the fundamental relationships correct. The second is that we can make our code look almost just like this using types (see [II. Types](#) ). We'll be doing this to create a domain layer in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS.

## Constraints

As you know from [II. Types](#) , we don't like to use primitive values for domain concepts. An `EmailAddress` is more than just a string. It has a set of constraints and validation rules that must first pass for it to be considered a valid `EmailAddress`. Let's ask some questions.

You: Can we talk about the credits for a moment? Is that what you called them?

Veronica: Yeah, we've been referring to them as credits. Credits for trading.

You: And are those whole numbers or can they also have decimals?

Veronica: Hmm, that probably depends.

You: On what exactly?

Veronica: Well, when we give traders credits for their vinyl that they trade in, we generally give out whole credits because it's easier. But since there's a monetary value assigned to a credit (a credit is roughly 3 dollars USD), it does make sense that there could be some decimal values.

You: And why is that?

Veronica: Because when traders purchase credits, they purchase them in 10, 25, 50 or 100 dollar increments in USD. Accounting for conversion rates and whatnot, the values would certainly get split into decimals.

With this information, we now know that the CreditsForTrade holds a floating point / decimal value.

```
data CreditsForTrade = decimal number
```

You'd also like to know if there is an upper or lower bound on the value for a trade.

You: What's the minimum amount of credits that someone can use in a trade? Is there a max?

Veronica: The minimum amount is 1 and we'll set the max to be 1000. That means the most expensive trade we allow is \$3000 USD.

We make a change to our definition.

```
data CreditsForTrade = decimal number between 1 and 1000
```

## Variants & state machines

We had a pretty basic definition for Offer.

```
data Offer =  
    AND OfferMethod  
    AND list of Vinyl
```

```
data OfferMethod =  
    CreditsForTrade  
    OR AutomaticallyPurchaseCredits
```

This misses some details that are obviously important to the domain expert.

It seems like an *Offer* has variants, or the idea of a lifecycle to it. If we were to think of it as a state machine, the offer starts out invalidated, and then becomes validated. It's only with the validated Offer that we do useful things with it.

We want to capture this somehow. This is the nature of making the implicit, explicit.

Taking from Scott Wlaschin's functional approach (I still recommend his approach to this initial design even if we take an object-oriented approach to implementation), we give each variant a new name; this is so that we model each step in the Offer state machine explicitly.

Therefore, an UnvalidatedOffer can be documented like so:

```

data UnvalidatedOffer =
    AND UnvalidatedOfferMethod
    AND list of UnvalidatedVinyl

data UnvalidatedOfferMethod =
    UnvalidatedCreditsForTrade
    OR AutomaticallyPurchaseCredits

```

Yes, the names are a little bit longer, but it comes with the advantage of removing all ambiguity from the design.

It has now become abundantly clear that at the very start of the use case, the `OfferMethod` and `Vinyl` aren't validated and that is a variant different than the *validated* one.

We can now also document what an `Offer` looks like after it's validated.

```

// Unvalidated
data UnvalidatedOffer =
    AND UnvalidatedOfferMethod
    AND list of UnvalidatedVinyl

data UnvalidatedOfferMethod =
    UnvalidatedCreditsForTrade
    OR AutomaticallyPurchaseCredits

// Validated
data ValidatedOffer =
    AND ValidatedOfferMethod
    AND list of ValidatedVinyl

data ValidatedOfferMethod =
    ValidatedCreditsForTrade
    OR AutomaticallyPurchaseCredits

```

Fantastic. Things are looking a lot more explicit now.

Now, about the notification that gets sent to the customer. Let's model that as well.

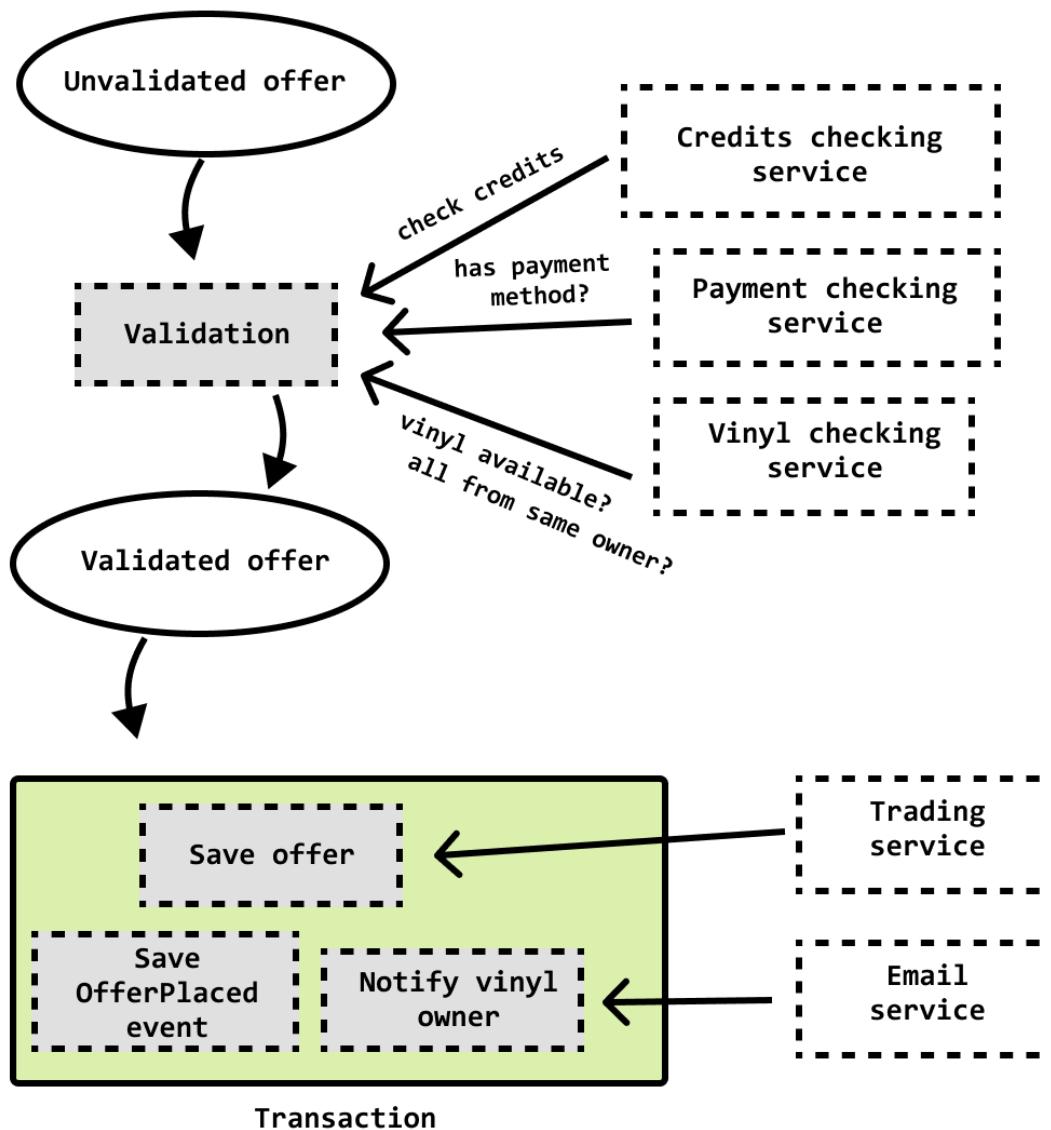
```

data OfferReceivedNotification =
    ValidatedOffer
    AND Recipient
    AND Message

```

If it seems complicated, take a moment to recognize that this is the true nature of the domain. This is the essential complexity. If it feels like we're doing too much here, I urge you to consider that we're not. We're just being explicit about what we would typically let slip through the cracks and into an implicit implementation.

We are merely representing the way the business works. This is the goal of Domain-Driven Design.



### Step 3 — Document the steps

The last piece of documentation we'll do is to get the steps laid out. If we can understand what we need to do, the job of coding is to merely translate this essential complexity as cleanly as the programming paradigm will allow, and to get it hooked up to infrastructure.

#### Use case algorithm: The highest level of abstraction

Let's write a *Use Case Algorithm* which illustrates the highest level of abstraction possible for

this feature.

```
Feature: "Make Offer"
  Input: UnvalidatedOrder
  Output:
    OfferPlaced event
    OR InsufficientCredits
    (when no credits and no auto-purchase credits)
    OR NoValidPaymentMethodOnFile
    (when no credits and auto-purchase is set)
    OR VinylNotAvailable
    (when vinyl no longer available for trade)
    OR MixedVinylOwnersInOffer
    (when more than one owner involved in trade)

    // Step 1
    do ValidateVinyl

    // Step 2
    do ValidateCreditsOrPaymentMethod

    // Step 3
    do CreateOffer

    // Step 4
    do SendNotificationToOwner

    // Step 5
    return OfferCreated event (if no errors exist)
```

Like that? The input of the use case is an `UnvalidatedOrder` and the output is either the `OfferPlaced` event or one of the many possible ways this can fail. Let's model the sub-steps now.

## Sub-steps

There are four sub-steps here for us to document, the first of which is `ValidateVinyl`. It takes in a list of `UnvalidatedVinyl` and returns an output of either `VinylNotAvailable`, `MixedVinylOwnersInOffer` or a list of `ValidatedVinyl`.

It also depends on a trading service for two dependencies:

- to `CheckIfVinylAvailableForTrade`
- and to `CheckVinylOwner`

Here's what that might look like:

```
Substep "ValidateVinyl"
  Input: list of UnvalidatedVinyl
  Output: list of ValidatedVinyl
```

```

    OR VinylNotAvailable
    OR MixedVinylOwnersInOffer
Dependencies: CheckIfVinylAvailableForTrade, CheckVinylOwner

for each UnvalidatedVinyl in list:
    CheckAvailableForTrade
    and CheckVinylOwner

if any vinyl not available for trade, then:
    return VinylNotAvailable

else if list of vinyl owners is more than one:
    return MixedVinylOwnersInOffer

else:
    return list of ValidatedVinyl

```

Straightforward, right?

The next sub-step validates the credits. It follows the exact same pattern.

```

Substep "ValidateCreditsOrPaymentMethod"
    Input: UnvalidatedOfferMethod
    Output: ValidatedOfferMethod
        OR InsufficientCredits
        OR NoValidPaymentMethodOnFile
Dependencies: CheckCredits, CheckValidPaymentMethod

CheckCredits
CheckValidPaymentMethod

if paying with credits only and has credits, then:
    return ValidatedOfferMethod
else:
    return InsufficientCredits

if has automatic payment enabled and has payment method, then:
    return ValidatedOfferMethod
else:
    return NoValidPaymentMethodOnFile

```

We also have the sub-step about creating the offer. This is pretty much a one-liner. It's self-explanatory.

```

Substep "CreateOffer"
    Input: ValidatedOffer
    Output: N/A

```

Create the offer

And lastly, we need to send the notification to the vinyl owner.

```
Substep "SendNotificationToOwner"
    Input: ValidatedOffer
    Output: N/A
    Dependencies: GetVinylOwnerDetails
```

GetVinylOwnerDetails

```
Use vinyl owner details to CreateAndScheduleEmail
```

## Summary

- Before we can decompose a feature and plan work on it, we need to understand its data, behavior, and namespace in detail. We do this by conducting a series of short interviews with domain experts.
- Pretend that you're an anthropologist and that you're there to learn the language, listen, ask questions, and capture the details of the use case from the domain expert's point of view.
- Remember, to deal with complexity: elucidate the essential, avoid the accidental. Use pseudocode to expose the essential complexity. Reject thinking about the solution with databases, classes, APIs, or any other technical details too early on in the process.
- Kick off the interview by asking about the input and output data.
- Document domain primitives, their constraints, and variants in the simplest possible way using AND and OR.
- Uncover all essential behavior using pseudocode and break it down into concrete steps.
- The main benefit of this approach is a complete, undistorted view of the application and business logic that is checkable by a domain expert. Now that we know the services, data, and non-functional requirements necessary for the feature, we can break the problem into smaller parts during the Iteration Planning Meeting.

## Exercises

■ Coming soon!

## References

### Articles

- Alistair Cockburn's "Use Case Template" — [http://cis.bentley.edu/lwaguespack/CS360\\_Site/Downloads/UseCaseTemplate.pdf](http://cis.bentley.edu/lwaguespack/CS360_Site/Downloads/UseCaseTemplate.pdf) (Cockburn).pdf

### Books

- Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F# by Scott Wlaschin
- Writing Effective Use Cases by Alistair Cockburn

## 22. Acceptance Tests

■ Acceptance tests are the most high-level, high-value, user-story-focused tests that we can write. They are also a contractual agreement between the customer and the developer for what constitutes a complete user story. Acceptance tests, when automated and passing in a production-like environment, are the definition of done.

In 21. Understanding a Story , we practically sketched out the entire algorithm for a use case. While that's helpful, and hopefully there are a lot less gaps in our understanding in how we'll realize it in actual code, we don't want to dive into code just yet.

What makes software development more of a trade and less of an art, academic, or mathematical experiment is that we can be intentional about our work. I don't like to do meaningless work, doing rework, or missing requirements. Our intention is to deliver a unit of functionality that is value to the customer. We should completely elucidate what that unit of functionality is instead of merely guessing and finding out that there's more to do or that we've missed something later.

Evidently, what we want here is a **contract for work**. A **definition of done**. A **picture of what the completeness for a user story looks like**. Without that, we can implement something to our best judgements, miss a lot of edge cases, requirements, but we'd likely be coding forever — or at least until we feel like we've done a good enough job. That's not good enough for me. I've done this plenty of times in the past and had clients come back and tell me that something wasn't working as intended.

In this chapter, we'll discuss *Acceptance Testing*: a practice for turning requirements into a code-based contract — one that gives meaning and value to every subsequent line of code that we write for a feature.

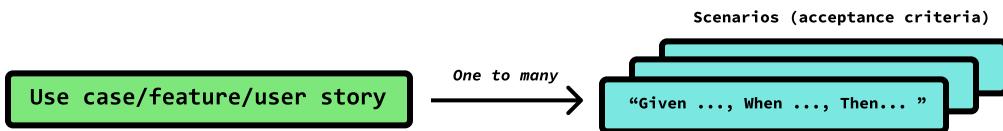
### Chapter goals

- Understand the goal of acceptance testing, the many benefits they bring, and why they're essential in an XP project
- Understand what acceptance tests have to do with BDD and how to write them in Given-When-Then style
- Understand the division of responsibilities for getting acceptance tests written and automated

### What is acceptance testing?

*Acceptance Tests* are a formalized way to describe the behavior of a story. They are how we:

- get the customer to commit to and agree upon the success criteria for a feature
- mitigate arguments like “you said the app was going to do X, Y, Z — we talked about this”
- catch features when they regress
- and prove with *certainty* that we've done what we agreed to doing



## A way to contractualize and exercise the success and failure scenarios

The goal is to take the user story and extrapolate scenarios out of it.

Remember how every feature has typically *one way* it will succeed and numerous ways that it can fail? Here's where we want to create a contract for the fact that CreateUser could pass if we successfully pass in a valid email and password but it could also fail due to an AccountAlreadyExists or an InvalidEmailOrPassword error as well.

*Acceptance Tests* are fluent, expressive, pretty close-to-English-style tests that tell us how the application should act in certain scenarios. The intended result for every scenario can be expressed in three parts:

- Preconditions
- Actions
- Postconditions

## Understanding (and appreciating) acceptance tests

### The road to acceptance tests (late 90s until now)

The road to acceptance testing wasn't always so simple. XP has always mandated that we use them to kick off the development of a feature, but actually getting them written was something of a challenge in the late 90s and early 2000s. It comes down to **understanding the responsibilities, tooling, ubiquity, and testing best practices**.

### A division of responsibilities

First off, it wasn't made readily apparent how acceptance tests actually came to fruition.

XP said that the *customer* writes the tests, but it didn't catch on that *writing the tests* and *automating them* are two entirely different tasks done by two entirely different roles. The customer writes them and the developer automates them.

### Early acceptance test tooling

Without a clean division of responsibilities, on came a slew of tools and formal languages and tools devoted specifically to **writing and automating acceptance tests**.

We had tools like FitNesse, SpecFlow, Cucumber — each with their own specific type of *formal language* that was expected to make it easier for teams to get acceptance tests written and automated.

Expectedly, many early 2000s developers found it hard to get customers to sit down and learn a new formal language, let alone write a whole suite of specifications with it. Some developers advocated *hard* for these tools, but like the least delicious bowl of chips at a party, they sat relatively untouched by customers if possible.

Think about it. How would you feel if you took your car into the shop and you had to read docs or take a class to learn how to place a work order for what you want. That's silly, isn't it? Just bad UX.

Customers want to describe the feature that they want in the most natural way. For most human beings, that'd be to use their native language. And for the vast amount of North American engineering tools, we'll assume that language is English.

### Failure to gain ubiquity

Because most of these tools tried to do *both* functions at the same time — writing the tests and automating them as well — and because the tools had technical formalities to how they'd work, it was hard for customers to learn how to express requirements naturally.

Instead, like we discussed in 20. Iteration Planning, it was just much easier for customers to simply talk about what they wanted, or to maybe put something in writing (although maybe not a formal structure). *Formal acceptance testing* never gained the widespread ubiquity that was originally imagined in XP.

### The need for TDD best practices

In the early 2000s, things started to happen in the TDD space which inadvertently changed the way we thought about *Acceptance Testing* as well.

Dan North, another one of our phronimos developers and a test-driven development practitioner, was struggling with figuring out how to teach programmers:

- What to test
- What not to test
- How much to test in a single go
- What to call your tests
- and how to understand why a test fails

Developers knew that you should start by writing a failing test, followed by making it compile, and then making it pass before refactoring it into something pretty. But the principles around what it meant to write good tests was something of a nebulous concept. Those who knew good test-writing principles were developers who had been lucky to absorb the tribal knowledge as it passed around in bootcamps and through trial and error. Without a ubiquitous understanding of *good TDD*, there was a lot of confusion throughout the industry.

To remedy this situation, Dan spent time practicing and researching a more structured way to implement and teach TDD — a way where commonly established agile practices (like *Acceptance Tests*) were baked in. Dan emerged from this era of *confusion-driven-development* with something called BDD: *Behavior-Driven Development*.

## BDD (behavior-driven development)

In 2006, Dan announced BDD (or, *TDD done right*) to the world.

BDD is a way to think about writing tests (*Unit, Integration, E2E*, whatever) that focus on the behavior, not the implementation details.

For example, a test like this would not be considered good in a BDD sense.

```
describe('calendar', () => {
  test ('save data into an array', () => {
    ...
  })
})
```

Why? Because *data* and *arrays* are technical concerns. That means nothing with respect to the domain. It also doesn't even *read* well. "Test save data into an array" isn't really a proper sentence.

Instead, BDD proposed that tests come from a place of speaking with respect to the domain, the business, and the observable behavior rather than the technical implementation details. Tests are not concerned at all about *how* behavior is realized. They are only concerned with expressing *what* should be realized. Notice the difference:

```
describe('calendar', () => {
  it ('contains all of the days for the month', () => {
    ...
  })
})
```

That's what I'm talking about. Some people like to think of BDD as being the amalgamation of DDD + TDD.

In Dan's proclamation to the programming world, BDD specified a few very important test-writing principles. It said:

1. Test names should be sentences
2. We should prefer a simple sentence format for tests to keep them focused
3. Expressive names help when tests fail
4. "Behavior" is a more useful word than a "test"
5. Prefer testing "behaviour" instead of the implementation
6. Evolve a design by probing for the next important behavior
7. Acceptance criteria should be executable

Now, all of these statements have gone on to help us figure out how to write better *Unit, Integration* and *E2E Tests* as well (and we explore them in detail in Part X: Advanced Test-Driven Development).

It's important to note that focusing on the behavior and being domain-driven can help us improve the quality of tests that we write, regardless of the scope of the test. Yes, even when we're writing *Unit Tests* against very technical concerns like a *StringUtil* helper class.

```

describe('textUtil', () => {
  describe('capitalization', () => {
    it('can capitalize text', () => { ... });
    it('capitalizes text with numbers and non-alphanumeric characters', () => { ... });

    // Continue to triangulate ...
  });

  describe(' ', () => {
    ...
  });
}
)

```

Dan himself says that BDD is like “using examples to talk through how an application behaves... And having conversations about those examples”.

■ **Triangulation (specification by example):** There’s a technique we learn about in Part IV: Test-Driven Development Basics called *Triangulation*. By providing more and more examples of various ways our class, method, or function should behave in specific scenarios, we improve the robustness of our code under test.

This is wonderful and all, but let’s discuss that last principle (#7), the notion that acceptance criteria should be executable — because well, that’s what we had been trying to do for quite some time. It’s great that we got better testing principles for the other types of tests, but what about writing *Acceptance Tests*? How did BDD help us with our *Acceptance Testing* conundrum?

### Given-When-Then (example)

The idea was to identify a “ubiquitous language” for analysis. A great starting point was what we often use to describe the business value of stories — the *As a, I Want, So That* format.

- **As a**[X — person or role benefiting from the feature]
- **I want**[Y — the feature]
- **So that**[Z — benefit or value of the feature]

This is always important to know, even if it’s not written down. But writing it down *does* help to single in on the valuable work to be done.

Dan and his team shifted to thinking about how to describe the acceptance criteria that would be necessary to realize the value of a story. They came up with the idea of *scenarios*. Each scenario starts in the following form:

```

Given some initial context (all the givens)
When an event occurs
Then ensure some outcomes

```

Let’s use a real-life example.

I use Notion to record all of the tasks that I need to complete for the week, but I also use Google Calendar to get event reminders and notifications. Here’s a small snippet of what

Monday and Tuesday look like.

Monday (13th)	12	...	+	Tuesday (14th)	5	...	+
Intention Setting				Meeting w/ CIBC			
<input checked="" type="checkbox"/> Done				<input type="checkbox"/> Done			
09:45 - 10:00				10:30 - 11:30			
Send September Invoice				Meeting w/ Kurt			
<input checked="" type="checkbox"/> Done				<input type="checkbox"/> Done			
10:00 - 10:15				11:30 - 12:00			
Pay my credit card bill down				All Hands Meeting			
<input checked="" type="checkbox"/> Done				<input type="checkbox"/> Done			
10:15 - 10:20				12:30 - 13:15			
Meditation: Observing the Phronimos & Ways of Being				Fix truncated TypeScript chapter?			
<input checked="" type="checkbox"/> Done				<input type="checkbox"/> Done			
10:20 - 10:30				1 - Estimated (> 2hr)			
10 Minute HIIT + 5 Minute Chest + Water + Coffee + Protein + Shower + Dressed				Read through the entirety of the book and make fixes/changes, especially to the phronesis chapter			
<input checked="" type="checkbox"/> Done				<input type="checkbox"/> Done			
10:30 - 11:15							

As a Notion user, I wanted to sync my tasks to my Google Calendar so that I could be notified when events take place and do all of my scheduling and task management from one place.

Using the Given-When-Then format, I can express the first scenario — when there are new tasks to sync — in this format.

Feature: Sync Notion tasks to Google Calendar

Scenario: Syncing new tasks

Given there are tasks in my tasks database  
And they don't exist in my calendar  
When I sync my tasks database to my calendar  
Then I should see them in my calendar

Notice the use of “and” to connect multiple givens or multiple outcomes in a natural way? This reads like English. It’s not just a better way to write tests. It’s a way that *developers* and *business people* can collaborate on a specification using language that’s about as close to the domain and in English as possible.

I can complete this feature specification by writing the rest of the scenarios:

Feature: Sync Notion tasks to Google Calendar

Scenario: Syncing new tasks

Given there are tasks in my tasks database  
And they don't exist in my calendar  
When I sync my tasks database to my calendar  
Then I should see them in my calendar

Scenario: Updating tasks

Given tasks from my tasks database already exist in my calendar  
And there are some changes to the tasks in my tasks database  
When I sync my tasks database to my calendar  
Then I should see the updated tasks in my calendar

Scenario: Deleting tasks

Given tasks from my tasks database already exist in my calendar  
And I have deleted them from my tasks database  
When I sync my tasks database to my calendar  
Then I should not see the tasks in my calendar

Scenario: Preventing duplicates

Given tasks from my tasks database already exist in my calendar  
When I sync my tasks database to my calendar  
Then I should not see duplicate tasks created in my calendar

## Writing acceptance tests

It’s great that we can write these tests in Given-When-Then format (also known as Gherkin) because that’s something *customers* should be able to do. But that’s only *half* the battle. How do *developers* turn these specs into executable code? We haven’t seen that yet.

Let’s talk about the tools first.

## Tooling

Regardless of which language you’re using, there are numerous libraries out there to give your test runner Given-When-Then style test support.

We also have to decide on the scope at which we’d like to write our *Acceptance Tests* (see 25. Testing Strategies). Regardless, in the JavaScript ecosystem, I like to use the following tools to write my tests:

- Jest — I default to Jest for *Unit* and *Integration* tests. If you choose to write your *Acceptance Tests* as coarse-grained *Unit Tests*, Jest works very well. I prefer this approach and devote later chapters to demonstrating how to do this. There's great TypeScript support with ts-jest and we can add Given-When-Then support with jest-cucumber.
- Cypress.io — I use Cypress for *E2E Tests*. If you want to write your *Acceptance Tests* at the *E2E* level, there . We can add Given-When-Then support with cypress-cucumber-preprocessor.
- Library or framework specific tools — React-Testing Library for example, helps you write slightly more declarative tests. It provides an API that promotes a focus on behavior instead of implementation details.

■ **Real-world example:** DDD-Forum, the application we build in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, will contain examples of real-world *Acceptance Tests* using the strategy described in 25. Testing Strategies.

### Format #1: Single-line tests

Most test libraries use single-line tests to demonstrate how their tools should be used. I'm sure you've seen tests like this:

```
describe ('login', () => {
  it(`takes me to the dashboard page after I login`, () => {
    ...
  })
})
```

These types of tests — the one-liners — they're OK when we're evolving a design with *Unit* or *Integration Tests*, but I find that they're not adequate for setting up a scenario to describe the state of the world when we're automating *Acceptance Tests*. For *Acceptance Tests*, one-liners are a clumsy way to express the specification.

Notice that this test doesn't say anything about the *Givens*? It is very easy to write bugs if we're not explicit about how we set up the state of the world and what the postconditions are.

### Format #2: Given-When-Then style Jest tests

This is probably the most awkward way to write *Acceptance Tests*, but it works in essence. Check it out.

```
describe('Feature: Login', () => {
  describe("Scenario: Successful login", () => {
    describe("Given I've previously created an account", () => {
      describe("When I attempt to login", () => {
        test("I should login and see the dashboard page", () => {
          ...
        })
      });
    })
  })
})
```

```

    describe("Scenario: Account doesn't exist", () => {
      ... // Continue
    })
})

```

Ah, that's quite better. We actually express the Given, When, and Then parts of the scenario under test. It would be better, however, if we could use the formal specification instead of having to encode it inside of strings in Jest.

### Format #3: Given-When-Then feature tests (preferred)

This is, in my opinion, the best way to write your *Acceptance Tests*. Assuming we're writing our *Acceptance Tests* as coarse-grained *Unit Tests* (more on this later), we can use Ben Compton's [`@jest-cucumber`](<https://github.com/bencompton/jest-cucumber>) library to write Jest tests that consume our Gherkin feature files and validate them against our actual test implementation.

Using the *Notion to Google Calendar Sync* example from before, that means for a feature file (test specification) like this:

```
# useCases/syncTasksToCalendar/SyncTasksToCalendar.feature
```

Feature: Sync Notion tasks to Google Calendar

```

Scenario: Sync new tasks
  Given there are tasks in my tasks database
  And they don't exist in my calendar
  When I sync my tasks database to my calendar
  Then I should see them in my calendar

```

You could expect to see a valid test implementation like as follows:

```

// useCases/syncTasksToCalendar/SyncTasksToCalendar.ts

import { defineFeature, loadFeature } from 'jest-cucumber';
import path from 'path'
import { SyncTasksToCalendar } from './SyncTasksToCalendar'
import { TasksDatabaseBuilder, TasksDatabaseSpy } from '../../../../../testUtils/tasksDatabaseBuilder'
import faker from 'faker'
import { SyncServiceSpy } from '../../../../../testUtils/syncServiceSpy'
import { ICalendarRepo } from '../../../../../services/calendar/calendarRepo';
import { CalendarRepoBuilder } from '../../../../../testUtils/calendarRepoBuilder'
import { FakeClock } from '../../../../../testUtils/fakeClock'
import { DateUtil } from '../../../../../shared/utils/DateUtil';

const feature = loadFeature(path.join(__dirname, './SyncTasksToCalendar.feature'));

defineFeature(feature, test => {
  let calendarRepo: ICalendarRepo;

```

```

let tasksDatabaseSpy: TasksDatabaseSpy;
let syncTasksToCalendar: SyncTasksToCalendar;
let syncServiceSpy: SyncServiceSpy;
let fakeClock = new FakeClock(DateUtil.createDate(2021, 8, 11));

beforeEach(() => {
  syncServiceSpy = new SyncServiceSpy();
})

test('Sync new tasks', ({ given, and, when, then }) => {

  given('there are tasks in my tasks database', () => {
    tasksDatabaseSpy = new TasksDatabaseBuilder(faker, fakeClock)
      .withAFullWeekOfTasks()
      .build();
  });

  and('they don\'t exist in my calendar', () => {
    calendarRepo = new CalendarRepoBuilder(faker)
      .withEmptyCalendar()
      .build();
  });

  // Act
  when('I sync my tasks database to my calendar', async () => {
    syncTasksToCalendar = new SyncTasksToCalendar(
      tasksDatabaseSpy, calendarRepo, syncServiceSpy
    )
    await syncTasksToCalendar.execute();
  });

  // Assert
  then('I should see them in my calendar', () => {
    expect(syncServiceSpy.getSyncPlan().creates.length)
      .toEqual(tasksDatabaseSpy.countTasks())
    expect(syncServiceSpy.getSyncPlan().updates.length).toEqual(0);
    expect(syncServiceSpy.getSyncPlan().deletes.length).toEqual(0);
  });
});
});

```

The great thing about this package is that it automatically spits out the skeleton of *Acceptance Test* code for you. All you have to do is write the logic within it.

Also, never mind the specific nuances behind how this test is actually realized. For now, don't worry about the concepts like *Spies*, *Builders*, *Fakers*, a *fake clock* (it's actually very interesting why you'd need this for testing) and so on. None of this important at the moment.

These are necessary nuances when we get into writing tests for code that mixes infrastructure and pure, plain ol' TypeScript or JavaScript which we like to refer to as *Core code*.

Just notice the fact that it's possible to use this Given-When-Then format to guide the design and implementation of a feature. In fact, that's exactly how this was developed. I first wrote the feature specification — not entirely sure I'd write the code to ensure things work — but used the test to guide me along.

■ **How can you write an Acceptance Test at the Unit Test scope?** I understand that to some developers, the notion of a *Unit Test* means “testing one class, and one class only”. That makes this particular example somewhat incendiary because it's clear that there are other objects involved as well. For this to make sense, we have to discuss the idea of a *Subject Under Test* (in Part IV: Test-Driven Development Basics), the correct conceptual model behind OO (23. Programming Paradigms), and the difference between *Core vs. Infrastructure Code* as well as Michael Feathers' definition of a *Unit Test*, which I believe is more helpful than the original definition used by Kent Beck (24. An Object-Oriented Architecture). Lots to discuss. It's no wonder testing is so hard.

## Frequently asked questions

### Who does what again?

Alright, so clear things up: the customer writes the acceptance tests and the developer automates it. That means there are two separate tasks to be done by two separate parties.

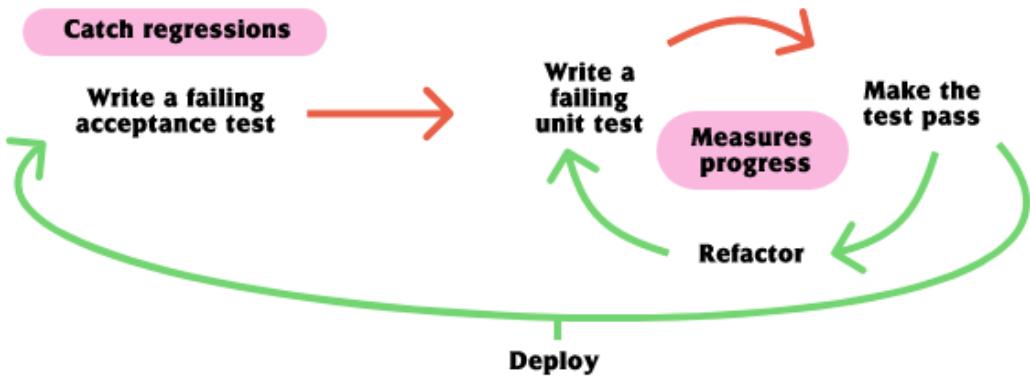
Ideally, you want to get the *Acceptance Tests* before you start building out a feature. In *Planning Extreme Programming*, Fowler recommends that the customer finish writing the *Acceptance Tests* during the first half of the sprint. But often, that's not what happens. So developers may start by writing their own *Acceptance Tests* based on what we learned through conversation or an interview with the domain expert or customer, but eventually, we'll need to make sure that the customer signs off on our tests.

It is 100x better if the customer is the one who writes the tests. We can help them get the hang of the Given-When-Then format, but it should be much easier than using rigid formal languages or esoteric testing tools.

### Cadence

When should we run *Acceptance Tests* tests? Again, it depends on how you write your tests. If you write them as *Unit Tests*, then you'll get all the benefits of *Unit Tests* like having them execute very quick.

In 13. Features (use-cases) are the key, we briefly talked about the idea of *Double Loop TDD* — having an inner and outer test loop. I like having a terminal window open which watches my code and re-runs every time I save it so that I can immediately know if I've broken an *Acceptance Test* (outer loop) and when all my regular *Unit Tests* (inner loop) pass.



## Double loop TDD

If you choose to write them as *Integration* or *E2E* style tests, they'll take a little bit longer to run, they involve potentially more setup (and teardown), and in some cases — there will be things we can't actually validate at this scope. However, it's a great idea to run these tests periodically and ideally, within the *Continuous Integration Server* when we push new code into the source control.

We'll continue the discussion of *Double Loop TDD* and considerations to keep in mind in 28. Test-Driven Development Workflow.

## Summary

- Acceptance tests are a way to create a contract for a feature. They confirm the necessary behavior that needs to work for us to consider the feature as completed.
  - Acceptance tests are written by customers in a loosely formal language known as Gherkin in Given-When-Then style, and can be automated by developers using a number of tools and libraries.
  - Acceptance tests are the outer-loop and unit tests are the inner-loop. This is called *Double Loop TDD* or even sometimes *Acceptance-Test-Driven Development*.

## Exercises

■ Coming soon

## References

Articles

- <https://proxy.c2.com/cgi/wiki?AcceptanceTest>

- <https://capiro.co.uk/writes-user-acceptance-tests/>
- <https://dannorth.net/introducing-bdd/>

## 23. Programming Paradigms

■ All programming paradigms have the same thing in common: they deal with sequence, selection, iteration, and indirection — just in different ways. By building correct conceptual models around structured, object-oriented, and functional programming, we make less big mistakes and learn discover that we can use the best techniques from each, regardless of the paradigm we decide on.

Let's say you decided to buy a fancy stool from a furniture store, but you have to take it home and assemble it. All good — you know how to assemble stuff.

You open up the box, take out the pieces, then pick up the manual and read it a little bit. After following the first few steps and noticing that things seem to be going pretty well, you begin to feel good about yourself. *It's coming together. I'm doing it. Niiiiice.*

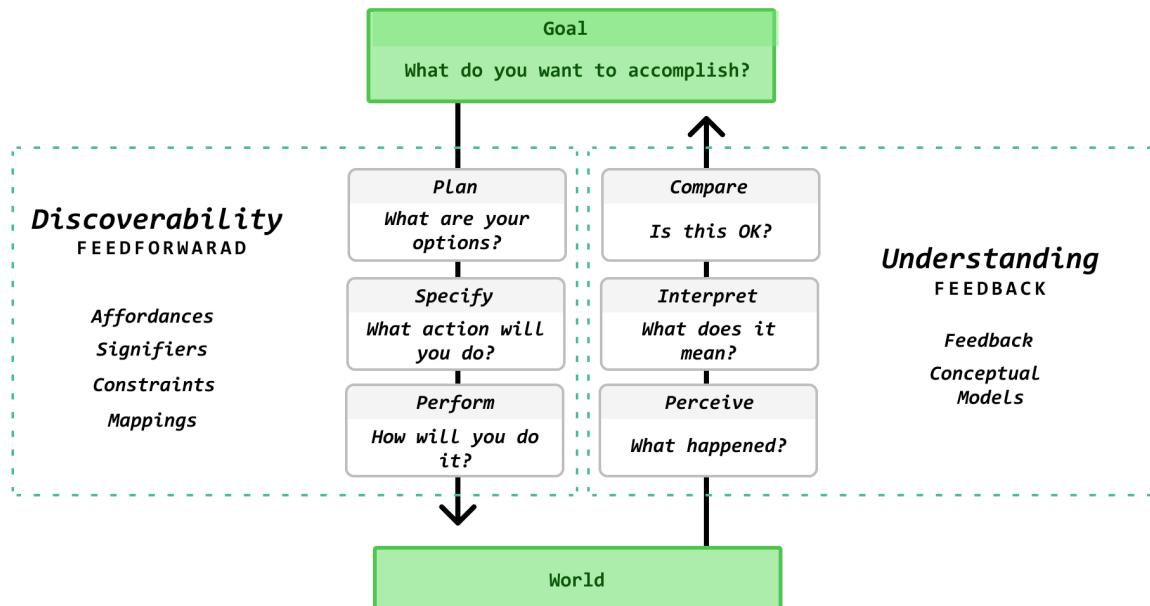
Because you can see where you're going with the design, you decide that reading the manual is superfluous. You put it down. Instead, you decide to use your intuition to figure out how to assemble the rest.

Fast forward thirty minutes later, the legs look a *little awkward*, you're missing two phillips head screws, and you notice you have too many 10mm flathead screws that you're not sure what to have done with. Reality kicks in: you've gone astray.

"Man, this stool (or [insert any programming paradigm, language, or library here]) is trash!"

It's almost silly to need to say this, but if we don't read the manual, it's just a matter of time before we make mistakes. And when it comes to writing code in a particular programming paradigm, the *manual* is comprised of **the main ideas behind it and how it was intended to be used**. To avoid disaster and misuse, **we need to create the correct conceptual model** for our paradigm of choice.

We learned about conceptual models in Part II: Humans & Code. We discovered that the way that we learn how to use something effectively isn't to understand every intricate detail. It's merely to build a conceptual model — a *good enough*, rough model of the tool — so that we know how to use it to accomplish our goals. Remember *discoverability* and *understanding*? Feedforward and feedback?



A reminder: this is what your brain needs to create a conceptual model.

Most of us know that turning the steering wheel in a car will force the car to go left and right, but a lesser number of are aware of the Ackermann steering geometry that makes it possible. That doesn't matter. I've been driving cars for years, and I only just now Google-d what that was (listen, I don't know anything about cars — I just try to keep the wheels on the pavement).

When a lot of us are learning a new programming language or paradigm, our initial conceptual model is created by our first feeling of "Yep, that did what I wanted it to do — so that's the way we do things. Got it."

What this also means is that if we were able to get by for a long time, successfully implementing what we needed, but also inadvertently leaving behind *code smells* (see Part V: Object-Oriented Design With Tests) like God classes, unmaintainable class hierarchies, and feature envy, **then the conceptual model we have built up in our minds is a flawed one.**

Just because the tool *lets you do something* doesn't mean you *should use it to do that* thing (like forcing a large bolt into a smaller socket or driving a car on the sidewalk).

The reality is: because there are so many different constructs in OO, there are many ways for us to do inadvertently do the wrong thing. This is what the idea of *constraints* in 5. Human-Centered Design For Developers was about. When there are too many options, it's more likely we'll make mistakes. By limiting the total number of possible things that can be done, we reduce that possibility.

The notion of less options is certainly something that makes functional programming look appealing, doesn't it? If everything is just *functions*, doesn't that make things easier? I tend

to hear comments like:

“Object-oriented programming is bad. It’s just fundamentally flawed.”

“Functional programming is much better.”

“In JavaScript, classes compile down to functions anyway.” — *ugh*

Well, merely using *functions* does not mean you’re doing functional programming. If we do nothing other than use functions instead of classes but still continue to use constructs like if statements and while loops, we’re actually a lot closer to another paradigm called *structural programming* than we are to *purely* functional programming.

Understanding your paradigm is important. Like the stool, it’s just a matter of time before you look up to realize you’ve got the pieces on wrong, and you need to take it apart and start all over again. Nobody wants to do that.

In this chapter, I’m going to give you a crash course on the main ideas behind the three dominant programming paradigms.

Why are we doing this *now*?

In our journey of learning techniques used by the phronimos developers, we’re practically ready to start coding. We understand the story, we have some notion of how we’ll test it, and it’s almost time to turn it into executable code.

Whether we use OO or FP doesn’t matter. All that matters is that we build using the paradigm in the way it was intended to be used and we understand the implications of stepping outside of the box.

## Chapter goals

Specifically, in this chapter, we’ll:

- Learn about and build conceptual models for the three dominant programming paradigms (structured, functional, object-oriented)
- Learn about the four basic programming constructs upon which all software is built
- Learn how the main paradigms impose constraints on the four aspects of computer code
- Understand why the most important design principles are paradigm-agnostic
- Learn what makes each programming paradigm uniquely special, how you can incorporate the best parts of each, and how to choose the best one for your team

## Structured programming

Software as we know it today starts with a Dutch programmer named Edsger Wybe Dijkstra.

Dijkstra was one of the first *official* “programmers”. There’s air quotes on the word “programmer” because back then, there was no such thing as a programmer, so when he took a job at the Mathematic Center of Amsterdam, he had to settle on the title of a “theoretical physicist” at the time.

Dijkstra decided he wanted to make a career out of this programming thing, and he made some of the most important discoveries — discoveries that kick-started the science and shaped it into what we know to be foundational today.

In 1968, Dijkstra discovered structured programming: the first of all paradigms to be widely adopted. Here's how it happened.

## Mathematical proofs

Realizing how challenging programming was and the fact that bugs seemed to happen a lot, Dijkstra realized that it was just not humanly possible to maintain all of the details of a program in one's head. Nor was it ideal to have to read through every line of code to confirm that things worked as intended. He needed a way to prove that his code was correct.

So he came up with the idea to apply mathematic proofs, the same ones that mathematicians use, to see if his simple algorithms could be proven correct using the same techniques.

While doing this, he realized that some statements, notably the GOTO statement, prevented modules of code from being decomposed into smaller units.

If you've never seen a GOTO statement, it's a statement that allows unconstrained access to a label in the code. If you've done x86 programming, it's pretty much the *jmp* (JMP) statement.

Here's an example I found on the internet:

```
#include <stdio.h>

int main()

{
    int sum = 0;
    for (int i = 0; i <= 10; i++) {
        sum = sum + i;
        if (i == 5) {
            goto addition; // Exit the loop and go directly to the
                           // addition label
        }
    }

    addition:           // You'll end up here
    printf("%d", sum);

    return 0;
}
```

While this seems relatively harmless at first (and maybe even kind of helpful), GOTO statements were the sole reason it was *impossible* to divide code up into smaller pieces to solve. Uh, that's kinda important, isn't it? Yes.

Evidently, Dijkstra proved that code *without* GOTO statements *was* capable of being broken up into smaller pieces and proven.

Furthermore, he also identified the minimum necessary set of control structures for program flow: **sequence**, **selection** (if/else/then) \*\*\*\*and **iteration** (while, for).

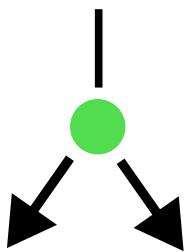
This was a tremendous discovery.

# Flow

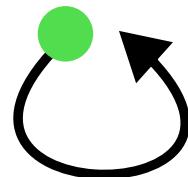
## Sequence



## Selection



## Iteration



Proving sequence and selection with the mathematic technique of enumeration, and iteration with induction, **structured programming was born**.

### Putting a stop to GOTOs

Realizing that we could all be writing *formally provable code* if we stopped using GOTO statements, not too much later in 1968, Dijkstra published a letter to the CACM (Communications of the ACM — the monthly journal of the Association for Computing Machinery) titled “Go To Statement Considered Harmful”.

After years of debate back and forth, mainstream programming languages saw GOTO statements removed entirely (or least suppressed). We removed the ability to perform “undisciplined direct transfer of control” (Martin).

### Functional decomposition

As programmers, one of the first things we learn how to do is to recursively break a large problem into smaller pieces so that we can tackle them one at a time. Functional decomposition — as it’s called — seems almost like second nature to so many of us today. Take note of this. Functional decomposition, along with those three control structures is the main idea behind structured programming.

### The birth of testing as we know it today

Alright, now how do we prove that code is correct? That’s what Djikstra was initially trying to do, right? And whatever happened to those mathematical proofs that Djikstra was talking about? Why do we use written proofs like enumeration and induction today?

At the end of the day, the proofs that Djikstra used were incredibly messy and laborious to do by hand. As you know, programs can get quite long; it just wasn't practical to be doing this for every module of code. We (thankfully) found a different way for testing (ie: Part IV: Test-Driven Development Basics and Part X: Advanced Test-Driven Development).

The key thing to note about the testing we do our programs is that it is rooted in science, not mathematics. That is, we use the scientific method to prove that our programs are working correctly.

In mathematics, we do this funny thing where we set up equations to succeed, and then rely on other decidedly agreed upon truths (called axioms) to prove them as truthful. Because of the law of addition (the axiom), we know that  $1 + 1 = 2$ .

However, in the wonderful world of science, we don't have really nice starting points. There's nothing we can confirm to be completely true.

There is no cosmic, universal truth that can explain *why* when I let go of my phone, it hits the floor. Instead, we have *good enough* laws that we regularly rely on (like Newton's law of gravity) because we're *unable to prove them false* through experiments. \*\*

If tomorrow, I drop my phone out of my hand and instead of hitting the floor, it stops in mid-air, hovers, grows arms, and with a booming voice says "you shouldn't have dropped me", then either I need serious psychiatric help, my phone is possessed (and I'd still need help), or we have to revisit many of the laws of physics.

But because that is extremely unlikely to happen, we'll just continue to assume that what we've discovered about gravity remains to be correct enough.

This, my friend, is exactly how we test code.

```
describe('fibonacci calculator', () => {
  it ('knows that 2 is a fibonacci number', () => {
    // Must fail to prove this statement wrong
    // for it to be correct
    expect(fibonacci(2)).toBeTruthy()
  })

  it ('knows that 4 is not a fibonacci number', () => {
    // Must fail to prove this statement wrong
    // for it to be correct
    expect(fibonacci(4)).toBeFalsy()
  })
})
```

Dijkstra once said, "Testing shows the presence, not the absence, of bugs." This means that our approach to testing code will be to apply a significant amount of effort by writing examples, setting up what we believe should happen, and if our tests don't fail, then we'll assume the code is — as we said — correct enough.

## What's the correct conceptual model?

With all this said, here's how you should think about structured programming:

- Whenever you use sequence in your code, that's structured programming — There's a more nuanced relationship between structured programming and *sequential programming* but let's just say that if you can assume that your code is moving in a sequential fashion, line after line, block to block, that's structured programming. You'll also note that it's impossible to selection (if/else/then) and iteration (for/while) without sequence. They are fundamentally structured programming concepts.
- It's a tool to functionally decompose problems into smaller parts — The idea is to make code more modular.
- To prove that algorithms are *correct enough*, we use tests — The only way to prove that structured programs are correct is to use tests.

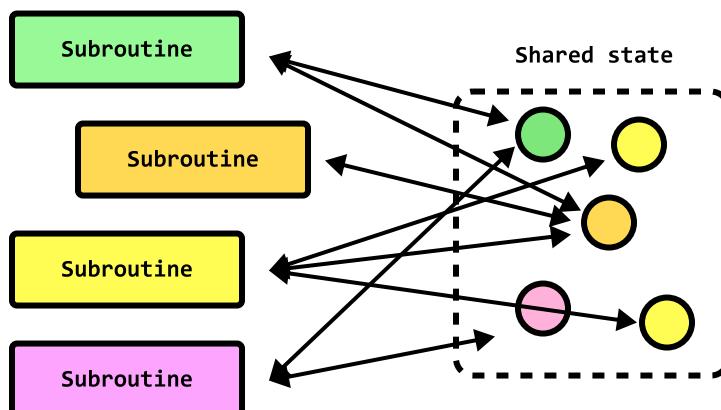
### Problem #1: Shared mutable state (coupling)

Structured programming got us pretty far, but there were some troublesome inadequacies. The first of which is the problem of *shared mutable state*.

All applications need to use a mixture of temporary and persistent state. Unfortunately, in the context of structured programming, persistent state was *shared* and *mutable* by other blocks of code. And there was little you could do about it.

Your options were to either put state somewhere global or pass data between subroutines. *Yuck.*

While declaring all your persistent state isn't all that looked down upon in single-file scripts for one-off quick-and-easy things, in business software, this is extremely frowned upon.



If you squint, this is essentially the coupling problem, my friend. We didn't have a good way to purely partition subroutines from each other. Since state required in another subroutine's scope was available to be inspected and mutated, they found themselves inherently connected and coupled.

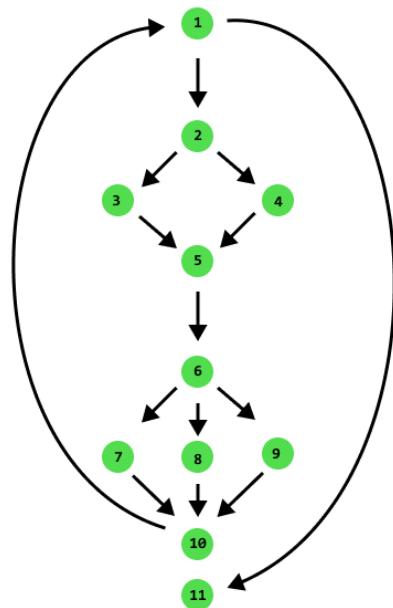
### Problem #2: Cyclomatic complexity

Another issue was high cyclomatic complexity.

Cyclomatic complexity is a way to measure the complexity of structured programs. It works by taking each instance of sequence, selection, and iteration (the three building blocks) and using them to document a *control-flow graph*: a type of directed graph that depicts the number of paths our program can take.

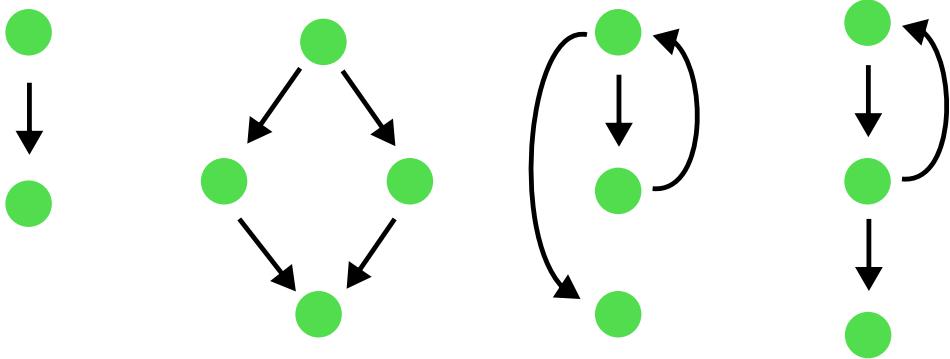
By counting the number of edges, nodes, and connected components, we can assign a program a numeric cyclomatic complexity score.

Node	Statement
(1)	while ( $x < 100$ ) {
(2)	if ( $a[x] \% 2 == 0$ ) {
(3)	parity = 0
(4)	}
(5)	else {
(6)	parity = 1
(7)	}
(8)	switch (parity) {
(9)	case 0:
(10)	println("a[" + i + "] is even");
(11)	case 1:
(12)	println("a[" + i + "] is odd");
(13)	default:
(14)	println("Unexpected error");
(15)	}
(16)	x++;
(17)	}
(18)	p = true;



If you look closely, you'll notice that each flow building block creates a different kind of behavior where sequence is the simplest, conditionals create new paths, and iteration sculpts out temporary paths.

## Sequence      If/then/else      While      For/until



I'm sure you could have guessed this, but studies have shown that there is a positive correlation between the defects a program has and the number of paths it takes. In other words, the more complex a program is — the higher the cyclomatic complexity — or, the more likely it is that it'll contain defects.

In purely structured programs, it wasn't uncommon to have lots of deep paths and *spaghetti code*. Again, this was largely due to the fact that the languages didn't introduce *safe ways* to decouple routines from each other and data structures in a safe and reliable way. Programs were very un-resilient and susceptible to ripple.

For example, imagine you were writing the code for an OS that made USB ports work. You'd need to ensure that every device plugged in can function and do its job, but in reality, there are a *lot* of different things that can be plugged in. What do you do? Write some code for each and every single one of those?

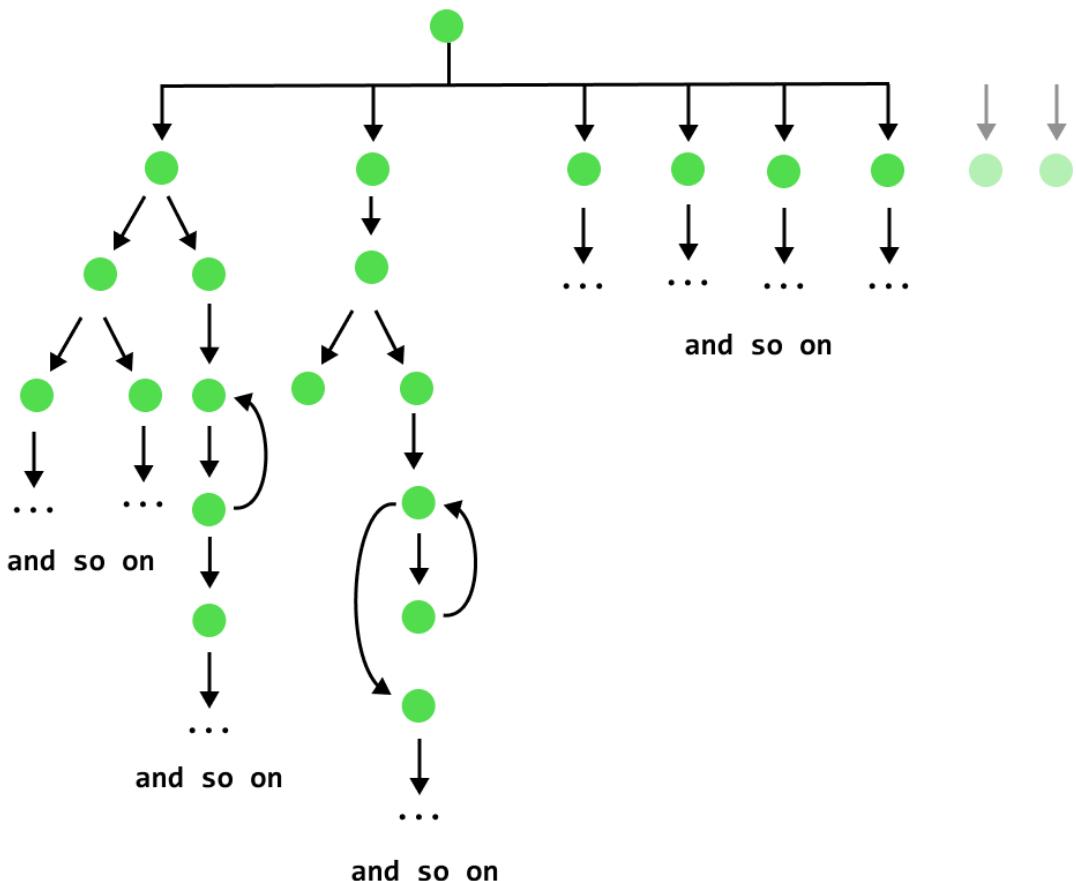
```

switch (deviceType) {
  case 'MOUSE':
    mouseSubroutine();
  case 'HEADPHONES':
    headphonesSubroutine();
  case 'MIDI_KEYBOARD':
    midiKeyboardSubroutine();
  case 'WEBCAM':
    webcamSubroutine();
  ...
}
  
```

That's ridiculous. What happens when there's a new type of USB device? Come in and write *more code*? No, this is *architecturally poor*.

First of all, this only continues to increase the cyclomatic complexity of our USB port code as we add support for more devices over time. We'd see our simple program blow up into an

enormous and ever-growing control-flow graph of subroutines on subroutines.



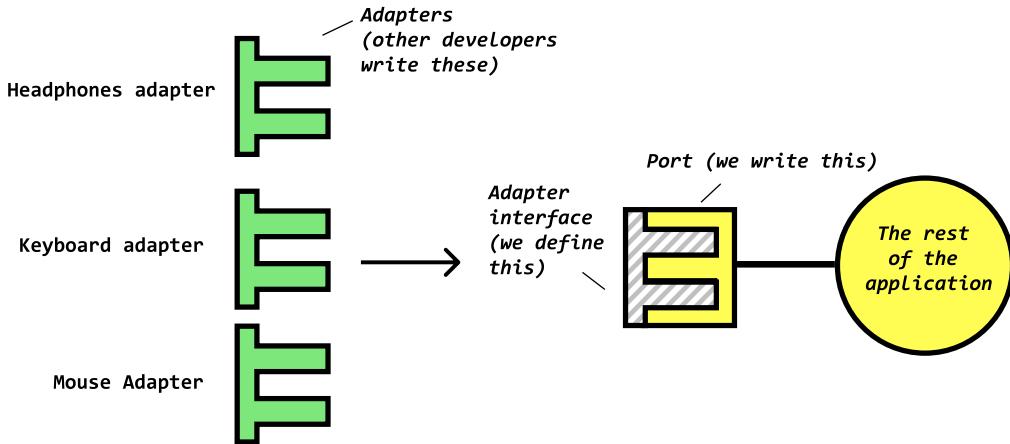
Second of all, the fact that we have to do more work every time a new device is created, is an impossible solution. There's no way that we could keep up with the real-world.

It seemed that if we had to provide all the future implementations, we'd only continue to drill ourselves deeper into an impossibly complex tree.

What if there was a way to invert this dependency relationship? What if we could just focus on defining a data structure — one that contains all the necessary things that a USB device needs to provide if they want it to work with the OS — and leave it to future USB device developers to implement them?

We are asking: what if we could **program against an interface instead of an implementation?**

It turns out we can! The idea of *ports* and *adapters* is probably the simplest way to understand this phenomenon. Imagine that all we have to do is to write the *port* and then define what a valid *adapter* is and leave it to others to implement those *adapters*.



This is the very notion of polymorphism and dependency inversion. It simplified our cyclo-matic complexity problems, but unfortunately, there wasn't a way to do it safely.

### Problem #3: Unsafe polymorphism

Polymorphism is the ability for an abstraction to take many shapes. While a lot of developers tend to think it's a uniquely object-oriented thing, it's been around long before we had OO.

In non-OO languages like C, we used *pointers* (a type of variable that stores the address of another variable or function) to reference functions that would eventually need to be implemented elsewhere. We'd do this in a file called a *header file*.

For example, a generic MIDI\_INSTRUMENT data structure (*port*) could contain pointers to three different functions:

```
// midi.h
struct MIDI_INSTRUMENT {
    void (*send_note)(int note);
    void (*on_note)(int note);
    void (*close)();
};
```

Now, it's up to the developer who writes the code for the MIDI instruments (*adapter*) to define those functions.

Take an electronic drumkit as a MIDI\_INSTRUMENT:

```
// electronic_drumkit.c
#include "midi.h"

void send_note(int note);
void on_note(int note);
void close();
```

```
struct MIDI_INSTRUMENT electronic_drumkit = {
    send_note, on_note, close
};
```

Or even perhaps a keyboard instead:

```
// keyboard.c
#include "midi.h"

void send_note(int note);
void on_note(int note);
void close();

struct MIDI_INSTRUMENT keyboard = {
    send_note, on_note, close
};
```

In the `electronic_drumkit` file, even though it's a `MIDI_INSTRUMENT`, we might implement the `on_note` function slightly differently than we implement it as if it were a keyboard.

In the electronic drumkit, when a note is received, we might want to light up the correct drum pad that responds to the midi key.

```
// electronic_drumkit.c

...
void on_note (int note) {
    // Light up the correct drum pad that corresponds to
    // the midi key
}
```

Versus in the keyboard, we may not actually want to do anything really.

```
// keyboard.c

...
void on_note (int note) {
    // Do nothing.
}
```

How do these get loaded up? Well, when you compile your program, you can provide whichever implementation you like. We just need to make sure that there's *some* valid implementation of the functions in the header file among the files being compiled.

While this seems somewhat similar to using interfaces in languages like TypeScript or Java, this is incredibly hacky.

Using pointers to functions is dangerous. Because there's nothing in the C language remind-

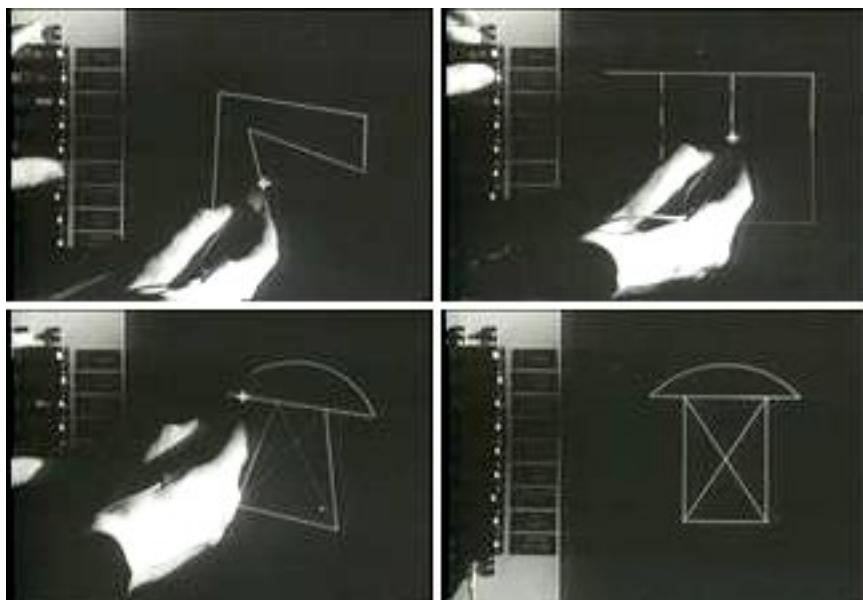
ing you to initialize the pointers. Since there are no *constraints* (5. Human-Centered Design For Developers) to guide you to do the right thing, you have to remember to do it yourself.

The programming language isn't working with us here. If you forget to implement the declarations, this can lead to hard to find bugs.

We needed *safe* polymorphism. This came around later in OO.

## Object-oriented

In 1963, Ivan Sutherland developed a program called Sketchpad. He didn't know at the time, but this little program would go on to influence the way humans interact with computers and create digital graphics using modern day applications.



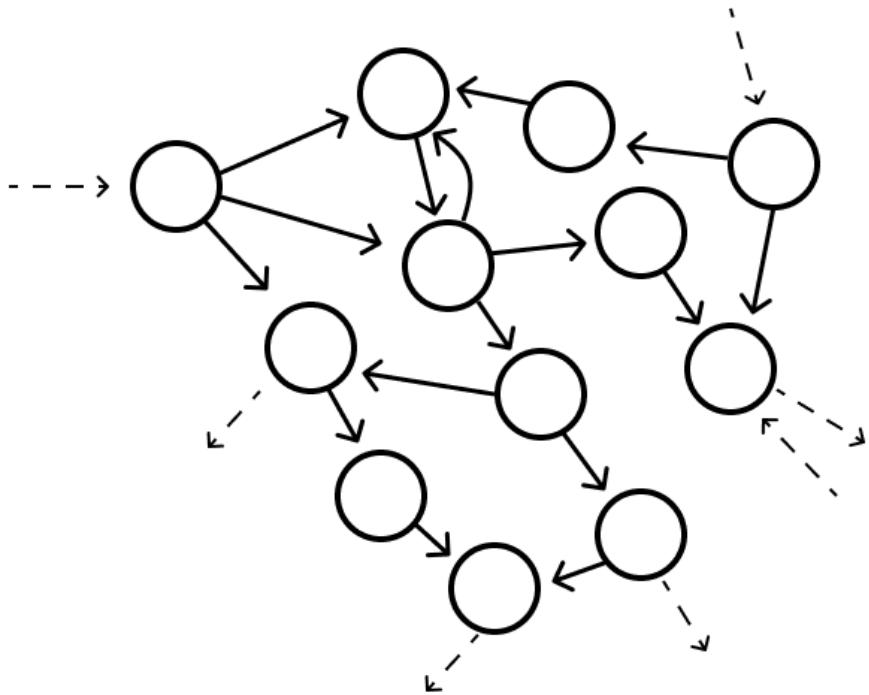
Not only was Sketchpad the first program ever to demonstrate a working graphical user interface (GUI), but it made use of concepts like "objects" and "occurrences" (instances). Sketchpad was one of the very first semblances of modern object-oriented programming in practice.

In 1966-67, researcher Alan Kay was likely aware of Sketchpad and put a stake in the ground. He coined the term "Object-Oriented Programming" (OOP).

## The main idea: A web of objects

Influenced by his background in cell biology and the design of Arpanet (a very early version of the internet), Alan conceptualized the idea of a **web of objects**.

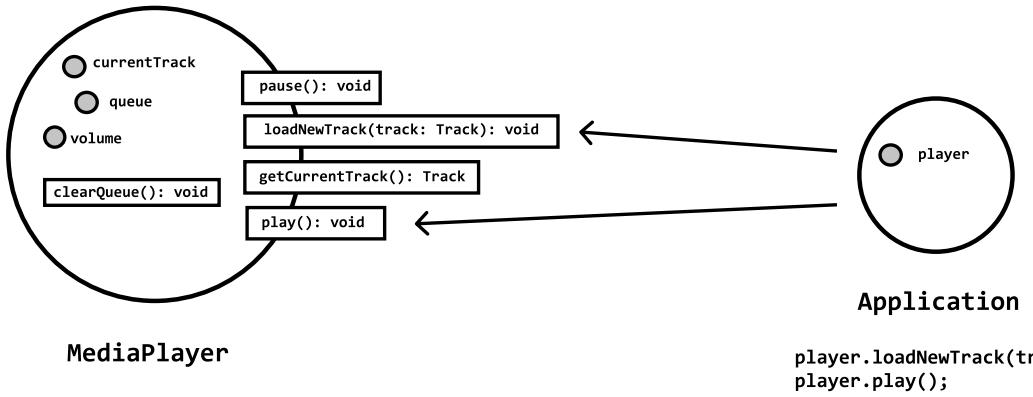
To Alan, OO was about messaging. Specifically, he envisioned individual mini-computers, each with their own isolated state (**encapsulation**), communicating with each other via the technique of **message passing**, and that could be easily **pluggable and interchangeable** (**ah, yes — polymorphic**).



What does a message in OO look like, you ask? A large number of us have been writing object-oriented code for a long time and have never thought about this before. **You may have to open your mind to a paradigm shift right here.**

In OO, objects communicate by sending messages to other objects. They also sometimes react to messages by returning a value or exception to the original sender. Good OO design principles recommend that a message be either a command or a query and less frequently both.

Each object has a *method* for handling every type of message that it understands. Read that again. This is the true philosophical meaning to a *method*. Methods aren't just *functions* within classes. They are what objects use to communicate with each other.



I used to have a different conceptual model for methods. I used to imagine that an object was a bag that we could reach inside of and nest deeper and deeper. This is a fairly common conceptual model, and we see it in code everywhere. For example, most OO languages allow us to *method chain* (also known as “train wreck code”) and write code like the following:

```

habitsPage.getAllHabits()
    .filter((h) => h.active)
    .forEach((h) =>
        h.getHabitState()
            .setValue(Boolean.FALSE.booleanValue())
    )
}

```

In Alan Kay’s OO, that is — the *actual* OO — this is very bad. If one object was trying to ask another object to do something (in this case, the `habitsPage`), why couldn’t it just ask `habitsPage` for what it really wanted? Craft an expressive message.

Reading behind the lines, it seems like what we actually wanted to ask the `habitsPage` to do is:

```
habitsPage.makeAllHabitsInactive();
```

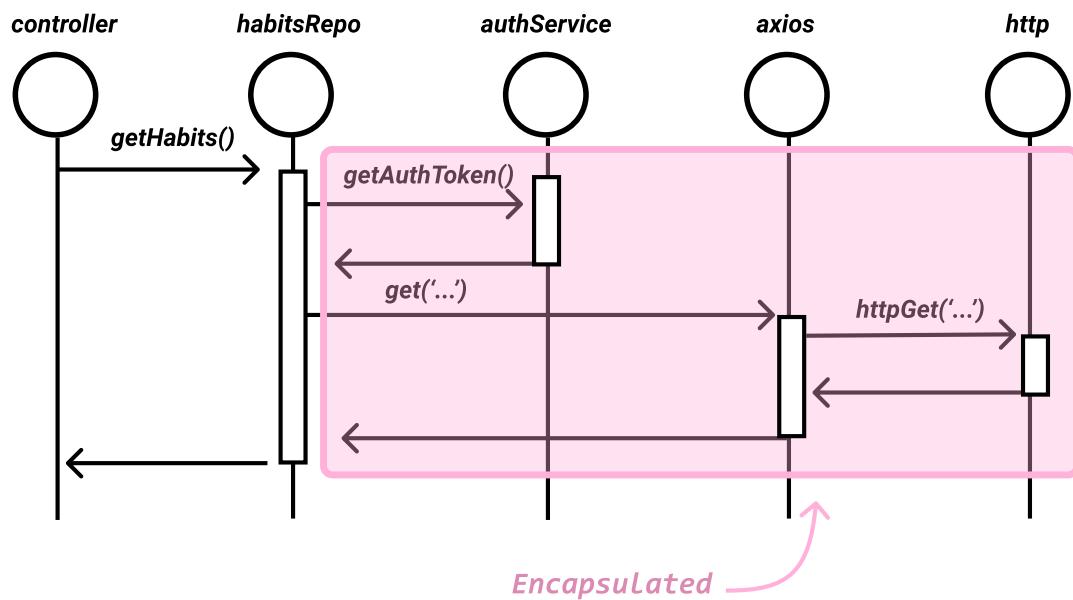
There are a number of design principles (Part VII: Design Principles) that build on this idea of being more declarative and saying what you want, not *how to get it* (like the *Law of Demeter* principle or the “*Tell, Don’t Ask*” principle).

With encapsulation, we **focus on the messages** and dictate that **the object receiving the message be responsible for figuring out how to do the work**.

A useful analogy is like when your parents asked you to go to your neighbour’s and ask them for sugar or some other ingredient. You’d go to the front door, ring the doorbell and politely ask them if they could get the ingredient for you. Then you’d wait patiently and hope that they come back with what you asked for. Now, if I was a bad kid, I’d blast past my neighbour,

go straight into their home, open the pantry, open the jars, and help myself to whatever I like. That's incredibly rude. That's kinda what we're doing with method chaining.

Another beautiful aspect of encapsulation is that **often, the object responsible for figuring out how to do the work isn't the object that does the bulk of the work**. With encapsulation, when an object receives a message (method call), we can offload tasks to other objects without explicitly needing to inform the calling object that this is what happened. To the calling object, all it cares about is that it gets what it asked for. Notice how this is illustrated in the following sequence diagram.



## Emergence

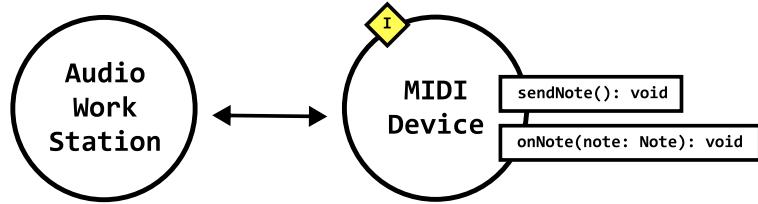
When we focus on the messaging and encapsulation aspects in OO systems, behavior emerges — and it does so *declaratively* instead of imperatively.

Steve Freeman says:

The behavior of the system is an emergent property of the composition of the objects—that is a) the choice of objects and b) how they are connected.

In other words, in a conversation between two objects, if we formalize the messages exchanged between them, we can safely replace them so long as the new ones we swap in **also know how to handle the messages in the same way**.

Let's think back to the MIDI device example. If we were building a program that enabled users to create electronic music with various pieces of MIDI hardware (like electronic drumkits, pianos, sequencers, etc), then we'd need to define the **common message structure** between our program and the devices. All MIDI devices need to know how to send and receive notes.



In OO languages like Java or TypeScript, we represent these concepts with either interfaces or abstract classes.

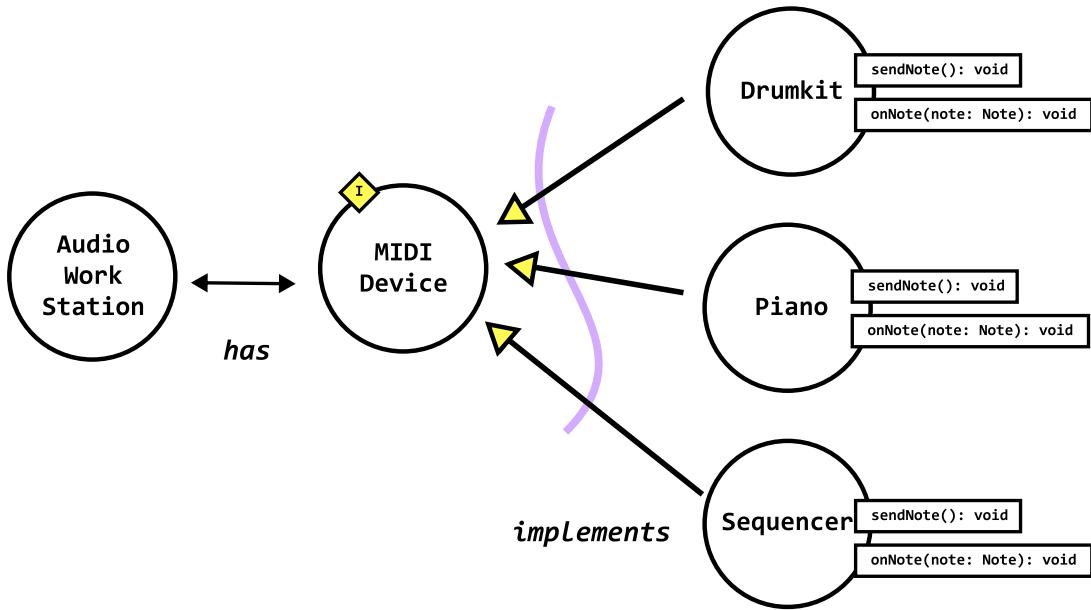
```

interface MIDIDevice {
    sendNote(): void;
    onNote(note: Note): void;
}

class AudioWorkStation {
    // Our program relies on the MIDIDevice interface, which
    // could be realized as a wide variety of actual devices
    // or mocks.
    private MIDIDevices: MIDIDevice[] ;
}

```

The interface, acting as the thing that defines the message structure, makes it clear to the MIDI device developers which methods *need* to be implemented to work with our program. So when a company that builds instruments for electronic music creators wants their hardware to work with our software, they write the adapter which implements the methods.



For example, the drumkit implementation done by company A might look like:

```

class Drumkit implements MIDIDevice {

    sendNote(): void {
        // Detect when a drum pad is being pressed
        // Then send the note
    }

    onNote(note: Note): void {
        // When a note comes in from the program
        // Then light up the correct pad on the kit
    }
}

```

A MIDI device can take many shapes in our program. But conversely to how this works in structured programming languages, interfaces and abstract classes give us a form of **safe polymorphism**.

Our code knows that *we are going to depend on something that knows how to handle these messages, but we don't know specifically what the implementation will be.*

This technique is more formally known as Dependency Inversion (or indirection) and architecturally speaking, it's the most important thing that OO gives us.

**The critical importance of Dependency Inversion:** Take special note of this principle. It is foundational to a great number techniques and practices such as decoupling dependencies, keeping unit tests fast, and inventing contracts that future objects must use to communicate. Arguably the most important principle for an Object-Oriented developer to master, we focus in on dependency inversion in 39. Keeping Unit Tests Pure with Mockist TDD and

The Art of Purity & Inversion - coming soon and Dependency Inversion Principle (DIP).

### **Sequence, selection, iteration, and indirection**

Structured programming gave us sequence, selection, and iteration — the three basic building blocks of **flow**, the way in which we implement our algorithms.

Object-oriented programming though, gave us **indirection**, an architectural technique. It gave us the ability to swap out dependencies without having to change the existing code. Instead of doing this unsafely like we used to in structured programming, we obtained a way to do it safely through the use of abstraction constructs.

With these four building blocks, we could now write components that were:

- Packaged into cohesive vertical slices with a single responsibility (Single responsibility principle)
- More maintainable since they could be strategically open for extension but closed for modification \*\*(Open-closed principle)
- Substitutable for other components, so long as they implemented the same methods (Liskov substitution principle)
- Declaratively communicating their intent with small, cohesive interfaces (Interface segregation principle)
- Able to **safely** transfer control to other components without needing to know their internal details (Dependency inversion principle)

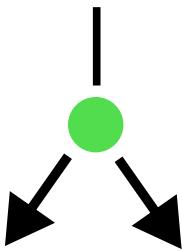
■ See more about the SOLID principles and more in Part VII: Design Principles.

# Flow

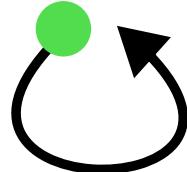
Sequence



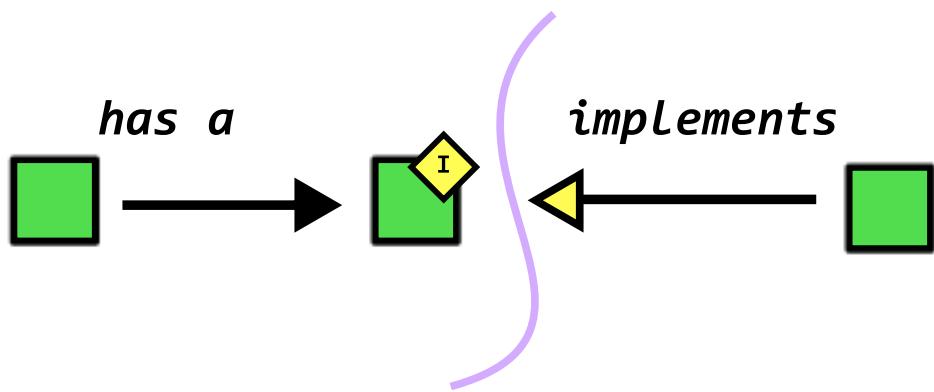
Selection



Iteration



Indirection



To this date, we have not identified any more comparable building blocks to the craft that is computer programming.

Software—the stuff of computer programs—is composed of sequence, selection, iteration, and indirection. Nothing more. Nothing less. — Robert C. Martin

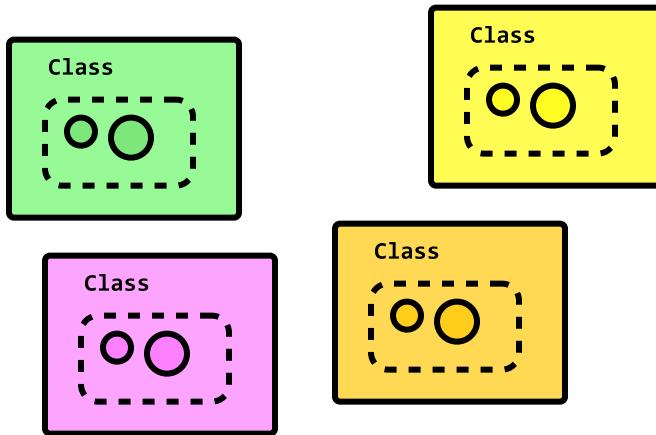
**How does OO solve the problems structured programming faced?**

Let's get explicit about the improvements here. The three main things that Alan Kay believes OO gave us were:

1. Encapsulation
2. Message passing AND
3. Dynamic binding (polymorphism — the ability for the program to adapt at runtime)

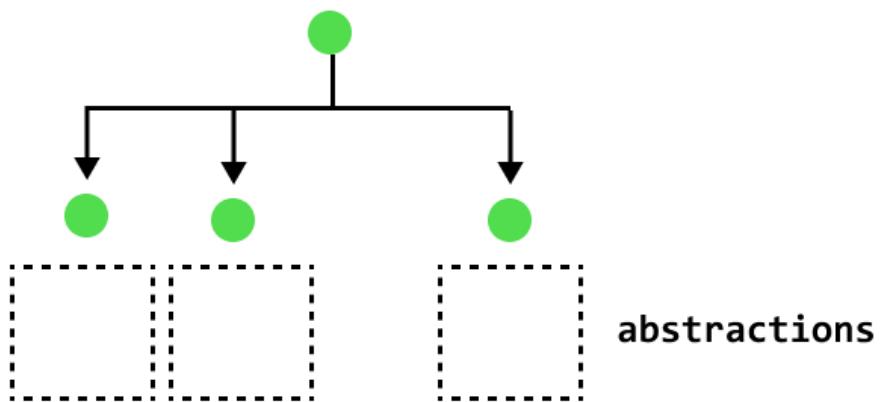
If we think back to the issues we were facing in structured programming, the first big one was the problem of shared mutable state and the tight coupling that it manifested. Encap-

sulation is the obvious answer to this problem. We can now encapsulate state within the object it belongs to most.



The second big problem was cyclomatic complexity. Code had a way of getting way too complex too fast. Encapsulation + dynamic binding is the answer to this. Now, not only can we encapsulate algorithms and data within the object it belongs to most (cohesion), but we can also choose to leave implementation details to be figured out later by programming against the abstraction (such as an interface) instead of a concretion.

With this technique, we cut the control flow graphs down dramatically. Instead of dealing with concrete classes here and now, we can accomplish key tasks by describing *future* abstractions. We can focus on problems at a high level and leave low-level modules to iron out the implementation details. This is the nature of architecture.



The third problem was unsafe polymorphism. Linking programs together was dangerous back then. Now, with constructs to define the message structures, and the fact that it can be checked at compile time, we can safely design flexible and reusable software.

## Inheritance — how to mess up doing OO

As I said earlier, just because the tool lets you do something doesn't mean it's right. Because there are quite a few different tools in OO, there are a *lot of ways* you can mess up your designs.

The most common way that developers do this is by using **inheritance** too much.

Inheritance is a language feature that allows you to reuse code by creating an *is a* relationship using the extends keyword. When we inherit a class, the subclass (also known as the derived class) learns how to realize the methods in the base class.

Here's the classic *Animal-Cat-Dog* example that a large number of us saw when we were starting our careers in OO (and that sent us down a path of creating a generally incorrect conceptual model for OO).

```
public class Animal {  
  
    private string name;  
  
    public setName (name: string): void {  
        this.name = name;  
    }  
  
    public void printName(){  
        console.log("My name is: " + name);  
    }  
}  
  
let animalOne: Animal = new Animal();  
let animalOne.setName("Betty");  
  
let animalTwo: Animal = new Animal();  
animalTwo.setName("Veronica");  
  
animalOne.printName(); // "My name is: Betty"  
animalTwo.printName(); // "My name is: Veronica"
```

And then we create the subclass and extend the Animal parent.

```
public class Cat extends Animal {  
  
    public meow() : void {  
        console.log("Meow!");  
    }  
}  
  
let myCat: Cat = new Cat();  
myCat.meow(); // "Meow!"
```

```
// These functions are inherited, so we can call them!
myCat.setName("Stanley");
myCat.printName();           // "My name is: Stanley"
```

Try to realize how far away this is from anything we've just discussed about the **main idea** of OO and the web of objects analogy we used. With that in mind, if this is how you were introduced to OO, it's not hard to see how you may have gone for a long time thinking that OO was about creating relationships and hierarchies between data that mimic the real-world.

Nonetheless, inheritance does exist, but all actuality, you should seldom use this technique. I can only really advocate for it in specific scenarios. Like when we're building a library or some infrastructure that is going to be depended on throughout the entire application.

As you may be able to ascertain, inheritance creates a *hard* source code dependency. This means there is an unusual amount of risk and importance on making sure that these classes have been created correctly. **The need to change them could ripple into all of the other classes that inherited from them.** Therefore, only extend classes that you're sure are going to be extremely stable.

## How to do OO properly

Remember that *values* influence *principles* which in turn, dictate good *practices*. So many developers get OO wrong because they're implementing practices without even understanding what the principles *or* the underlying values are. It's great that you've studied the SOLID principles and other design principles, but if you really want to intuitively know what's right and wrong, we have to start with the values. Starting from the values, we'll be able to develop our own principles, know when to use the established ones, and really truly respect the practices.

So, leave the hierarchy idea behind. Let's focus on that web of objects again. Specifically, here's what's important moving forward.

### Value #1: Messaging

Ask yourself the following:

- Can I declaratively express the behavior/communication I want to happen between objects (even if they don't exist yet)?
- Can I use tests to guide the design and validity of my messages/methods?
- Can I tell if my class is overly coupled?
- Can I use Part VI: Design Patterns to express common communication patterns?

### Value #2: Encapsulation (and the right amount thereof)

Ask yourself the following:

- Can I tell if my class is cohesive?
- Is this class too big? Should I break it into another class? How?
- Should all these instance variables really be in this class? Should they be in another class instead?
- Are the methods expressive? Will another developer understand?

## **Value #3: Dynamic binding (polymorphism)**

Ask yourself the following:

- Do I know how and where to create boundaries in my code?
- Do I know how to design an understandable abstraction?
- Do I know how to use design patterns to express common structural and behavioral patterns?

■ **How to grow:** Do some honest self-examination here. Ask yourself each question and try to answer it. If there are any that you're not able to, that's OK, but be honest. Knowing yourself is the first step to improving yourself. Future sections in the book take you through these topics so that you can master them and formulate your own opinions on them. For example, in Part IV: Test-Driven Development Basics, we cover the basics of TDD and in Part V: Object-Oriented Design With Tests, we learn about good OO design practices, mockist TDD, *code smells*, and how to detect and refactor them.

## **Functional**

Functional programming is the oldest form of programming.

Its essence comes from something called Lambda calculus, a system that predates programming itself. It was developed by Alonzo Church in the 1930s and used to express computations as functions. Eventually, we learned that we could apply the same techniques to programming and voila — we ended up with an incredibly declarative way to write code.

For example, try telling me you don't get a general idea of what we're trying to accomplish with the following lines of code:

```
capitalize(trim(" hello world ")); // "HELLOW WORLD"
```

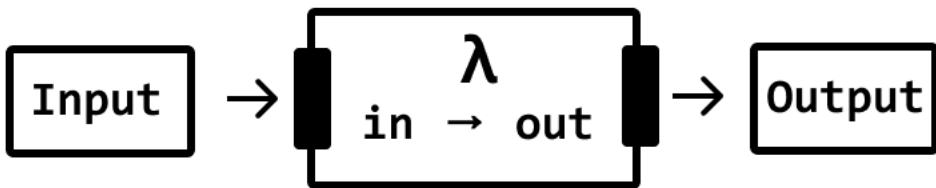
## **A completely different conceptual model**

Originating in mathematics, functional programming approaches the idea of computing from an entirely different angle.

Wikipedia says that functional programming is:

“ a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements.”

Sardonically, Scott Wlaschin says functional programming is “*programming as if functions really mattered*”.



If you're coming from OO where we've used decomposing the problem by breaking it down into smaller objects, in FP, we break them down into functions. And where you're used to reducing coupling in OO using dependency inversion, in FP, we use a comparable technique called *parameterization*.

If you're coming from OO (or at least structural, procedural programming — which is likely because that's the only other option), functional programming requires a complete paradigm shift if you're going to be successful with it. The way we approach problems in FP is fundamentally different to the way we're used to thinking about them in procedural style languages like Java, TypeScript, and so on.

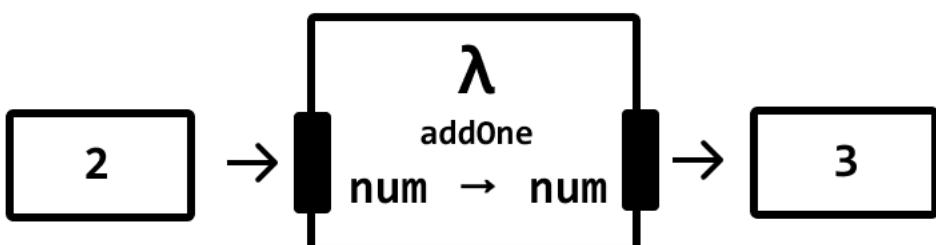
There is a lot that can be said about functional programming, but I'll focus on helping us understand what's different and how we can use FP to build fully-fledged software systems.

## Function basics

Starting from the very bottom, functions map input values to outputs.

For example, the following function takes in a number and adds the value 1 to it.

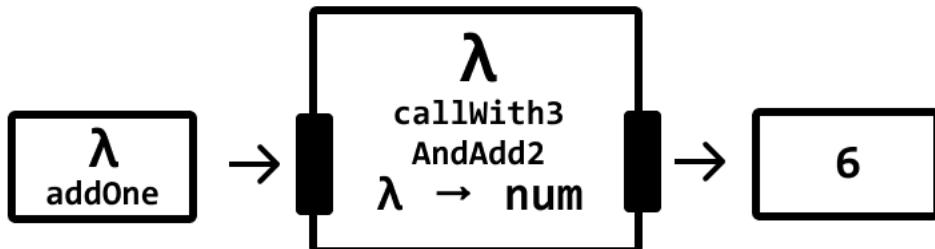
```
const add1 = (num: number) => num + 1;
add1 (2); // 3
```



Functions themselves can *also* be passed in as input values.

In this next example, `callWith3AndAdd2` is a function that takes in a function, calls it with 3 and adds 2.

```
const callWith3AndAdd2 = (f: Function) => f(3) + 2;
callWith3AndAdd2(add1); // 6
```



Just as functions can be passed in as inputs, they can also be returned as outputs as well.

An example I got from Scott is the scenario where you have three different functions adding three different numbers:

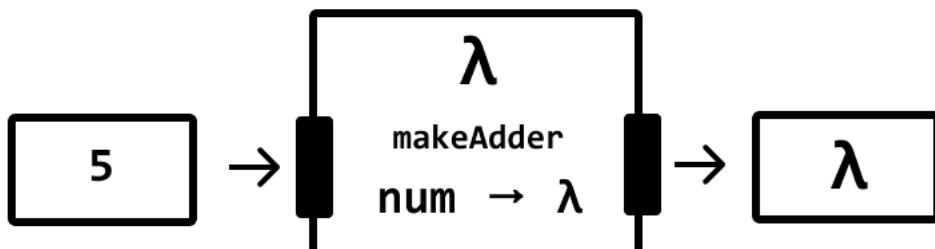
```
const add1 = (num: number) => num + 1;
const add2 = (num: number) => num + 2;
const add3 = (num: number) => num + 3;
```

How can we remove the duplication here? The answer is to create a function that returns another function with the ability to add a number that we bake in:

```
let makeAdder = (numberToAdd: number) =>
  // return a lambda (unnamed, anonymous function)
  (x: number) => numberToAdd + x;
```

Now, we can do this:

```
let add5 = makeAdder(5);
add5(1); // 6
```



Functions can also be passed in as parameters to change the behavior of a function. If you're a JavaScript developer, some of the most common examples are the array methods. Here's an example of us negating an array of integers based on an anonymous function we provide to the `map` array method.

```
[1,2,3,4,5] .map((num) => -num);
```

In programming languages where we can use functions as inputs, outputs, or parameters, we call functions **first-class citizens**. They're called that because they can be treated the same way that we treat conventional primitives like strings or integers.

### Not the same sequence, selection, iteration and indirection

What's the difference between these two ways of implementing the same observable behavior?

```
// Solution one
function doubleItems (arr) {
  let results = []
  for (let i = 0; i < arr.length; i++){
    results.push(arr[i] * 2)
  }
  return results
}

// Solution two
const doubleItems = (arr) => arr.map((item) => item * 2);
```

If you said the first was **imperative** and the second was **declarative**, you're right.

In procedural languages, we use statements to move line-by-line, instruction to instruction, assigning state and changing it to express what we want the code to do. Scholars call this **imperative programming**. In structured and object-oriented programs, statements are things of assignment, selection, and iteration. These are what we use to create behavior. Every line from the beginning to the end of that first `doubleItems` implementation is a statement.

In functional programming however, we write code with *expressions* and *declarations* instead of statements.



## Imperative      vs      Declarative

### Focuses on

How to do things                  What to do

### Describes behavior using

Statements                  Expressions

### Program execution involves

Defining variables  
and changing their  
values                  Evaluating the result  
based on the input

For example, in F#, a functional programming language, if we wanted to use selection (if/then/else or switch), we'd do something like this:

```
let tenDividedBy n =
    match n with
    | 5 -> 2
    | 4 -> 2
    | 3 -> 3
    | 2 -> 5
    | 1 -> 10
```

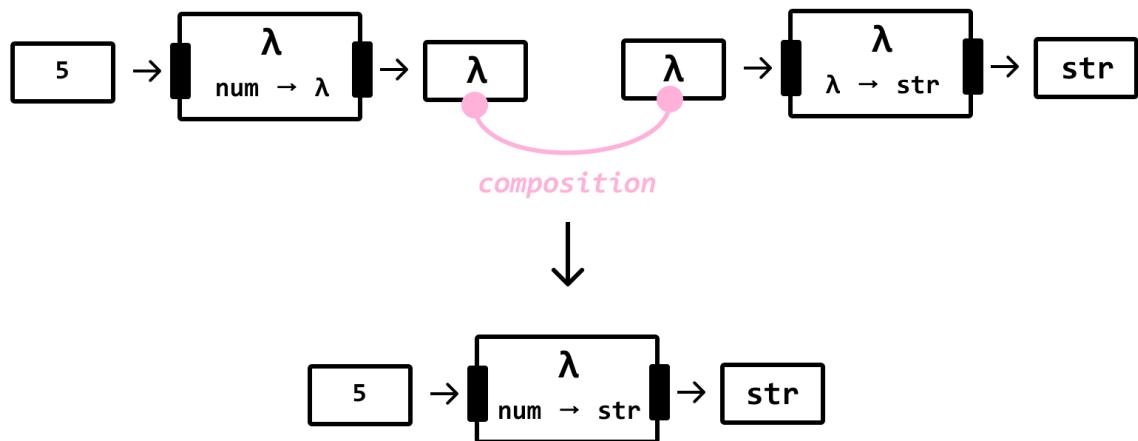
The key is to write smaller expressions and compose them together to get them to do things that in essence, behave like sequence, selection, iteration, and indirection.

### Composition

I was always curious about how real-life systems were built using functional programming. I'd only ever seen small, simple examples like we walked through just now.

The answer is composition, or *function composition*, to be specific. When we can map one output of a function to the input of another one, we accomplish functional composition. It

looks something like this.

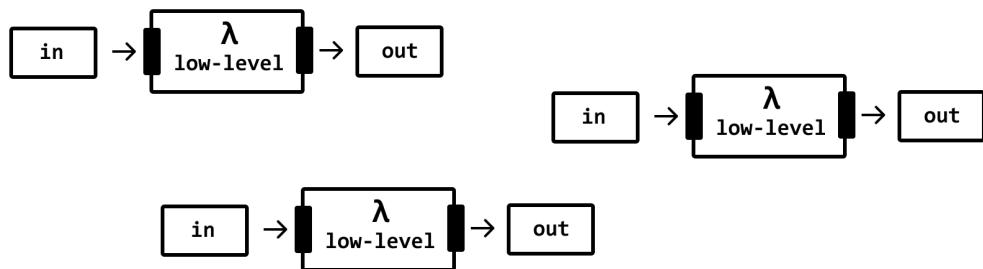


Now, how to build up fully-fledged computer systems? You merely compose over and over.

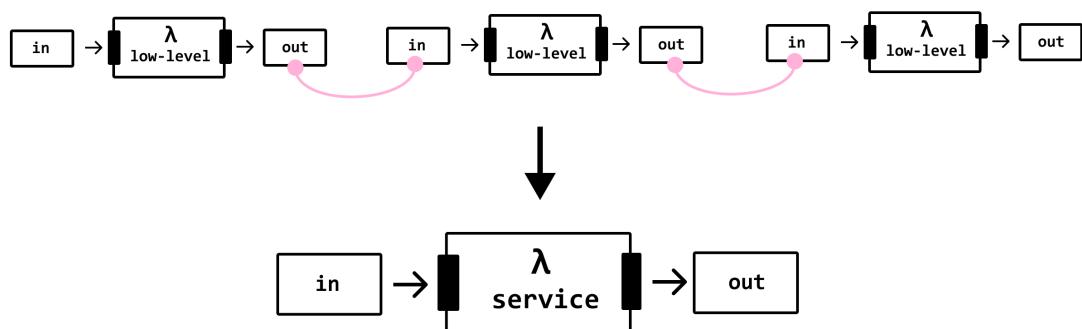
### Using composition to build real-world systems

Here's how it works in theory.

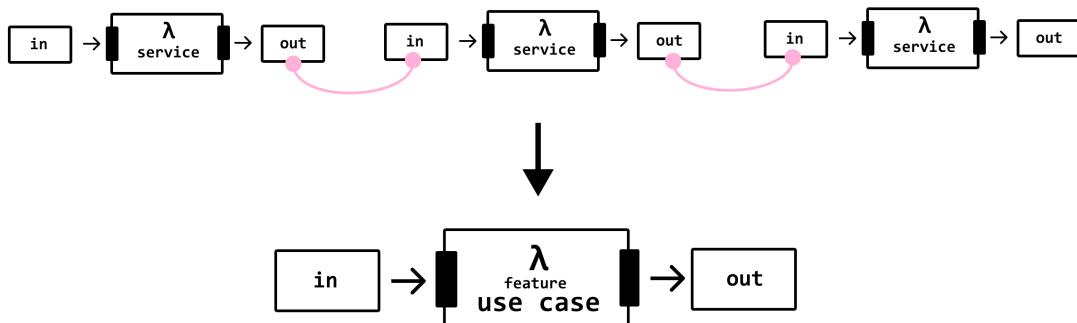
We start with a bunch of small, low-level functions that do basic things:



We compose those small low level functions together into a single service function:

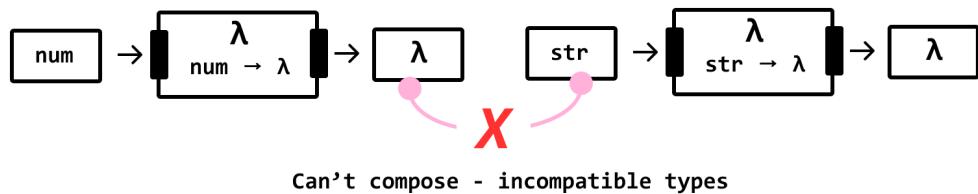


Then we take those service functions and glue them together using special sequence, selection, and iteration functions to create **use case / feature** functions. These are our user stories turned into functional code.



Finally, we take these use case functions and hook them up to a controller that selects the correct use case based on the API call.

The biggest challenge in this entire endeavor is that often times, functions don't nicely fit together. The input type of one function might not match up with the output type of another. It feels like trying to force a lego or puzzle piece into a space that doesn't fit.



The solutions here are to either:

- Convert both sides to be “lowest common multiple” — the easiest side to convert
- Standardize a uniform shape for function composition across the entire application

This is, in essence, how real world systems are built. Compose together functions that handle a request in the controller, handle various levels of application and domain logic, and handle persisting or failing results in a database. All without state or sequence. Just I/O to I/O from edge to edge.

We won't go into how this works in detail (the switching and conditional logic can get pretty complex) — instead, if you're interested, I recommend you check out Scott Wlaschin's *Domain Modeling Made Functional* book to learn the functional technique. There's a lot of overlap with what we cover in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS and I think you'll find *railway-oriented programming* incredibly fascinating.

## Why functional programming?

**Better predictability.** Without state and sequence, we write software in a way that improves the predictability of our code. With procedural style programs, we had to write unit tests to ensure that they were behaving correctly. If we can get our functional programs to compile, we write less unit tests and focus on *Acceptance Testing* that we get the correct output data based on the inputs. Of course, this places weight on ensuring our logic utilities work as intended and are understandable (see 5. Human-Centered Design For Developers). That may be the biggest challenge yet.

**We push state changes the boundaries.** When we do this, we reduce the probability of issues like race conditions or concurrency problems. Why? Because you can't ever have a race condition or deadlock if there isn't anything to lock in the first place. This is part of the reason why we prefer the *Unit of Work* pattern (see Unit of Work) in our web requests — to make the only time we mutate data be when we're committing the atomic transaction at the very end of an API call (see Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS).

**Scalability.** Taking it even further, architecturally speaking, the architectural pattern Event Sourcing (10. Architectural Patterns) that powers banks, massive-scale websites, and even your source code control system, inherits the same immutability principles from functional programming. With it, we store the transactions, but not the state. When state is required, we apply all the transactions from the beginning of time to compute it.

**Readability?** This is highly debatable. Some say it's subjective to ones experiences with imperative-style programming. I lean in the direction that FP is not nearly as readable as imperative or object-oriented style code. Based on our understanding of *Knowledge in the Head*, those of us accustomed to making sense of code by reading line-by-line statements find OO more readable. Strengthening our understanding of line-by-line reasoning is the fact that statements are the basis of the English language and the way we tend to describe phenomena. We have Aristotle to thank for that. We think and describe in terms of nouns (as objects) and verbs (as behavior/methods). "Move the car". "Eat your lunch". However, some of the more mathematically experienced developers may find functional code to be much more precise and efficient to read. However, it is an entirely different approach to thinking about and describing phenomena. One you may need practice with.

■ **Consider this:** If the English language were constructed around functional expressions instead of line-by-line statements, recipes would be written as "A pie is 45 minutes of 400F heat applied to 200mg of final pie dough."

## Design principles are paradigm-agnostic

Sometimes developers think that the only way to accomplish immutability is to use the functional programming paradigm. And other times, developers believe the only way to perform encapsulation or polymorphism is to use object-oriented programming.

The truth is that all of these techniques, design principles, and whatnot — are paradigm-agnostic. While some techniques may be implemented differently depending on the paradigm, and that it may be more or less trivial to do, we are still dealing with sequence, selection, iteration, and indirection, regardless of the paradigm.

■ In Part VII: Design Principles, we learn how to use design principles that come from both

the OO and FP paradigms.

## Choosing a programming paradigm

In 21. Understanding a Story , we saw how to gather the essential complexity for a feature by writing it out as declarative pseudocode. The goal now is to codify that essential complexity in the most effective way possible.

There are a wide range of factors that will influence which programming paradigm you're able to use from:

- the skill level of other developers on your team
- familiarity with the programming paradigms and their appropriate conceptual models
- the language you're using and the support it has for the programming paradigm you would like to use

Personally, I believe that functional programming is the *cleanest* way to codify the essential and avoid accidental complexity. However, object-oriented programming is still the most popular way to develop software, and it's most likely that you'll find yourself on teams of other object-oriented developers.

I believe that the best thing any developer who knows the basics of OO and wants to get better should do is to master OO, learn how to apply the functional programming principles to it, and learn functional programming on your own time.

There is immense value and much to be learned by each. As Uncle Bob puts it, structured programming taught us to restrict the use of direct transfer of control, object-oriented programming gave us the ability to impose discipline over indirect transfer of control, and functional programming taught us to impose discipline against assignment and how we handle state.

As an industry, over the last 40 years, the programming paradigms have taught us a lot about what *not* to do. Perhaps the best thing is to use the best aspects of each paradigms at the same time.

## Summary

- Structured programming gave us sequence, selection, and iteration. These are the algorithmic building blocks we use on a day to day basis.
- Object-oriented programming is about a web of objects. It gave us safe encapsulation, message passing, and dynamic binding (polymorphism or just indirection). Indirection is an architectural tool that we use to safely cross boundaries.
- Functional programming is an entirely different way to think about building software, and if it contains state or sequence (statements instead of expressions), it is no longer purely functional code. Functional programming's lack of moving parts demonstrates the value of rigorously pushing state to the boundaries.
- It is recommended that you use either object-oriented programming or functional programming, but it is more likely that you will use OO in workplace. With diligence, you can still use all the best principles of FP in OO as well.

## Exercises

■ Coming soon!

## Resources

### Articles

- [https://en.wikipedia.org/wiki/Control-flow\\_graph](https://en.wikipedia.org/wiki/Control-flow_graph)
- [https://en.wikipedia.org/wiki/Cyclomatic\\_complexity](https://en.wikipedia.org/wiki/Cyclomatic_complexity)
- <https://richardeng.medium.com/oop-vs-fp-1a3da34d2030>
- [https://www.reddit.com/r/AskComputerScience/comments/aarnie/why\\_do\\_alot\\_of\\_people\\_hate\\_oop/](https://www.reddit.com/r/AskComputerScience/comments/aarnie/why_do_alot_of_people_hate_oop/)
- <https://yarax.medium.com/fp-vs-oop-in-search-of-truth-7328b4de86e9#:~:text=FP%20has%20encapsulation%20on%20the,level%20of%20classes%20and%20methods.&text=It's%20true%3A%20usually%2C%20classes%20are%20of%20children%20classes%20as%20well>
- <https://courses.cs.washington.edu/courses/cse341/04wi/lectures/09-ml-modules.html>
- <https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f>
- <https://medium.com/weekly-webtips/imperative-vs-declarative-programming-in-javascript-25511b90cdb7>
- <https://codeburst.io/a-beginner-friendly-intro-to-functional-programming-4f69aa109569>

### Books

- Clean Architecture by Robert C. Martin
- Domain Modeling Made Functional by Scott Wlaschin
- Functional Programming for the Real World (With examples in F# and C#) by Tomas Petricek and Jon Skeet
- Hands-on Functional Programming with TypeScript by Remo H. Jansen

### Videos

- The Functional Evolution of Object-Oriented Programming
- [https://www.youtube.com/watch?v=AjR2Rc9wQ6s&ab\\_channel=StefanMischook](https://www.youtube.com/watch?v=AjR2Rc9wQ6s&ab_channel=StefanMischook)

## 24. An Object-Oriented Architecture

■ Architecture is meant to support the functional and non-functional requirements. A good architecture is one that does this but also adheres to the most well-known design and architectural principles. In object-oriented business applications, the most important non-functional requirement is the ability to test business logic complexity effectively. I argue that a layered architecture is the superior technique to accomplish this. It lets us write code in a test-first and domain-driven way.

Let's keep our eyes on the prize here.

We're going to use OO **the right way** to build out features. How do we get started? We're supposed to TDD our way through them, right? How do we kick-start that process?

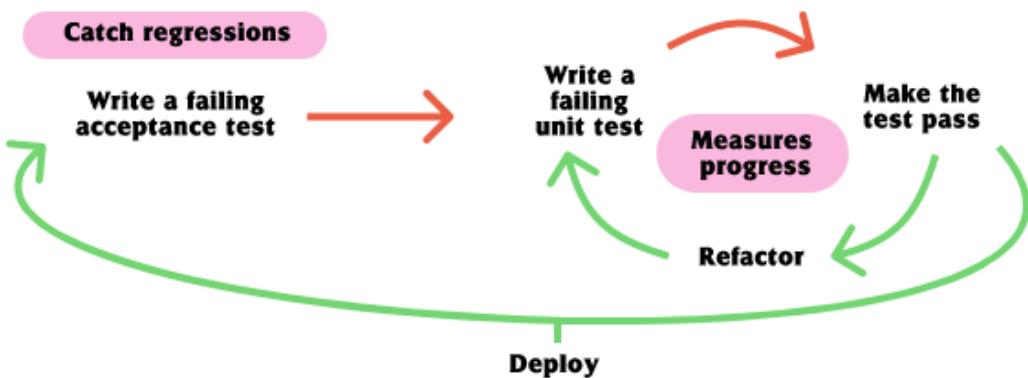
In XP, our goal is to TDD our way through a story. Using *Acceptance Tests*, we use a technique called *Double Loop TDD*.

It starts by first writing a failing acceptance test from the *outer boundary* of the application (now — what *outer boundary* means to you and which *form of tests* you use for your outer boundary *Acceptance Tests* could differ depending on our 25. Testing Strategies).

Regardless, after the automated acceptance test fails, we move in a layer, and continuously write *inner loop* red-green-refactor *Unit Tests* until the outer loop passes.

Once both loops are green, we're done with the feature. Best part? We have tests to prove it too. Now it's time to deploy it. We push it into a CI server, that builds the code, runs the tests, and deploys it if everything passes.

That's where we're trying to end up.



But we have to back up a little bit. How can we even write the first test if we don't have the infrastructure to support it?

Yes, we're going to want to install our testing tools, like Jest and Cypress, of course, but what about the *structure* of the code itself? What about the architecture? TDD is “doing the simplest possible thing that could work”, but we still need to think about design. In fact, when I do TDD, I'm *constantly designing*. On the whiteboard, on paper, in discussion, and in the patterns I use in my solutions.

Here's what we need to do before we can write the first test:

1. Understand the problem
2. Design a rough sketch of our initial architecture
3. Build, deploy, and test a walking skeleton (and use that as our test architecture)



The context of the first test (from “Growing Object-Oriented Software Guided by Tests — Steve Freeman”)

With respect to 1. *Understanding the problem*, we’ve already done a lot of work on that in 16. Learning the Domain, and 21. Understanding a Story . We understand the problem in the sense of data, behaviour, and namespaces (or domain events/aggregates, use cases, and sub-domains in a Domain-Driven Design sense) and we’ve uncovered a lot of the non-functional requirements too.

Skipping the second step for a moment, the third is to *Build, deploy, and test a walking skeleton*. The walking skeleton is essentially the **test infrastructure** in which we perform our acceptance-test-driven development loop within. In the next chapter, 26. The Walking Skeleton , we discuss getting this up and running.

Now, the second step — creating a broad-stroke design of the architecture. At this point, we have to ask ourselves: how do we determine the *shape* of the skeleton overall? What does the initial architecture look like ?

The focus of this chapter is to discuss the **design of an initial object-oriented architecture** that we’ll use \*\*\*\* to inform how we build the platform upon which we’ll kick off the TDD loop with.

## Chapter goals

Specifically, we will:

- Learn how to choose an architecture based on the functional and non-functional requirements
- Understand why we prefer a layered architecture instead of merely using MVC for business applications
- Discuss how and why to decouple core code from infrastructure code
- Understand how we handle requests in a layered architecture

## Deciding on an architecture

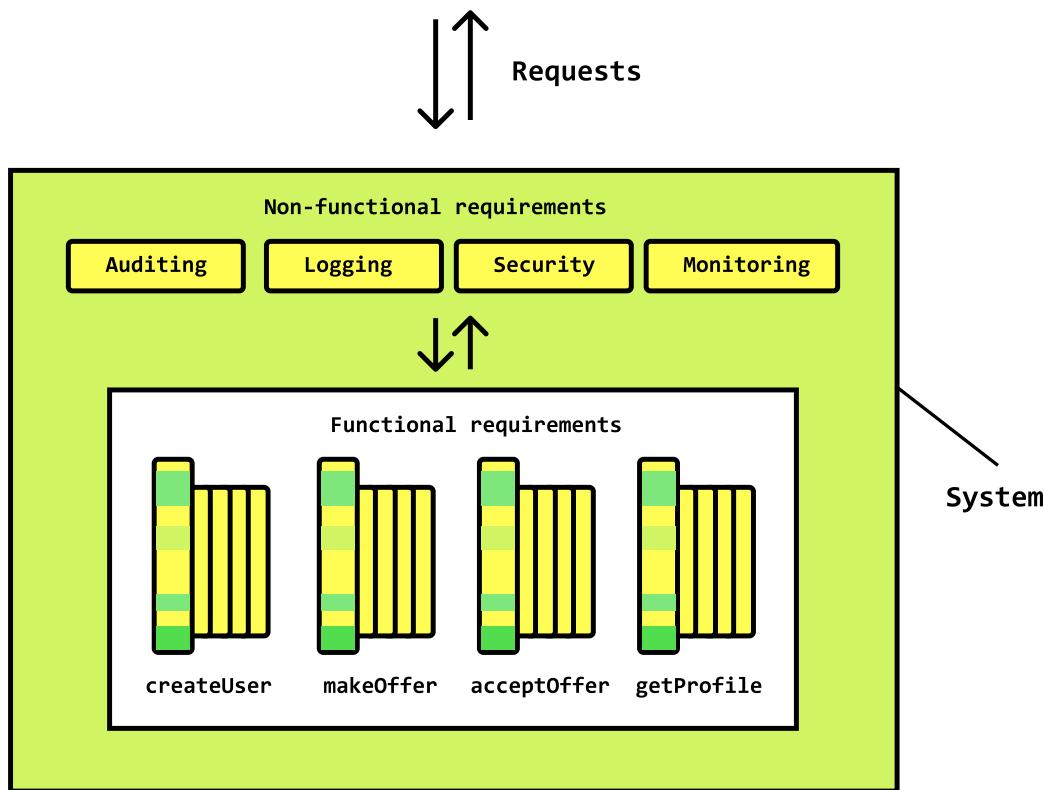
How do we decide on an architecture, overall? What are the *principles* at play that influence how to make this decision?

This is an *excellent question*. If you’ll remember, this question is one of the main reasons why I started writing this book. I was *personally* struggling with MVC while building an application that had a very complex domain. It would have benefited from a different architectural style.

Architecture is about the stuff that's hard to change if we get it wrong early on. That said, if I had to make a recommendation on how to make a decision overall, here's what I would do.

## I. Use the requirements

I know I've said it already, but the requirements are the biggest clues as to what you need. If you're worried about pre-optimization, remember YAGNI, and let the requirements dictate what is absolutely necessary.



How do functional requirements help me make architectural choices?

What's a functional requirement again? They're the type of requirement that specify that the "system shall do *something*". It's a feature that can naturally be tested *functionally*, where the same input (same preconditions and same input data) yields the same output every single time. **Functional requirements typically help us identify the external services, APIs, and dependencies that we'll need.**

The critically important pattern we use to represent these are appropriately named *Use Cases* (see Use Case). Consider the following ones:

- **Example: S3 - Amazon for file uploads.** Perhaps you want to do *direct-to-storage* instead of doing file uploads yourself. This would tell you that you have to think about how we're going to handle storage, and storage is a major architectural component.
- **Example: Real-time chat.** We could look into services that do this, or we could build it ourselves.

- **Examples: Transcribing home movies, image/video processing jobs.** Similarly to real-time chat, if we want to roll this all ourselves, it could be quite the endeavour. Perhaps, if it's our *core domain*, this is something we'd do. Otherwise, we can use some services and see how it works out.

What about non-functional requirements? How do they influence architecture?

Non-functional requirements specify that the “system shall be *a certain way*”. It considers the overall property of the system as a whole, not just related to a particular aspect, feature, or function.

These kinds of requirements can come from a variety of different places. Every couple of years, users come to expect better website experiences. Faster, more real-time, such, and such. Also consider what the customer — the stakeholders — need in a legal, business, and competitive contexts. For example, you'll sometimes find that stakeholders *absolutely need* certain software quality attributes like security, reliability, flexibility, speed, or compliance so that they can work with certain customers, retain users, respond quickly, and prevent disastrously costly situations. A good time to discuss this is when we learn stories.

Non-functional requirements often give us hints to the *patterns and architectural components (like queues, caches, etc)* that we'll need to design for the system to be a certain way.

Take the image/video-processing job example again for a site like YouTube or Instagram. Functionally, users need to be able to upload their media. Non-functionally, the customer wants users to be able to continue to move around the site while their media is being uploaded. To decouple the requests and the work to be done, this introduces the need for a *Job Queue* as an important architectural component. You see?

A few other common non-functional examples are:

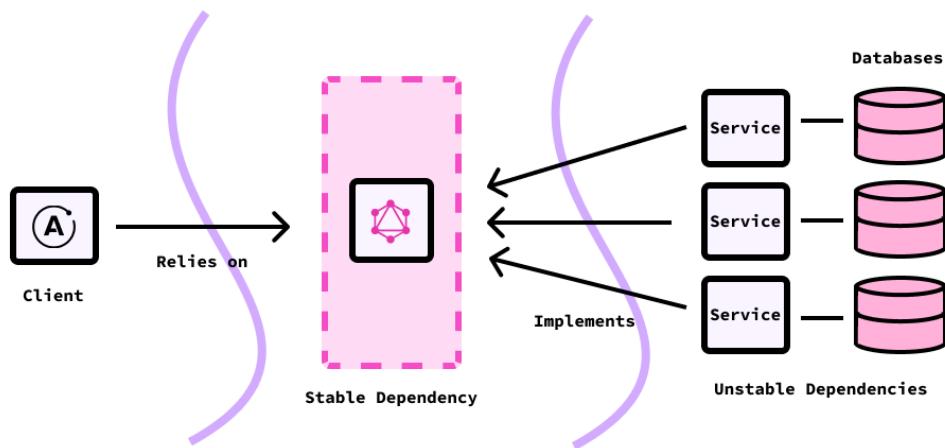
- **Example: Auditing.** Many companies need to maintain the history of everything that happened in the application for auditing or business analysis purposes. What's the solution here? Try architectural patterns like event sourcing, CQRS, or merely saving events in an event table. Alternatively, structured logging and efficient log-ingestion tools could work.
- **Example: Access control (security).** When we execute use cases, we *assume* that they're authenticated and authorized. As a cross-cutting concern, we'll need to devote a layer to determining if a user is allowed to perform a use case or not — or to what *degree*. There are many approaches we can take here.
- **Example: Reliability.** Perhaps this could refer to uptime — in which case, redundancy is a good idea. Maybe it also refers to having the proper mechanisms in place so that we can roll-back changes quickly, or prevent developer errors that can crash the application from getting into production. CIs, GraphQL schema checks, pre-commit hooks, and so on are good things to use.
- **Example: Tracing, monitoring, logging.** Once things are live, we need a way to understand how the system behaves, especially when things go wrong. Monitoring is generally the ability to aggregate and analyze metrics. Tracing is the ability to follow a program's flow and data progression. Logging is about writing error data and state transformations so that we can investigate later. All three are different but important. You will find various libraries and approaches to handling these things.

## 2. Consult design principles

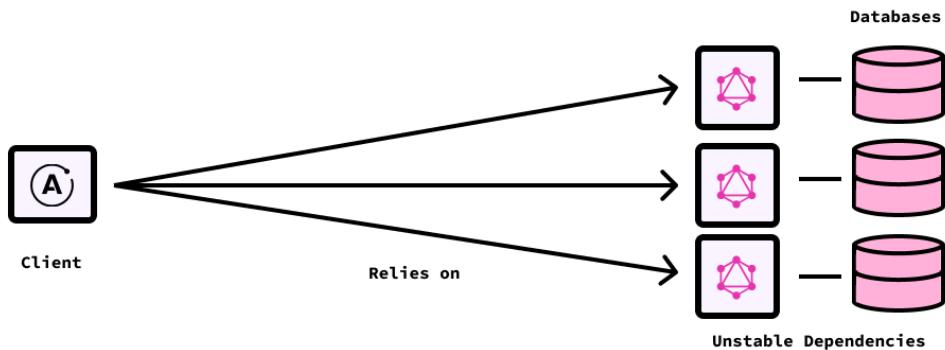
The next recommendation is to consult design principles.

Design principles all, in some way shape or form, help us make better decisions about coupling and cohesion — two ultimate measurements of software quality. While the requirements are the first thing we should look to, having a solid knowledge of design principles can help you avoid silly mistakes. We tend to think of design principles like SOLID at the *code level*. Luckily, they can just as readily be applied at the *architectural level*.

For example, if you wanted to use GraphQL, a smart architectural decision would be to create your GraphQL schema based on the needs of the client applications so that the client can rely on it, and the server can implement it (see [PrincipledGraphQL](#)). Then, deploy it separately from both the client and the server in the middle of your architecture and have it act like a traditional object-oriented interface. This is architectural dependency inversion. And as such, you get the same benefits: clients stay decoupled from server, and the internals of both sides can evolve independently, as long as the interface (the GraphQL API) remains the same.



A bad architectural GraphQL decision for building client applications would be to use an automatic GraphQL API generator that generates your API based on the structure of your database schema. If clients rely on the API directly, it couples them straight to the database. How? Well, any time your database schema (an implementation detail) changes, it also changes the GraphQL API. And if your GraphQL changes, that has the potential to break any clients that depend on it.



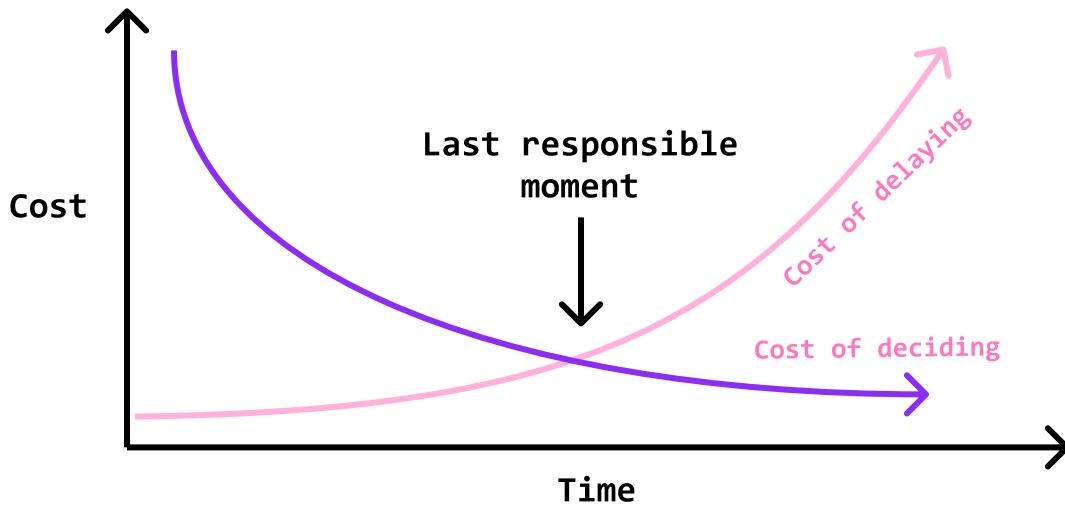
Design principles are invaluable. Just, pure wisdom. Learn them, do what we discussed in 3. A 5000ft View of Software Design and try to see how they operate at the class and architectural level — you'll start seeing *all* your designs in a higher resolution.

■ **Learn more:** We learn about the various design principles in Part VII: Design Principles.

### 3. Consult architectural principles

Architectural principles are more aligned to helping us evaluate decisions that we make throughout the duration of a project. They have more to do with the way we organize teams, how we save money, and how we'll actually spend our day to day time working.

For example, there are a set of principles referred to as The Principles of Economics. One of them is called the *Principle of Last Responsible Moment* (aka the *Cost of Delay*). It basically specifies that we should attempt to hold off on making big, crucial decisions for as long as possible until the last possible moment when we've narrowed down our options and are better informed.



Why? So that we can prevent making irreversible decisions until the cost of not making a decision becomes greater than the cost of making one. Make a decision too *early*: you'll likely be wrong and cause rework. Make a decision too *late*: miss opportunities and cause rework.

Another one that crops up *all the time* is called Conway's Law. It describes this phenomenon where when we build software, we first learn the different groups/teams/roles it serves, and then divide the app up into separate parts similar to how those groups *normally* communicate in real life. We actually saw this in 16. Learning the Domain with our *Event Storming* exercise and the way the subdomains were structured.

These are all good principles specifically about the high-level stuff.

**Learn more:** We discuss architectural principles in 8. Architectural Principles from Part VIII: Architecture Essentials.

#### 4. Consult architectural patterns

As we covered in 3. A 5000ft View of Software Design, design patterns are solutions to common problems at the *class-level*. Architectural patterns are similar, except that they're broader in scope.

Some patterns are better suited to problems based on the domain, the type of software being built, and the non-functional requirements than others. For example, if you had to build speech recognition or vehicle tracking software, the *Blackboard pattern* is one that we as a software community have discovered works particularly well for that problem.

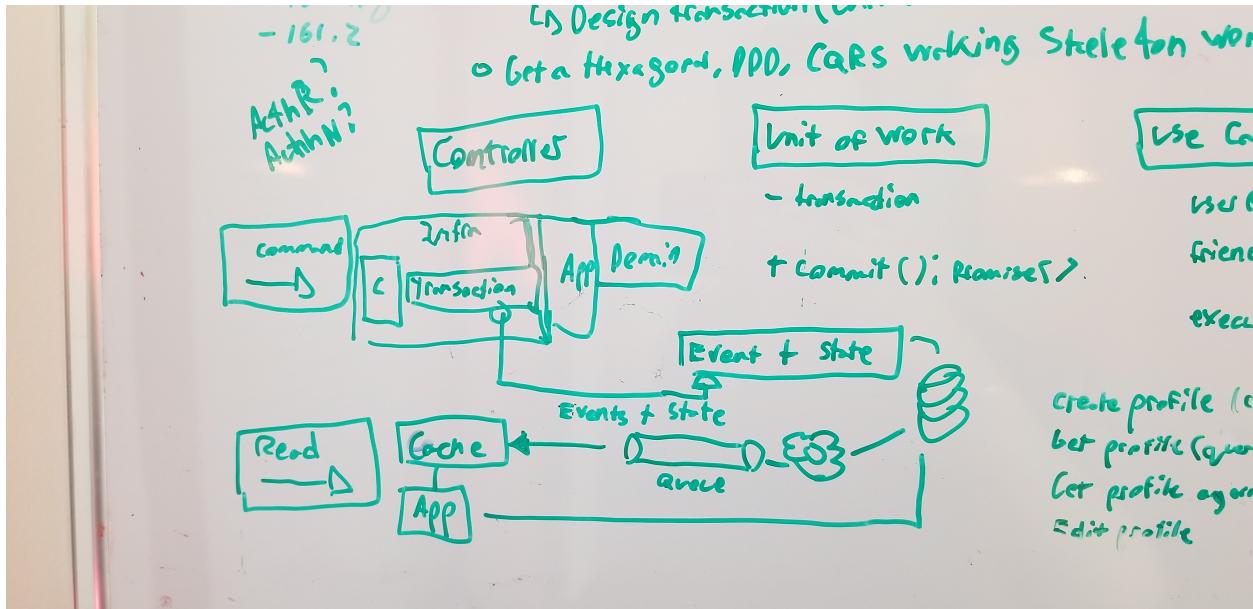
My advice here is to merely be aware of the top ten architectural patterns we discuss in this book, know what problems they're uniquely equipped to solve well, and then study them in more depth when the opportunity presents itself.

**Learn more:** We discuss the most common architectural patterns in 10. Architectural Patterns from Part VIII: Architecture Essentials.

## 5. Draw it out on a whiteboard

I'm not exactly sure what it is about drawing your designs out on a whiteboard (perhaps the fact that you can't BS yourself), but if we're **unable** to at the very minimum express what our design should look like spatially or visually, then it's unlikely we'll be able to express what it should look like in code.

Use a whiteboard to draw out your designs so that you can have a big-picture understanding of what it is that you'll be building.



## Improving your architecture skills

If you get good at extracting user stories (especially the non-functional ones) from customers, you'll be in a really good position to know what kind of system you'll need to build.

As I outlined above, there is some baseline knowledge required here. To improve your early design decisions — your architectural decisions — you need to know some theory. Learn design and architectural principles. Learn about different architectural styles and the specific patterns you can implement with them. Then learn which non-functional requirements they tend to be good at meeting. We'll get to all of this later on in the book. Baby steps.

## An object-oriented architecture for business applications

Let's discuss an architecture for the 95% of us that are web developers building task-based UIs and business applications using object-oriented programming.

I've frequently said that we're trying to build testable, flexible and maintainable software. Let's dive into that again quickly. Why are we doing that? Well, because *testability*, *flexibility*, and *Maintainability* are developer-focused non-functional requirements. That's how we do good work, right? We need an architecture to support these things.

You'll be intrigued to know that the broad category of architectural style capable of giving us these three things (specifically *testability*) goes by the name of Layered Architectures.

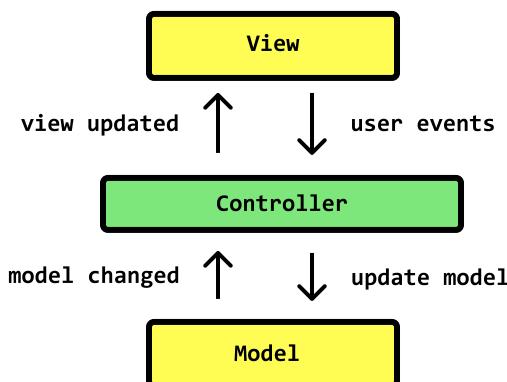
The most well-known pattern (implementation of the style) is known as MVC. I guarantee you've used it.

### MVC: The trivial layered architecture

In MVC, the model is everything above the database up to, but not including the controller. The model contains business logic and the types that make up the application core (the functionality).

The view is the front-end, which in web applications is often CSS, HTML and JavaScript.

The controller sits in between the controller and the model, handling requests from the view by passing it off to the model, which in turn, interacts with the database. The controller then formats a response to pass to the view.



The model-view-controller architectural pattern (a type of layered architecture implementation).

The primary benefit of *any* layered architecture like this is Separation of Concerns. That's not specific to the MVC pattern (there are other layered patterns as well). In a layered architecture, we isolate layers from each other, which means that they can be changed independently. In this, the API becomes the contract (the boundary) that separates the layers from each other.

Specifically, MVC gives us the three non-functional requirements we're looking for in these ways:

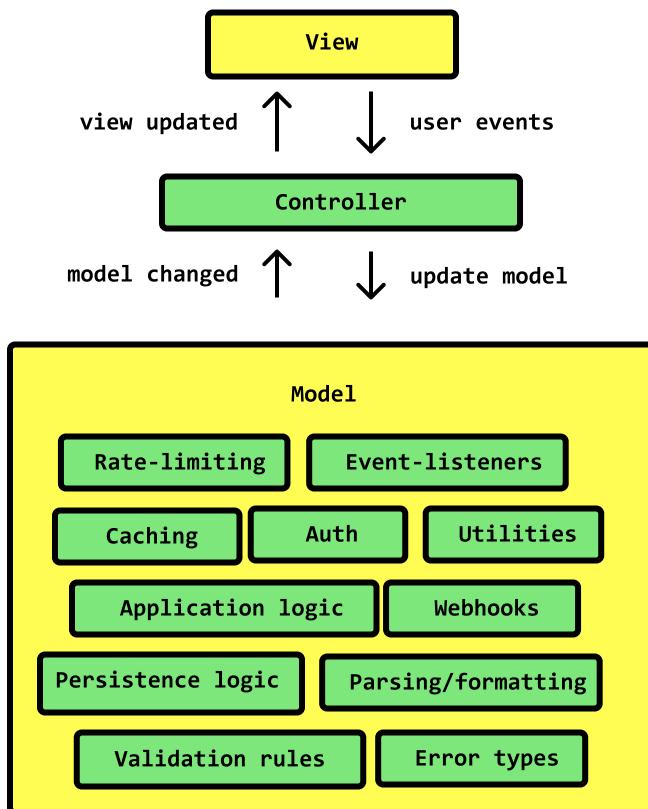
- **Flexibility** — Easier to update and change each layer separately since a proper boundary ensures that changes in one layer don't affect the other.
- **Maintainability** — We can assign different teams to the layers — frontend team and backend team. Each team can focus on their own respective part of the product, which reduces cognitive load.
- **Testability** — We can test the front-end separately from how we test the back-end.

## What's the problem with MVC?

The biggest problem we face with MVC has to do with the *model* portion. **The testing options are limited.** What do I mean when I say that? When most of us learn MVC and build out the model component, we typically do so in a way that mixes all kinds of logic and concerns in such a way that is not easily testable. It's not uncommon to find controllers that have validation logic, business rules, event handling logic, parsing & formulating responses, web server logic, and persistence code all in the same place. That is, **trivial implementations of MVC promote a mixture of core code and infrastructure code.**

*Core code* is code that has to do with the features and domain logic. It knows nothing of databases, caches, or web servers. Infrastructure code is the code which actually connects the core code to the real world through these various technologies, database adapters, system clocks, and so on. *Core code* is the family jewels, the most valuable, and what gets written through <sup>16</sup>. Learning the Domain. Infrastructure code is what makes it buzz and whirr for real. We'll continue to elaborate on *core* and *infrastructure* momentarily, but just know that *core* code is the code that contains the most value.

This mixture tells us that the model does a *lot* of stuff. How do we organize it effectively? Often, we do so arbitrarily. We write classes called SomethingService or SomethingHelpers. We tend to package by infrastructure (see 6. Organizing things).



Honestly, if we're not practicing TDD, we can get pretty far with this because it makes no

difference to us what category a certain type of code is. *If we can merely get the controller hooked up to the database and return a status code, we're done!* With this thought pattern, perhaps our idea of testing is to merely add a few *End-to-End Tests* or to test that things work properly by going through the public API and hitting the entire backend as if it's a black-box. In tests like this, we test from the outside, going through the RESTful or GraphQL API and looking at the results. Yes, these tests can be expensive in the sense that they take long to run and are complicated to set up, but they can work. They may not really reveal *why* something is failing internally, but they do still provide value.

In a test-first approach, we strive to do much better than that. We write fine-grained tests — white-box tests (that test the inside) — rather than just testing at the boundary. Why? A lot can go wrong on the *inside* of a system. As we've discussed in 23. Programming Paradigms, a single message (method call) in an OO system can ask any number of objects to do some work (which can *also* then ask *more* objects to do work). This means it's possible to have quite a few different moving parts and state to account for. We shouldn't assume that just because things seem to work on the outside, that we're completely void of issues on the inside. That's naive.

The way to gain confidence in our code is to strive to have as much of it covered by tests as possible. We want low-level *Unit Tests* and high-level *End-to-End Tests*. We want options. Options let us create a better testing strategy.

When we *Acceptance Test*, we test the *features* of our application. Some phronimos developers recommend you write your *Acceptance Tests* to be fast, cheap, and by definition, as I've stated a little earlier — this sounds a lot like the idea of a *Unit Test*.

The big problem that most of us face in realizing this lofty idea of fast *Acceptance Tests* as *Unit Tests* in a basic MVC architecture is that **we can't write unit tests to test features because (fast) core code is coupled to (slow) infrastructure code.**

## You can't unit test infrastructure

One of the things that makes testing hard is that everyone has a different name for everything. In lieu of that, perhaps it would help to understand a *Unit Test* by defining what it is *not*. According to Michael Feathers, a test is not a unit test if:

1. It talks to the database
2. It communicates across the network
3. It touches the file system
4. It can't run at the same time as any of your other unit tests
5. You have to do special things to your environment (such as editing config files) to run it.

If you zoom out, these rules are all saying the same thing. You can't unit test **infrastructure** concerns. If your code has infrastructure mixed in and coupled with the code you want to unit test — the (pure) application core — then it's impossible. All the tests you write will be slow, pull expensive infrastructure along with it for the ride, require complex spin-up, spin-down, and resetting every test, and tests that should have taken milliseconds will end up taking seconds (or worse, minutes).

Take a look at the following code I copied off of the internet of a RESTful API build with Express and see if it's something we can test with a *Unit Test*.

```
// routes/exercise.ts

// Assume that `Person` is a Sequelize (ORM) model
import { Person } from '../models'

...

app.post('/api/exercise/new-user', function(req, res) {
  let username = req.body.username;

  Person.findOne({ username }, (err, findData) => {
    if (findData == null) {

      //no user currently, make new
      const person = new Person({
        username,
        exercise : []
      });

      person.save((err,data) => {
        if (err) {
          return res.json({error: err});
        }
        return res.json({
          "username": findData.username,
          "id": findData.shortId
        });
      });
    } else {
      // username taken, show their id
      return res.json({
        error:"This username is taken",
        "id":findData.shortId
      });
    }
  });
})
```

If you guessed *no*, then you're correct. Why? It breaks rule #1 — it talks to a database. Also, since the only way to execute this is to go through an Express web server, it breaks rule #2 as well. So, not only is business logic *implicitly* mixed in here (we implicitly express a `UsernameTaken` error), we're passing back database errors directly (with poor HTTP response consistency and concern-invoking from a security perspective), but even if we wanted to *Unit Test* this, we couldn't.

I want you to start to see what can and cannot be tested with unit tests. If you go and look through your old projects, if you were writing code like I was, you'd find that pretty much the only thing that could have been unit tested were utility classes like the following:

```
// shared/utils/textUtil.ts

export class TextUtils {

    public static toTitleCase (val: string): string {
        return val.substr(0, 1).toUpperCase() + val.substr(1).toLowerCase();
    }

    public static contains (text: string, match: string): boolean {
        return text.indexOf(match) !== -1;
    }

    public static isNumber (text: string): boolean {
        return !isNaN(Number(text));
    }

    public static pad (n: number | string, width: number, z?: string) {
        z = z || '0';
        n = n + '';
        return n.length >= width
            ? n
            : new Array(width - n.length + 1).join(z) + n;
    }

    public static kebabCase (str: string): string {
        return str
            .replace(/([A-Z])([A-Z])/g, '$1-$2')
            .replace(/([a-z])([A-Z])/g, '$1-$2')
            .replace(/\s+/g, '-')
            .toLowerCase()
    }

    public static replaceAll (str: string, match: string, replacement: string): string {
        let newStr = str;

        while (newStr.indexOf(match) !== -1) {
            newStr = newStr.replace(match, replacement);
        }

        return newStr;
    }
}
```

You have to be really diligent here. You'll even find that if you use the *system clock* throughout your codebase, that's inadvertently using code that relies on I/O. It changes every-time you run it. That means you'll also find it hard to test (see Clock (time)).

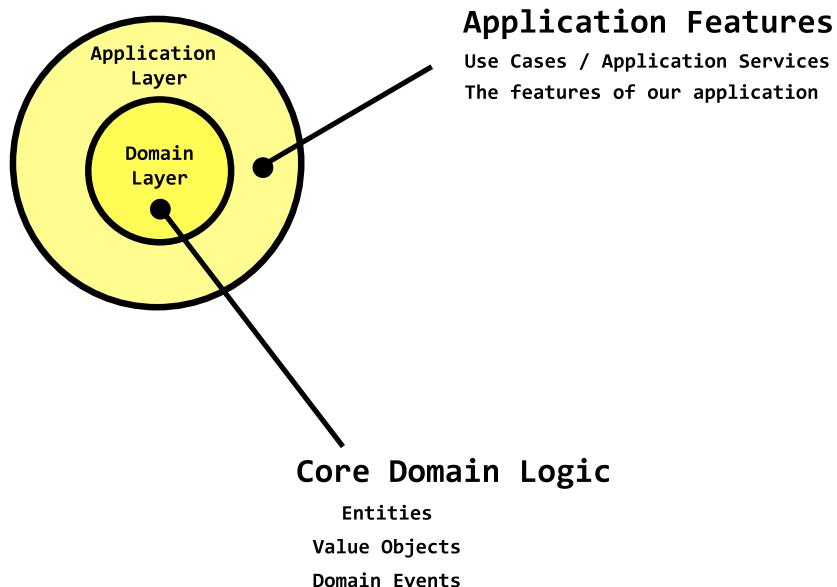
```
const createTask = (taskDetails) => {
  // ...
  return {
    // Other properties
    startedAt: new Date()
  }
}
```

Evidently, the most valuable code (which is the the code containing our features) cannot be unit tested in traditional MVC architectures. And maybe you're OK with that, but I'm not. And if you're building a backend, you shouldn't be either, in most cases.

I often work on projects where the most important code gets very complex, very fast, and I like having a safety net to fall back to a point where I know things are working the way I intended them to.

### Core code vs. infrastructure code

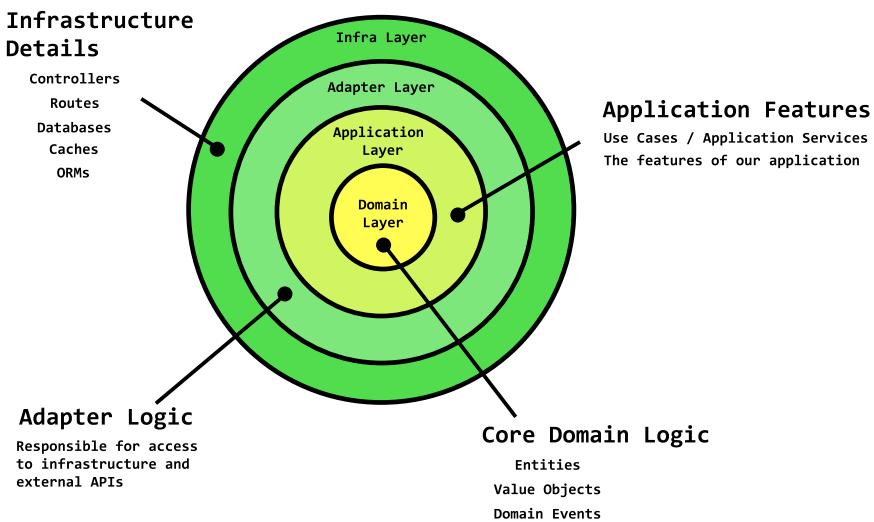
*Core code* is the heart of our software. In more robust layered architectural patterns like the *Clean* or *Onion Architecture*, core code refers to the *application* and *domain layer*.



The application layer is a layer of code responsible for the application logic (which is what we've written as pseudocode in 21. Understanding a Story ). The domain layer then, is responsible for the structure of data, validation logic, invariant logic, and core business rules. These are things that have more to do with the *domain itself* rather than the *application* rules.

Core code has zero concrete dependencies to concerns like databases, web servers, and so on. It's the first place we should think about unit testing. And if we keep it pure, we can.

What other kinds of logic are there? Obviously, our application doesn't do much unless it's connected to the real world and a real database. So, there's logic responsible for fetching things and saving them to a database. There's also logic for transforming domain objects between the format it's saved as in the database, the format it's coming in as through a web request, and the format it's used as in our actual business logic. There's web server logic that handles the request and sends control off to the right place. There's a lot more too. This type of logic is infrastructure logic, and it belongs to the *infrastructure layer* in a clean architecture.

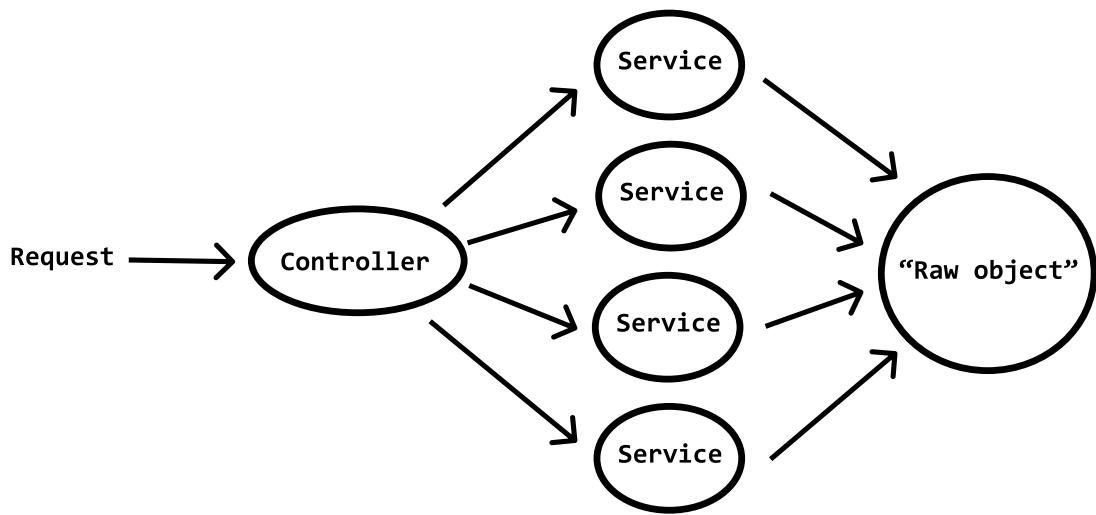


## How requests work in an OO architecture: Transaction scripts vs. domain models

How exactly does a web request work? We have this idea of *layers*, but what does that really look like?

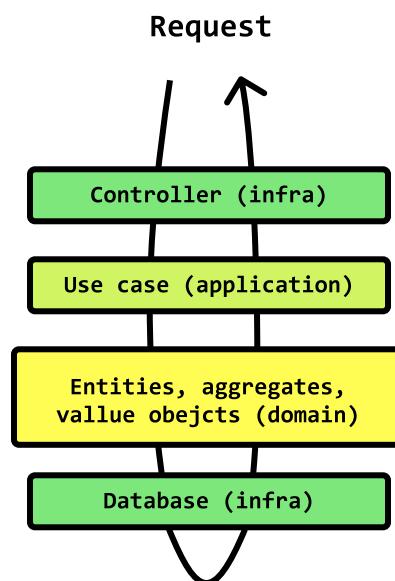
There are two ways to handle web requests. You have the *Transaction Script* (also sometimes referred to as the *Anemic Domain Model*) pattern and the *Domain Model* pattern.

The *Transaction Script* pattern is just procedural code, right from the controller. As is common in *Structured Programming*, each web request is a procedure of code that may call other subroutines (or \*\*service classes) to complete the request. The core code and business rules are split all over the subroutines/services. Some may also live in the controller and some may live in the services.



### Transaction Script approach.

With the *Domain Model* approach, features are vertical slices that cut through each layer. A web API request coming in starts at the controller. The controller does controller things and then passes a *Data Transfer Object* (the data for the request) into an application layer *Use Case*. From within the *Use Case*, we use *Repositories* and \*Services \*\*\*\*\*to retrieve the appropriate domain objects and potentially alter their state in some way. If all is good, we save the domain objects to the database and the request returns.



Domain Model approach.

Here's the skeleton of what a *Use Case* could look like. See if you can notice what's wrong with it.

```
// Concrete dependencies
class UserRepo { ... }
class TraderRepo { ... }

export class MakeOffer extends UseCase<MakeOfferInput, MakeOfferResult> {

    constructor (
        private userRepo: UserRepo,
        private traderRepo: TraderRepo
    ) {

    }

    public async execute (data: MakeOfferInput): MakeOfferResult {
        // Use case logic here
    }
}
```

This may seem a bit odd. When you look at the order of the dependencies this way, it's as if the application layer concern (the *Use Case*) actually relies on the database, which is an infrastructure layer concern. Why is the database the deepest layer in the request? Isn't that what we're trying to avoid — having the core code rely on infrastructure code?

How are we supposed to unit test the features (the core code) if it depends on a database?

The key is *dependency inversion*. Safe polymorphism. OO's gift to us.

### Dependency inversion & the dependency rule

Someone once said that most problems in software design can be solved by adding a layer of indirection. And that's exactly what we'll do.

Instead of relying directly on repositories, we rely on *abstractions* to them. Interfaces.

```
// Abstractions
interface IUserRepo { ... }
interface ITradeRepo { ... }

export class MakeOffer extends UseCase<MakeOfferInput, MakeOfferResult> {

    constructor (
        private userRepo: IUserRepo,
        private traderRepo: ITradeRepo
    ) {

    }
}
```

```

public async execute (data: MakeOfferInput): MakeOfferResult {
    // Use case logic here
}
}

```

Why? Because that's how we accomplish dependency inversion. Now, we can pass in *real* concrete dependencies.

```

const sequelizeUserRepo: IUserRepo = new SequelizeUserRepo();
const mongoDbTradeRepo: ITradeRepo = new MongoDBTradeRepo();

const makeOffer = new MakeOffer(sequelizeUserRepo, mongoDbTradeRepo);

```

Or, we can pass in fake, in-memory mock objects for use during testing. Both are valid. This is how we get options.

```

const mockUserRepo: IUserRepo = new MockUserRepo();
const mockTradeRepo: ITradeRepo = new MongoDBTradeRepo();

const makeOffer = new MakeOffer(mockUserRepo, mockTradeRepo);

```

The *adapter layer* in our layered architecture describes the contract \*\*that infrastructure components must satisfy in order to be compatible with our application (and evidently, domain layer) code. The adapter layer pretty much contains the definitions (interfaces/abstractions) for how to write the *adapters* so that they can connect to our application's *ports*. The adapter layer is what connects our code code to the infrastructure layer, all in a loosely coupled fashion.

There's form to this technique. It's called **the dependency rule**. It specifies that an inner layer concern cannot rely on a concern from an upper layer. That means it's a violation for domain layer code to rely on application layer code. It's also a violation for application layer code to rely on infrastructure. However, it is completely valid for infrastructure code to rely on application and domain layer code. And that's what we do. That's why our controllers look like this:

```

export class CreateUserController {
    private useCase: CreateUserUseCase; // use case is lower than infra - OK

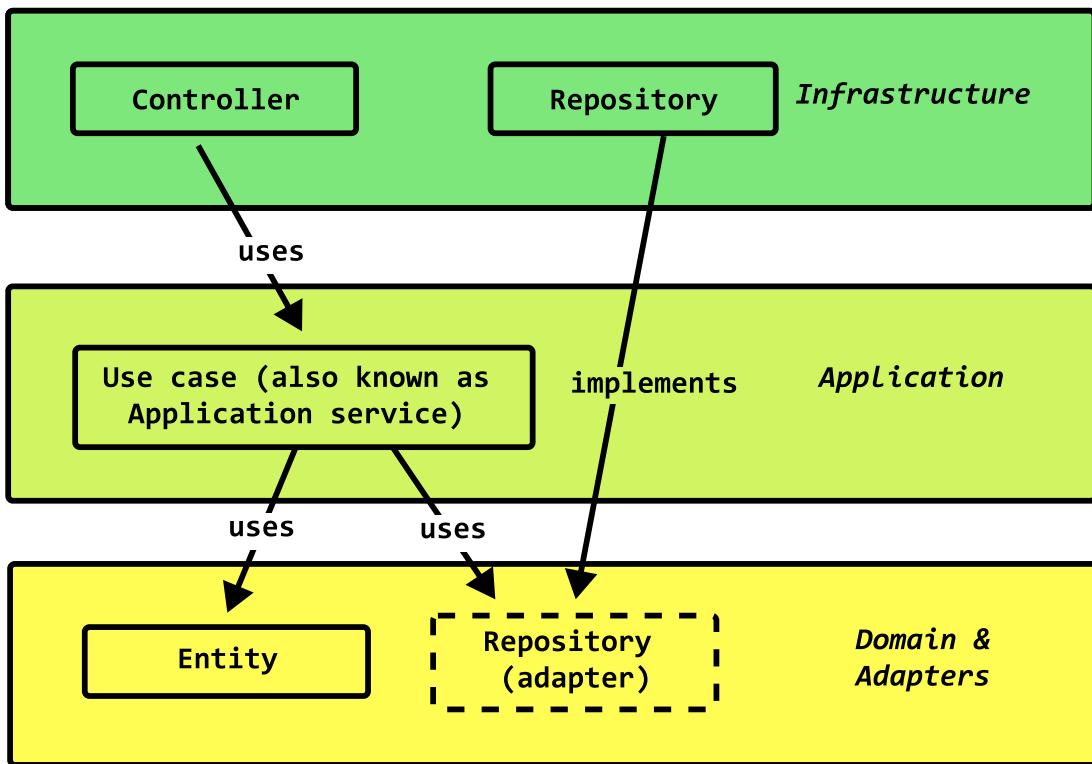
    constructor (useCase: CreateUserUseCase) {
        this.useCase = useCase
    }

    public async execute (req, res) {
        // Validate
        const result = await this.useCase(req.body);

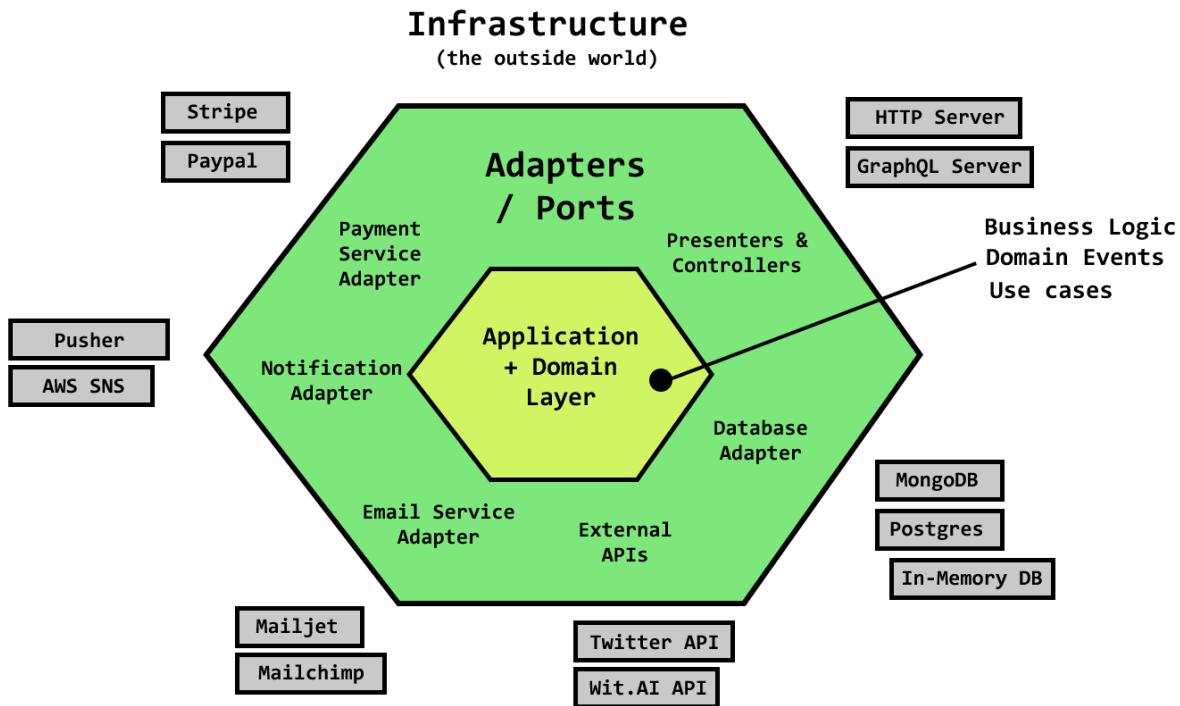
        // Etc
    }
}

```

The general layout for how our tactical patterns interact with each other between layers in a layered architecture is as follows:



In similar variations of the *Clean Architecture*, like Ports & Adapters or the Hexagonal Architecture, we refer to the interface as a *port* and the concrete implementation as an *adapter*. This is how we create a *plug-in* architecture where any infrastructural component can be configured to work with our application core.



A general understanding of this theory is good for the rest of *Phronesis*, but don't fret if you don't fully grasp it 100%. We truly know it if we do it. In later chapters, we'll walk through it step-by-step. You can also see a real-world example of how this works at <https://github.com/stemmlerjs/ddd-forum>.

### Why decouple core from infrastructure code again?

Let's backtrack. Why do we want to do this? The two big reasons are that:

- It give us a strong, stable, technical foundation to \*\*focus on the essential complexity and perform **domain-first development**
- It opens up an entire world of testing possibilities, which **makes it much easier to perform test-driven development**

But it also allows us to:

- Delay the decision on exactly which type of web server, database, transactional email provider, or caching technology until it's absolutely necessary to decide. We can always use an in-memory implementation of the *port* for initial development efforts.

### Deployment as a modular monolith

How will we organize the code? Should we start with microservices? No, we don't want to start with the idea of microservices right off the bat. I think starting off with **well-organized Modular Monolith** until it reaches the critical mass and the socio-technical opportunity at which it makes sense for us to break into separate teams and deployments is good advice.

We apply the principles of 6. Organizing things and package subdomains into modules which then contain the use cases, packaged by feature.

In Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, (more specifically, Context Mapping a Modular Monolith), we go into more detail as to how *Subdomains* and *Bounded Contexts* are realized in a monolithic deployment.

## Summary

- Before we can kick off the TDD loop, we need to understand the problem, design a sketch of our architecture, and then build, deploy, and test a walking skeleton. The walking skeleton is the framework upon which we'll deliver value with fully acceptance tested features in.
- Functional requirements fundamentally tell us the types of components we'll need. Non-functional components give us hints to the correct architectural patterns and tools we should use. To improve your architecture skills, learn Part VII: Design Principles, 8. Architectural Principles, 10. Architectural Patterns, and seek to understand what *non-functional requirements* tools and technologies (like Kubernetes or Apache Kafka) are being used to solve.
- The biggest challenge with traditional MVC alone is that it promotes the use of the *Transaction Script* pattern which results in a mixture of code and infrastructure concerns, reducing our ability to write test-first and domain-first code.
- We can separate the concerns of core and infrastructure by enforcing an adapter layer with dependency inversion. Such an architecture opens up the door to better 25. Testing Strategies and layer of code that purely represents the essential complexity.
- I recommend deploying your application as a single bounded context and organizing your subdomains into well defined modules with explicit communication boundaries. That is, I recommend starting with a modular monolith.
- By using the *Shared Kernel* context mapping pattern, our subdomains rely on the same database and infrastructure.
- The primary method of communication between subdomains is through the form of *Domain Events*. We'll rely on a queue or event listener/handler of some sort to accomplish this.

## References

### Articles

<https://towardsdatascience.com/5-key-principles-of-software-architecture-e5379cb10fd5>  
<https://www.simplethread.com/code-generation-should-be-the-nuclear-option/>  
[https://egertonconsulting.com/improving-reliability-performance-through-good-design/?doing\\_wp\\_cron=1629047231.3773291110992431640625](https://egertonconsulting.com/improving-reliability-performance-through-good-design/?doing_wp_cron=1629047231.3773291110992431640625)

## 25. Testing Strategies

■ A layered application opens up the number of testing possibilities. Before development starts, it's a great idea to have an understanding of what you'll need to test and how you'll write those tests.

You want to know *what* to test and *how* to test it. For example, what should we test with respect to the database? What about the GraphQL API? What are we testing exactly? What about our acceptance tests? At which *scope* should we write them? End-to-end? Integration?

There are a *lot* of questions here. These are questions that are often times left unanswered until it's far too late.

In the previous chapter, we took our understanding of how OO works and, at a high-level, saw that separating core code from infrastructure code lets us:

1. Use a *Domain Model* to write pure, domain-focused code and
2. Write different types of tests for different aspects of our codebase in a decoupled way.

In this chapter, we're going to examine that second benefit at a high-level.

## Chapter goals

Specifically, we will:

- Briefly cover the different types of tests one can write
- Discover all of the various concerns that need to be tested to gain confidence that our code is working correctly
- Discuss a testing strategy for decoupled applications that leads to confidence in our code and a high test coverage percentages

■ **Note:** This chapter is focused on *what to do*, not *how* to do it. See Part X: Advanced Test-Driven Development for detailed examples of how to write each test.

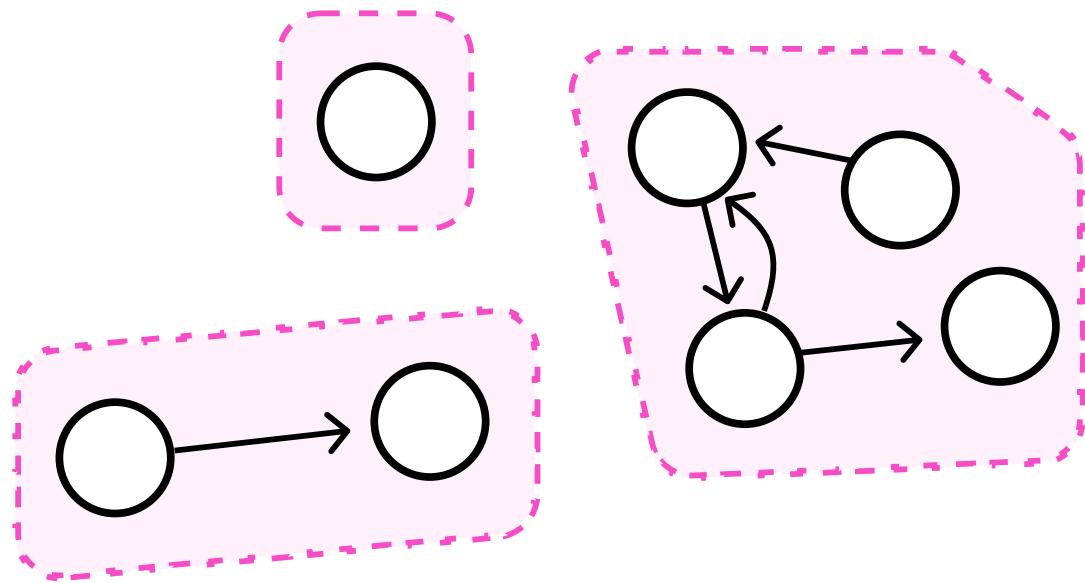
## Test types

Let's run through the different definitions of each type of test.

### Unit

We defined a unit test previously based what Michael Feather says it was *not* (it doesn't talk to a database or file system, doesn't communicate via the network, doesn't need to run at the same time as other tests, and doesn't require special setup to run).

Some developers call tests unit tests when you're only testing a single function, class, or object at a time. That sometimes works, but if you think back to our understanding of OO, sometimes we want to test a class that talks to another one.



**■ Subject under test:** Since objects can refer to each other and can be dependency injected with new ones at run time, we refer to the specific object we're interested in testing as the *Subject Under Test (SUT)*.

These are all unit tests. To me, what really makes a unit test a unit test is what Michael Feathers said — but even more simply, **a unit test is a test that tests the contract of a class or function that is completely core code, never infrastructure code.**

## Integration

While unit testing is fairly agreed upon, an integration test means a lot of different things to a lot of different people.

Kent C. Dodds says that an integration test is one “confirms that several units of code work together in harmony”. That’s true, but I think it needs to be more specific. It still sounds like our *unit tests*. This is highly opinionated, and you’ll be hard pressed to find a single answer anywhere, but testing how two different modules of *core code* work together isn’t necessarily an integration test. That could be a unit test.

Generally speaking, **an integration test is one where we want to make sure that some other body of code, system, or dependency that we haven’t personally written and are not responsible for, works with our code.**

We are testing the *integration* with our code and some other maintainer, vendor, library, or API’s code.

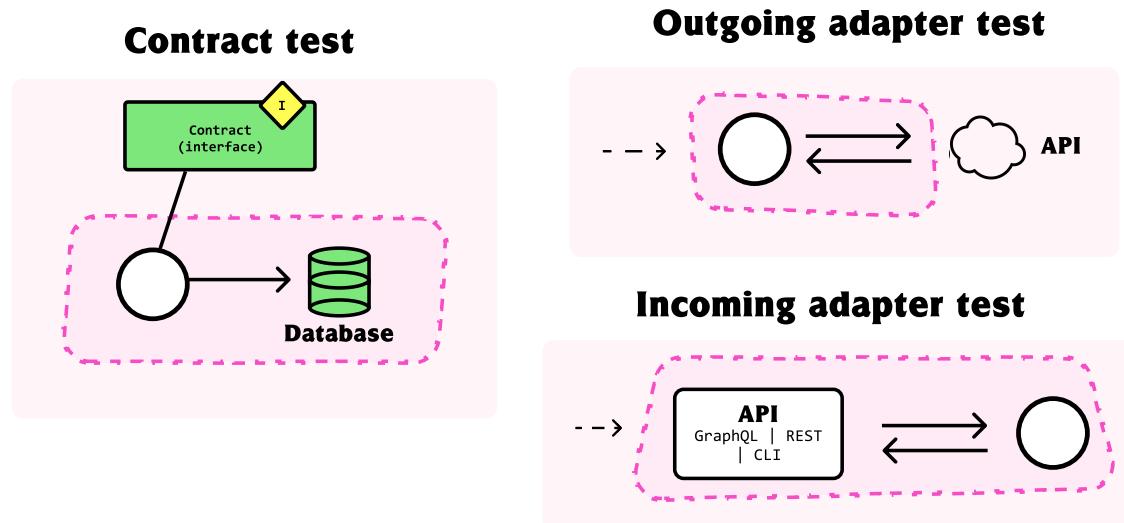
For example, on the front-end, consider we `npm install`’d a library for on-screen notifica-

tions after you successfully complete something or an error happens. An integration test may want to make sure that the notifications are visible on the screen when called. Why wouldn't it work? Hey, you just downloaded it from some person off the internet. Who knows if it works. And who know if it will *continue* to work if you keep it up to date? People deprecate things. People change their public APIs. So an integration test on the front-end would confirm that our code can still work with their code.

On the backend, we have even more external dependencies (adapters) to make our application work. Depending on if the dependency is owned by us (like a database), shared by the entirety of the company (like a bus or a queue) or completely external to us (like Stripe, Auth0, PayPal or some other external API), we may call our integration test something vastly different.

A few different flavours of integration tests are:

- **Contract tests** — a test that ensures that a concrete implementation of an adapter adheres to everything the interface says it must do (example: do all the *Repository* methods work?)
- **Outgoing adapter tests** — a test that ensures that an external dependency can be connected to, and if possible, verifies that we can perform the operations we need to against it (example: can I connect to the Stripe API? What about AWS S3? Do file uploads work properly?)
- **Incoming adapter tests** — a test that ensures that an incoming request handler (like a RESTful API controller, GraphQL API, or websocket connection) calls the correct application core code.



Notice how each of these do slightly different things, but they are still ensuring that *outside* code that we didn't write — infrastructural code — works with ours.

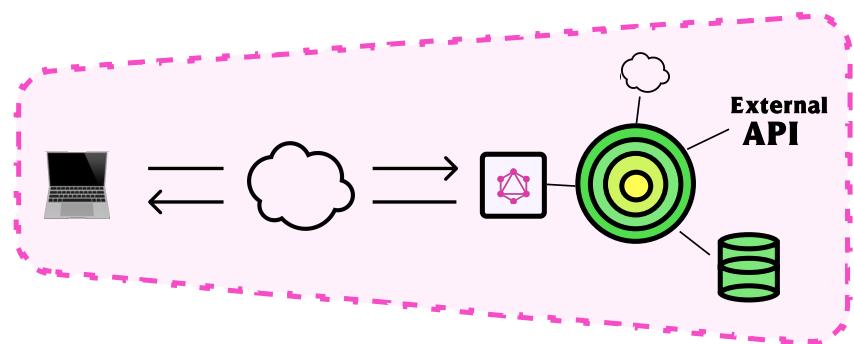
### End-to-end

An *end-to-end* test is a test written from the perspective of a real user. It exercises the entirety

of the system by moving around through the UI and accomplishing tasks similar to the way a real person would, by *interacting* and *perceiving*.

- **Interacting:** Button presses, keyboard clicks, what have you. We want to simulate realism so we only test the UI through what the user can do.
- **Perceiving:** We verify correctness based on what we can observe from the UI such as the fact that “we can see the list of todos” or that we “end up on the /dashboard page after logging in”.

It should be apparent that there is a *lot* that cannot be tested this way. For example, we can’t test the nuances of how our client-side auth logic works under the hood. Or maybe we can, but because end-to-end tests are supposed to be focused on what a *user* does, it’s clear that that’s something that should probably be covered by a *unit test* instead. End-to-end tests are solely focused on what a human being can do and see based on their interactions.



■ **Flakiness & fragility:** Flaky end-to-end tests are ones that seem to work sometimes, but randomly fail other times. This used to be a problem that existed due to the tooling that was available to write such tests. However, in recent years, end-to-end tooling has improved substantially and flakiness isn’t as much of a problem. These days, end-to-end test fragility is the bigger problem. It is lesser the fault of the tooling and has to do more with how we structure our tests. Traditionally, developers write end-to-end tests directly against HTML elements, hooking into attributes like `id` and `className` to *interact* and *perceive* them programmatically. Referencing HTML attributes directly in your test implementations breaks BDD principle that we should “test against behaviour, not implementation” and the general design principle that states that we should Encapsulate what varies. In Stable End-to-End tests with the Page Object Pattern from Part X: Advanced Test-Driven Development, we again learn how to use dependency inversion to create a contract to enforce stability — this time, we create stability between our tests and our implementations.

## Acceptance

By now, you know that acceptance tests are customer-specified tests that we use as receipts to create the definition of done for a user story. As expanded on in 22. Acceptance Tests, they’re best written in Given-When-Then format (formally known as Gherkin) like this:

Feature: Sync Notion tasks to Google Calendar

Scenario: Sync new tasks to Google Calendar

Given there are tasks in my tasks database

And they don't exist in my calendar

When I sync my tasks database to my calendar

Then I should see them in my calendar

Scenario: Updating tasks

Given tasks from my tasks database already exist in my calendar

And there are some changes to the tasks in my tasks database

When I sync my tasks database to my calendar

Then I should see the updated tasks in my calendar

Scenario: Deleting tasks

Given tasks from my tasks database already exist in my calendar

And I have deleted them from my tasks database

When I sync my tasks database to my calendar

Then I should not see the tasks in my calendar

Scenario: Preventing duplicates

Given tasks from my tasks database already exist in my calendar

When I sync my tasks database to my calendar

Then I should not see duplicate tasks created in my calendar

It's not entirely necessary to write your acceptance tests like this, but it certainly helps to because the customer can easily read it.

The last important thing to discuss about acceptance tests are the *scope* that we write them at.

1. **End-to-end acceptance testing:** Do we write them at the *front-end* as end-to-end tests going from the front-end to the back-end?
2. **Integration-style acceptance testing:** Do we write them as integration tests (from the API), going through a GraphQL or RESTful API into the backend?
3. **Coarse-grained unit test style acceptance tests:** Or do we write them as unit tests (through the *Use Cases*), testing only the core code?

There's no single correct answer here. While it does all depends on your testing strategy and how confident you feel that your other bases are covered, the reality is that each approach has advantages and disadvantages. For example, while end-to-end tests cover more ground, they are slower, harder to set up, and can't verify everything — how do you verify that an *account verification email* was sent after a *createUser* use case in an end-to-end context?

Based on the writings from Thomas Pierrain and Matthias Noback in addition to my own experimentation, I believe the coarse-grained unit test is the best approach. I'll elaborate on my reasoning below.

## Concerns: What do we need to test?

We've covered the main test types. There are other types of tests like *Smoke Tests* or *Exploratory Tests* but these are the most important ones that we need to be aware of for now.

With some of the vocabulary out of the way, let's take a closer look at our layered architecture and make sure we're clear on exactly what we need to have tests for.

■ **Quiz yourself:** For each of these, think about the type of test (or tests) that makes most sense to use.

### Features (queries)

When we ask for data through the API, what are we really concerned about testing? Well, there's:

- the fact that we can see the data
- the fact that the data is correct (ie: I created two posts, can I see them both?)
- the security and scope of the data (ie: Do I have permission to look at this? Does this user own this data? Is this data private to the owner's account? How much can public users see?)

### Features (commands)

Commands are different from queries because they perform *state-changing* effects on the system in some way. We want to test that the features:

- fail when they should fail (ie: I didn't pass in a valid email for `createUser`, do I get an `InvalidDetailsError`?) and
- succeed when they should succeed and that they
- adhere to security rules (ie: am I allowed to perform this action?) and that they
- do, in fact, invoke the appropriate **state-changing** calls to change the system in some way, and with the correct arguments (ie: Does the system attempt to send an email, bill the customer, and save the new records and domain events to the database?)

Depending on the way we test commands, we probably don't want to *actually* write to a database, send real emails, or bill real customers all the time. This is where we make use of `to`. Using Test Doubles - coming soon. Learning how to use test doubles well will take some practice. We've devoted an entire chapter to discussing them in Part X: Advanced Test-Driven Development.

### Core code

Everything else that isn't infrastructure code is core code. We'd like to have tests for them. This includes:

- Utilities and core classes
- Custom authentication/authorization code
- Middleware logic

## Infrastructure

Remember that things like databases, caches, external APIs, and components that are coupled to i/o or the network is *infrastructure* code. This is code that we don't own and that we didn't write tests for — someone else did.

While someone else wrote tests for them (think about the developers who wrote the tests for the MySQL or Postgres databases), we have to test that they work the way we want them to within the context of our application.

The way we'll be able to test them has to do with whether they're *managed* or *unmanaged* dependencies.

- A **managed dependency** is one owned entirely by our application (like your application's database — there is a 1-to-1 relationship here). We have full control over it so we have a lot of options for testing.
- An **unmanaged dependency** is one where we *don't* own it (think external API or even just an internal, shared piece of infrastructure like a queue). In this case, the way we test how our application works with it relies entirely on what options the dependency gives us.

Let's walk through a couple common ones.

### Database (infra)

Your application has a database. What are we interested in testing?

1. That we can connect to it.
2. That our *Repositories* adhere to the contract. Meaning, we can save, getAll, getOne and whatever else is in the interface.

### GraphQL or RESTful API (infra)

Let's assume that you only have one application and the GraphQL or RESTful API is what connects it to the outside world. The most important things to test are:

- **The correct command/queries called:** Ensure that when you call a *command*, it actually gets to the application layer use case. Similarly, we want to know that when you perform a query, the correct function that execute the query is called. We're not too concerned about the result of the command or the query at this point — just that we've connected things together properly.
- **Bad client requests:** When someone makes a request that won't allow us call the application layer use cases, we want to let them know that in a uniform, structured way. Here, we're testing that our API can elegantly fail.

### Shared infrastructural components (infra)

Sometimes you have a shared piece of infrastructure like an email server, a cache, a queue, service bus, or perhaps even something like an Elasticsearch instance, for example.

Because it's used by other applications, we have to be careful about the way that we test it. We can't just wipe it clean and start from initial state every time. What are we interested in testing for?

- That we can connect to it
- That our application is using it properly
- That our application calls it at the correct time (like at the end of a successful feature)

## **External APIs and integrations (infra)**

The last common type of dependency are the external ones which we certainly don't have access to like Stripe, AWS S3, and so on. What we want to test for each of these is going to be slightly different.

With something like Stripe for example, we want to know that payments are going to work in our app, right? But how do we get that confidence? We don't want to make real payments every time we run our tests. It depends on the service itself. Sometimes there are options available. For something like Stripe that may have a Sandbox environment, that's an option. For something like AWS S3 for image uploads - maybe we just periodically clean uploads?

There are a number of approaches that can be taken here. We'll cover them all in more detail later.

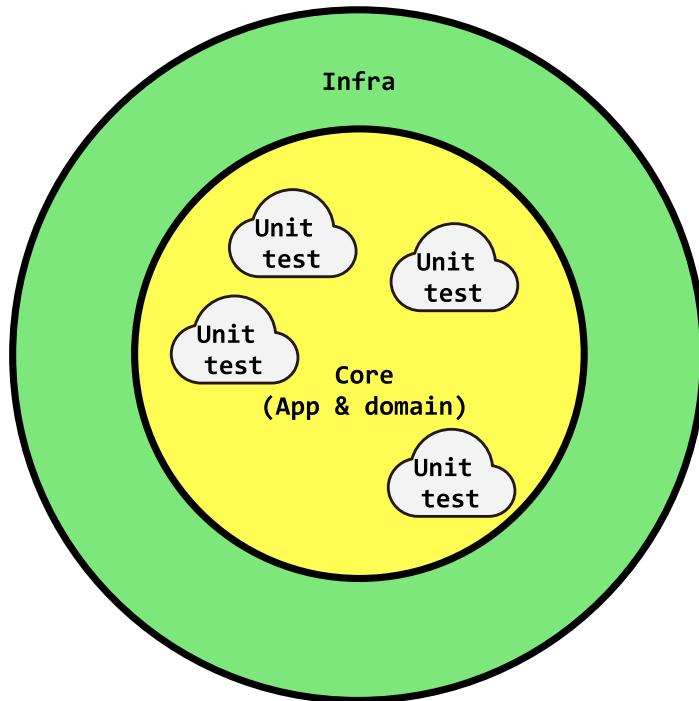
## **Testing strategy for a decoupled application**

What I'd now like to share with you is a testing strategy that seems to work very well for decoupled applications. It's not perfect, because no testing strategy completely is, but hopefully, this should inspire you to consider how your own.

Additionally, if you find yourself working on more complicated front-ends (beyond list/detail-view apps), you can apply a similar approach to carving out a reasonable testing strategy.

## **Unit test all domain layer code**

Starting with the simplest one, let's make sure that we unit test all of our utilities and what-not that isn't related to *features*. This is your middleware, helpers, in addition to your *Value Objects*, *Entities*, and any other domain layer code you've written, and so on.



You should seldom need to use mocks (although maybe stubs sparingly) to test this code because it should be pure, cohesive, and loosely coupled.

**Examples:** In Part IV: Test-Driven Development Basics, we learn the fundamentals of unit testing which you can use to test the behaviour of pure code-like objects or functions. In Part X: Advanced Test-Driven Development, we demonstrate how to unit test specific domain object patterns like *Value Objects*, *Entities*, *Aggregates*, *Domain services*.

### Use case tests: acceptance test application layer features as unit tests

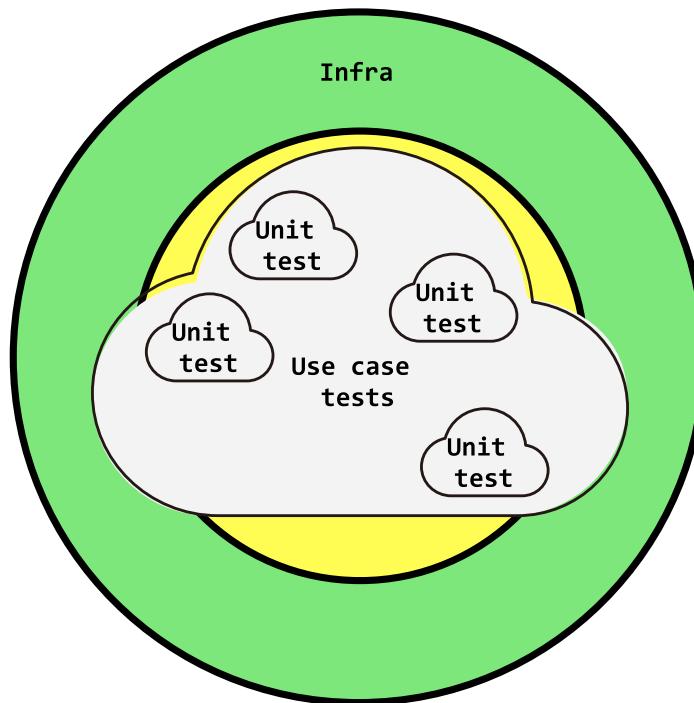
Next up we have our features — the acceptance tests. I recommend writing these tests which exercise the application layer *Use Cases* as unit tests. Because use cases rely on interfaces to *Repositories* and other adapters, you will have to mock them out to verify that the correct state-changing methods were called without actually making those changes to the infrastructure.

```
// Arrange
let userRepoSpy = new UserRepoSpy();
let mailerSpy = new MailerSpy();
let createUser = new CreateUser(userRepoSpy, mailerSpy);

// Act - execute the `createUser` use case, for example*
await createUser.execute({ ... })
```

```
// Assert
assert(userRepoSpy.save).hasBeenCalled();
assert(mailerspy.sendEmail).hasBeenCalled();
assert(mailerspy.getSentEmails().length).toBe(1);
assert(mailerspy.getSentEmails()[0].destinationAddress)
    .toEqual('testuser@gmail.com');
```

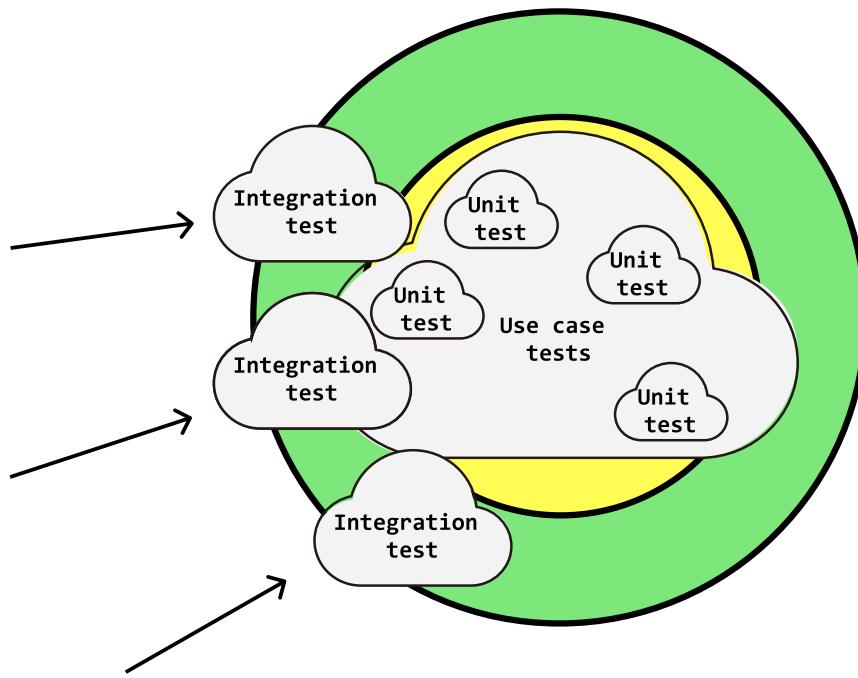
This example uses *Spies*, which is a type of mock object. While we won't discuss it in depth here (but we will here), the general rule of thumb for testing with mock objects is that we should only use mocks to **validate that a state-changing effect** has happened. That is, we use **mocks for commands**. To deal with queries, we use stubs.



**Examples:** We build out *Use Cases* in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS and learn how to use write them in a test-driven way in Use Case Testing.

### Integration test input adapters

Here, we test that the incoming communication mechanism (such as an HTTP request from a web server, GraphQL API, web-hook or some other incoming port) calls the correct application layer use case. Use an HTTP testing framework like Supertest to make real requests to your application. You can stub the responses for the use case because the behaviour is covered in use case tests.



■ **Examples:** We'll demonstrate how to integration test input adapters in Integration Testing Input Adapters: GraphQL, REST, CLI from Part X: Advanced Test-Driven Development.

### Integration test outgoing adapters

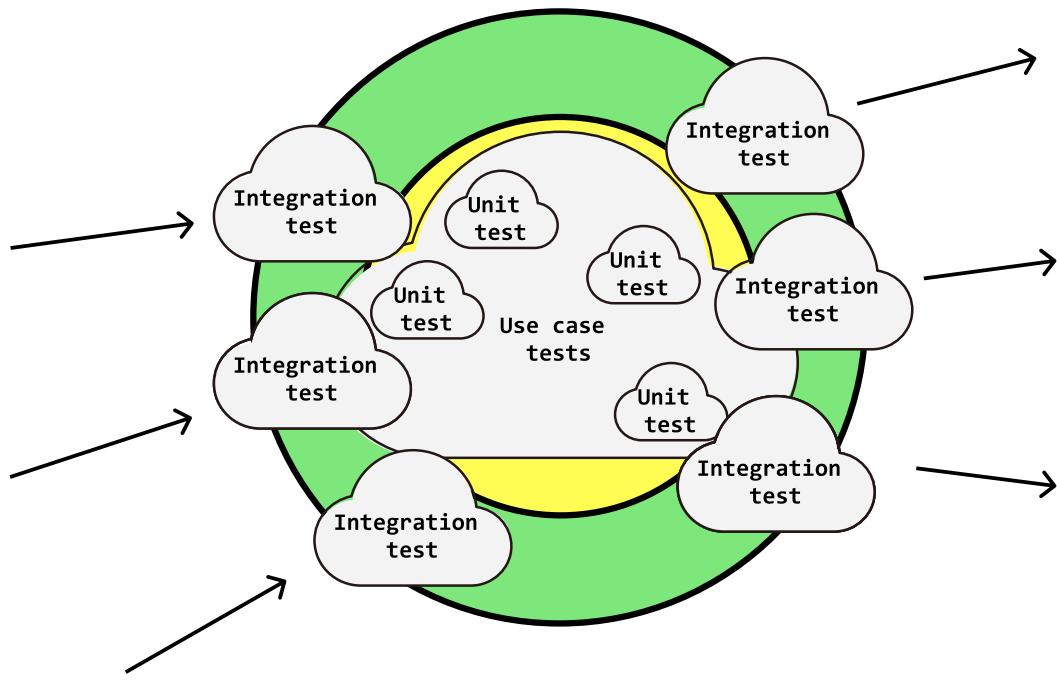
For outgoing adapters to *managed* dependencies like databases, use contract tests.

For outgoing adapters to *unmanaged* dependencies — integrations with external APIs, services, or shared infrastructure that costs money or are expensive to set up and test — we have a few different options.

As Matthias Noback writes, the most viable options are to:

1. Write the test against the real service
2. Write the test against a sandbox environment the third party offers
3. Write the test against a fake server (or Docker install)

You may have to use a mixture of a few. For example, I'll happily do #1 for AWS S3. For Stripe, I'll do #2. I will rarely do #3, though these days, with Docker, it's not a bad option.



## **End-to-end test for queries and additional confidence**

Lastly, end-to-end tests.

Since we have already so much covered by the other forms of tests, end-to-end tests help to ensure that everything works together.

The key is to write end-to-end tests that execute in the most production-like environment possible, whether this be a staging server or test server that you periodically clean every now and then. The goal is to confirm that your end to end tests can run through and do all of the things that a user would do, and then that's it.

Treat the entire application like a black box. Don't mock anything. Use stubs to clean up the creation of request data if necessary.

Because this is the most natural form of perceiving, it's here that we can test to ensure that we can see data in a declarative way. Using tools like Cypress or React Testing Library, you can query for data on screen regardless of the type of element it lives in. This leads to more stable tests.

## **Summary**

- The four main types of tests that we should know about are unit, integration, end-to-end and acceptance tests.
- In layered architecture, we can use a mixture of each type of test to cover each of the various concerns which need to be tested. This gives us a chance to produce software with a very high test coverage percentage.

## References

### Books

- Advanced Web Architecture by Matthias Noback
- Growing Object-Oriented Software Guided by Tests by Steve Freeman

### Articles

<https://kentcdodds.com/blog/static-vs-unit-vs-integration-vs-e2e-tests>

<http://tpierrain.blogspot.com/2021/03/outside-in-diamond-tdd-2-anatomy-of.html>

<https://martinfowler.com/articles/practical-test-pyramid.html>

## 26. The Walking Skeleton

■ A walking skeleton is a thin implementation of the system that performs a function from end-to-end. It links together the main architectural components (UI, database, messaging, etc). It exposes the most difficult pain-points early, provides the foundation to kick start our TDD process with acceptance tests, and gives us the power evolve the functionality and supporting architecture in parallel.

We're trying to kick off this test-driven workflow and evidently, there have been a few prerequisites.

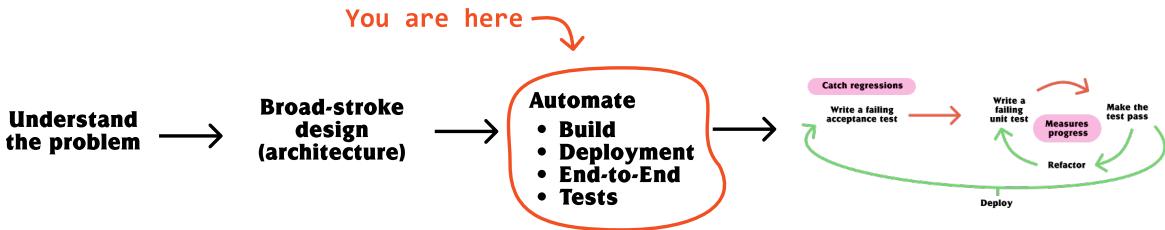
The first of concerns we needed to clear up was how we're actually going to test our application. Since most applications include a mix of core and infrastructure code, we realized that we needed an object-oriented architecture that promotes separation of concerns. Why? Because of the testing options it affords.

In 25. Testing Strategies, we got crystal clear as to the types of tests that can be written within a layered architecture. At this point, we can expect a lot less ambiguity as to how we will *safely* and *consistently* develop working functionality over time. Because we haven't practiced it just yet, we should at least *theoretically* understand how to deal with testing the features, infrastructure, and remaining core code in our application.

With that said, here's where we should be to advance:

1. You've got a high-level picture of your architecture and the main components it consists of — check
2. You understand how you're going to write your acceptance tests and cover gaps in the rest of your application — check
3. You have the user stories (and ideally, customer-written acceptance tests) that you're going to be building for the first functionality-focused iteration — check

That's great. What else needs to be done to start the test-driven process?



Well, the problem is that we don't actually have any infrastructure — a platform upon which we can write our first acceptance test. At this point of the project (*Iteration Zero*), it's common to have a number of developers ready to get started building things out, but there's no real structure, no way for the application to be divided up and built up top of.

Therefore, we're going to need *just enough infrastructure* for us to get busy.

"Alright, well how do you know when enough is enough infrastructure? Didn't you say that you shouldn't spend entire iterations building out infrastructure? Isn't that a violation of YAGNI? Didn't you say you say we should be feature-driven?"

Yes, that's a good point. I did say you need to be feature-driven, and spending entire sprints just building out infrastructure for the sake of building it out is not only wasteful, but irresponsible.

Instead of just getting into a trance of building out infrastructure for the sake of it, we use a more intentional approach. What we want to do is **build, test, and deploy a walking skeleton**: something that will allow us to expose uncertainty early, confirm that everything can in fact work together, kick off our test-driven development workflows for future feature-iterations, and deploy tested features to users in an automated fashion.

## Chapter goals

In this chapter, we will:

- Discuss the walking skeleton concept and the benefits of using one to start a project
- Learn how to choose the functionality to exercise the initial skeleton with
- Walk through a high-level demonstration of a minimal end-to-end function used to sculpt out and validate the initial architecture

## What is a walking skeleton?

First referred to as *Tracer Bullets* in *The Pragmatic Programmer*, a walking skeleton is a thin slice of functionality that executes from end-to-end, threading all the way through each of the main architectural components. The walking skeleton merely helps us validate that our choices of packages, libraries, vendors, and tooling will actually work.

Once we've done that, we ensure that we can build, test, and deploy that functionality in a production-like environment.

## Why end-to-end?

“But it worked on *my* machine!” Ever heard that?

It is extremely common to see that everything works locally. We locally install databases, caches, services; maybe we use Docker to install them instead of having to deal with trying to get the correct versions, and so on. Being able to run your tests against local infrastructure is a good start, but the only way to prove that the system completely works is try it out end-to-end.

This means you *really* want to reduce any sort of mocking or stubbing of components (unless of course, it’ll cost you money to run your tests — like if one of your components is a premium stock market data API).

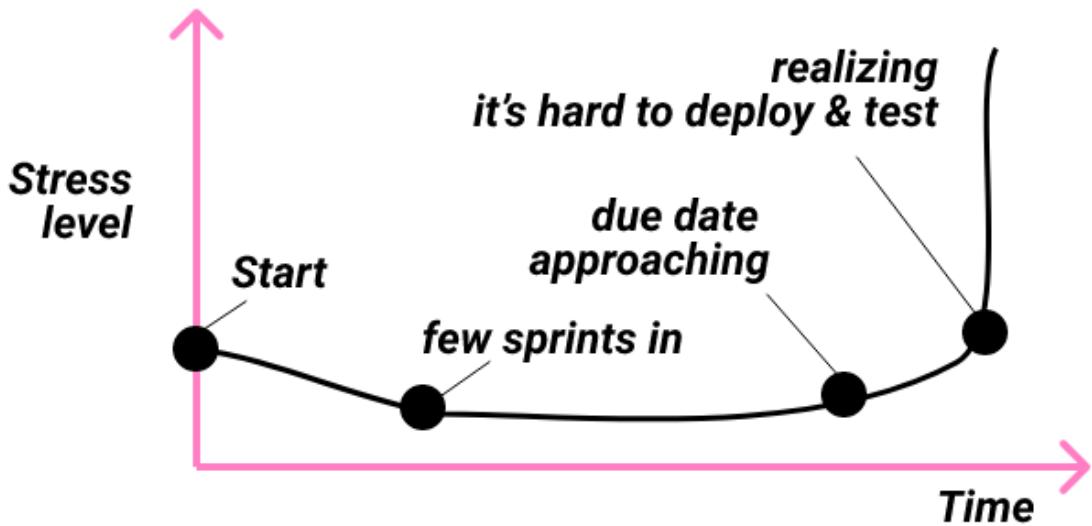
The benefits of an end-to-end approach are that we:

1. Build out the initial front-end testing architecture
2. Build out the initial back-end testing architecture
3. But most importantly, we expose uncertainty early by building, testing, and deploying it early.

## Exposing deployment and connection issues early

It’s only when we actually have to try to get things working for real, and it comes time to deploy it so that users can actually start using our features (that we’re proud of that work locally) does the pressure set in.

How many times have you run into dreaded CORS errors? It’s pretty annoying, but trying to fix that before a deadline can be borderline nerve-wracking. You do not want to be in a position right before the deadline where you’re fumbling with database security settings on a production server so that your ORM can connect to it. You also don’t want to suddenly realize that AWS SES has some limitation that means you have to drastically change the way your event-handling code works.



We want to figure all this stuff out as soon as possible so that we can mitigate as much pain, suffering, and embarrassment as possible. Let's gracefully cross the finish line once we've done great work, shall we?

It may also be the case that we're trying some new functionality or some new architectural pattern that we've never used before. A primary goal is the walking skeleton is to validate that our architecture *can* work, and to figure that out early. We won't always be right, but if you can learn where there are issues earlier on in the process, that's always going to be better (see The Principle of Last Responsible Moment).

As well, deploying in environment that resembles a production-like one is ideal because if its going to take the infrastructure team two weeks to set everything up for us, let's get the ball rolling on that as soon as possible.

### **What do you mean by production-like environment?**

A production-like environment means real servers. Or, at least real in the sense of however you were going to deploy it for real. Regardless of if your production environment is running on a real server or serverless lambda functions, you need to be deploying to an environment that lets you test your end-to-end tests similarly to how a real-world user would use the application in production.

Therefore, if your production environment connects to a real database, a real cache, and a real ElasticSearch instance, then replicate your production environment as closely as you can in the environment that runs your end-to-end tests. This is how to gain the most confidence.

### **Why automated builds?**

This is part of a larger DevOps philosophical discussion (Part XI: Above and Beyond), but

very simply put — in XP, you want to ensure that at any point in time, you have a deployable system. If the *Acceptance Tests* pass, we'd like to assume that the system can be merged to production and used by real users (or at the bare minimum, presented to the customers).

Automating this process is how we employ the *Continuous Integration* XP (and DevOps) principle. We refer to continuous integration as the practice of enabling developers to commit their code into the same repository several times a day. Ultimately, this is another way to expose uncertainty early. How? Because if there are conflicts in the codebase, they're discovered earlier than later. Gone are the days of everyone sitting around a shared computer trying to figure out how they're going to get the code that everyone wrote that week to work together.

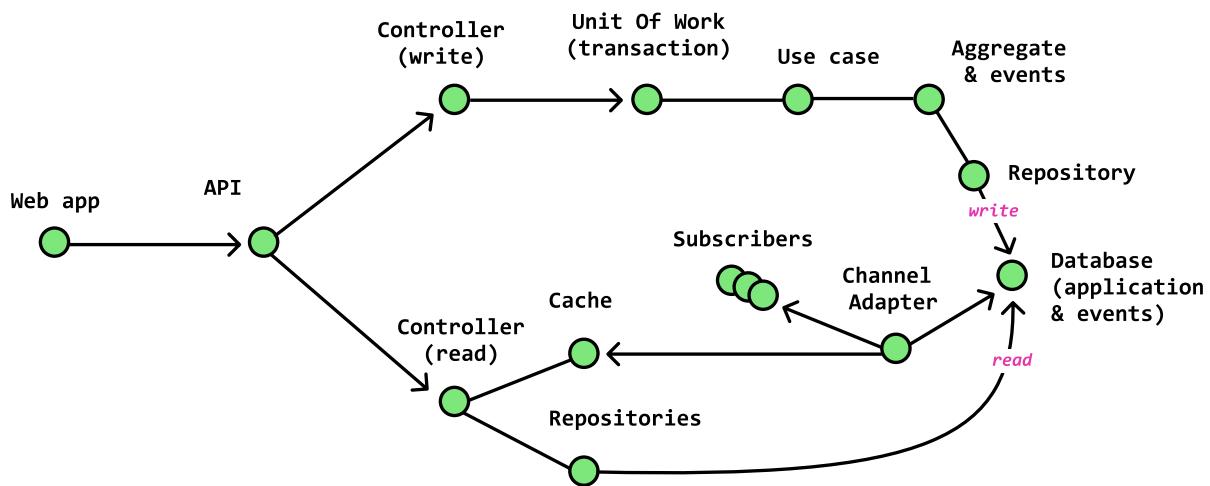
There is no single one way to set up your automated environments, but a common approach is to employ a staging environment that looks just like a production one. In a staging environment, you can move around and play with the new features at your leisure. A testing environment could be a precursor to the staging environment. The difference is that your testing environment merely takes your code, puts it on real application servers, and *headless-ly* runs the end-to-end tests from a continuous integration server.

There are a variety of services out there you can use to automate the build and deploy process like GitHub, GitLab, Jenkins, or CircleCI — take your pick.

### What do we do about the components not covered by end-to-end test?

The minimal end-to-end functionality we choose to sculpt out our architecture is unlikely to pass through *every* single conceivable component that you need. You may find that you still have a number of major architectural components (like a queue, cache, monitoring software, logging, reporting/etc) that aren't connected via your simple end-to-end tests.

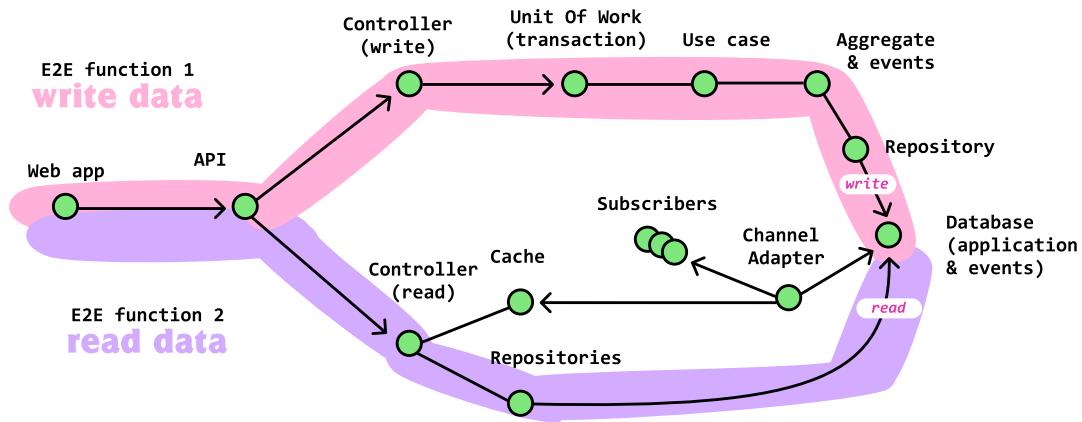
Let's do a simple walkthrough. In Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, we use the CQRS pattern. Drawing all the pieces out, our idea of the skeleton could look like this:



One of the characteristics of a CQRS architecture is that it has separate *read* and *write* paths.

There are a lot of benefits to that, but it's not important to get into at the moment.

Anyway, if we had an end-to-end test which merely confirmed that we could create some data, have it save to the database, and then see it on screen, we'd have to write a test that does a *write* through the write path and a *read* through the read one.



This is great, we've managed to thread through a lot of the bones in the skeleton to sculpt out our architecture, but there are a few pieces still left un-sculpted out. That's OK. We can always *Integration Test* these components later on. *E2E Tests* can't cover everything, but they do a great job of getting us started.

### High-level demonstration (example)

In this section, let's imagine I'm building an application called StudySpots: a way for you to find places to study in your city. The application is concerned with ranking and rating coffee shops, parks, libraries, and other places to study near you. Users can submit spots, discuss details, and recommend places to work.

On the front-end, users should be able to get spots in their city and select a number of filters like *noise rating*, *hasBathrooms*, *indoor/outdoor* and so on.



We start from the front-end so that we can get our front-end testing architecture started and point it towards a currently non-existent backend API.

After we install the necessary libraries, tools, and ensure our simple front-end E2E tests are failing for the correct reasons, we'll move to the backend and get things started there.

### **Small steps**

Building out your entire architecture is not something you should consider to be done in a single go. There's just way too much to mentally account for. You have to install dependencies, make layout decisions, set up your database, get seeder and migration scripts running, and among other things — make sure you have a good development environment to boot.

Incremental development is about slicing up all the work to be done into smaller parts that can be done little by little. This is something you'd like to do in your *Iteration Planning Meeting* for the initial sprint (see 20. Iteration Planning).

It's OK for this process to take a few days. If you're using new technologies, you're going to be learning a lot about them as you go along.

### **Choose the simplest scenario(s)**

We said that the idea is to choose the simplest possible scenario to get started with our walking skeleton. A short list of user stories (features) for StudySpots could be comprised of the

following:

- ViewAllSpots
- SubmitSpot
- EditSpot
- RateSpot
- CreateAccount

Looking through the list, we decide that *ViewAllSpots* is one of the simplest things that we can do *solely because we can come up with a simple scenario to test it.* \*\* I want to test that I can “View all study spots in Toronto by default”. We can merely seed some of my favourite spots to study in Toronto in the database and test that it can be shown on screen.

In a CQRS architecture, such a feature would cut through the *read* path which gives us an opportunity to set up our front-end, our testing architecture with Cypress, the (GraphQL or RESTful) API which receives the HTTP request, the database, ORM, and the repository interface and related implementations.

■ **The simplest Acceptance Test scenario:** I like to think that the *E2E Test* we should use to build the walking skeleton is the *simplest* scenario of all the scenarios for any *Acceptance Test*. For example, if we had five stories and *Acceptance Tests*, and each one have six scenarios, we look for the simplest scenario in all five of those *Acceptance Tests* and use that to build out the skeleton. We can always come back and finish up the rest of the scenarios for that *Acceptance Test* once we’re focused on functionality-iterations.

## A failing end-to-end test

Let’s say I’m using React.js, Apollo Client, and Cypress.io on the front-end. After installing everything, I write my first E2E test, but I do it in *Given-When-Then (Gherkin)* style like we demonstrated in 22. Acceptance Tests.

■ **BDD-style E2E tests:** I prefer to write all of my high-level tests in BDD-style. My intention is to keep the tests as declarative as possible. You can use `[@cypress-cucumber-preprocessor]`(<https://www.npmjs.com/package/cypress-cucumber-preprocessor>) to learn how to get Given-When-Then clauses in Cypress.

Here’s what my feature definition file looks like:

```
Feature: View all study spots
```

```
I want to see study spots
```

```
Scenario: View all study spots in Toronto by default
```

```
  Given I'm on the map page
```

```
  And location is disabled
```

```
  When the page loads
```

```
  Then the current location should be Toronto
```

```
  And at least one spot should be displayed
```

When I run my Cypress tests, this should fail because there is no test implementation yet. Let’s go ahead and write that.

## Use Page Objects to write declarative E2E tests

Lots of developers struggle with E2E testing because they write tests that are prone to ripple. When we write tests against pages, we tend to refer to their internals (HTML elements) in order to interact with the page.

The problem is that by doing this, we end up with brittle tests since changing the UI means changing the HTML, and your tests rely on the HTML directly.

The solution is to again, just like interfaces at the class level, and just like GraphQL at the architectural level, to add a layer of indirection in-between your test specifications and your HTML. We define a contract that specifies how elements get added to the UI.

We can use something called the *Page Object* pattern: a declarative way to write front-end tests.

Page objects use declarative *expressions* (instead of imperative *statements*) to define what should happen. The benefit is that it lets us perform operations against a page without needing to sift through HTML.

Therefore, a test implementation for the test specification could look like the following:

```
/// <reference types="cypress" />

import {
  And,
  Given, Then, When,
} from "cypress-cucumber-preprocessor/steps";
import { MapPage } from "../../pageObjects/mapPage/mapPage";

// Page object pattern
let mapPage = new MapPage();

/**
 * Scenario: View all study spots in Toronto by default
 */

Given(`I'm on the map page`, () => {
  mapPage.load();
})

And(`location is disabled`, () => {
  mapPage.useDefaultMapLocation()
})

When('the page loads', () => {
  mapPage.waitUntilLoaded();
})

Then(`the current location should be Toronto`, () => {
```

```

    mapPage.getCurrentLocationInput()
      .should('exist')
      .should('have.value', 'Toronto')
  }

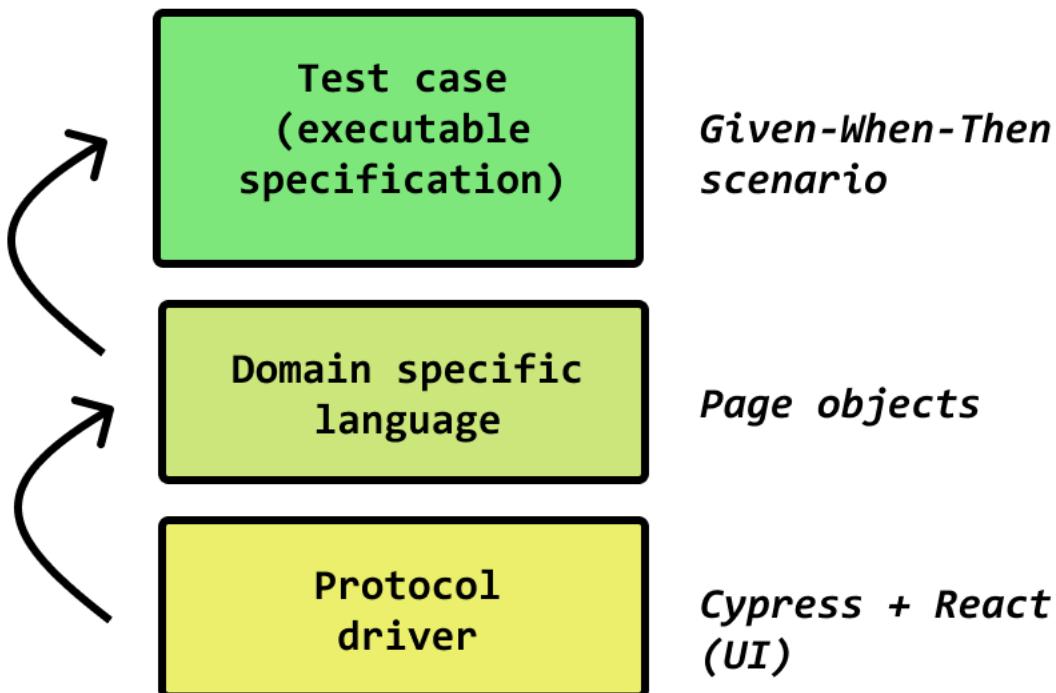
And('at least one spot should be displayed', () => {
  mapPage.getCurrentStudySpotsInViewOnMap()
    .should('have.length.greaterThan', 0)
})

```

As I wrote this test, I wasn't even sure about how the actual code which implements the test would work. At this point, all I was focused on was making sure that the way I'd expressed how my test should work makes sense.

Some call this **programming by wishful thinking**, but I just think of it as doing BDD or the real way we write declarative code in OO.

Once I resolve the compilation errors, I'm presented with a failing E2E test. I then have to implement the MapPage page object and within it, I start to make architectural decisions about how *pages* will come to exist in my application and how I'll be able to traverse elements within pages to find the elements I need in a relatively structured way.



There's a general philosophy to why we structure tests in this layered fashion (and Dave Farley talks about it in detail here). We dive deeper on *Page Objects* and related testing concerns

in Part X: Advanced Test-Driven Development.

### Fetching the non-existent spots

At this point, the test is failing because we don't have anything developed just yet. Therefore, we start by putting together a React component that makes an API call to a non-existent backend to fetch non-existent study spots from a non-existent GraphQL API.

```
// pages/MapPage.tsx

import { gql, useQuery } from '@apollo/client';
import React from 'react';
import PageContainer from '../../shared/components/PageContainer';
import { Map } from '../components/Map';
import SearchAndFiltersPanel from '../components/SearchAndFiltersPanel';

export interface MapPageProps {
  mapboxToken: string;
}

const GET_SPOTS = gql`query GetSpotsByCity ($city: String) {
  spots (filters: { city: $city }) {
    title
    id
    latitude
    longitude
    address
  }
}`

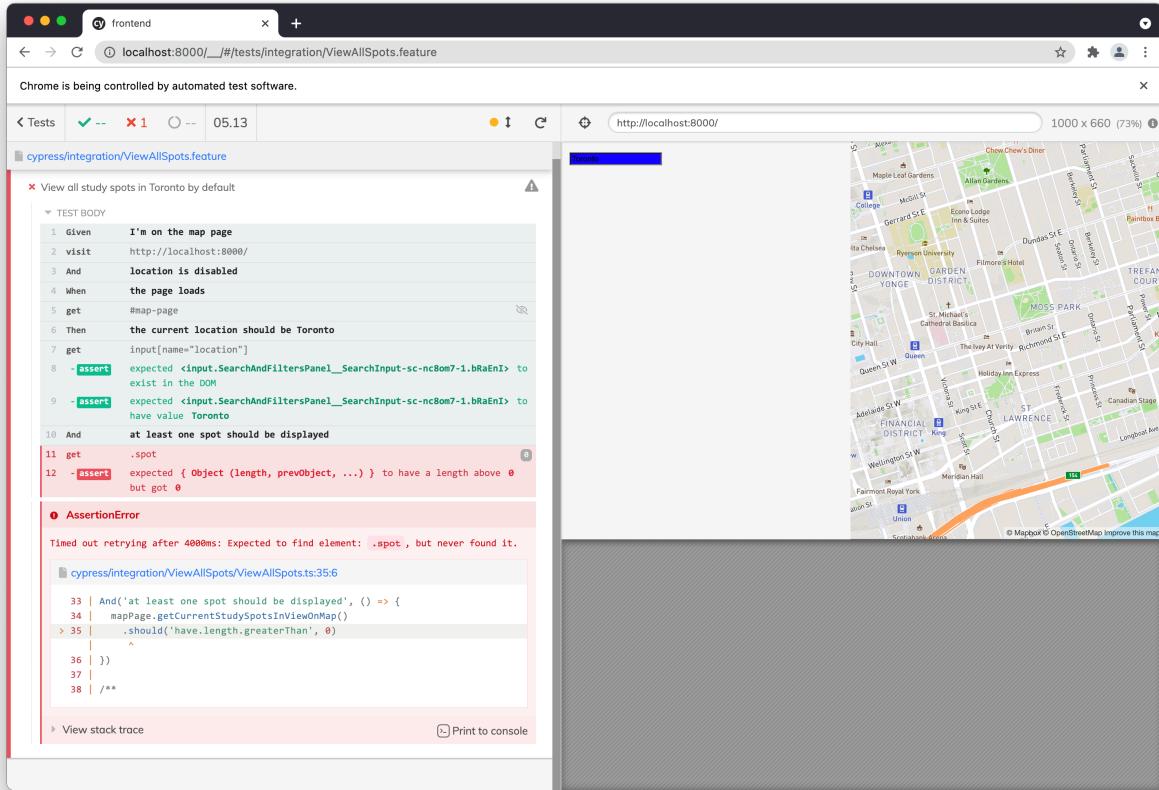
const MapPage: React.FC<MapPageProps> = ({ mapboxToken }) => {
  const city = 'Toronto';

  const { data, error, loading } = useQuery(GET_SPOTS, {
    variables: {
      city
    }
  });

  return (
    <PageContainer pageId="map-page">
      <SearchAndFiltersPanel />
      <Map accessToken={mapboxToken}/>
    </PageContainer>
  )
}
```

```
export default MapPage;
```

In Cypress, we should still see that our test is failing.



So far so good. We've managed to get a front-end test architecture setup for E2E tests. Now it's time we jump over to the backend and do the same over there. Once this E2E test passes because we've set up our backend, we should stop and take a look at our progress because we've just built the walking skeleton.

## Backend: the bare minimum

There's a lot of work to do on the backend, so again — it's important to step through it slowly and carefully.

Now, of course — you can do whatever you want, but at this point, I see two really nice options for how we can go about building out the backend so that our E2E test passes. We can either:

1. Start a new *Double Loop TDD/Acceptance-Test Driven Development loop* (see the next chapter on 28. Test-Driven Development Workflow) \*\*using the backend equivalent scenario. In this case, I might write a *SearchSpots* use case + *feature* file combination and use the *Use Case Test* approach we learned about in 25. Testing Strategies. To exercise the database, I may also write an *Integration Test* to confirm that things are connected correctly.
2. Just do the bare minimum to make the E2E test pass.

In my opinion, I think you should do #2 — the bare minimum to get the E2E test to pass *first*. Why? There's just a lot going on. Let's focus on getting the build-test-deploy infrastructure set up and running.

We'll spend more time using *Integration Tests* to confirm that the infrastructure adapters (databases, caches, 3rd party APIs) adhere to a contract that we like, later on when we move to building features in functionality iterations.

## Summary

- We use a walking skeleton to help us expose uncertainty early, verify that our choices result in an architecture that works, get our TDD development workflows started, and set up automated builds.
- Testing end-to-end in a production-like environment gives us the most confidence that our code works and forces us make some initial decisions about our code.
- Opt for the simplest scenario in a user story's acceptance test as the first slice of functionality to write your acceptance test that creates the walking skeleton.
- To make your failing end-to-end test pass, you'll encounter a large number of tasks to complete. Take your time and opt for the bare minimum that will prove to be a good starting point for future feature-iterations.
- Once the skeleton can be built, tested, and deployed in an automated way, you're done and ready to move onto acceptance testing features.

## References

### Articles

- <https://www.henricodolfing.com/2018/04/start-your-project-with-walking-skeleton.html>
- <https://wiki.c2.com/?WalkingSkeleton>
- <https://wiki.c2.com/?WishfulThinking>
- <https://martinfowler.com/bliki/PageObject.html>

### Books

- Growing Object-Oriented Software Guided By Tests by Steve Freeman
- Agile Technical Practices Distilled
- The Pragmatic Programmer — discusses this topic under a different name (Tracer Bullets) — <https://flylib.com/books/en/I.3I5.I.25/I/>

### Videos

- “How to Write Acceptance Tests” by Dave Farley

## 27. Pair Programming

■ Pair programming is a way to produce higher-quality designs by reducing the feedback loop. The phronimos developers recommend pair programming as a foundational technique.

Having someone look over your work is really useful. For most of us, the only time someone really checks our work is when we create pull requests. That's a missed opportunity. We could shorten the feedback loop dramatically.

Pair programming is one of the original XP techniques. It actually mandated that *all work* be done in pairs. You may consider that a bit excessive and a waste of time, but think about it — how much time do we spend going back and forth updating pull requests based on feedback when we could have produced a better design the first time around?

With willingness, maturity, and a few soft skills for working with humans, anyone can learn how to do pair programming: one of the best XP techniques for producing high quality code.

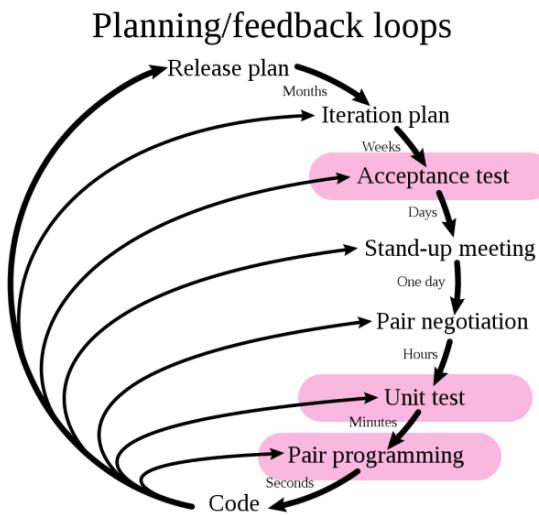
## Chapter goals

In this chapter, we will:

- Understand what pair programming is, how it improves design, and how it works
- Learn different switching techniques
- Learn a few ways to make pair programming sessions enjoyable and effective

## What is it?

Pair programming is what it sounds like. Instead of writing code independently, you write code in pairs. With this disciplined approach, the goal is to reduce the feedback loop (since feedback is the fundamental tool, after all).



The longer we wait to get feedback on our designs, the more painful it is to know we've done something in a way that could have been cleaner, simpler, or more correct. What if you could get instant feedback? *While* you're coding? That's what pair programming is about.

It works by taking turns writing code and discussing our design decisions in real-time with a peer. This lets you examine the code and react to potential design deficiencies earlier.

It helps you:

- Reduce frustration when you're stuck
- Consider edge cases you may not have considered
- Find and mitigate potential bugs
- Discuss how your code could potentially result in one or more of the symptoms of complexity discussed in Part I: Up to Speed.

## Roles

There are two roles in a pair programming session: the *driver* and the *navigator*.

- **Driver** — The driver is responsible for actually coding. They use their hands to type, move the mouse and so on.
- **Navigator** — The navigator reviews the driver's work and constantly looks at the code strategically. They ask questions like: “are there any mistakes?”, “could we design this better?”, “will the solution require changes elsewhere?”, “could we make that more understandable?”

## Switching techniques

One way to differentiate pair programming techniques is how we handle switching roles.

### Using a clock

This is probably the simplest way to implement pair programming. Use a chess clock or a pomodoro timer (pomofocus.io is my favourite) to switch between roles. Once the timer expires, take a moment to wrap up your thought, and then switch roles.

### Ping pong/popcorn

With the ping pong/popcorn technique, pairs take turns writing tests and making them pass. For example:

1. A Driver (B Navigator). Write a new test and see that it fails for the correct reason.
2. B Driver (A Navigator). Implement the code needed to pass the test.
3. B Driver (A Navigator). Write the next test and see that it fails for the correct reason.
4. A Driver (B Navigator). Implement the code needed to pass the test.
5. A Driver (B Navigator). Write the next test and see that it fails for the correct reason.
6. Continue in this pattern

### Strong-style pairing

This one is really cool but requires a pretty high level of maturity and experience with TDD. For an idea to go into the computer, **it needs to pass through someone else's hands** (Llewellyn Falco).

Fundamentally, this means that the driver needs to trust *and understand* the directions given by the navigator. The rules are that:

- the driver is only allowed to ask questions to understand what needs to be done
- the driver is not allowed to question *why* we're taking the direction we're taking until the idea is fully expressed — ie: when the navigator says the idea has been implemented and that they're now open to suggestions on how to proceed
- the driver lets the navigator know when they're ready to implement the next instruction

This style is really interesting.

Apparently, it's most effective when the navigator understands the entire picture of how to implement a feature, and that would make sense (see 21. Understanding a Story ).

With this approach, the driver isn't concerned with learning the ideas until they're actually implementing it in the code.

As I said before, this implies a certain level of trust and maturity. Sometimes, all you have to do is ask for a small, temporary window of trust.

### **What's the output of a pair programming session?**

For each passing test, you make a commit. Then, when appropriate (like finishing a feature), you create a pull request to merge it into production.

Why bother with pull requests? Isn't that what pair programming solves? The need to do that?

Well, no. Pull requests are always a great idea. We get the chance to incorporate feedback from others that were not in the pair programming session in addition to getting another look at our code later.

### **How often should you pair program?**

Some companies pair program *as much as possible*, around 95 percent of the time. Some companies like to pair only in the morning or at certain times of the day. Some companies pair 3 out of 5 days a week and leave the rest of the time for programmers to do solo coding.

There's no real right answer to this, but as Engineering Manager Brad Pederson at a company called *Stats Perform* says, "once people have become used to the practice over a two- or three-week span, I have found that the majority do prefer pairing."

## **Considerations**

- **Power dynamics:** If you're a senior or more experienced developer, it's going to be mostly in your hands to ensure that your pairing partner feels comfortable. Be patient, encouraging, and stick to the timer.
- **Maturity:** Pair programming requires a fair amount of maturity to keep the conversation on track, follow the rules, and to be respectful and encouraging. It's important to leave ego at home for this one.

- **Let the timer remind you:** Do not just rely on your partner pointing out that you need to change roles, as he or she may not feel comfortable requesting it.
- **Allow time for individual work as well:** There are valid reasons for wanting to code individually instead. As Sandro Mancuso writes in *Agile Technical Practices Distilled*, in his experience, pair programming can at times be exhausting, you may want to explore a half-thought out idea but you're not ready to share it, or sometimes you may feel like you're not absorbing the techniques you're learning about from an experienced developers because you're not doing it yourself. Sandro and the other authors of *Agile Technical Practices Distilled* recommend that pair programming be the type of exercise that you opt into, not enforce. I think there are some good middle grounds. I've heard of companies that pair programming three out of five days a week. Some do it from noon till the end of the day.

## What a good pair looks like

An effective pair is one where they can communicate their thoughts and solutions clearly.

As Mike Tecson from Club Labs says,

"They know how to strike the right balance between communicating solutions, requesting feedback and driving forward with the task without becoming too chatty. [They] are socially aware. They are first and foremost engineers, but they also know how to pick up on social cues that help them foster collaboration and avoid conflict."

Ah yes, further proof that software development is not the isolated effort that lots of people think it is from the outside. Mike also goes on to say,

"Lastly, an effective pair takes initiative and strives to learn. They do not rely on their partner to drive solutions, but rather are looking to contribute."

## Summary

- Pair programming is a way to produce higher-quality designs by reducing the feedback loop. It works by taking turns being the *driver* or the *navigator*.
- This technique is highly effective but requires a fair amount of maturity and understanding of social cues.

## Resources

### Articles

- <https://builtin.com/software-engineering-perspectives/pair-programming>
- "Llewellyn's strong-style pairing," <https://llewellynfalco.blogspot.com/2014/06/llewellyns-strong-style-pairing.html>
- <https://www.quora.com/Which-companies-use-pair-programming>

### Books

- Agile Technical Practices Distilled by Marco Consolaro and Pedro M. Santos

## 28. Test-Driven Development Workflow

■ The best way to write testable code is to write the test first. In a real-world project, we maintain two TDD loops: an outer and inner loop. One for the outer *Acceptance Test*, and one for the inner *Unit Test* loops. Once all the tests pass on both loops, we're done implementing the application core. This technique is called Double Loop TDD. Following that, we improve the code by filling confidence gaps with *Integration* and *E2E* tests. TDD gives us the opportunity to safely exercise the entirety of one's design skills: from testing, to object-design, design patterns, principles, refactoring, and architecture.

A lot of musicians I know tend to spend a lot of money on expensive equipment like instruments, speakers, programs, pedals, and all kinds of other fun things — sometimes to ridiculous proportions.

When it comes to doing a good job in any trade, tools matter for sure, but I think it's more about the technique. I've heard, in my opinion, some of the most engaging music created by folks that had nothing but a tape recorder and a couple of instruments. Perhaps that's why Grammy award-winner and Rock & Roll Hall of Fame-r, Trent Reznor, decided to forgo his nearly multi-million dollar studio for a laptop with some basic programs on it to create a record that became the band's first number-one album on the Canadian Albums Chart over a career of 25 years.

In this trade of software design, if you're serious about mastering it, I believe that TDD is the vital technique that will take you there.

TDD has many benefits, but one major aspect is that it creates a scenario where we can integrate everything we've learned (and will learn) about design — all the wisdom of principles, rules, patterns, and what we've learned makes for testable, flexible, and maintainable code — in a safe, risk averse way.

### Chapter goals

In this chapter, we will:

- Formally discuss test-driven development and the rules and schools of thought to TDD.
- Recap critical prerequisites in order to use TDD to develop a web application and discuss what makes TDD hard to perform in the real-world and
- Explain how we use both the Mockist (London-style) and Classic (Boston-style) TDD schools of thought to implement Double Loop TDD.
- Learn how to rely on our testing strategy to increase confidence in our codebase after we've finished the *Acceptance Test*.
- Learn why you should be refactoring constantly and why the refactor step of the TDD loop is the key place where design happens
- Wrap up Phronesis and grasp how the remaining parts of the book integrate back into the phronimos techniques we've covered

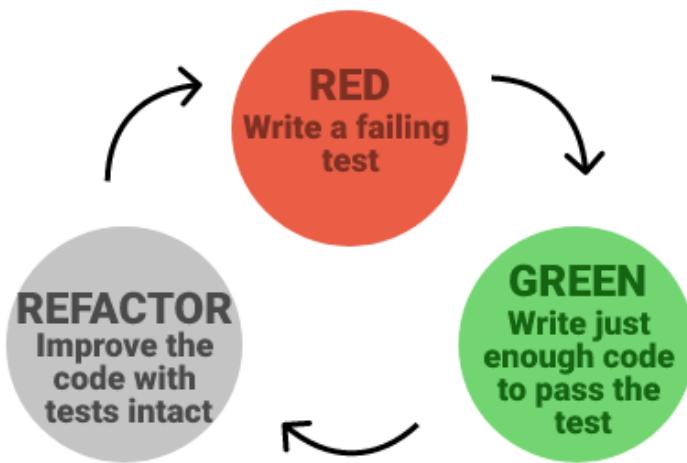
### Test-Driven Development rules & considerations

#### Red-Green-Refactor

As you know, TDD (test-driven development), is a technique — or a process — for developing software. The goal is to keep code quality high and keep you productive, even as projects grow to be really large and complex.

It works by following the *red-green-refactor* loop:

- Red — Write a failing test
- Green — Write just enough code that will pass the failing test
- Refactor — Criticize the design and refactor the code, keeping the tests intact



This is a *technique* that can be replicated and followed along with. When you're writing the failing test, you're focusing on making sure that the test makes sense and that you understand how you'll be able to write the solution to prove that that solution is correct. If we have no idea how we'll make the test pass, we learn that right away. TDD gives us immediate consequence detection at each step.

### **Don't get ahead of yourself (the three laws of TDD)**

Some equally important TDD rules to follow are:

- 1. You are not allowed to write any production code unless it is to make a failing test pass**

Because it is so much harder to write tests *after* the code has been written, the idea is to start with the tests.

- 2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.**

Your goal is to use the tests to drive the design. When you write your first test, you'll often refer to things that don't quite yet exist. Here's an example:

```
describe('elevator', () => {
  it ('starts at ground floor', () => {
    let elevator = new Elevator();
    expect(elevator.getFloor()).toEqual(Elevator.Floors.Ground)
  })
})
```

If I did nothing other than just wrote those lines of code, then my code would *not* be compiling. The rule says to merely make the code compile — so this means to create the Elevator class, give it a getFloor method and a static Floors enum object so that this code will compile. But do nothing more. The test will fail, but the code will pass. You're in *red* mode.

### **3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test**

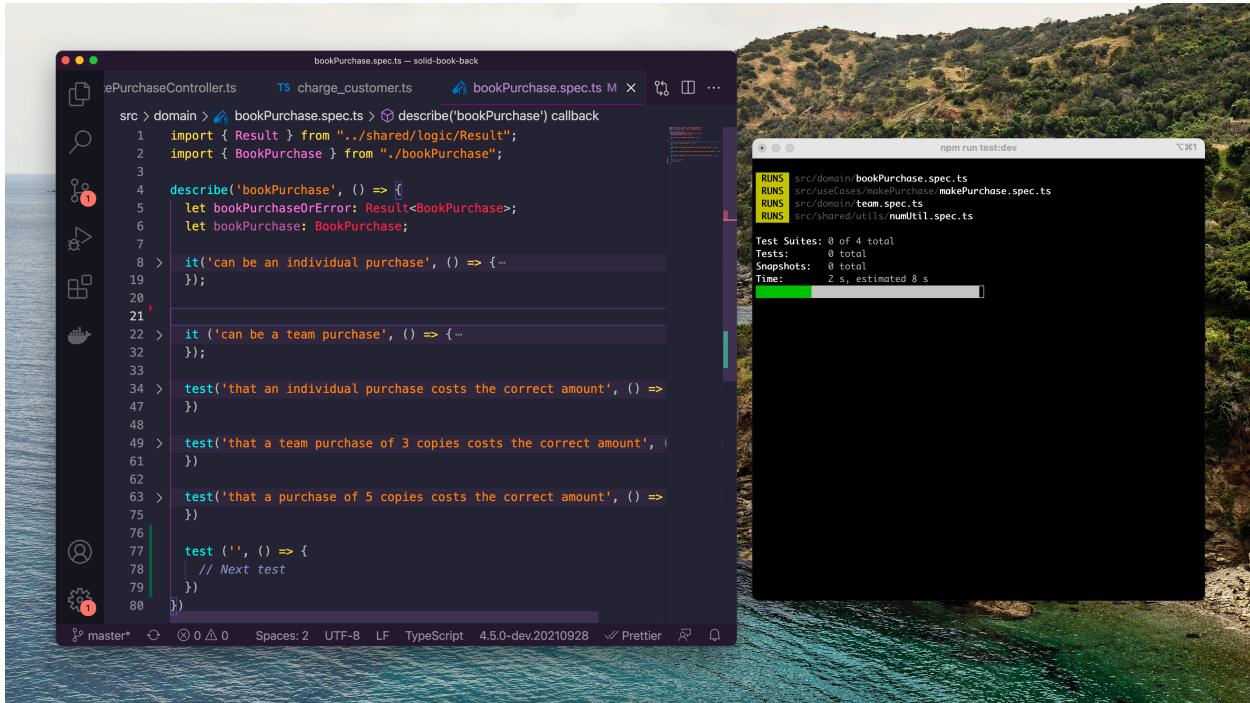
Once you make the failing test pass, STOP. Don't write any more code. Stop. Take a moment to contemplate your design since you've moved from *red* to *green* and now you're in *refactor*. Once you decide you want to write some more code, do so by first writing a new failing test (*red*).

Now, of course — sometimes you can get ahead of yourself, and I sometimes do — but because I've been doing this for a while now, I have a good understanding of how far away from the boat I'm getting and know when I'll be able to swim back safely (write the necessary tests) or if I'm in trouble.

### **Have your tests running alongside you as you code**

To get the instant feedback we've been talking about, you want your tests running at the same time as you're coding. You'll want to set up a script that re-runs your tests every time you save your code.

I like to put my editor on one side of the screen and have my terminal with my tests running on the other side. If you have multiple monitors, that's even better.



## Committing after a passing test

It's also recommended to commit your code every time you make a new test pass. Why? This allows you to go back in time if you've made a mistake or a potential refactoring messes things up.

Although sometimes, when you're on a roll and writing a bunch of tests and making them pass quickly, it's easier to just make all those tests pass and then commit them as a group.

## TDD Prerequisites

There are a large number of things that makes TDD hard to do. Perhaps after having seen the gradual approach we've taken to getting up to the point where we can write our first test, you can appreciate the challenge.

To recap, here's what we had to do to ramp up:

## Reasonable testing strategy thought out

In 25. Testing Strategies, we learned that in order to truly test an object-oriented architecture, we need to first understand what an OO architecture looks like, separate core code from infrastructure, and then decide on the appropriate types of test to cover untested surface area. We had to know this before we wrote our first line of code.

## Create a build-test-deploy test architecture

Then, in 26. The Walking Skeleton we had to develop the skeleton of our test architecture on both the front-end and on the back-end.

## **Separate scripts for acceptance/unit, integration, and E2E tests**

Another part of setting up the initial test architecture is ensuring that we have separate scripts for the different types of tests that we need to run. Why? Because we want as immediate feedback as we can get with our unit and acceptance tests. We also want feedback with our E2E and integration tests as well, but since they take longer, let's not let them slow down our tests that *will* execute fast.

We'll discuss in more detail in Part X: Advanced Test-Driven Development, but I like to name my unit tests with `.spec` and separate my integration tests by naming them with `.ispec` file extensions so that I can have separate Jest configuration scripts for each.

## **Can seamlessly switch between environments**

In addition to having different test scripts, we need to ensure that we can effectively change the context in which our code runs.

For example, lets say you want to write some E2E tests to test a `MakePurchase` use case that utilizes a Stripe API key. Now, because you don't want to be making *real purchases* every time your run your E2E tests, you'll need a way to switch between:

- your live key for production and
- your sandbox key for development

Additionally, there may be other services that you don't want to interact with when you're running E2E tests on your local development machine. You probably don't want to write to your production database with dummy data, you probably don't want to send real emails out.

As we've learned in 24. An Object-Oriented Architecture, with the power of polymorphism, we can swap dependencies out at runtime, but we want to do so in a clean and coherent way. The last thing you want is `process.env` instances all over your codebase, as merely calling `process.env` is a way to instantly turn whatever code that uses it into *infrastructure code* (since `process.env` is an I/O or a Cross-cutting concern).

■ **Coming up:** We discuss this further in 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - coming soon.

These are all decisions that need to be made ahead of time when we decide on our testing strategies and build out our test architecture but before we write our first tests.

## **Why real-world TDD is so hard**

Not only did we have to implement those prerequisites, but throughout the latter half of Part III: Phronesis, we got a chance to see just how complex and challenging the landscape truly is for us today. Some of the

## **Different TDD challenges in front-end vs. back-end development**

TDD on the back-end is — in a way — easier. It's easier because sometimes, you're just working with pure *core code*. No dependencies to anything else. On the front-end, however,

TDD can sometimes be harder because you're coding within the confines of libraries and frameworks. There's less *core code*, fundamentally.

You have to ask yourself whether it's worth it to try to decouple core from infrastructure code on the front-end or if you just want to test the majority of your code using E2E tests instead. Depending on the nature of the front-end application you're building, this may very well be OK. For list-detail-view applications where we're merely fetching data and presenting it, **you can validate correctness by observing and clicking**. However, in more complex applications like dragging and dropping, games with lots of client-side logic and low-level domain complexities — things that cannot be tested by merely observing and clicking — it does presents a similar need for testing code the way we test within a server-side hexagonal/clean object-oriented architecture.

### **As an industry, we struggle to agree on testing terminology**

What's an integration test? What's a unit test exactly? What *isn't* an integration test? What *isn't* a unit test? Depending on your TDD school of thought, they may mean entirely different things to you. This definitely poses a challenge when we're trying to have discussions with others. Instead, what I've found works best is to get very clear about where the gaps are, and who *cares* what you call your tests — just make sure the gaps which are hurting your test confidence are filled.

### **Many fail to recognize that TDD starts with architecture**

You saw how much work we had to do ahead of time to even write our first test. From requirements to broad stroke architecture to testing strategy and the walking skeleton — imagine the challenge of trying to write tests after already having completed two or three iterations. Not easy.

### **Understanding what to test (and how to test it) is hard**

Should we test the database? How to we handle the fact that we rely on an API that costs money to use? How do we test that? How to we make our tests run fast?

The 25. Testing Strategies present some answers here, but like anything else, you have to do it to know it.

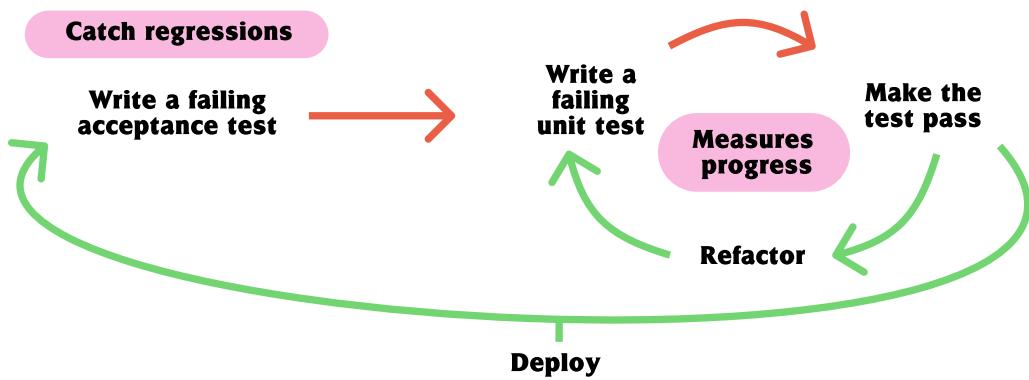
And then there's the topic of mocks and mocking. You'll hear a lot of absolute language in the industry around mocking. Some people will tell you to never mock dependencies. I don't agree with that statement.

What I've found is that depending on your TDD school of though — if you're only aware of *one* — you'll either see mocks as a useful tool you can use in certain scenarios, or you'll think that they were created by the devil itself.

Mocks (and the act of mocking) is useful as a part of a larger testing strategy. Allow me to explain the two different schools of thought for TDD and how they work together when we implement *Double Loop TDD*.

## Double Loop TDD & the two TDD schools of thought

Recall that the idea of *Double Loop TDD* is to main two loops.



When we're writing tests for the *outside* acceptance test loop, we're usually coding from the outside-in. When we're writing tests for the *inner* unit test loop, we're coding inside-out.

Inside-Out and Outside-In. These are also names for two schools of thought for TDD.

### Classic (Inside-Out/Chicago) and Mockist (Outside-In/London) TDD

Classic TDD (also known as *Inside Out* or *Chicago style TDD*) is a style of unit testing where we start from the unit tests, and build *outwards*, fleshing out the internal details of the objects we know we need. This is probably the most straightforward and commonly known form of TDD. We aim to master this in Part IV: Test-Driven Development Basics.

Whereas with Mockist TDD (also known as *Outside In* or *London style TDD*), we start at the boundaries of the application (typically the controller or use case) and end up coding *inwards*, creating *mock objects* for **infrastructural dependencies** needed to fulfill the acceptance test. We aim to master this in Part X: Advanced Test-Driven Development.

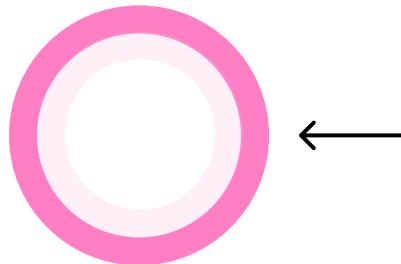
## Classic TDD

Chicago style  
Inside-out



## Mockist TDD

London style  
Outside-In



Both styles of TDD are useful. By using a mixture of both inside-out and outside-in, we can compose an elegant approach to TDD our way through a feature.

### Filling in gaps: Adding E2E and Integration Tests

For each *Acceptance Test* we've finished, we have to take a good and honest look at the feature and ask ourselves how confident we feel about it working in production. 95% of the time, you should be lacking some confidence. This is because the feature — the vertical slice or use case — relies on real-world infrastructure and 3rd party APIs (infrastructure adapters). To more easily exercise different scenarios for the use *Use Case*, we've mocked out these dependencies.

This is good. We know the application core functions the way it should. Now, we need to start the *integration* process. We need to see how our feature will work when we integrate it with the outside world. We will fill in the gaps with *Integration* and *E2E* tests next.

### Integration tests

As we work through making our *Acceptance Test* pass, we'll run into various infrastructure adapters and domain objects. The domain objects (things like User, Offer, Location and so on) we can test the creation and behavior of these things using *Unit Tests* as we're going along. But with infrastructure adapters — we should run a separate *Integration* test script. We want to test

Some people call the **ingress integration tests** *Driver Tests*. These are the tests which confirm that the correct application core code is called when a webhook, GraphQL API, or RESTful API is called. They also verify the requests return the correct errors and response codes when passed in bad client input.

And then others are called **outgoing integration tests** *Contract Tests*. These are the kinds of tests that verify that your databases, caches, or APIs and services behave according to a contract we've defined.

If a CustomerRepo interface required for a MakePurchase *Use Case* looks like this...

```

interface CustomerRepo {
  getCustomerById (id: string): Promise<Customer | Nothing>;
  getCustomerByEmail (email: string): Promise<Customer | Nothing>;
  save (customer: Customer): Promise<void>;
}

```

... then we'll need to confirm that when we create a *real* CustomerRepo, perhaps using Firebase as our database of choice...

```

class FirebaseCustomerRepo implements CustomerRepo {
  getCustomerById (id: string): Promise<Customer | Nothing> {
    ... // Implementation here
  }

  getCustomerByEmail (email: string): Promise<Customer | Nothing> {
    ... // Implementation here
  }

  save (customer: Customer): Promise<void> {
    ... // Implementation here
  }
}

```

... that we have tests which ensure that the all implementations (real and mock) behave as they should — that they adhere to the contract of the interface.

```

// Rough integration test
describe ('customerRepo', () => {

  let customerRepos: CustomerRepo[] = [];

  beforeEach(() => {
    customerRepos = [
      new FirebaseCustomerRepo(), // The real repo
      new MockCustomerRepo()     // The fake repo
    ]
  })

  it ('can get a customer by id', async () => {
    let customer = new Customer(...);

    // Test all implementations
    for (let customerRepo of customerRepos) {
      await customerRepo.save(customer)
      let customerResult = await customerRepo
        .getCustomerById(customer.getId());
      expect(customerResult).toBeDefined();
    }
  })
})

```

```
        expect(customerResult.getId()).toEqual(customer.getId());
    }
}

...
})
```

## E2E tests

Now, it's good and all that we have our integrations working, but how do we know that if a real user logged on and tried to use our application, it'd work? We don't. We need to test end-to-end.

How should we write the *E2E Test* if we just finished an *Acceptance Test* at the *Use Case* level? Well, you can actually copy the *Acceptance Test* spec and set it up in the front-end as an *E2E Test*. How we go about verifying correctness might change a little bit, because on the back-end, we verify correctness by checking values returned by functions and methods. On the front-end, we verify correctness by *perceiving* what's on screen.

It may also be quite challenging to write *E2E Tests* as *Acceptance Tests* for certain features, like say a Password Reset feature. How would you get access to the email that gets sent to you with the code to reset your password?

Nowadays, with E2E testing tools like Cypress, they advise for you to periodically run your E2E tests locally so that you can handle situations like this more elegantly. The Cypress docs say:

“The [most] important reason why you want to test against local servers, is the ability to **control them**. When your application is running in production you can’t control it.

When it's running in development you can:

- take shortcuts
- seed data by running executable scripts
- expose test environment specific routes
- disable security features which make automation difficult
- reset state on the server / database

With that said - you still have the option to have it **both ways**.

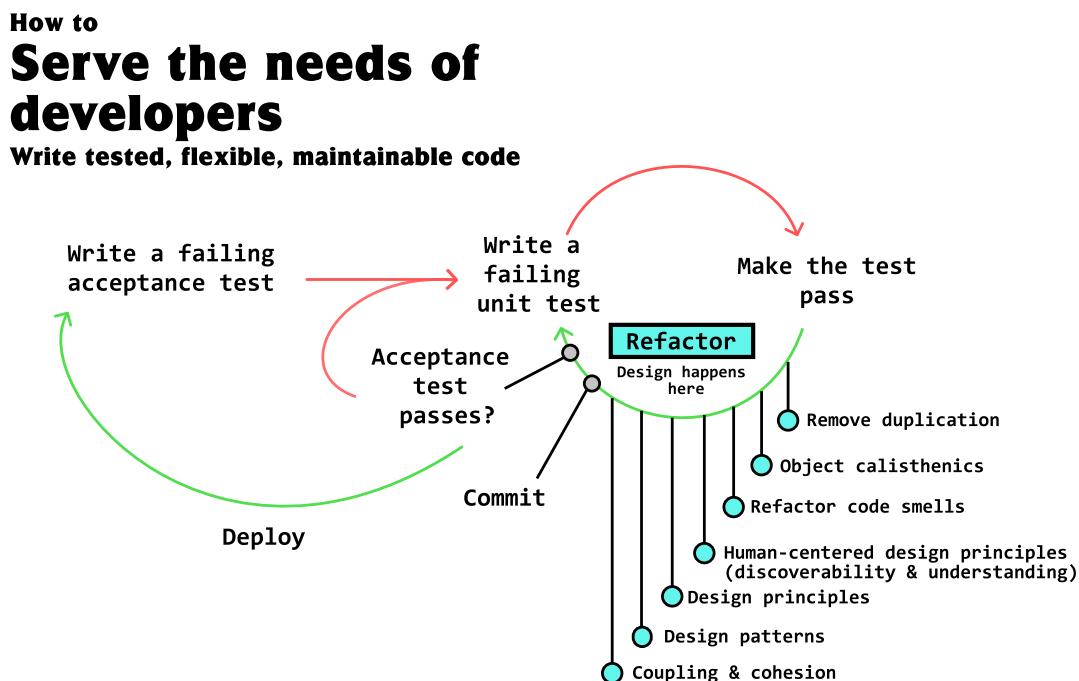
Many of our users run the *majority* of their integration tests against a local development server, but then reserve a smaller set of **smoke tests** that run only against a deployed production app.”

I believe this is OK, but it's not perfect. Steve Freeman still recommends that you test E2E in the most production-like environment as possible. That means *not* your local machine. Maybe you deploy to a staging environment and it runs a specific type of *Acceptance Tests*. Maybe you decide you'll run some tests on your local machine, and others, against servers and in staging. I wish I could tell you that there's a simple, one-off solution, but unfortunately, there is no silver bullet here. No perfect one-size-fits-all solution. The only solution is understanding the problem and communicating how we're going to handle it.

# Refactoring is where design happens

Refactoring is the final step of the TDD cycle before we decide to write a new failing test. Refactoring is actually something you do several times every hour, not something you wait until a few sprints later to do.

It is within the refactor step that we have the opportunity to improve our designs with the safety of knowing that we can fall back to a commit that works.



So far, we have a good understanding of what constitutes clean code and more specifically, code that humans can work with. The refactor step gives you an opportunity to apply our learnings from Part II: Humans & Code to your codebase after you make the test turn green from red.

In Part IV: Test-Driven Development Basics, we cover the core (classic) TDD techniques and focus on the fundamentals, like removing duplication, triangulation, how to write good tests.

In Part V: Object-Oriented Design With Tests, we learn about Object Calisthenics, code smells, and how to use tests to guide the design of objects.

In Part VII: Design Principles, we'll further improve the refactor step because we'll work on learning about the most important design principles

You typically don't want to implement *Design Patterns* unnecessarily. They can be additional complexity. In Part VI: Design Patterns, we learn about the most common ones and identify when and how to refactor to patterns.

In Part VIII: Architecture Essentials, we revisit the two most essential metrics with respect to gauging the *structural* quality of your code: coupling and cohesion. Through practice, you'll be able to tell when you need to refactor your code to improve these two metrics.

## Summary

- TDD is the fundamental technique that gives us the ability to *integrate the principles of good software design* in a safe, repeatable, non-reversible way.
- There are quite a few steps involved to preparing to do TDD correctly in a real-world web application. At the very minimum, you need a testing strategy, a testing architecture, separate scripts, and the ability to switch between environments.
- Real-world TDD is hard for a number of reasons: front-end vs. backend TDD is different, we are still struggling to agree on testing terminology, we tend to start TDD too late, and understanding what to test (and how to test it) is not widely known.
- TDD has two schools of thought: Classic (Inside-Out) and Mockist (Outside-In). When we perform Double Loop TDD, we use both.
- We use Double Loop TDD to build and test the application core, and we add Integration and E2E Tests to gain confidence that our code will work correctly in the real-world.
- We should be refactoring constantly. Each refactoring is an opportunity to integrate everything we've learned about good design. TDD gives us a safety net to fall back on if things go astray.

## In conclusion of Phronesis

We've reached the end of our journey exploring the Phronimos techniques. We've learned how to learn the domain, identify the features, set them up as user stories and convert them into passing acceptance tests within an object-oriented architecture.

If that sounds like a lot, it really is. We covered many topics at a high-level in Phronesis. However, now we can say we understand how we truly accomplish the two-sided goal of software design. We know the business practices necessary for the customer and we know *about* the technical practices that'll keep future maintainers happy.

My goal then, for the remainder of the book, is to dive deeper into all of the technical topics. Throughout the rest of the book, we'll explore each topic at the appropriate low-level detail so that Phronesis acts as a reminder of how to be in the world as a software developer.

## References

### Articles

- <http://www.extremeprogramming.org/rules.html>
- <https://twitter.com/GeePawHill/status/1416489659799060486>
- <http://tpierrain.blogspot.com/2021/03/outside-in-diamond-tdd-1-style-made.html>
- <http://tpierrain.blogspot.com/2021/03/outside-in-diamond-tdd-2-anatomy-of.html>
- <https://khalilstemmler.com/articles/test-driven-development/introduction-to-tdd/>

- <https://khalilstemmler.com/articles/test-driven-development/use-case-tests-mocking/>

## Part IV: Test-Driven Development Basics

■ Tests have a lot of benefits. As we learned in *phronesis*, the best way to write testable code is to start by writing a failing test. In this section, you master the basics of the TDD technique, starting with the first of two TDD schools of thought: Classic TDD.

### 29. Getting Started with Classic Test-Driven Development

■ TDD is a critical technique for building well-crafted software. Acting as the highest level of abstraction, we describe behavior — the essential complexity — using English and write the simplest possible object-oriented code to realize it. In this chapter, we learn the core mechanics of the *classic* TDD school of thought. Here, we begin to hone our design skills with real exercises.

#### Chapter goals

In this chapter, we:

- Learn about Classic TDD and why we'll be mastering the classic school of thought first
- Recap the core mechanics of Classic TDD
- Discuss methods for pushing the design forward using tests
- Learn why you prefer using real examples to abstract concepts in your tests
- Present some unit testing exercises you can use to start building your skills

#### About Classic TDD

As we discussed in the 28. Test-Driven Development Workflow, there are two schools of thought for TDD: *classic* and *mockist*.

Classic TDD — created originally by Kent Beck and known as the Detroit/Chicago school of thought for TDD — is what we're going to be learning first.

#### How is Classic TDD different from Mockist TDD?

What makes Classic TDD *classic* is the absence of mocking. We verify our classes or functions by testing them exactly as they occur without mocking out dependencies. This means that if some class we wish to test relied on the use of a database, we'd be testing that class *with* the database connection as well. While gives you a greater level of confidence that your code is working correctly, for code involving *infrastructure code*, it makes test setup and teardown harder and it makes them run slower as well slower. And as we discussed in 25. Testing Strategies, such a test that involves infrastructure is actually \*\*an integration test.

In the real world, we want to use a mixture of *classic* and *mockist* tests for different types of tests. Classic TDD is usually the right tool for the job if:

- We wish to unit test a pure function or classes that doesn't contain any infrastructure dependencies
- We're writing integration tests, and we are concerned with verifying that our integrations work — therefore, we mock nothing

When we get to the *mockist* approach in the later part of Part V: Object-Oriented Design With Tests, you'll learn that it works best when we're trying to isolate a *subject under test (SUT)* containing dependencies like APIs and databases (see Use Case Testing).

■ **Mockist TDD:** We learn about how to isolate a SUT using Mockist TDD in 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - coming soon.

### Why start with Classic TDD?

We first need to internalize the TDD technique. We have to get used to first writing the test and using our words — English, or whatever spoken language you know — to evolve designs. If we can't express what we want to build with English, then we'll be sure to encounter troubles when we attempt to express it in code. At this point, the practice we need is within learning to write tests driven by behavior and example (essential complexity) instead of getting too caught up in the weeds of implementation (accidental complexity).

We start with classic TDD because it's the simplest for beginners. At first, we'll use it to solve mostly procedural problems. As we practice, we'll move onto object-oriented problems.

### TDD is mostly a design tool

Yes, we end up with tests, but make no mistake — TDD is a design tool that helps you write better code, especially *object-oriented* code. While you can complete the examples in Part IV: Test-Driven Development Basics using functions, in Part V: Object-Oriented Design With Tests, you'll want to switch to using classes predominantly because we'll explore many topics and techniques necessary to build up your repertoire of OO design skills.

### You must do the examples

Throughout this chapter (and the next two sections of the book), you'll see a number of TDD exercises for you to try out. I strongly urge you to run through them yourself. The only way to get good at this TDD is to do the examples. Don't treat it as solely intellectual stimulation. You need to put in a lot of reps yourself. Some object-modellers practice TDD examples every morning just to keep sharp at it. I don't think you need to go to those extremes, but at least attempt to solve each problem once.

If you need a starter project, you can always use the simple typescript starter @stemmlerjs/simple-typescript-starter.

■ **Solutions:** Lastly, if you get stuck, you can find the solutions in stemmlerjs/solidbook-tdd-examples.

Now then, let's get into it.

## The core TDD mechanics (recap)

In 28. Test-Driven Development Workflow, we learned two important sets of rules to do TDD properly. As a reminder, they were:

- **Red-Green-Refactor**
  - Red — Write a failing test
  - Green — Write just enough code that will pass the failing test
  - Refactor — Criticize the design and refactor the code, keeping the tests intact
- **The three laws of TDD**
  1. You are not allowed to write any production code unless it is to make a failing test pass
  2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
  3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test

■ **Rules:** Remember that rules come from principles and principles come from values. While it is true that rules are susceptible to dogmatism, we should first master the rules so that we can better understand the principles. At that point, we can break the rules when we see fit.

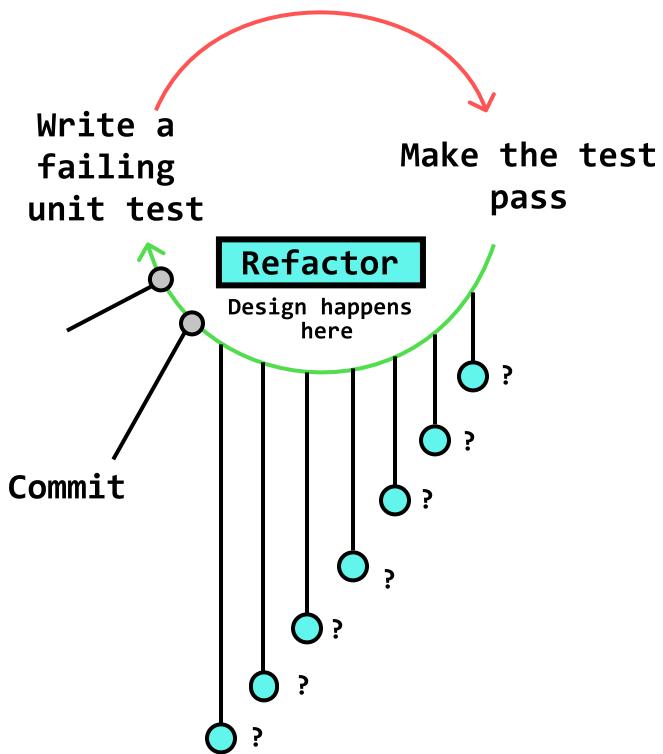
■ **Don't break the rules yet:** For the time being, don't break the TDD design rules we list. As we advance in the following chapters, we'll gradually build onto them.

## Design happens during refactoring

If we were to kick off your TDD journey today using only these two sets of rules, we'd be able to at least get started writing our first few tests. However, neither rule advises us in how to adequately perform the *refactor* stage of the red-green-refactor loop.

Since **the refactor step is where we perform design**, if we were told to merely refactor our designs, that'd leave the floor open for us to do *anything*. We might add new abstractions, split code into different files, add *nice-to-haves*, use *design patterns*, and so on.

Let's not do any of that just yet.



Since we're just starting out, we need to constrain the surface area for potential refactoring. Let's start small. We'll gradually incorporate additional rules to inform our refactoring efforts.

Our first rule will be to adhere to **The Rule of Three**.

### The rule of three

The rule of three dictates that you should **only clean up duplication when you come across it three times.** \*\*\*\*Why?

You've probably heard of the DRY principle — do not repeat yourself. The idea is to remove duplicate code when you see it. This is a good thing to do because duplicated code can result in coupling and cohesion problems. It's a good idea to refactor duplication into the appropriate abstraction to have a single place to maintain potential changes.

However, the problem with cleaning up duplication immediately is that **it is way easier to invent the wrong abstraction than it is for us to invent the right one.**

This is the idea of *premature optimization* discussed in 1. Complexity & the World of Software Design. Instead, what we should do is *wait*. Let duplication occur *twice* for now. Yes, twice. **A little bit of duplication is 10x better than an incorrect, complex, or wrong abstraction.**

However, once you see duplication three times, go ahead and refactor it.

This will take a little bit of discipline to put into practice, admittedly. You'll have to train your brain to handle some temporary messiness in your code. But that's OK, because if we follow the TDD process, it'll get cleaned up when it needs to.

■ **Design rule:** Remove duplication using The Rule of Three. When you see duplication for the third time, refactor it.

■ **Tip:** Refactoring using *The Rule of Three* applies to your test code as well as it does to your production code. For example, if you notice yourself doing the same setup code three times in your tests, think to refactor it out to a `beforeEach` abstraction in the test runner.

## How to push the design forward with TDD

Now that we know the rules, let's talk about how this works in practice. Assuming you've written a *failing test*, what are the different ways we can use TDD to guide the design?

### Making a test pass (red to green)

As Jason Gorman says, "In TDD, our goal is to discover the design in baby steps, verifying that it works and cleaning up our code as we go." To make a failing test pass, we have two strategies. They are:

1. Fake it
2. Obvious implementation

**Fake it:** With the *fake it* technique, we fake the implementation to make the test pass. Here's a simple example. Assume a test against a `StringUtil` class's ability to check text length. We could write a test like:

```
describe('StringUtil', () => {
  it('knows that the length of "hello" is less than 10', () => {
    expect(StringUtil.isLessThanLength("hello", 10)).toBeTrue();
  })
})
```

The faked implementation would look like the following:

```
class StringUtil {
  public static isLessThanLength (text: string, length: number): boolean {
    return true; // This is the simplest thing we can do!
  }
}
```

Since this is the only test we currently have, it is extremely easy for us to make this failing test turn green. However, as we add more tests, we'll obviously be forced to change the implementation to make all of the tests pass.

**Obvious implementation:** The second method of making a test pass is to provide the *obvious implementation*. Sometimes, we know exactly how to make the test pass. Perhaps we have the solution in our head already. For example, taking the same test against a `StringUtil` class, we could have written:

```
class TextUtil {  
    public static isLessThanLength (text: string, length: number): boolean {  
        return text.length < length;  
    }  
}
```

That works! And it's still pretty simple, right? Yes, but we want to be careful here. Some implementations — while they may seem simple — are more complex than others. And when we introduce overly complex implementations too soon, we run the risk of running getting stuck.

We should prefer the simplest possible *transformation* possible to continue to flesh out more behavior through expressive tests.

■ **Transformations:** We discuss code transformations, problems that arise, and how to prevent them when using TDD in 31. Avoiding Impasses with the Transformation Priority Premise.

### Writing the next test (green to red)

When you're ready to write the next test, you can:

1. Move onto new \*\*behavior or
2. Add another example (related to the *previously* completed test, perhaps to explore edge cases)

Regardless, what we're doing is performing something called **triangulation**.

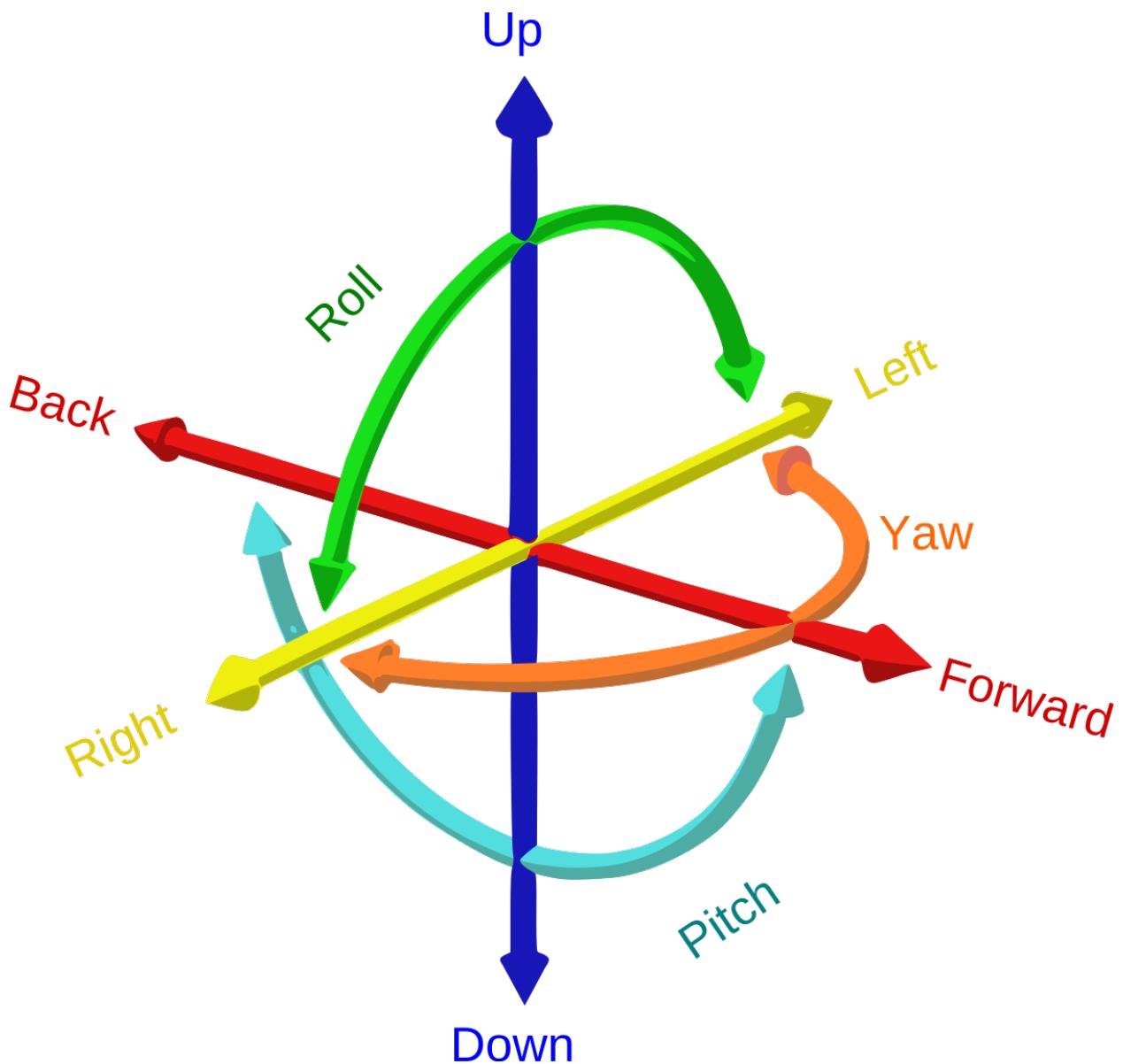
### Triangulation

Triangulation is way to describe the fact that as we add more tests, we're sculpting the design towards a general, robust, and hardened solution that knows how to handle whatever we throw at it.

The authors of the book *Agile Technical Practices* \*\* \_\_relate the concept of triangulation to the idea of *degrees of freedom* from mechanics\_..

■ **Degrees of Freedom:** In three dimensional space, there are six degrees of freedom. This describes all of the ways that an object can move. The types of degrees of freedom are: forward/backward (surge), up/down (heave), left/right (sway), yaw (normal axis), pitch (transverse axis), and roll (longitudinal axis).

Cars have three degrees of freedom: surge, sway, and yaw. Planes also have three: pitch, roll, and yaw. Boats, however, have all six.



If we were to implement a Car class, we wouldn't be done until we've implemented all three of the different degrees of freedom. This is the necessary behavior for the car.

By triangulating using more tests, **we carve out the various degrees of freedom** — or the capabilities — of our designs.

To move the design forward, performing #1 - *Move onto new behavior* would be like finishing the forward/back behavior and then moving onto implementing left/right. Performing #2 - *Add another example* is to spend more time on the forward/back functionality, perhaps by evolving what happens when we try to go back when the car is turned off or out of gas.

### Naming tests

### Basic testing pattern

In 22. Acceptance Tests, we explored a couple of different ways to write tests — specifically *acceptance tests*. To write *unit tests*, we'll find it easiest to stick with the “**Format #1: Single-**

“line tests” pattern. Using Jest, it looks like the following:

```
describe('guitar', () => {
  it('should do something when', () => {
    // .. Implementation
  })

  ...
})
```

Or you could write it as:

```
describe('guitar', () => {
  test('when I do something, then something happens', () => {
    // .. Implementation
  })

  ...
})
```

You’ll notice that, in this format, we use the words *when* and *then*, but not *given*. For the majority of our beginner unit testing examples, we won’t be relying on state, therefore — there will have been *no precondition* — no *\*\*given*.

Example:

- Testing that a stateless `add(a: number, b: number)` function works properly doesn’t require a *given*, but testing that *a resumable chess game* has all the pieces in the right place at the start of a game *would* require a *given*.

### Write your tests based on behavior, not implementation

When naming tests, keep in mind that we’d like to:

- Use behavior-driven test names that mean something with respect to the domain
  - If the domain has to do with *fishing*, use *fishing* terminology
  - If you’re testing a calculator object, use *mathematics* terminology
- Avoid using technical names and referring to primitives

■ **Design rule:** Write BDD-style tests using the language of the domain instead. Avoid technical names and leaking implementation details in test names.

One of the examples in this chapter is to build a palindrome checker. Here’s an TDD solution to this problem that I found on the internet. Try to take notice of how it violates this rule.

```
it("should be able to manage its state", () => {
  const palindrome = new Palindrome('1');
  expect(palindrome).toHaveProperty("data", 1);
})
```

In 23. Programming Paradigms, we talked about how we should assume encapsulation and only access values through the use of methods. The first issue is that this test leaks

implementation details — it assumes the implicit by accessing a value through the `toHaveProperty` API instead of through a method.

Secondly, and most importantly — **we don't need state for this problem**. This can be done statelessly. But since the test was written from an implementation-detail standpoint instead of a *behavior of the domain* standpoint, it's easy to see how state was introduced. **Focus on what needs to happen instead of how it needs to happen**. At the end of this chapter, we demonstrate a better way to do this.

■ **Naming principles:** Revisit 8. Naming things and 22. Acceptance Tests to review BDD and good naming principles.

### Prefer concrete examples instead of abstract statements

By providing lots of examples, we can evolve the design to understand what it is and what it does. Sometimes, we're designing something that is based on abstract or novel concepts. Maybe the domain is something new to us. For better understandability, we should provide concrete examples.

What do I mean by this? Here's an example.

```
// Instead of  
'can synchronize midi pedals'  
  
// Prefer  
'when I sync a delay and a reverb pedal, I get a single audio output channel'
```

And here's an even simpler one.

```
// Instead of  
'can add two numbers'  
  
// Prefer  
'addition'  
  'when I add 2 + 2, I get 4 back'  
  'when I add -2 + 2, I get 0 back'
```

By providing concrete examples, we end up with tests that are more easily readable, less abstract, and it's more conducive to write potential edge case tests this way.

■ **Design rule:** Prefer evolving the design with concrete examples instead of abstract statements.

### One example per test

When we use abstract statements, it tends to lead us down the road of maintaining multiple examples within a single test. While some tools allow you to do this, I would recommend against it so that test failures are easily traceable. If the tooling allows you to do this in a traceable way, by all means — go ahead.

■ **Design rule:** Prefer one example per test so that tests are readable and failures are easily traceable.

## Exercises

That's enough rules for now. Let's do some examples!

■ **Reminder:** You *must* practice on your own time to get good at TDD.

### Palindrome (partial demonstration)

■ **Description:** Create a palindrome checker. It should be able to detect that a string is a palindrome: that is, it is the same word or phrase in reverse. Words like "mom" are palindromes. Words like "bill" aren't palindromes. It should still know that something is a palindrome, even if the casing is off. That means that "Mom" is still a palindrome. It should also be able to detect palindromes in phrases like "Was It A Rat I Saw" and "Never Odd or Even" too, regardless of spacing.

Let's pull in the simple typescript starter repo and get building:

```
git clone https://github.com/stemmlerjs/simple-typescript-starter.git
cd simple-typescript-starter
npm install && npm run test:dev
```

■ If you prefer to see the demonstration *visually*, you can watch me code it here.

I. **Write the failing test.** The first test I'll write is that the palindrome checker can tell that "mom" is a palindrome. I will:

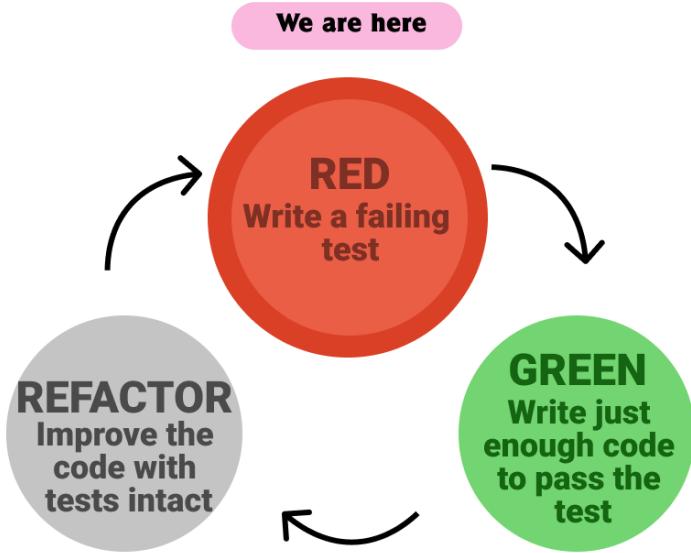
- Write the test name using the requirements
- Pretend that something called `palindromeChecker` exists and that it has an `isAPalindrome` method on it.
- Expect the method to return `true` for `mom`
- Save

```
// index.spec.ts
describe('palindrome checker', () => {

  it('should be able to tell that "mom" is a palindrome', () => {
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); // 
  });

})
```

At this point, our test should be failing because nothing named `palindromeChecker` exists and it doesn't have an `isAPalindromeMethod`.



**2. Write the simplest code to make the test pass.** Let's move to green. In this next step, we:

- Create a PalindromeChecker class
- Give it an isAPalindrome method
- Return true (the simplest thing that would work)
- and import it in our test

Implementing this, the index.ts containing our PalindromeChecker looks like this:

```
// index.ts
export class PalindromeChecker {
  isAPalindrome (str: string): boolean {
    return true; // This is the simplest thing!
  }
}
```

And our test file now looks like this:

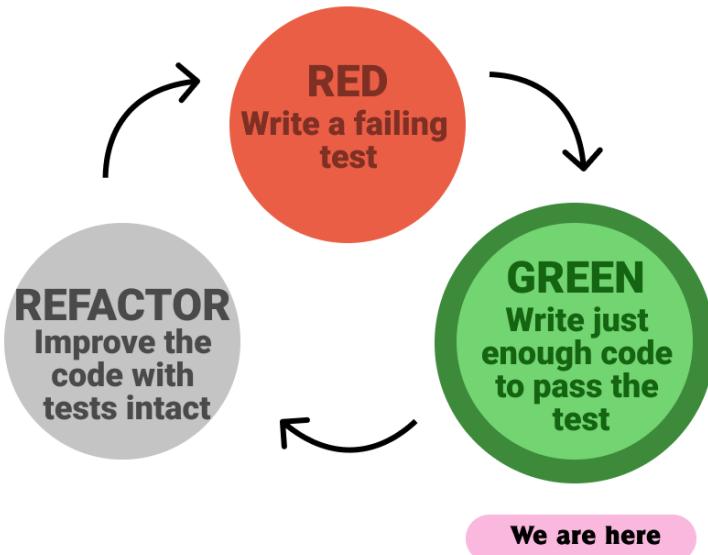
```
// index.spec.ts
import { PalindromeChecker } from './index'

describe('palindrome checker', () => {

  it('should be able to tell that "mom" is a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); //
  });

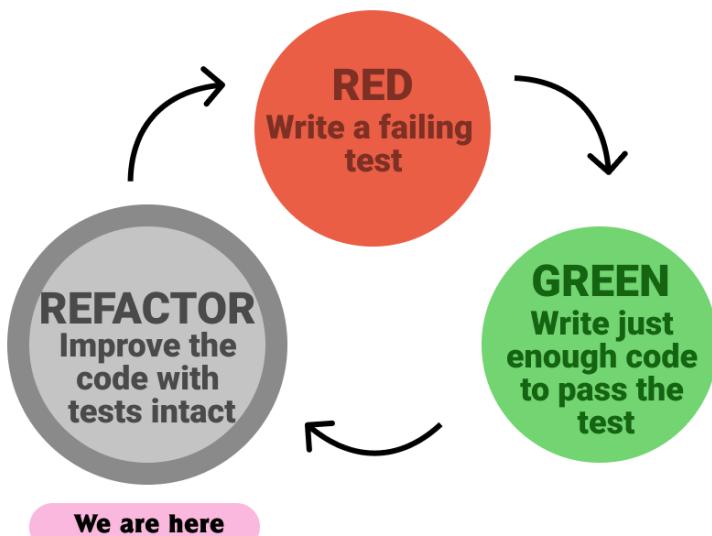
})
```

Our test passes .



Let's move to the next step.

**3. Refactor.** When refactoring, keep a lookout for duplication (at least three times).



However... There's none here yet, so let's move to the next failing test.

**4. The next failing test.** Let's grab the next test. We're going to test that "bill" *isn't* a palindrome. Our tests should look something like this now.

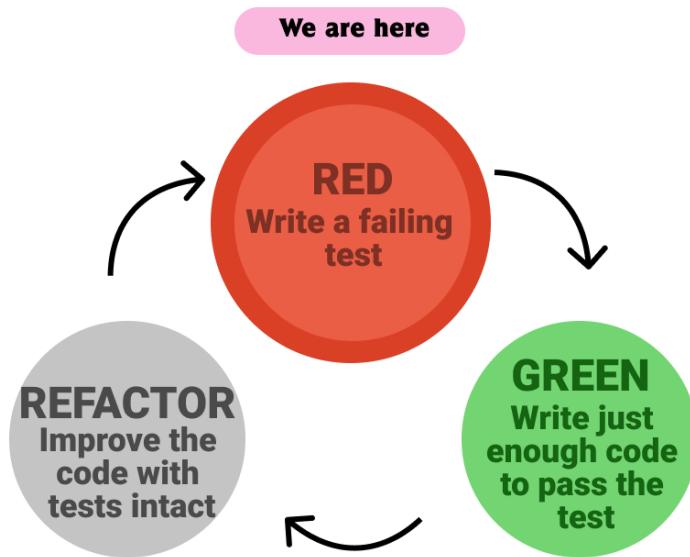
```
// index.spec.ts
import { PalindromeChecker } from './index'

describe('palindrome checker', () => {

  it('should be able to tell that "mom" is a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); // 
  });

  it('should be able to tell that "bill" isn't a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy(); // 
  });
})
```

Since our test fails, we're in the *red* phase.



Let's turn it green.

**5. Write the simplest code to make the test pass.** Currently, this is what our `PalindromeChecker` class looks like:

```
// index.ts
export class PalindromeChecker {
  isAPalindrome (str: string): boolean {
    return true; // This is the simplest thing!
  }
}
```

There are several ways to go from Red to Green. If we're really writing the *simplest possible thing that would work*, then we'd likely find no simpler thing to do than this:

```
// index.ts
export class PalindromeChecker {
    isAPalindrome (str: string): boolean {
        if (str === 'mom') {
            return true;
        } else {
            return false;
        }
    }
}
```

But we know where this leads — so let's implement the *Obvious Implementation* of reversing the string and seeing if it matches the originally provided one.

```
// index.ts
export class PalindromeChecker {
    isAPalindrome (str: string): boolean {
        const reversed = str.split("").reverse().join("");
        return reversed === str;
    }
}
```

And if we save that, we should notice that our test passes.

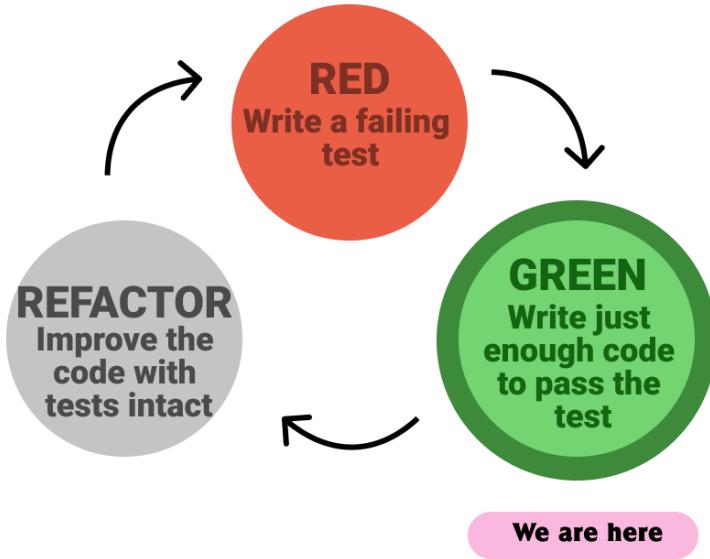
```
// index.spec.ts
import { PalindromeChecker } from './index'

describe('palindrome checker', () => {

    it('should be able to tell that "mom" is a palindrome', () => {
        const palindromeChecker = new PalindromeChecker();
        expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); // 
    });

    it('should be able to tell that "bill" isn't a palindrome', () => {
        const palindromeChecker = new PalindromeChecker();
        expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy(); // 
    });
})
```

Nice. Now we're in *Green* again.



**6. Refactor.** Refactoring doesn't just mean refactoring production code. It means refactoring our test code too. And you may have noticed that there's some duplication in our test setup.

```
// index.spec.ts
import { PalindromeChecker } from './index'

describe('palindrome checker', () => {

  it('should be able to tell that "mom" is a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); //
  });

  it('should be able to tell that "bill" isn't a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy(); //
  });
})
```

However, we're going to use the *Rule of Three* to remove duplication. So we'll leave it for now.

**7. The next failing test.** The last test I'll demonstrate is the test case that specifies that a word is still a palindrome, even if it includes capitals and lowercase letters. The business rule here dictates that a word is a palindrome regardless of if it is capitalized or not. Let's write the failing test.

```
// index.spec.ts
import { PalindromeChecker } from './index'
```

```

describe('palindrome checker', () => {

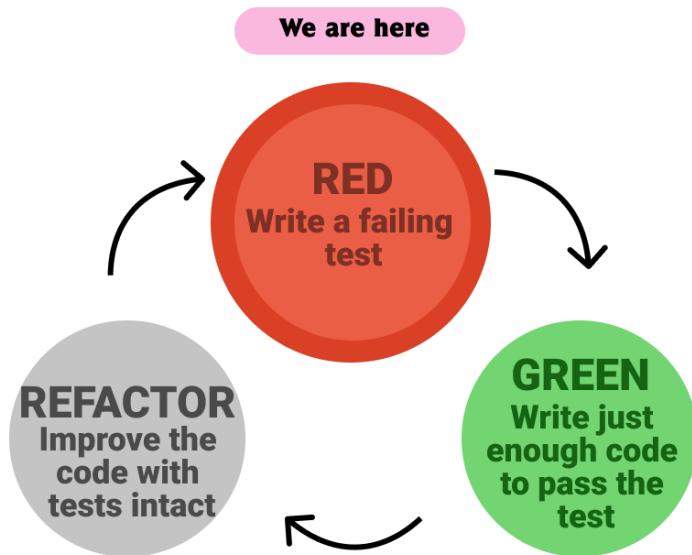
  it('should be able to tell that "mom" is a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); // 
  });

  it('should be able to tell that "bill" isn't a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy(); // 
  });

  it('should still detect a palindrome even if the casing is off', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome("Mom")).toBeTruthy(); // 
  });
})

```

Aha! Look. We have that duplication we were talking about a moment ago. **But we're not in the refactor phase yet.** We're about to turn this *Red* test *Green*.



We impose a little bit of discipline here and shift our focus to making this test pass.

**8. Write the simplest code to make the test pass.** At first glance, the simplest thing to do is to merely add `.toLowerCase()` to both the original and the reversed strings. So let's do that.

```

// index.ts
export class PalindromeChecker {

```

```

isAPalindrome (str: string): boolean {
  const reversed = str.split("").reverse().join("");
  return reversed.toLowerCase() === str.toLowerCase();
}

```

And if we save it and check our tests, we should see it pass as well.

```

// index.spec.ts
import { PalindromeChecker } from './index'

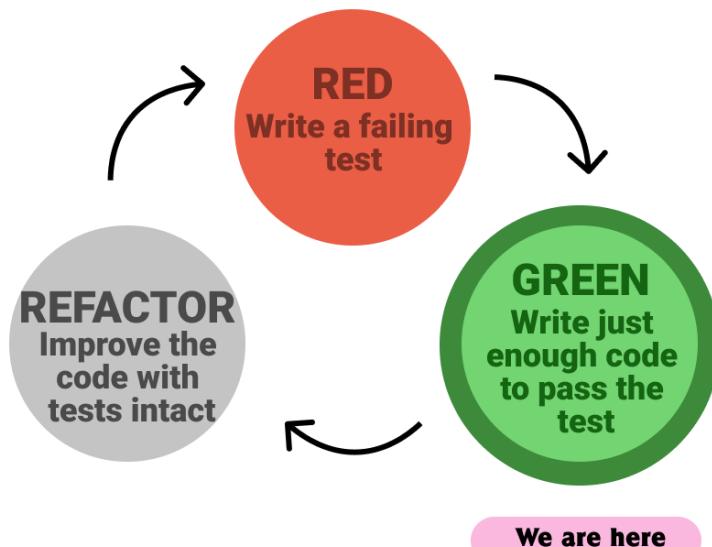
describe('palindrome checker', () => {

  it('should be able to tell that "mom" is a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); // 
  });

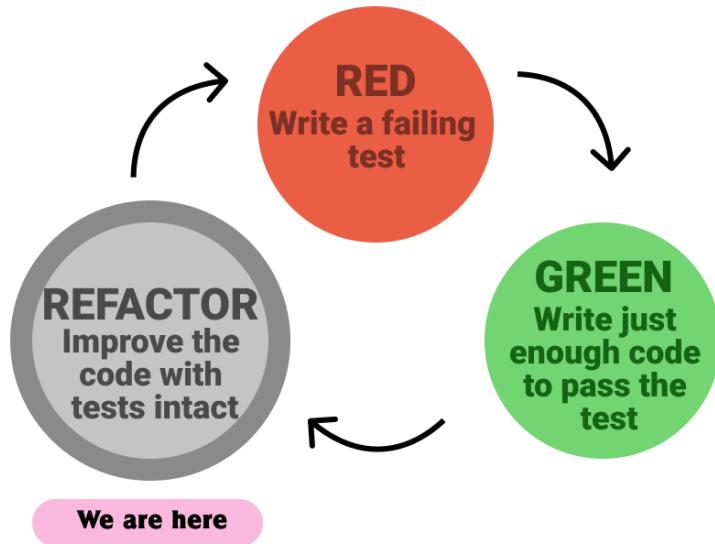
  it('should be able to tell that "bill" isn't a palindrome', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy(); // 
  });

  it('should still detect a palindrome even if the casing is off', () => {
    const palindromeChecker = new PalindromeChecker();
    expect(palindromeChecker.isAPalindrome("Mom")).toBeTruthy(); // 
  });
})

```



**9. Refactor.** And now that we're in the refactor phase, we can look to improving that duplication we saw earlier.



Jest, our test runner, has a way to specify things that we must do *before each* test. And creating the PalindromeChecker is what we'd like to do here.

We can clean up our code with the following:

```
// index.spec.ts
import { PalindromeChecker } from './index'

describe('palindrome checker', () => {

  let palindromeChecker: PalindromeChecker;

  beforeEach(() => {
    palindromeChecker = new PalindromeChecker();
  })

  it('should be able to tell that "mom" is a palindrome', () => {
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy(); //
  });

  it('should be able to tell that "bill" isn't a palindrome', () => {
    expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy(); //
  });

  it('should still detect a palindrome even if the casing is off', () => {
    expect(palindromeChecker.isAPalindrome("Mom")).toBeTruthy(); //
  });
}
```

```
    });
})
```

That looks good to me!

**Continue.** We can continue on like this by writing more and more failing tests, making the code more generic to make them pass, and improving the design.

I'll leave the next two tests for you to do.

```
// index.spec.ts
import { PalindromeChecker } from './index'

describe('palindrome checker', () => {
  let palindromeChecker: PalindromeChecker;

  beforeEach(() => {
    palindromeChecker = new PalindromeChecker();
  })

  it('should be able to tell that "mom" is a palindrome', () => {
    expect(palindromeChecker.isAPalindrome('mom')).toBeTruthy();
  });

  it('should be able to tell that "bill" isnt a palindrome', () => {
    expect(palindromeChecker.isAPalindrome('bill')).toBeFalsy();
  });

  it('should still detect a palindrome even if the casing is off', () => {
    expect(palindromeChecker.isAPalindrome("Mom")).toBeTruthy();
  });

  it('should be able to tell that "Was It A Rat I Saw" is a palindrome', () => {
    // You do this one!
  });

  it('should be able to tell that "Never Odd or Even" is palindrome', () => {
    // And this one!
  })
})
```

## Fizz Buzz

**Description:** The definitive TDD exercise. Create a function (or a class) that accepts only the numbers from 1 to 100 and throws an error otherwise. For multiples of three, it should return the word “Fizz”. For multiples of five, it should return the word “Buzz”, and for multiples of both three and five, it should return “FizzBuzz”. For anything not a multiple of three or five, it should return the inputted number, but as a string.

## Nth Fibonacci

■ **Description:** The Fibonacci sequence is an interesting mathematical phenomenon. Starting from 0 and 1, each number is the sum of the two preceding ones. This means that the first ten numbers in the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Notice that the 2 can be found by adding the two numbers before it ( $1 + 1$ )? And notice that the 3 can be found by adding the two numbers before it as well ( $1 + 2$ )? Assuming indexes start at 0, write a function (or a class) that accepts any positive integer index and generates the Fibonacci number for that nth position.

Example: `fibonacci(3)` should return 2.

## Summary

### TDD considerations so far

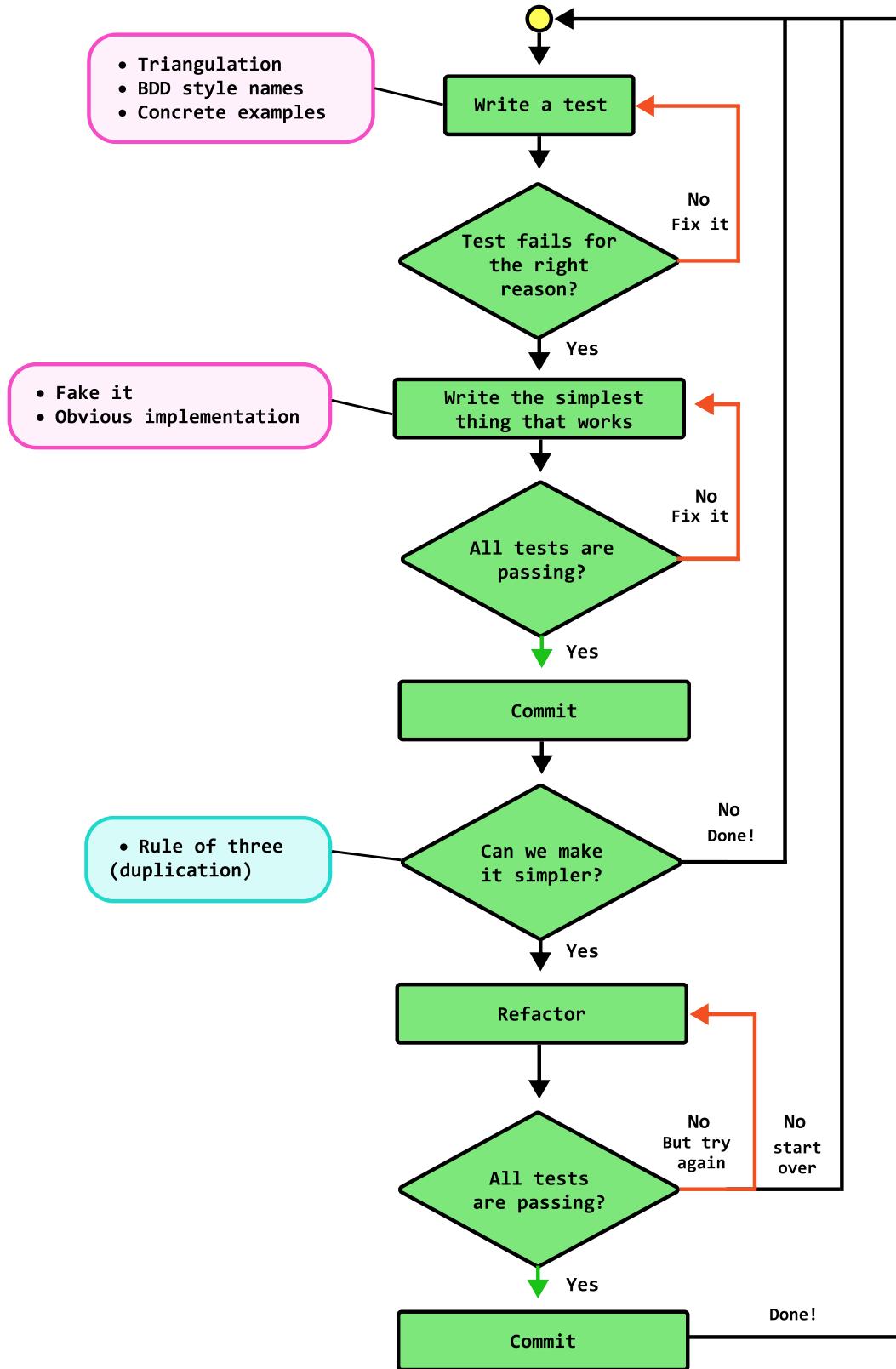
Here's a handy-dandy list of everything we've learned so far.

- **When writing a test**
  - Use behavior-focused tests that are meaningful to the domain
    - \* Avoid leaking implementation details
    - \* Avoid using technical language
  - Prefer concrete examples instead of abstract statements
  - Prefer one example per test to keep tests readable and failures traceable
  - Before continuing, ensure that the test is failing for the right reason — compilation errors or missing imports are not valid reasons for failure.
- **To make a failing test pass**
  - Use the fake it or obvious implementation strategies
- **After the test passes**
  - Use the rule of three to refactor duplication
- **To write the next test**
  - Triangulate new behavior with a different example
  - Triangulate the same behavior with different examples

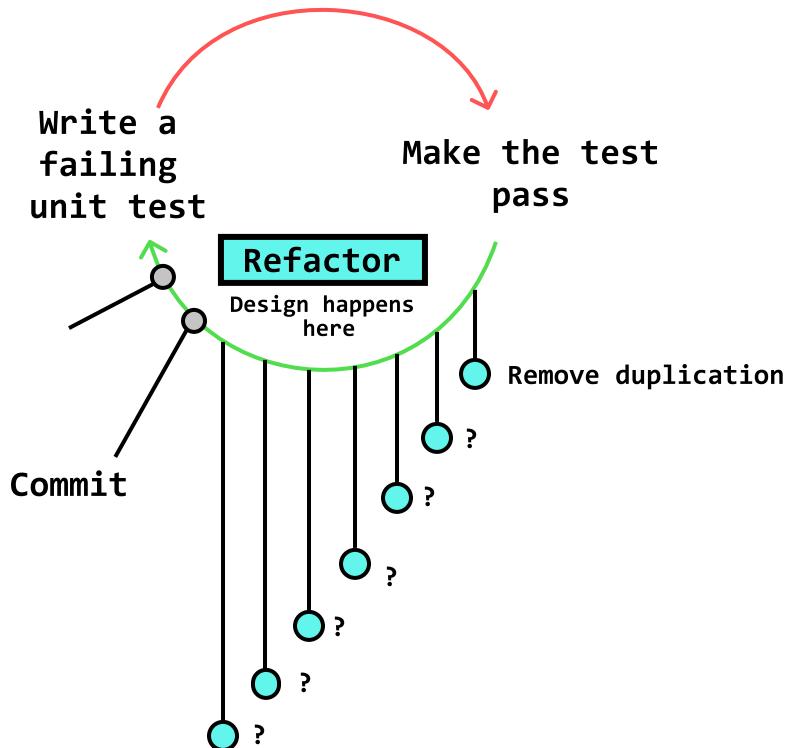
## Current test-driven workflow and design rules

Let's take a look at what our workflow looks like so far. Here's a workflow diagram originally inspired by Rachel M. Carmena \*\*\*\*(@bberrycarmen)'s drawings of Kent Beck's TDD workflow but slightly changed from the one reworked by the authors of the excellent *Agile Technical Practices* book.

As we progress with our TDD and OO skills, we'll add new learnings and techniques to this workflow to optimize it, specifically within the *design phase*.



If you'll recall, the design aspect of TDD is focused on making things simpler. This happens predominantly in the refactor step. For now, all we're concerned about is removing duplication (while adhering to the rule of three). We'll add onto this in the next chapter.



## References

### Articles

- Introduction to Test-Driven Development (TDD) with Classic TDD Example
- The Mysterious Art Of Triangulation
- Fake It — C2 Wiki

### Books

- Agile Technical Practices Distilled by Marco Consolaro and Pedro M. Santos
- Test-Driven Development by Example by Kent Beck

## 30. Working Backwards using Arrange-Act-Assert

Some tests are really tricky to write. We may not know how to express the behavior right away. Or maybe we're stuck on the name. In this chapter, we learn how to get unstuck with challenging tests by writing the test *backwards*. By first focusing on a way to prove that the test *could* work, we tend to construct solutions more naturally.

## Chapter goals

Here, we will:

- Learn three test-writing challenges you're likely to encounter at this point
- Learn how to structure your tests into act-arrange-assert phases
- Learn how to write your tests backwards
- Practice writing tests backwards to tap into our design creativity

## Three likely challenges you'll encounter

If you're just starting with TDD, it's possible that after the last chapter, you may have encountered one of the three following challenges:

**Not sure how to name the test** — Sometimes we know exactly what it is that we want to do, but naming the test is tricky. This often occurs if we're trying to come up with abstract, clean, generic test names.

- Instead, *use concrete examples to guide test names.*
- Remember that *naming is a process* (see 8. Naming things)

**Not sure how to express the behavior** — We know what to name the test, and we know how to prove that it'd work, but as for how to *make it work*, we have little idea.

- A concept called the *Transformation Priority Premise* can give you plenty of hints for the simplest possible solution. We cover this in the next chapter.
- *OO design patterns and OO principles also help.* We cover this in future sections.

**No idea how to prove correctness** — Say we know how to express the behavior, and we can name the test, but as for actually proving that it works? We're stuck.

- *Write the test backwards using the Arrange-Act-Assert test structure.*

## Arrange-Act-Assert

What is the *arrange-act-assert* test structure? It's a way to organize your tests into three parts based on the same Given-When-Then structure we explored in 22. Acceptance Tests.

```
// Arrange (Given) - preconditions
let palindromeChecker = new PalindromeChecker();

// Act (When) - act out the behavior
let result = palindromeChecker.isAPalindrome('mom');

// Assert (Then) - postconditions
expect(result).toBe(true);
```

In the Arrange phase, we set up all of the variables and class instances we're going to need to implement the test. In the Act phase, we plug everything together and act out the behavior. Finally, in the Assert phase, we verify that the behavior we wanted to happen has occurred. If it didn't, the test should fail.

## Benefits of organizing tests into act-arrange-assert

There are a few benefits to approaching our tests this way:

- We develop a clear separation between what we're testing and what our test setup is.
- We make test smells more apparent. For example, it is easier to tell if we're testing too many things at once — most stateless tests can be written in about three lines. Additionally, if we have assertions halfway through the test, this is easy to spot; they should be at the end instead.

## Do I have to do this?

Not necessarily, but it does help in the long run. In the previous chapter's examples, most tests could have been written to perform the arrange-act-assert as one-liners. This is common for tests against stateless classes and functions.

However, when we get into more complex tests — tests that involve statefulness, dependencies, and pushing our SUT (system under test) down more subtle success and failure paths, the arrange (given) part of the test gets a little more fleshed out. Sometimes, test setup can be so involved that it takes up way too much space in our test files. Such a problem could rely on the help of useful patterns like the *Builder pattern* to simplify test setup. In any case, keeping tests clean and having distinct phases to your tests is beneficial.

## Keep it clean — comments not necessary

If you really need to — while you're learning at least — go ahead and use comments to carve out the sections. I sometimes use arrange-act-assert comments when I'm gearing up to write a test that I'm really not sure how it's going to work and really have to think about it. However, make an effort to remove your comments afterwards to keep your tests compact, tidy, and easy to read (see my philosophy on 9. Comments for a larger discussion).

```
let palindromeChecker = new PalindromeChecker();
let result = palindromeChecker.isAPalindrome('mom');
expect(result).toBe(true);
```

## Writing your tests backwards

Often, the hardest part about writing a test is figuring out *how we're going to verify that it will have worked*. When we write tests backwards starting from the assert phase, we start by deciding how we'll be able to verify that the test passes. Surprisingly, I find that technique maps it easier to tap into your creativity — which is something I believe to be mostly unconscious. You'll find that when you write tests backwards, it informs your design in ways that you never would have otherwise considered.

With this approach, you don't waste time inventing abstractions that you don't need. You don't get overly ceremonious. You don't introduce unnecessary state. The best way I can describe what we're doing here is the technique sometimes known as “programming by wishful thinking”: expressing what we want to happen in a *declarative* style. It sort of looks like English, but it's code — code that we know can **eventually** resolve to executable code.

For example, if we were building a Tic Tac Toe game and the behavior was that it “should know that it’s O’s turn to go after X”, starting backwards, we could do something like this:

```
// Assert - this is easy, it's pretty much just
// turning the requirement into declarative code
expect(game.getCurrentTurn()).toBe('O');

// Act
game.chooseMark({ row: 0, column: 0 });

// Arrange - by default, it'll be X's turn
let game = new Game();
```

Where did I get the idea for detecting the currentTurn? I’m not sure. My unconscious mind. But that’s what I’m going with in the *Assert* phase.

Then moving to the *Act* phase, I’m thinking about the easiest public interface one could use to interact with the game object.

“Let’s use rows and columns”, I say internally.

And so it is. For this to work, the game would need to somehow keep track of whose turn it is. That’s not important to me right now. All I care about is that this test makes sense — and it does.

With the test sketched out, we flip it around. The final test looks like:

```
describe('tic tac toe', () => {
  test("should know that it's O's turn to go after X", () => {
    let game = new Game();
    game.chooseMark({ row: 0, column: 0 });
    expect(game.getCurrentTurn()).toBe('O');
  });
})
```

You have to trust yourself a little bit — take a *baby step* of faith — in these difficult scenarios. The good thing news? If you can’t figure out how to make the test pass, you can always revert back to the previous commit.

■ **Design rule:** When you’re stuck on how to verify correctness, try writing your tests backwards starting from the assert, to act, arrange, then finish by writing or cleaning up the name of the test.

## Exercises

Adding this new design rule to your toolbox, we’re going to continue along with some relatively straight-forward exercises, but if you get stuck — you know what to do.

### Stats calculator

■ **Description:** Without using system Math library functions, process a sequence of integers to determine the following statistics:

- minimum value
- maximum value
- number of

elements in the sequence • average value

For example: [2, 4, 21, -8, 53, 40] • minimum value = -8 • maximum value = 53 • number of elements in the sequence = 6 • average value = 18.666666666667

## Password validator

■ **Description:** Write a function (or a class) for validating passwords. Passwords must meet the following criteria:

- Between 5 and 15 characters long
- Contains at least one digit
- Contains at least one upper case letter

Return an object containing a boolean result and an errors key that — when provided with an invalid password — contains an error message for all errors in occurrence. There can be multiple errors at a single time.

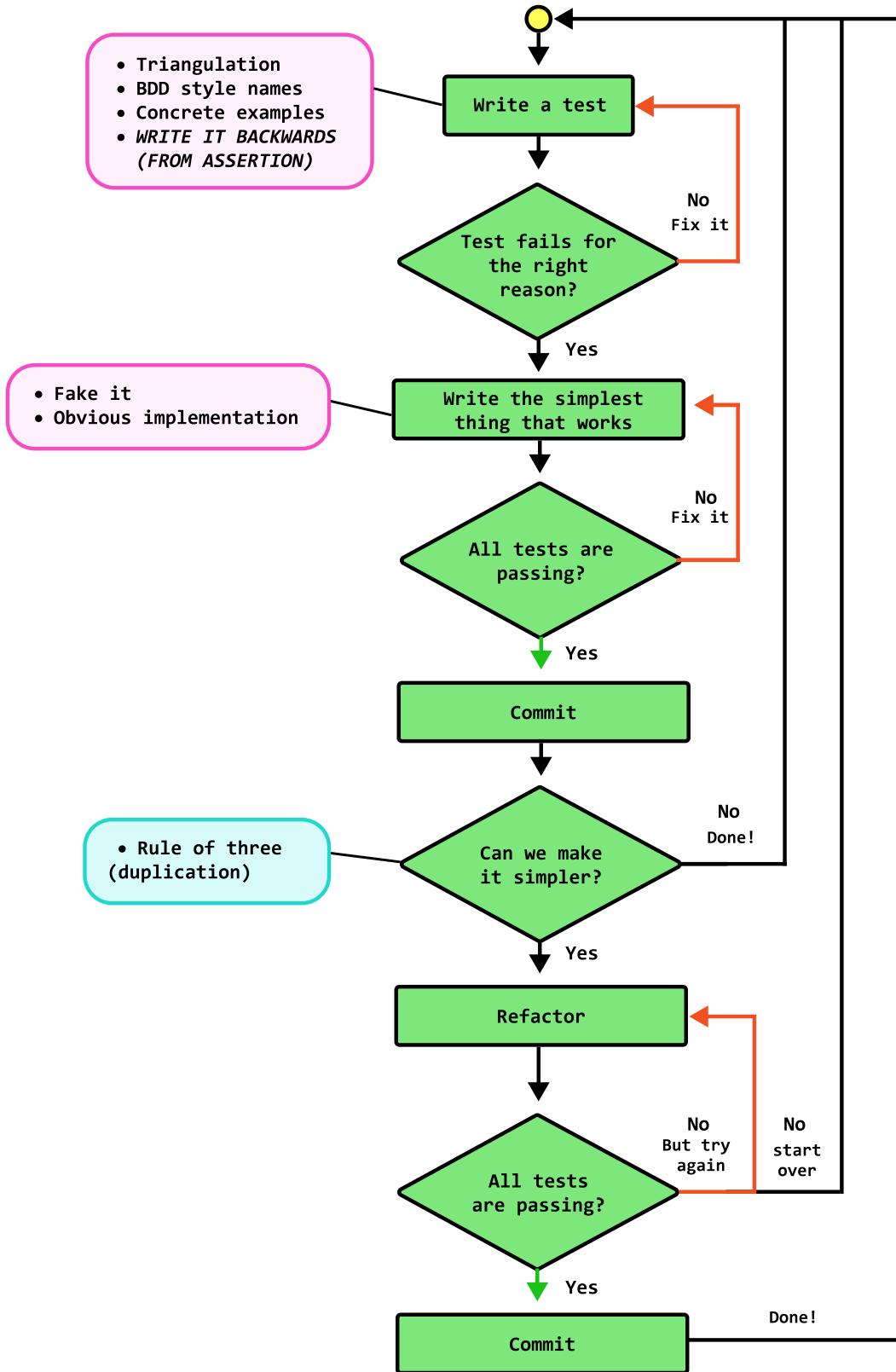
## Summary

### TDD considerations so far

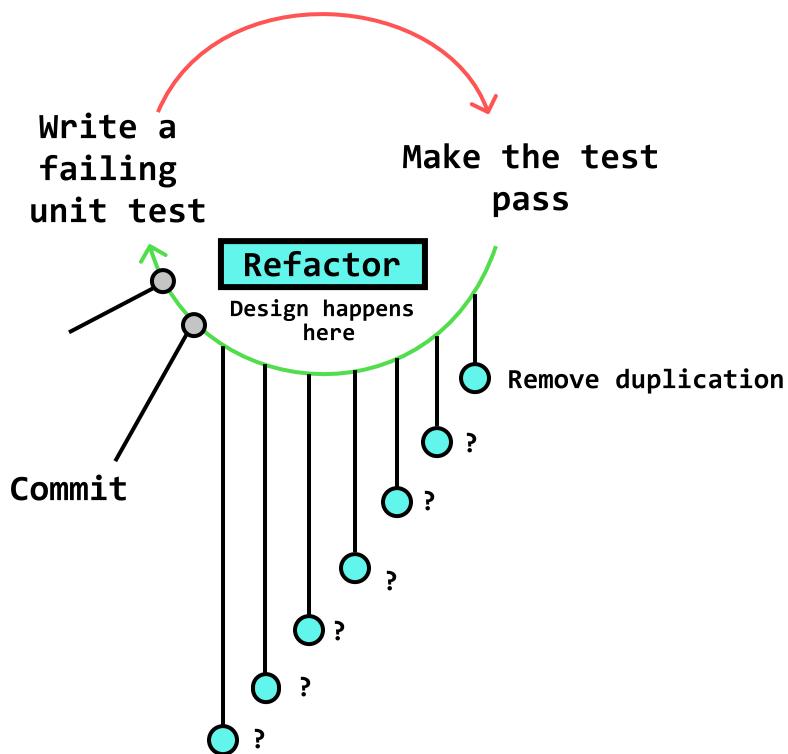
- **When writing a test**
  - Use behavior-focused tests that are meaningful to the domain
    - \* Avoid leaking implementation details
    - \* Avoid using technical language
  - Prefer concrete examples instead of abstract statements
  - Prefer one example per test to keep tests readable and failures traceable
  - Before continuing, ensure that the test is failing for the right reason — compilation errors or missing imports are not valid reasons for failure.
  - Structure your tests into arrange, act, and assert phases (**new!**)
    - \* Arrange (given) sets up the preconditions
    - \* Act (when) performs the action on the subject under test
    - \* Assert (then) deduces that the results occurred
  - Work backwards from the assertion (**new!**)
    - \* Start by writing the assertion; it's OK if you can't think of a good test name yet.
    - \* Write the act phase
    - \* Write the arrange phase, if necessary
    - \* Lastly, name or rename the test
- **To make a failing test pass**
  - Use the fake it or obvious implementation strategies
- **After the test passes**
  - Use the rule of three to refactor duplication
- **To write the next test**
  - Triangulate new behavior with a different example
  - Triangulate the same behavior with different examples

## Current test-driven workflow and design rules

Let's update our workflow with one more consideration when writing a test.



And as for our design skills, no change just yet!



## Resources

### Articles

- Arrange-Act-Assert: A Pattern for Writing Good Tests

### Books

- Agile Software Development Principles, Patterns, and Practices by Robert C. Martin
- Agile Technical Practices Distilled by Marco Consolaro and Pedro M. Santos
- Test-Driven Development by Example by Kent Beck
- Growing Object-Oriented Software Guided by Tests by Steve Freeman

## 31. Avoiding Impasses with the Transformation Priority Premise

To turn a failing test green, we transform our design with small changes. But some transformations are better than others. Some pave the way to successfully implementing all the desired behavior, while others create impasses — deadlocks — that prevent us from continuing. In this chapter, we learn about the Transformation Priority Premise (TPP): a way to optimize our designs and prevent getting stuck.

## Chapter goals

Here, we will:

- Learn about transformations in TDD
- Discuss problems with the obvious implementation strategy
- Learn how to use the Transformation Priority Premise to keep our solutions as simple as possible.

## Transformations

In TDD, there are two ways code gets changed: through *transformations* and *refactoring*.

**Transformations** are small code changes that modify how the code behaves. They also have the effect of directing the behavior of the code from being specific (especially when we use the *Fake It* strategy) to being more *generic*.

For example, if we only had one test for an `add(x, y)` function, then we could simply *Fake It* with a hard-coded value, like so:

```
describe('add', () => {
  it('should return 25 when adding 10 + 5', () => {
    expect(add(10, 5)).toEqual(25);
  })
});

// Implementation
function add (x, y) {
  return 25;
}
```

But upon adding a subsequent example, we are inclined to **transform** the solution towards a more generic solution.

```
describe('add', () => {
  it('should return 25 when adding 10 + 5', () => {
    expect(add(10, 5)).toEqual(25);
  })

  it('should return 30 when adding 10 + 20', () => {
    expect(add(10, 20)).toEqual(30);
  })
});

// Implementation
function add (x, y) {
  return x + y; // transformed
}
```

How are transformations different from refactoring? Both transformations and refactoring change the code, but transformations are about adding new behavior, while refactoring

is about keeping the behavior the same. Refactoring is merely about changing the structure.

## Revisiting how we go from red to green

Let's revisit how we go from red to green. If you'll recall, there are only two ways to do this:

1. Fake it — *hardcode a solution*
2. Obvious implementation — *implement whatever comes to mind*

We can use the *Fake It* solution for the first few tests but when duplication appears, we must naturally switch to the *Obvious Implementation*.

For such trivial problems like the `add(x, y)` one, we can skip straight to the *Obvious Implementation* since it can feel agonizing to perform *Fake It* when we know ahead of time what we're going to do: use the `+` operator.

But for less trivial problems — let's say we were building a checkers game and we were about to implement the logic which dictates that you *must jump* your opponent's piece when the opportunity presents itself — well, these transformations aren't <sup>\*\*</sup>exactly simple.

### Problems with obvious implementation

What's wrong with the obvious implementation transformation strategy? There are several drawbacks.

1. **Firstly, there is no universal *obvious implementation* for every test.** Two developers using TDD to solve the same problem may design their tests and the resulting implementation in completely different ways. What's obvious to one person may not be obvious to the next.
2. **Not all implementations are made equal.** Let me ask you a question. Which transformation is better? A transformation where you have to *completely rework 80% of the code already written against 15 existing tests* or a transformation where you have to change one line of code and add two more? The answer is obvious. It's the latter. Transformations that force us to rip apart much of what we've already written can be tricky, and often — they just don't work. If we get stuck, we have to revert back to the last commit and try again.

■ **Impasses/deadlocks:** Stuck? If you're not able to add the next test because of the way you've written code to pass previous tests, we call such a situation an *impasse* or a *deadlock* in TDD. You want to avoid this as much as possible.

1. **Complex transformations — especially early ones — increase the likelihood of an impasse.** If, in our second test, we jumped straight to using the *Object Pool* design pattern, well — there's a good chance that's not necessary. It's probably likely that we could have continued *Fake It* for a little while longer or use much simpler transformations.

■ **Design rule:** To prevent impasses, always prefer the simplest possible transformation.

That last design rule comes from a good place, but it's not very helpful without any concrete guidance. So then let's learn about a new concept which can guide our transformations.

## Transformation Priority Premise (TPP)

The Transformation Priority Premise (TPP) is a list of all possible transformations that can occur when performing TDD. Listed in order of preferred transformations, the transformations listed further towards the bottom of the list are more expensive and complex to implement. The bottom transformations are less preferred than the ones closer to the top of the list.

### TPP table

#	Transformation	Starting code	Ending code
1	<code>{}</code> -> nil	<code>{}</code>	<code>[return] nil</code>
2	Nil -> constant	<code>[return] nil</code>	<code>[return] "0"</code>
3	Constant -> constant+	<code>[return] "0"</code>	<code>[return] "0" + "1"</code>
4	Constant -> scalar	<code>[return] "0" + "1"</code>	<code>[return] argument</code>
5	Statement -> statements	<code>[return] argument</code>	<code>[return] argument.substring(2)</code>
6	Unconditional -> conditional	<code>[return] argument</code>	<code>if (condition) [return] "0" else [return] "1"</code>
7	Scalar -> array	<code>apple</code>	<code>[apple, banana]</code>
8	Array -> container	<code>[apple, banana]</code>	<code>~</code>
9	Statement -> tail recursion	<code>x + y</code>	<code>x + recursion</code>
10	If -> loop	<code>if (condition)</code>	<code>loop (condition)</code>
11	Statement -> recursion	<code>x + recursion</code>	<code>recursion</code>
12	Expression -> function	<code>(y2-y1)/(x2-x1)</code>	<code>calculateSlope(x1, x2, y1, y2)</code>
13	Variable -> mutation	<code>position</code>	<code>var position = [1,1]; position = [1,2];</code>

### How this works — constraint upon the obvious implementation

Remember the idea of **constraints** from Part II: Humans & Code? We previously learned that **constraints** are a good way to make it easier for humans to figure out the right sequence of steps to take. Well, here is again.

The TPP imposes constraints upon the *Obvious Implementation*. It explicitly defines what **preferred transformations** look like (and perhaps more importantly, what a bad transformations look like).

It's always going to be simpler to go from using constants to scalars rather than from constants to conditionals or to recursion.

Before TPP, we needed to completely rely on our unconscious intuition — our gut — to figure out how to get new behavior working. Now, we have a structured way to think about transformations.

e.g: “Can I do this with a scalar (argument)? No? Okay, what about if apply a transformation to conditionals instead? Will that work? Ah yes, let’s use that.”

■ **Tip:** You can also use TPP when refactoring to think about how to refactor your transformations back down to simpler transformations.

### TPP transformations

Let’s walk through each transformation.

**Transformation 1: `{}` → nil.** This is the most straightforward transformation. Assuming we haven’t implemented anything yet, the most straightforward thing we can do is to merely return null, an empty string, or nothing.

```
return;          // Nothing
return null;    // Null
return '';      // Empty string;
```

**Transformation 2: Nil → constant.** Next, we go from returning nothing to a constant.

```
return 1;        // Constant
```

**Transformation 3: Constant → constant+.** Then, we return several constants, a combination of constants, or merely a more complex constant.

```
// More complex constants
return 1 + 2;
return { value: 1 }
```

**Transformation 4: Constant → scalar.** Next up, we make use of function or method arguments.

```
function demo (arg) {
  return arg + 2;
}
```

**Transformation 5: Statement → statements.** Here's where it starts to get a little bit different. Up until now, we were only using `return`. There were no *statements*. No sequence. In this transformation, we get to use **unconditional statements**.

■ **Unconditional statements:** If you'll recall from 23. Programming Paradigms, high-level programming languages give us sequence, selection, iteration and indirection. When we're talking about *selection*, we're really talking about *conditional statements*. Where `if/else` and `switch` are examples of *conditional statements* that break program flow into separate paths (see 23. Programming Paradigms), unconditional statements are those that *do not*. At this point, it means that we may *perform assignment* and we may *make calls to other methods or functions*. These are sequential actions that flow through one code path. That's the idea: maintaining a single code path at this point.

Here's what that might look like:

```
function demo (arg) {
  return min(max(0, arg), 10);    // subroutine
}

function demo (arg) {
  var temp = arg + 2;            // assignment
  return someFunction(temp, arg); // subroutine
}
```

**Transformation 6: Unconditional → conditional.** As we discussed in 23. Programming Paradigms, splitting code paths increases the cyclomatic complexity of a program, so early on, we'd like to prevent that as much as possible. However, it may eventually become unavoidable. In this transformation, we finally introduce conditional statements (like `if/else` and `switch`). You can also use the ternary operator if you like.

```

function demo () {
    if (condition) {
        return 0;
    } else {
        return 1;
    }
}

function demo () {
    return condition ? 0 : 1;
}

```

**Transformation 7: Scalar → array.** In this next transformation, we move from conditionals to performing conditional logic using an array.

```

function demo (index) {
    var arr = [0, 1, 2, 3, 4, 5, 6];
    return arr[index];
}

```

**Transformation 8: Array → container.** Often, an array isn't the best abstraction to use for conditional logic. In this transformation, we use a slightly more complex data structure to do the selection work — be it a class or an object from a function.

```

function demo (index, elIndex) {
    var page = new Page(); // a more complex container than an array
    return page.getElement(index, elIndex);
}

```

**Transformation 9: Statement → tail recursion.** A big jump happens here in this transformation. Here, we introduce a new concept: tail recursion. Instead of introducing iteration with `for` or `while` loops right away, we introduce it with tail recursion.

```

function tailrecsum(x, running_total = 0) {
    if (x === 0) {
        return running_total;
    } else {
        return tailrecsum(x - 1, running_total + x);
    }
}

```

■ **Tail recursion:** In general, tail recursion is different from regular recursion in the sense that with each evaluation of a recursive call, we're pushing a value along into further recursive iterations. A simplified way of understanding it is that the original recursive call is the last expression evaluated. Read this for a more detailed explanation.

**Transformation 10: If → loop.** After tail recursion, the next complexity is to actually use loops.

```

function demo (arg) {
    for (let i = 0; i < arg; i++) {
        //
    }
}

```

**Transformation 11: Statement → recursion.** After tail recursion and loops comes actual recursion. In *original* recursion, the compiler needs to get to the bottom of the recursive chain *before* it can start evaluating the values.

```

function recsumrecursive(x) {
    if (x === 0) {
        return 0;
    } else {
        return x + recsum(x - 1); // original recursion
    }
}

```

**Transformation 12: Expression → function.** In this transformation, we're now allowed to take some of the more complex logic and put it into subroutines (functions or methods). Decomposition.

```

function demo (name) {
    return Greeting.getGreeting(name);
}

```

**Transformation 13: Variable → mutation.** Lastly, the concept of mutation — changing state. This should be the absolute last type of transformation that we reach to reach for. Why? Because we know the headaches we sometimes deal with when state is involved.

```

function demo() {
    let name = 'khalil';
    name = 'Bill';
}

```

### Applying TPP to the Fibonacci exercise

Here's a table depicting the transformations when applying TPP to the *Nth Fibonacci* exercise from 29. Getting Started with Classic Test-Driven Development.

Input	Expected output	Transformation type	Implementation
0	0	{ } -> nil { }	<i>Doesn't work</i>
0	0	Nil -> constant	return 0
1	1	Constant -> scalar	return index
2	1	Unconditional -> conditional	if number < 2 { return index } else { return index - 1 }
3	2	Unconditional -> conditional	<i>No change</i>
4	3	Unconditional -> conditional	<i>No change</i>
5	5	Scalar -> array	let f = [0, 1, 1, 2, 3, 5]; return f[index];
6	8	Scalar -> array	let f = [0, 1, 1, 2, 3, 5, 8]; return f[index];
7	13	Scalar -> array	let f = [0, 1, 1, 2, 3, 5, 8, 13]; return f[index];
8	21	Statement -> tail recursion	if index < 2 { return index } else { return fibonacci(index - 1) + fibonacci(index - 2) }

■ **Design rule:** When you find yourself stuck, try applying the next transformation in the TPP table. You don't have to follow this to a tee, but keep it in the back of your mind as you go about implementing production code to pass a test.

## Exercises

Let's take what we know about TPP and try to apply it to some slightly more complex exercises.

### Fizz Buzz

To warm up, perform the Fizz Buzz kata again, but with what we learned about TPP.

■ **Description:** Create a function (or a class) that accepts only the numbers from 1 to 100 and throws an error otherwise. For multiples of three, it should return the word "Fizz". For multiples of five, it should return the word "Buzz", and for multiples of both three and five, it should return "FizzBuzz". For anything not a multiple of three or five, it should return the inputted number, but as a string.

### Recently Used List

Note that this is the first exercise we've done so far which specifically requires the use of state. Don't worry if your object-oriented designs aren't great. We'll work on it in the next section.

■ **Description:** Develop a recently-used-list class to hold strings uniquely in Last-In-First-Out order. The most recently added item is first, the least recently added item is last. Items can be looked up by index, which counts from zero. Items in the list are unique, so duplicate insertions are moved rather than added. A recently-used-list is also initially empty.

■ **Optional extras:** Here are a few more pieces of specification to make it more interesting.

- Null insertions (empty strings) are not allowed.
- A bounded capacity can be specified, so there is an upper limit to the number of items contained, with the least recently added items dropped on overflow.

- While getting items by index, supplied index-value should be within the bounds of List [eg. if maximum item counts of list is 5 then supplied index is less than 4 as index starts from 0 (zero)]
- Negative index value not allowed [ $>0$ ]
- Size limit is must if not supplied make 5 as default [0-4]

## Tennis

■ **Description:** We're going to build a tennis game/simulation for the console.

*Serving & receiving:* At the start of the game, the server and the receiver are decided. In real life, we do a coin toss, but for our game, you could merely refer to the players as *server* and *receiver*.

*Scoring:* In Tennis, there are 4 points. The first point — which represents zero — is called “love”. It goes “love”, “15”, “30”, and then “40”. Whenever the ball isn’t served back, whoever’s side it was last on, the other player gets a point.

*Winning:* In order to win a game, you have to win by two. That means that if the score is tied 40:40 in a “deuce” (four points each player), then the next person to score wins an “advantage”. If the player with the advantage scores again, then the game is over and they win. However, if the player with the “advantage” loses the next point, it goes back to “Deuce”.

I know what you’re thinking, that could go on forever! And you’re right. In fact, the longest recorded tennis match lasted just over 11 hours!

You’ll want to test all of these scenarios. Remember to focus on the behavior.

## Summary

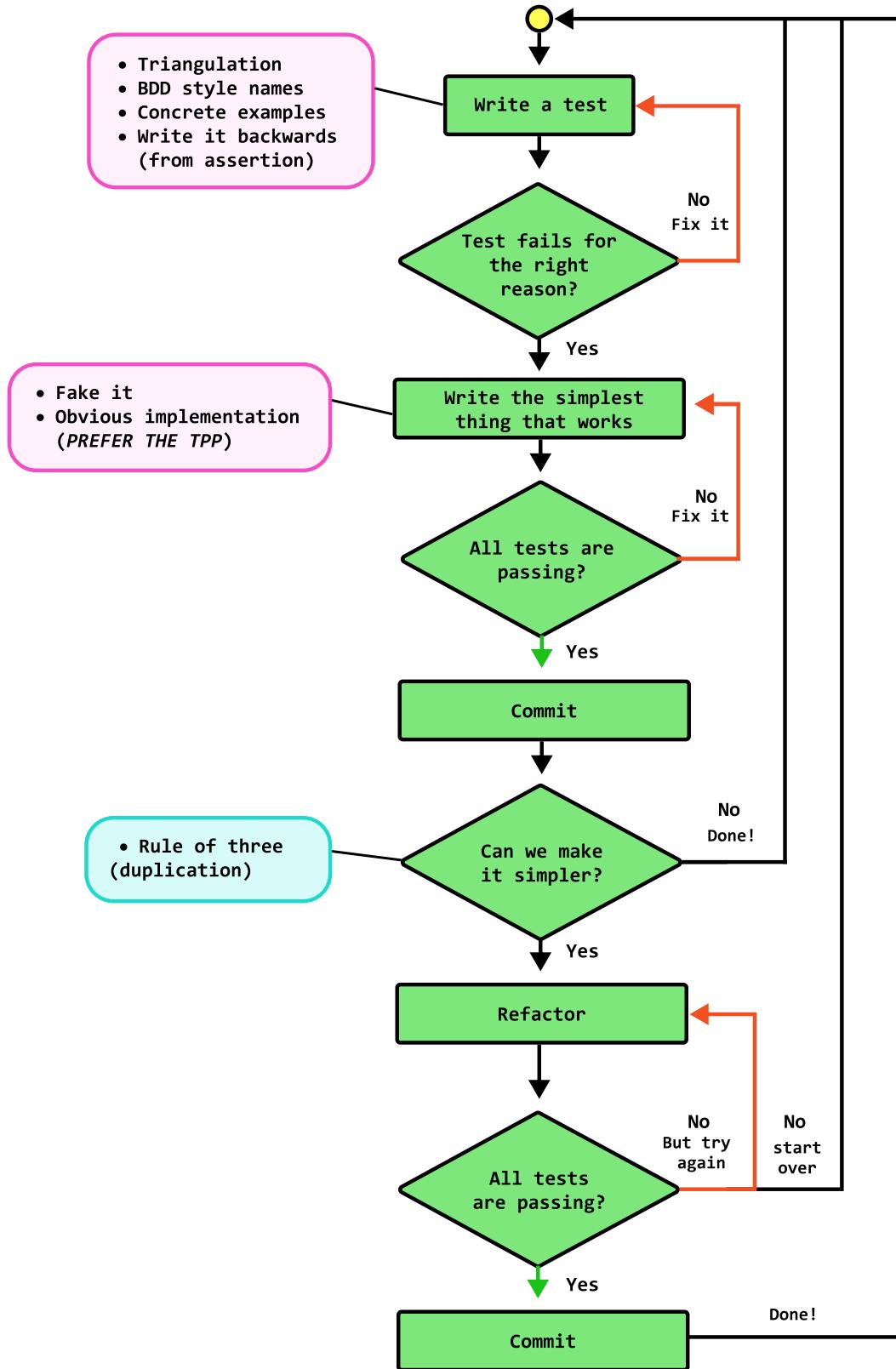
### TDD considerations so far

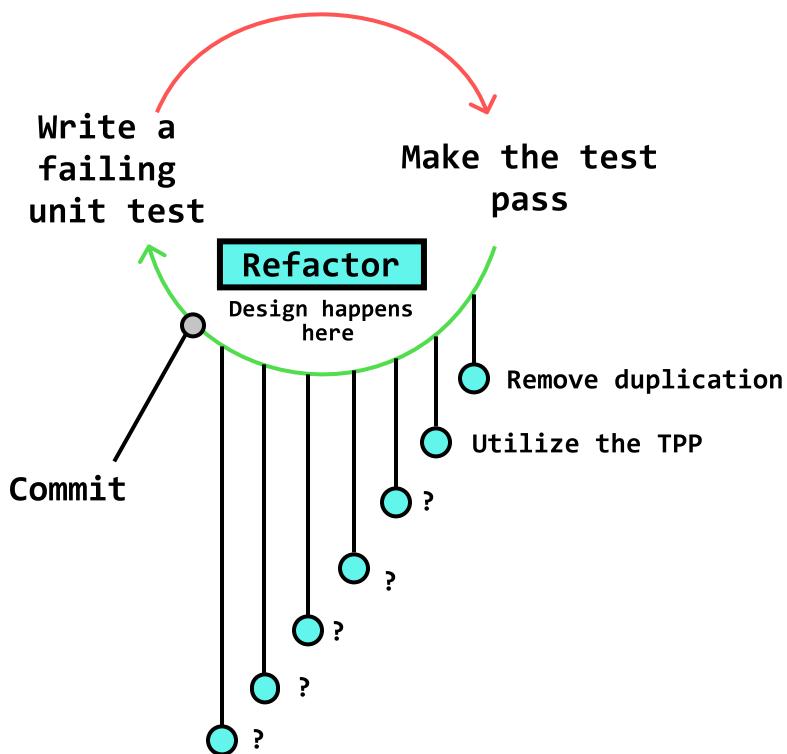
- **When writing a test**
  - Use behavior-focused tests that are meaningful to the domain
    - \* Avoid leaking implementation details
    - \* Avoid using technical language
  - Prefer concrete examples instead of abstract statements
  - Prefer one example per test to keep tests readable and failures traceable
  - Before continuing, ensure that the test is failing for the right reason — compilation errors or missing imports are not valid reasons for failure.
  - Structure your tests into arrange, act, and assert phases
  - Work backwards from the assertion
- **To make a failing test pass**
  - Use the fake it or obvious implementation strategies
  - Use the transformation priority premise to inspire better transformations (**new!**)
- **After the test passes**
  - Use the rule of three to refactor duplication
  - Prefer refactoring to transformations lower on the TPP chart (**new!**)
- **To write the next test**
  - Triangulate new behavior with a different example

- Triangulate the same behavior with different examples

### **Current test-driven workflow and design rules**

Our test-driven workflow hasn't changed too much. Instead of doing unconsciously obvious implementations, we can lightly rely on the TPP to prefer better transformations.





## What's next?

At this point, you should feel comfortable with using TDD to solve some exercises. The goal of this section was to get you up and running with the foundational TDD mechanics so that we can effectively learn how to guide object-oriented designs with tests. Tests are like guide-rails for OO design.

Before continuing, ensure you have a good feel for this and can walk through the TDD workflow using all of the techniques and considerations we learned about in this section.

Next, we're going to learn about *design*. Object-oriented design.

## References

### Articles

- [https://en.wikipedia.org/wiki/Transformation\\_Priority\\_Premise](https://en.wikipedia.org/wiki/Transformation_Priority_Premise)
- <https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>
- [https://science.jrank.org/programming/Transformation\\_Priority\\_Premis.html](https://science.jrank.org/programming/Transformation_Priority_Premis.html)
- <https://stackoverflow.com/questions/33923/what-is-tail-recursion>
- <https://www.javacodegeeks.com/2013/01/tdd-and-the-transformation-priority-premise.html>
- <https://www.codurance.com/publications/2015/05/18/applying-transformation-priority-premise-to-roman-numerals-kata>

## Books

- Agile Software Development Principles, Patterns, and Practices by Robert C. Martin
- Agile Technical Practices Distilled by Marco Consolaro and Pedro M. Santos
- Test-Driven Development by Example by Kent Beck
- Growing Object-Oriented Software Guided by Tests by Steve Freeman

## Part V: Object-Oriented Design With Tests

■ **Important notice on this section (deprecation):** This section is here *as is* for now. While I'm refactoring the book towards the *new version*, I've released this section because some of it may prove to be valuable to readers. Download a copy of the book to save it as it. It will change drastically over the next few months. This will likely not be here. ■

■ Most developers struggle with object-oriented programming. Building on top of our basic understanding of object-oriented architectures and how to use TDD to safely guide our solutions, we shift our awareness to the foundational topic called Responsibility-driven design. With a focus on roles, responsibilities, and collaborations, we learn the world's most effective way to develop object-oriented software.

## 32. Why & How to Learn Object-Oriented Software Design

■ Much of design is about *structure* and *relationships* — the essence of the object-oriented paradigm. By learning OO properly, we remedy many frustrating design challenges faced by both front-end and back-end developers. OO design skills significantly increase our confidence to take on more ambitious projects.

In the late 2000s, at the time, Kobe Bryant was undeniably known as the best basketball player in the world.

Among his incredible list of achievements included:

- Taking the Lakers to win three consecutive NBA championships.
- Leading the NBA in scoring from 2005-06.
- Knocking out a whopping 81 points against my home team, the Toronto Raptors (ouch) — the second-highest number of points ever scored in a game in the history of the NBA.

What was the key to Bryant's greatness? Was it that he studied the greats? Was it that he knew the flashiest moves? No. It was where he put his awareness — his energy, his focus.

Sports trainer Alan Stein, Jr., once shared that Bryant would rigorously practice “the most basic footwork in offensive moves” — the type of stuff Alan “routinely taught to middle school-age players.”

When asked why he was practicing the basics over and over, Bryant smiled and replied with all seriousness,

“*Why do you think I’m the best in the world? Because I never get bored with the basics.*”

Getting lost in the sea of new tools, technologies, languages, frameworks, and libraries is easy. While it's ideal to stay up to date, if we'd like to be good at what we do, we shouldn't consider ourselves exempt from the basics. We must put in reps.

*Alright, then. What are these basics of which you speak, Khalil?* To spin up a REST API? Implement the quicksort algorithm? Some recursive magic?

There are many different ways to answer this. But if there's one thing I know for sure, **OO — the most widespread programming paradigm — is a significant part of the basics, and we shouldn't neglect it.**

OO teaches us much about design, whether you continue using it or not. We can't deny this. In today's programming climate, you must know how it works and how to use it effectively; only then can you understand what you're leaving behind when you pursue the purely functional approach.

Look. I know you've probably studied object-oriented programming in school. You're probably at the very least aware of concepts like polymorphism, encapsulation, and inheritance (curse those terrible *animal-dog* examples). Now, if you're similar to me, whatever you've learned hasn't been tremendously helpful in practice. Perhaps, you're still nervous about when to use that `extends` keyword. Or maybe you're one of the thousands of developers who feel that classes tend to get confusing and out of hand too quickly.

It's also likely you're unsure about things like:

- When to use an interface or abstract class
- When to use an abstract method
- How to use mocks and stubs
- When *not* to use mocks and stubs
- And when to *actually* use inheritance and polymorphism

For those that lack the OO basics, it's not uncommon to feel like you:

- Don't know how to structure things
- Don't know when to break files up
- Don't know where to put decision-making logic
- Have to constantly re-learn new ways to build when your favorite libraries or frameworks change their best practices
- Experience mass amounts of confusion and imposter syndrome making things that aren't web apps (games, robots, desktop applications, etc.)
- Struggle to handle cross-cutting concerns like routing and auth in front-end architectures
- Rely a lot on the frameworks, and when you get stuck, you're *really* stuck

I'm sure this list barely scrapes the surface of problems you've likely faced or are currently facing.

Well, forget the pain and suffering you've endured with object-oriented programming. Erase it from your mind. We're going to re-learn. Properly.

The OO paradigm can teach us a lot about design. So let's dive deep into the skill that is *Object Design*.

## Chapter goals

In this introductory chapter, we:

- Learn why tests aren't enough.
- Learn about *Object Design* and its parts: *analysis*, *design*, and *programming*.
- Discuss how learning object design affects your front-end or back-end development skills.
- Discuss a framework or learning path to object-oriented software design mastery.

## Current challenge: design

In Part III: Phronesis, we saw how the object-oriented paradigm compared to the others in 23. Programming Paradigms. We even learned how 24. An Object-Oriented Architecture could work in practice. Then in Part IV: Test-Driven Development Basics, we learned how to use the classic TDD approach to watch our designs *emerge* — using \*\*both “programming by wishful thinking” and the more linear Transformation Priority Premise techniques.

This is a great baseline.

At this point, you should feel confident about using tests to make functions and drive the design of classes that contain core code.

There is a problem, though. While tests can guide our designs, they aren't a substitute for design.

### Tests aren't design

Let's think back to the goal of software design (“to build products that **serve the needs of the customer** and can be **cost-effectively changed by developers**”). Breaking it down, we can see how far we've come. We'll notice that TDD gets us further, but it's not enough.

“The needs of the customer” refers to the notions of:

- ■ Functional requirements (i.e., do the features such as “makeOffer” and “createUser” work?)
- ■ Non-functional requirements

Whereas “can be cost-effectively changed by developers” means:

- ■ a very specific set of non-functional requirements (ie: testability, reusability, maintainability, and flexibility)

A **test-driven approach to software development can help us more reliably implement functional requirements**; this is great because we have a safety net to try out designs until they work.

Here's the rub. Without an understanding of design, it's (1) **likely that the code we write to implement the functional requirements will be messy**, and (2) **it's unlikely the test-driven approach will help us realize non-functional requirements**. Why's that?

Functional requirements-wise, TPP helps us solve problems with as little programmatic complexity as possible, but that doesn't help us design understandable abstractions.

Things like scalability and portability don't *just happen* while you're coding. These concerns must be thought about and designed for as a part of the architecture early on (see 26. The Walking Skeleton ).

Not only that, but if we shift to the Simple Design point of view, TDD only helps us with the first two points:

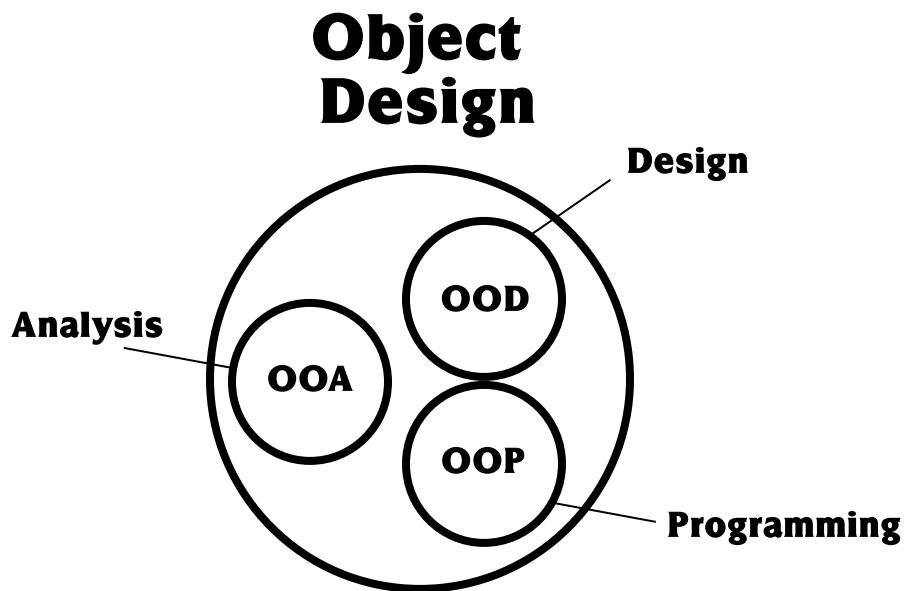
1. ■ We have tests that confirm the code works (functional behavior)
2. ■ We have no duplication (can create abstractions on the fly)
3. The code is as expressive as possible (abstractions have high cohesion)
4. Fewer elements exist, and they communicate in coherent, reasonable ways (abstractions have low coupling)

Folks, we are missing *design*. It's not enough for us to merely create abstractions that work - we need clear, expressive, minimal ones. And they need to address the non-functionals somehow as well!

## Introducing object design

The term "*Object Design*" or *Object-oriented Design* encapsulates all the techniques, tools, frameworks, and analysis activities used to develop object-oriented software.

Most developers think that OO is just about *programming*, and that's a big mistake. Object design has three distinct and equally important parts: *analysis*, *design*, and *programming*, and each is important to study, practice, and master.



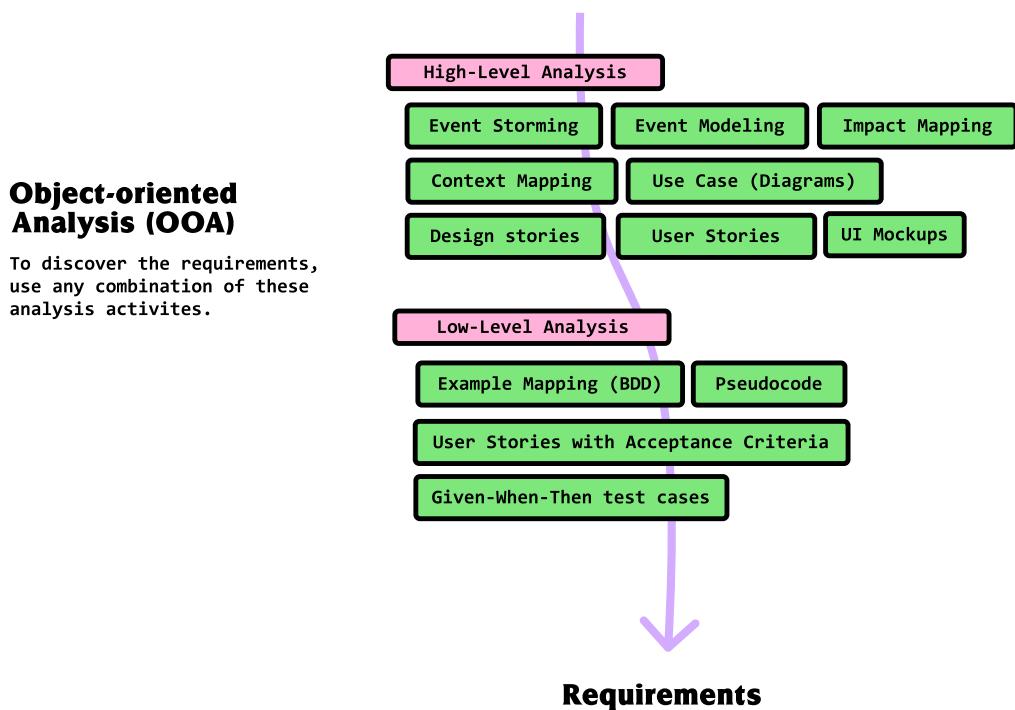
It's no wonder many of us struggle with OOP — we're unaware that two parts come before it! Perhaps the quickest fix I have for developers frustrated with their object-oriented code is to say, “stop writing object-oriented code and learn about *object-oriented analysis* (*OOA*) and *object-oriented design* (*OOD*) first.”

Doing OOP without understanding OOAD (object-oriented analysis & design) is like a kid trying to build a treehouse with a bucket of nails, a hammer, and some wood. Yes, they may know how to hammer timber and nails together to make *something*, but let's be real — you're not going to set foot in that treehouse, sitting thirty feet in the air.

## Analysis

**Goal: Identify the requirements for which we will craft a design.**

To start, we're in the same place we always are. Complexity. Entropy. Chaos. A place of potential. Our first weapon in the defence against complexity is analysis.



Before the Agile revolution, object-oriented analysis (OOA) was a phase at the beginning of the Waterfall method. As you may know, Waterfall had problems. In the real world, requirements change constantly. To keep up with a reality of constant change, most of the industry prefers an iterative Agile approach.

What does this mean for OOA? Same things we discussed in Part III: Phronesis. **If, at any point, the requirements were to change, we have to back up and do some analysis.**

Changing requirements means re-organizing priorities, new objects, refactoring the design of existing objects, and changing how objects collaborate. Extreme circumstances can mean architectural changes if new non-functional requirements (NFRs) emerge (and that's why you want to find NFRs early!)

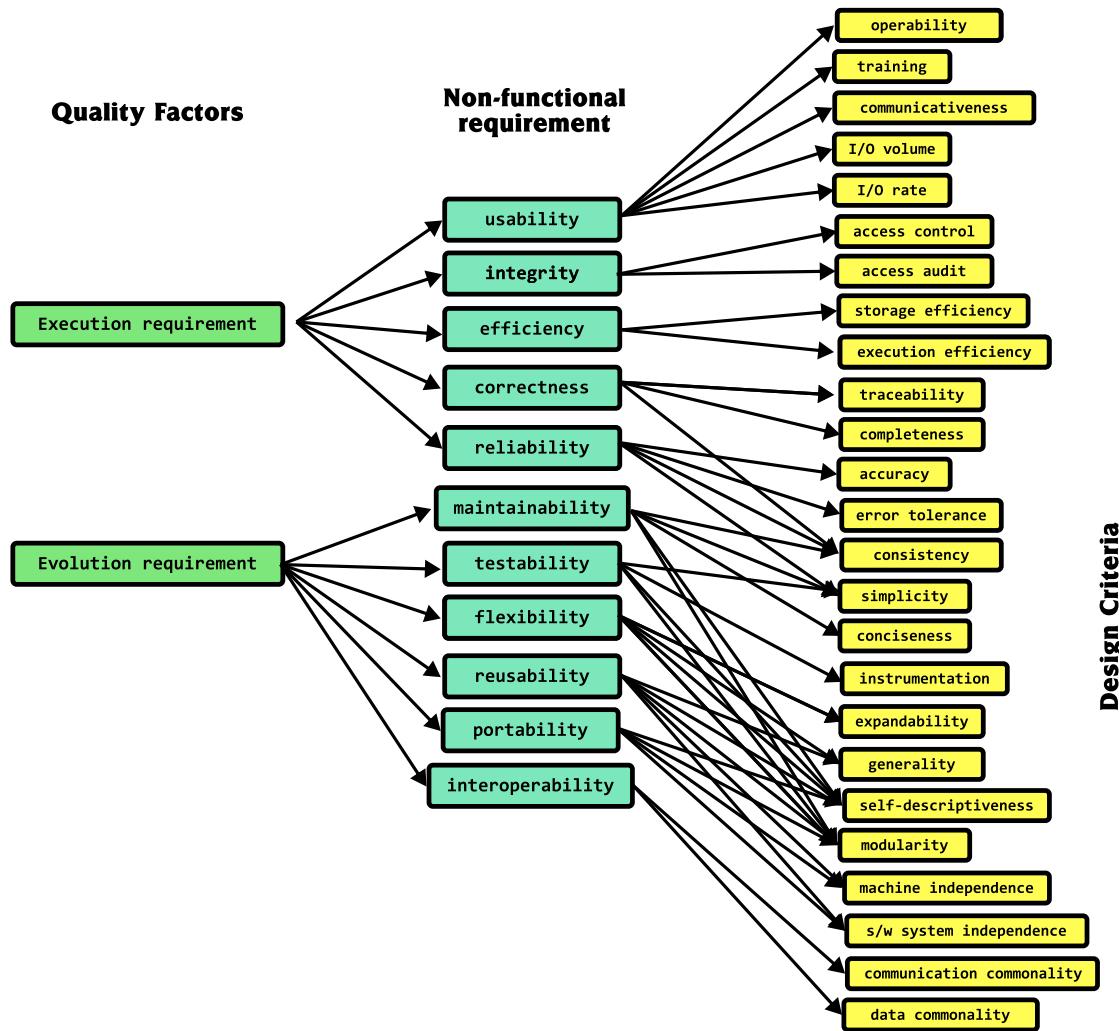
It's important to take a quick detour to understand non-functional requirements in more detail.

## Non-functional requirements: evaluation vs. execution qualities

When we speak of non-functional requirements, there are two broad-stroke categories that we can divide them into: execution qualities and evaluation qualities.

- **Execution:** Meets specific run-time behaviours and user-facing needs — ie: usability, efficiency, correctness, reliability
- **Evaluation:** Makes code easier for us to work with — ie: maintainability, testability, flexibility

See below for an incomplete list of non-functional requirements divided into one of these two categories.



Reworked from Steve Easterbrook's Lecture on Non-Functional Requirements (via University of Toronto). Here, "Product operation" = execution qualities and "Product revision" and "Product transition" = evolution qualities. Originally from McCall's NFR list via Vliet 2000, pp111-3.

As you can see, there are a lot of NFRs. Not all of them are necessary for each project, but some of them are critical. Make or break. This is a huge reason why we need the analysis step.

## Techniques for analysis

Over the years, many techniques have emerged. In this book, we've seen a few of these so far.

**For the big picture** — understanding the problem and creating a high-level solution, I suggest you rely on the use of:

- Event Storming
- Event Modelling
- Impact Mapping
- Context Mapping
- Design Stories (see 33. Responsibility Driven Design 101 - Rough Notes for examples)
- Use Case (Diagrams)
- and UI Mockups

As discussed in 16. Learning the Domain, these techniques help ensure we're on track to build something useful and hone in on the work we need to do.

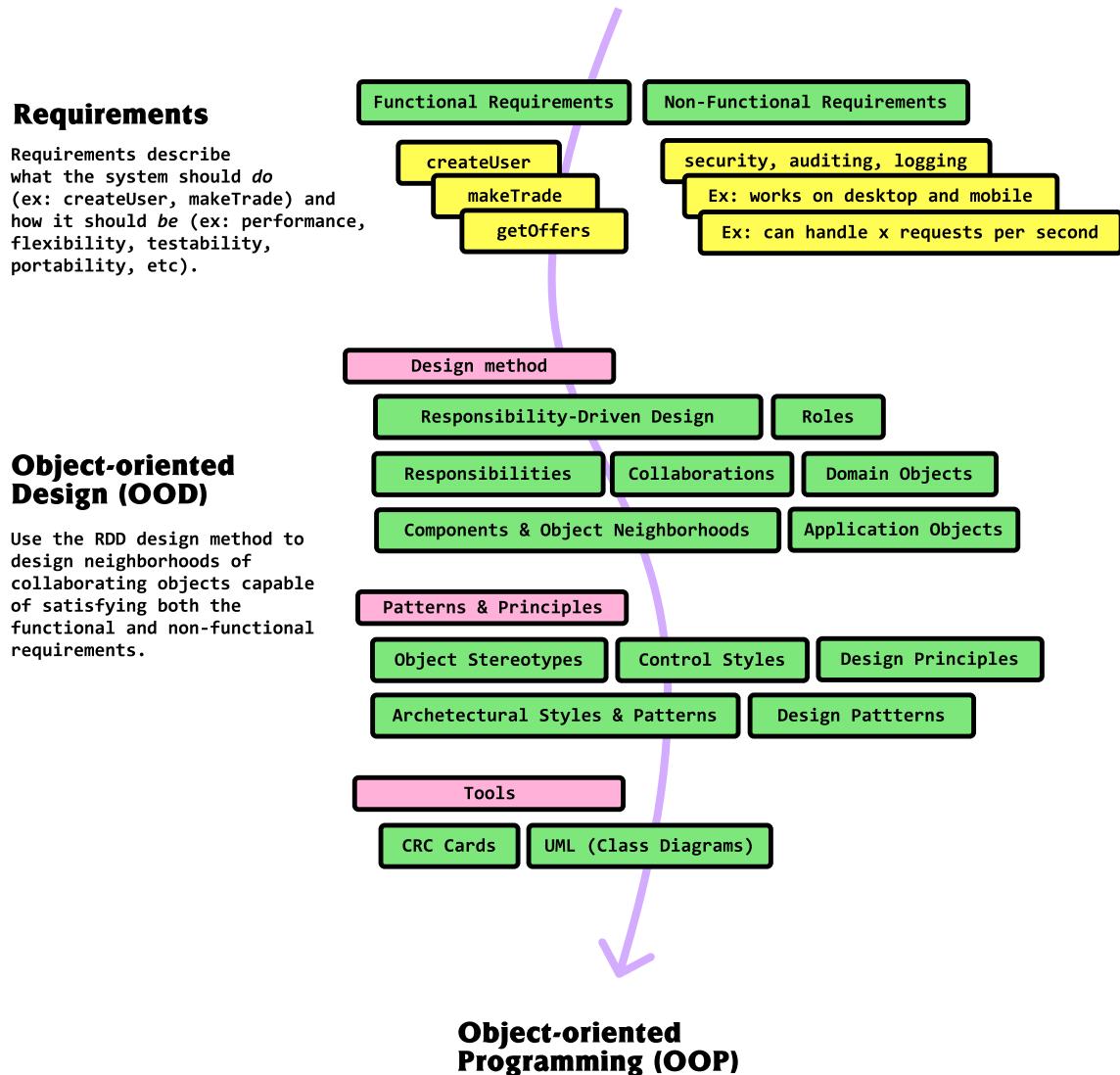
Once you have a requirement, **to get to the nitty-gritty details of what should happen**, I tend to use the following techniques:

- Use case pseudocode as demonstrated in 21. Understanding a Story
- BDD-style Given-When-Then tests (see 22. Acceptance Tests)
- Example Mapping
- UML sequence diagrams
- Rough sketches

These techniques give us the appropriate level of detail to move over to the next phase. With the requirements laid out, how do we turn that into code? The intermediate step: design.

## Design

**Goal: Divide requirements into neighborhoods of collaborating objects — each with well-defined responsibilities.**



From here, we take the requirements and create a system from them. The end result here is a bunch of *object candidates* (rough ideas of objects, interfaces, components and how they could interact). These help us when we sit down to write our first test.

This means we don't actually *have to* write any code during this time (although we sometimes write tests and exploratory code to try out potential designs and get new ideas for candidates).

Candidates are either invented or integrated from libraries and frameworks and designated with the handling of unique aspects of the requirements.

Our collection of *object candidates* involve objects we both invent and integrate from libraries and frameworks to handle the functional requirements (features) and non-functional requirements (the observable system as a whole).

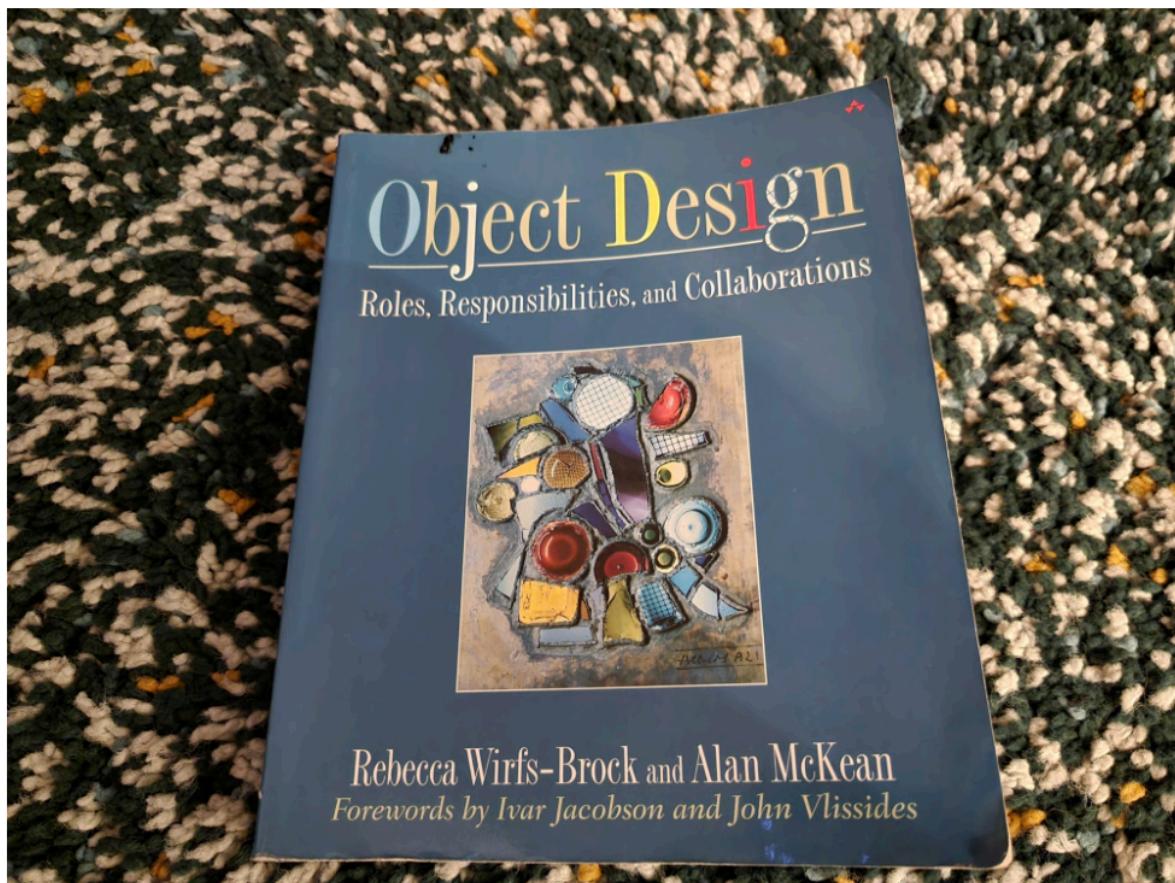
Sounds great right? Imagine that. The ability to sketch out what we need before we code it. How do we do this? Through the use of a design method called **Responsibility-Driven**

## Design.

### Responsibility Driven Design

Design is more of an art than a science. That can make it hard to come up with a repeatable approach to building the right thing. Design *methods* however, are learnable philosophies containing procedures and techniques that can lend massive amounts of clarity to and facilitate the way we work.

That's what **Responsibility-Driven Design** is — the programming world's most influential, effective, and intentional approach to designing object-oriented software. I know, big claim. But it's true.



My copy of the legendary Object Design book. My second favourite blue programming book. This book is so rare that it's nearly \$200 USD nowadays!

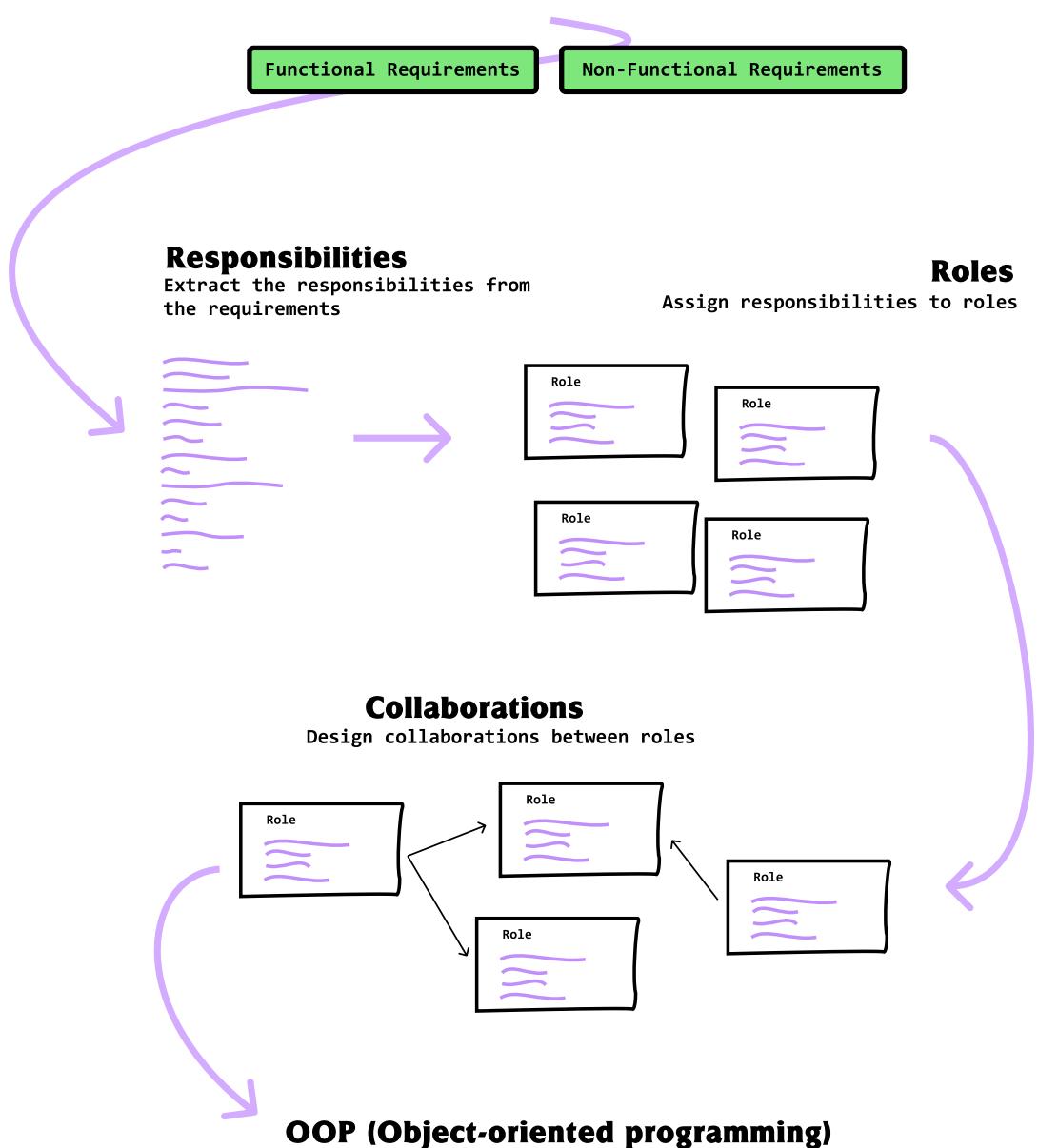
RDD, originally invented by Rebecca Wirfs-Brock in the late 80s and modernized in her 2002 book titled *Object Design*, is a learnable and repeatable design method for building object-oriented applications.

We'll talk more about RDD, but in essence, the idea is that we convert requirements into *responsibilities*, assign them *roles* (to be filled by objects), and then design how those roles *collaborate*.

It's a shift of perspective, but thinking about program structure and communication this way has huge implications for virtually every other topic we'll explore in this book. As you'll go on to see, the ideas of RDD underpin the tactical patterns in DDD, the clean/hexagonal architecture, and many other architectural patterns. In this light, you'll also find that design principles act as a shortcut to the philosophy of RDD.

# The Responsibility-Driven Design Process

## Start



The RDD process involves converting requirements into responsibilities, assigning them to roles, then deciding how they'll collaborate.

### Object design is about closing the gap

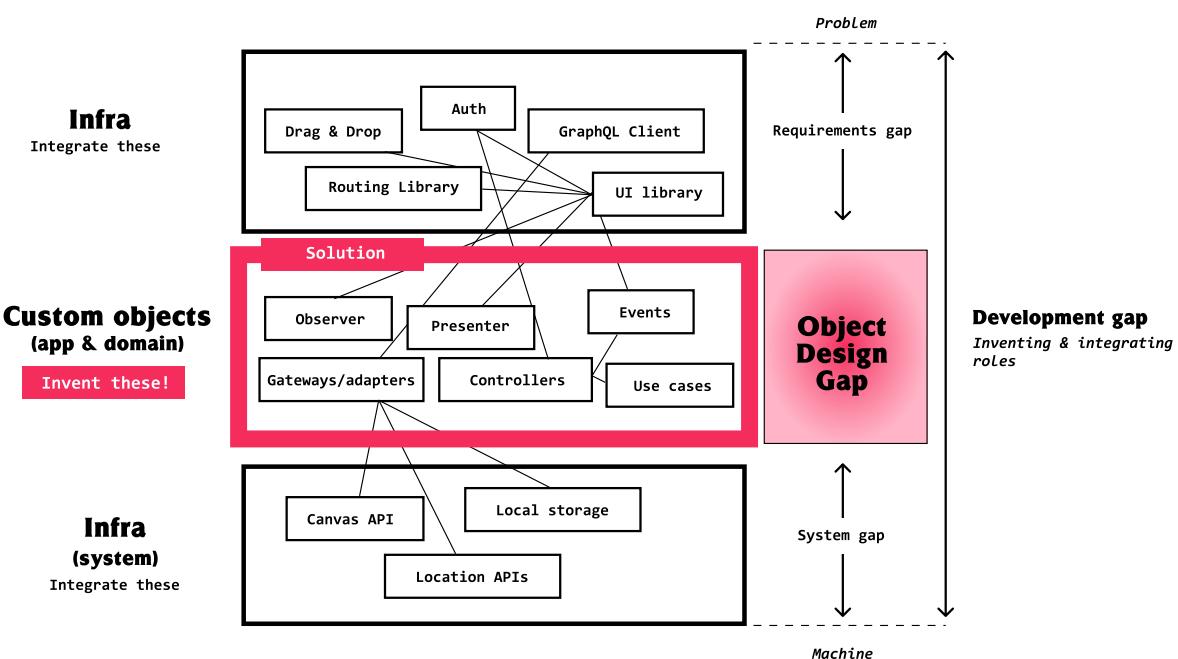
As abstraction designers, we are either inventing objects or integrating existing ones.

What do I mean by integrating existing objects? Well, you don't want to build things from

scratch that you can merely import or download from the internet. That's right. I'm talking about frameworks, libraries, components, and platforms. They're amazing, but we want to become conscious of their use and the implications.

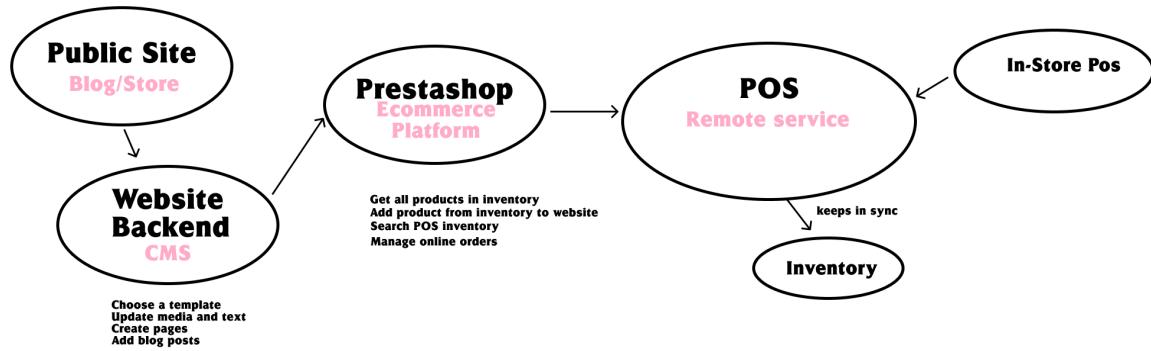
Say we need drag-and-drop capabilities. Now also say that we're aware of a drag-and-drop library. From an RDD point of view, relying on a library is to integrate *role(s)* that carry out the *responsibilities* we need, into our application. Our effort goes towards the conscious design of how roles *collaborate* with our other *roles* (eventually filled by objects). Same goes with working auth, state management, notifications, or routing into our app.

In this light, object design \*\*is purely about closing the gap.



Or let's say we're building an open-world video game. What are some of the things we have to handle? Handing user input, showing our character jump when we press the jump button, showing screens, rendering buildings, characters, among a few. Among these responsibilities are themes that would suck to have to build ourselves: rendering and physics. Instead, we can find a popular video game engine and physics library to carry out the responsibilities of *collision detection*, *applying gravity to objects*, and so on.

Let's go a level higher now. Consider building an eCommerce website for a brick-and-mortar store with a point-of-sale system set up. Here, our task is to see what components are available, see if we can integrate them, or, if not — invent them. Using the RDD process, we'd realize we could pull a CMS off the shelf and allow that to work as the back-end for a static site instead of building one ourselves.



Are you starting to see how useful this is?

Whether we're building front-end applications, trying to tie things together and design how **control** works, or we're designing distributed systems, object design is the prime skill. Why? Again, because **design is about structure and relationships**.

RDD is a game-changer. It brings clarity to the invention, outsourcing, and integration of objects and components.

- We learn about *Control Styles* in 33. Responsibility Driven Design 101 - Rough Notes
- We walk through an example of using RDD to design a checkers game in 34. The RDD Process By Example - Rough Notes.

### You'll do a lot of design outside of the editor (CRC cards, whiteboards, etc)

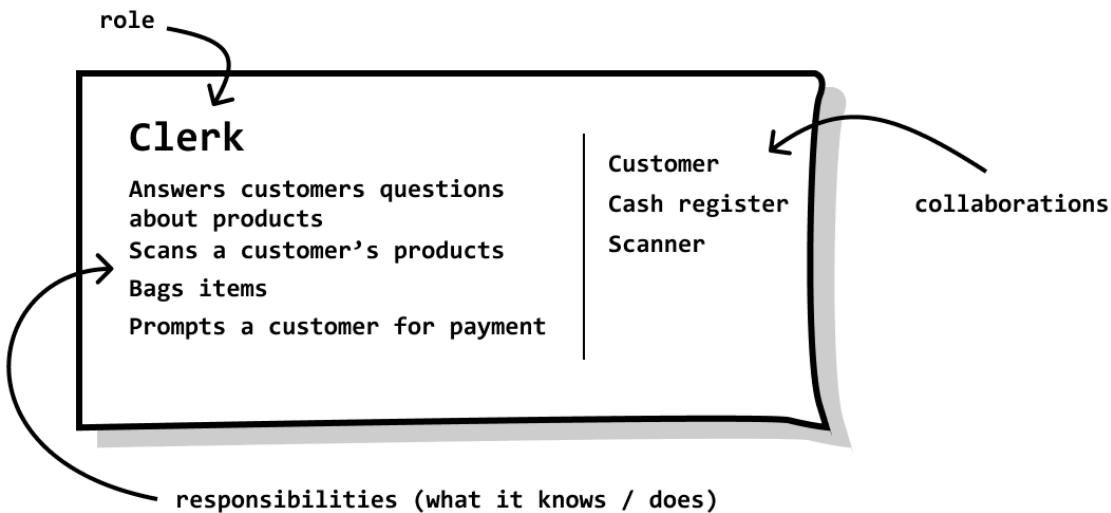
So if we're not coding at this point, how are we building our designs?

In this stage, the whiteboard is your friend. Notecards, paper, and markdown files are too. There are several ways to express your designs, but a popular tool is the CRC (Class-responsibility-collaboration) cards.

Popularized by Ward Cunningham and Kent Beck, CRC cards are a quick and cheap way for us to express our initial designs and see how collaborations may work.

Here's how they work:

1. On top of the card, write the **class name**
2. On the left, write the **responsibilities** of the class
3. On the right, write the **collaborators** (other classes) with which this class interacts to fulfill its responsibilities



It's important to note that CRC cards, just like all other design documents, are not something you need to stay faithful to. We merely use them to test things out and get ideas out of our heads so that we can play with them. Stay nimble.

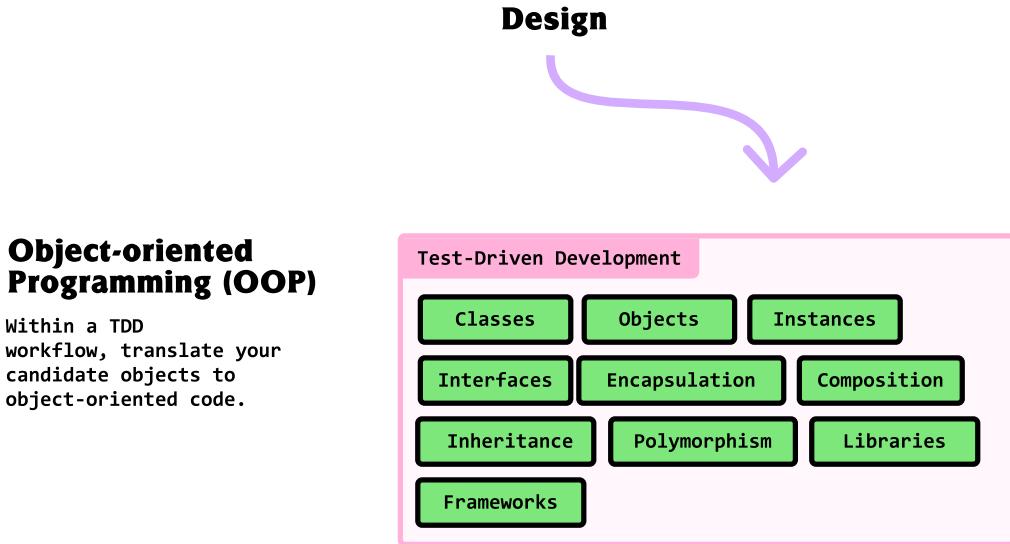
■ **Don't want to use CRC cards?:** That's cool. Use paper. Use markdown files. Use a virtual whiteboard. If you're using VS Code, install the draw.io extension; you can draw potential designs and then check them into source code.

■ **UML:** UML (the unified modelling language) has a number of different types of diagrams (like class diagrams and sequence diagrams). While we won't be doing a deep dive on UML in this book, we will call out the important aspects of how they work when we use them. Alternatively, you can download and print out a UML class diagram cheat sheet [here](#).

## Programming

### Goal: Translate designs into classes and interacting objects

The object-oriented programming process is merely a translation process — one done within the context of a TDD loop (usually).



In this stage, we rely on an understanding of the basic object machinery concepts to realize our designs.

Without analysis and design, **we'll remain fragmented object designers**. Instead of a focus on the machinery — trying to retrofit them into our problems, we need the new philosophy of RDD to express our designs in the first place. Then we use the typical object-oriented programming concepts to realize them.

The design specification that we start the coding from can a combination of any of the following:

- CRC cards
- UML class diagram
- Whiteboard drawing
- Markdown file full of roles, responsibilities, and collaborations
- Some designs in your head (but don't rely on this — get it onto paper)

With time building things, experience, and familiarity (with patterns, principles, and styles), you will more rapidly notice code smells, potential design pattern tradeoffs, and architectural styles that will lead to the non-functional requirements we need.

This is what mastery looks like.

- In 38. Detecting Code Smells & Anti-Patterns - Rough Notes, we learn how to detect code smells before they can become problems.
- In Part VI: Design Patterns, we learn about design patterns, tradeoffs, and what non-functional requirements they can help us with.

### Guide your code with tests

In one of the recommended readings from our phronimos developers, *Growing Object-Oriented Software Guided by Tests*, we learn how to use tests to guide our designs. It's

important that, while we may “program by wishful thinking”, that we keep RDD in mind. The classic version of TDD that we learned in Part IV: Test-Driven Development Basics doesn’t discriminate against our design philosophy, but eventually, we’re going to have to learn how to make effective use of mocks and stubs with *Mockist TDD*. For this, we must think in terms of roles, responsibilities, and collaborations.

- We learn how to perform *Mockist TDD* in 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - Rough Notes.

### **Why object design? It's foundational to our continued journey towards mastery**

Object design is the foundation for so many advanced topics in software development. Want to learn how to design effective systems? Distributed ones? Event-driven ones? Truly bespoke ones? This is the start of that journey.

■ **Becoming a front-end UI architect:** If you’re currently a front-end developer, this skill will propel you. For example, if you’re concerned with front-end architecture and you’re struggling to get your head out of tools like Redux or React Hooks; to design a good architecture for auth, errors, and structuring components for extensibility for change — well, there’s much to be gained here. This is the foundational skill.

### **How to learn object design: The Object Design Mastery Framework**

All skills take experience to learn. What I’ve done here is broken down my personal journey learning these topics along with the experiences you need to have to learn them. With this, you can see where you are in the mastery process and know how to progress to the next phase.

No skill I’ve ever acquired (coding, music production, social skills, or marketing) has ever come to me without a mixture of about 80% action, 20% theory. What I’m saying here is: **you must spend time building things if you wish to get good at this.**

This is what 2. Software Craftsmanship is about. Like Kobe Bryant, we have to put in the reps.

#### **Phase 1: Awareness**

**A focus on mastering the fundamentals of Responsibility-Driven Design to write intentional object-oriented code.**

At this stage, you’re aware of a few different forms of analysis. Since you’ve read Part III: Phronesis, you know how to learn a domain and identify requirements. Next is to learn the RDD philosophy and practice discovering roles, responsibilities, and collaborations (using rough sketches, CRC cards, a whiteboard, or UML) to create candidate designs. Before writing any object-oriented code, focus on this.

#### **Related chapters:**

- In 33. Responsibility Driven Design 101 - Rough Notes, we learn the philosophy of RDD. We come to fully understand what object-oriented designs are made of.

- In 34. The RDD Process By Example - Rough Notes, we walk through the process of using RDD to design real object-oriented software starting from a *Design Story* (a description of the problem in words). You'll learn how to find objects, assign responsibilities, decide on collaborations, and make decisions about a control style\*\*\*.
- In 35. Mapping Concepts & Designs to Object-Oriented Code - Rough Notes, we revisit the principles of object-oriented programming but from a practical, intentional perspective based on what we've learned about RDD. Here, we'll learn how to translate our object-oriented designs into flexible object-oriented code.

#### **Recommended projects:**

- Build a console or backend application using a design created using the Responsibility-Driven Design method.
- Build a front-end application using the Responsibility-Driven Design method.

### **Phase 2: Quality stewardship**

**A focus on maintaining the quality of our designs and preventing potentially bad designs before they become problematic.**

Now that we know how to design the RDD way, let's tighten up our designs; pull things back towards Simple Design. Here, we learn to detect common code smells, anti-patterns, and refactor them before they become problematic.

#### **Related chapters:**

- In 36. Better Objects with Object Calisthenics - Rough Notes, we learn a number of temporary rules we can use to improve the coupling and cohesion of our objects.
- In 37. Refactoring - Rough Notes, we learn about common refactors to make to improve the design to improve the design of existing code.
- In 38. Detecting Code Smells & Anti-Patterns - Rough Notes, we learn about code smells, anti-patterns and how to prevent them using object calisthenic rules and how to repair them with refactoring techniques.

#### **Recommended projects:**

- Find an old project of yours and refactor it using what you've learned about RDD, object calisthenics, code smells, and anti-patterns.

### **Phase 3: Test dexterity**

**A focus on keeping objects *humble*, separating core code from infrastructure code, and learning how to test all of the meaningful parts of your applications.**

With a strong background in RDD, the way you see testing changes.

As we discussed in 25. Testing Strategies, you're not going to use the same types of tests for everything. Here, we'll develop the ability to lean on a variety of testing techniques (such as the *Mockist* version of TDD) to effectively test collaborations, stateful code, and even legacy code.

You'll know when and how to switch techniques and ways to keep your tests readable and maintainable.

#### **Related chapters:**

- In 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - Rough Notes, we learn how to detect and decouple from subtle forms of infrastructure most commonly known to bleed into core code.
- In 40. Using Test Doubles - Rough Notes, we learn how and when to test classes and functions based on their behavior instead of just the resulting state (return values).
- In 41. Testing Legacy Code - Rough Notes, we learn how to introduce tests to previously untested code using techniques such as *seams*, *Characterization tests*, and *Golden Master* tests.

#### **Recommended projects:**

- Start or work on an existing application and ensure it is complete with unit, integration, and acceptance tests.

### **Phase 4: Pattern recognition**

**A focus on identifying opportunities to use patterns to improve evaluation qualities.**

Recall that evaluation qualities have to do with NFRs related to making code easier for us to work with. At this stage, we should be able to identify opportunities to refactor designs to patterns for the explicit purpose of improving a non-functional requirement.

#### **Related chapters:**

- In Part VI: Design Patterns, we cover the variety of design patterns (creational, structural, behavioural), learning about tradeoffs, and common real-world opportunities for their use.

#### **Recommended projects:**

- Go through existing personal projects and simplify designs by refactoring to patterns.

### **Phase 5: Naturalization**

**A focus on developing macro-level design intuition instead of relying on micro-level concepts.**

We'll have covered a lot of different topics by now. Upon scrutiny, it can be noticed that there's a relationship between many of the rules, principles, and techniques we've studied and practiced.

Design intuition should emerge. At this point, you know how to craft abstractions that developers find easy to use. A solid understanding of the interplay between coupling, cohesion, and connascence (the macro-level concepts) is evident. This opens up the door for more creativity and effective experimentation.

#### **Related chapters:**

- In Part VII: Design Principles, we develop our design intuition, learn about the most important software design principles and learn how to recognize and apply them in real-world scenarios.

### **Recommended projects:**

- Build and deploy a real-world application to the internet and iterate upon by adding new features.
- Mentor a junior developer.

## **Phase 6: System architecture**

**A focus on developing the correct architecture to meet required execution qualities.**

Execution qualities are about run-time behaviour and user-facing needs like reliability, resilience, efficiency, and so on.

At this stage, you should know how to design a system from (new or existing) components — using architectural patterns and styles that help meet the target non-functional requirements.

### **Related chapters:**

- In Part VIII: Architecture Essentials, we learn essential architectural principles that govern effective use of components. We also learn how and when to use the most influential architectural styles and patterns based on their tradeoffs.

### **Recommended projects:**

- Build a server-less application out of serverless functions and serverless architectural components.
- Build an application using CQRS or Event Sourcing.
- Integrate monitoring and logging into an application.
- Integrate caching into a new or existing backend architecture.

## **Summary**

- A test-driven approach to software development can help us implement the functional requirements, but without an understanding of design, solutions can be messy and usually fail to realize non-functional requirements.
- OO is the paradigm of design. It's about inventing abstractions, structures, and defining the relationships between them that meet the requirements.
- Highly effective for large and complex problems, OO gives us the design confidence and foundation necessary to address many common developer pain-points on both ends of the stack.
- Object design is comprised of three parts: *analysis, design, and programming*.
- Analysis involves identifying the requirements.
- Design involves using the Responsibility-Driven Design method to convert requirements into *responsibilities*, organize them into *roles*, and define how they *collaborate*.

- Design is about filling the responsibility gaps. We fill roles with objects by either **inventing** them \*\*\*\* or **integrating** them as components, libraries, frameworks, tools, etc.
- The final step, programming, is about the translation of *roles, responsibilities* and *collaborations* into object-oriented code using object machinery such as classes and interfaces.
- We can learn object design by going through the The Object Design Mastery Process.
- Learning how to write effective object-oriented code can teach us design patterns, help us cultivate a deep understanding of design principles, and discover when to rely on various architectural styles and patterns.

## Resources

### Books

- Object Design: Roles, Responsibilities, and Collaborations by Rebecca Wirfs-Brock and Alan McKean
- Fundamentals of Computer Science using Java by David Hughes

### Articles

- Object Design Lecture by Bernd Bruegge
- “An Aristotelian Understanding of Object-Oriented Programming” by Derek Rayside
- Responsibility-Driven Design via Wikipedia
- Class-Responsibility-Collaborations Card via Wikipedia
- [https://www.andypatterns.com/index.php/blog/blackboard\\_architectural\\_pattern/](https://www.andypatterns.com/index.php/blog/blackboard_architectural_pattern/)
- “Case against OOP is understated, not overstated” by Henri Tuholo
- How the Late Basketball Legend Kobe Bryant Mastered His Craft by Damon Brown
- “What makes for good object oriented design?” on Quora
- Programmers need a muse

## 33. Responsibility Driven Design 101 - Rough Notes

■ Responsibility-Driven Design is the influential object-oriented design method with which we can use to intentionally drive the design of object-oriented software. What follows is the essential philosophy of RDD, from responsibilities to stereotypes, components, object neighbourhoods and beyond. Use the RDD methodology to convert requirements into neighbourhoods of collaborating objects.

### Introduction

In psychology, the concept of *frames* describes the way you see the world.

Take being stuck on a train. Where one person may resist the situation, grow frustrated and treat it as a net negative occurrence, another may see it as an opportunity! *Ah, wonderful — a chance to meditate, catch up on some reading, or start a conversation with a stranger. How lovely.*

Essentially, your frames are your bearings on the world, your interpretation of reality, your cognitive biases.

In this book, I've asked you to adopt a number of **phronimos frames** on software development. These frames probably challenged the way you normally did things.

So commonly, when I sit down and craft designs with developers, I find that there's a tendency to see software as a Rube Goldberg machine of library concepts, frameworks, and platform features. I also note a tendency to jump straight to thinking about things like API endpoint names, databases, and which state management library we're going to use on the front-end. This conditioning to focus on the imperative is one of the first things I want to decompose. We've done some work here already.

With the notion that **13. Features (use-cases) are the key**, we shift to a **focus on the behaviour** instead of database-first, API-first, or tooling-first ways. This drives us to increase the importance of learning things like BDD, DDD, acceptance testing, identifying the use cases, and now, to develop object-oriented software, object-oriented design.

If we're already thinking in terms of behaviour, then the next shift we need to make is easy. It's a shift to see behaviour as **responsibilities**. Personally, this frame shift came from reading the legendary *Object Design* book, particularly the following quote:

“Object-oriented applications are composed of objects that come and go, assuming their roles and fulfilling their *responsibilities*.” — Rebecca Wirfs-Brock

Meditating on this statement, seeing behaviour as responsibilities, we illuminate the path to mastery for many topics including testing, when to mock, stub, how to implement non-functional requirements, deal with architecture on both sides of the stack, and use frameworks, libraries, and many other tools effectively. Responsibilities are the next step.

I'd like you to have a similar revelation. So in this chapter, we're going to learn the philosophy of Responsibility-Driven Design. I hope you never see code the same way again.

## Chapter goals

In this chapter, we:

- Learn the philosophy of Responsibility-Driven Design.
- Learn how to think about objects from an RDD point of view.
- Learn about the six object stereo object stereotypes and how they can help you more invent and understand how objects should collaborate.
- Learn how contracts, concretions, classes, and contract technology work to convert designs into potentially polymorphic constructs.
- Learn how object neighborhoods evolve and turn into components.
- Discuss control styles: strategies for communication between object neighborhoods.
- Learn why effective use of frameworks and libraries is to merely fill in the responsibility gaps, and why this can sometimes be harder than we expect.

## Object basics

Let's break down this new frame of object-oriented software development by revisiting a concept that might seem silly to talk about at this point, but we're doing it anyway: objects.

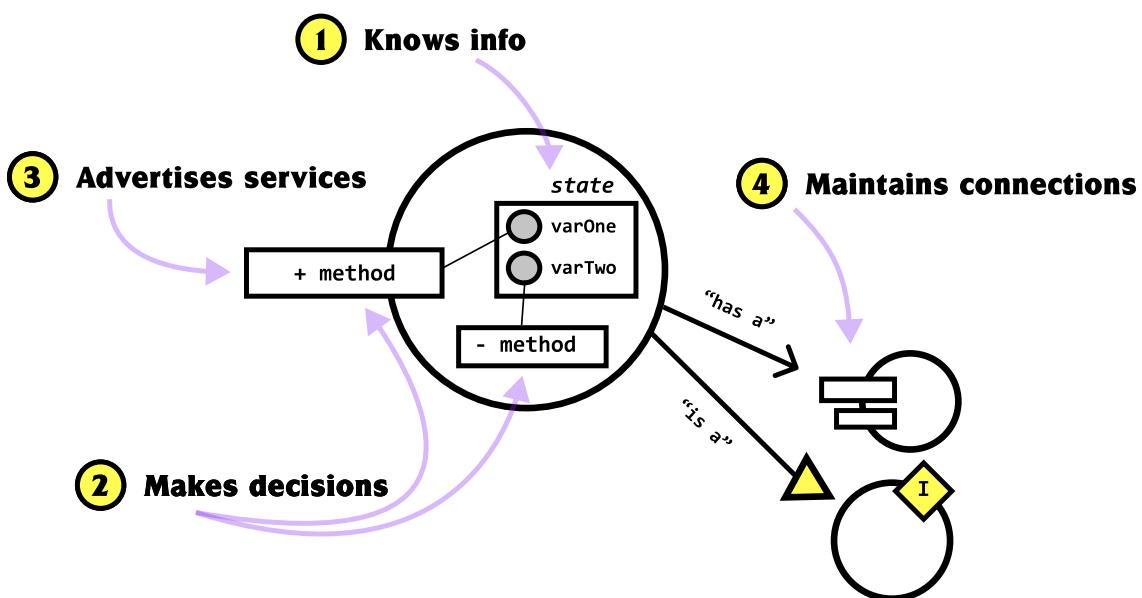
## Object capabilities

Hopefully this doesn't come as a surprise to you, but in the world of object software, the **object** is the fundamental tool.

Objects have four main purposes.

1. **To know information:** We know this. Information objects hold state called state.
2. **To make decisions:** They use methods (often reliant on state) to define meaningful business rules, algorithms, and scripts that help decide on the right thing to do.
3. **To advertise their services:** Objects are only useful when they're used by other objects. To do this, we introduce public methods and make them easy and understandable to use.
4. **To maintain connections to other objects:** Objects offload some of their tasks and decision-making to other objects. That's how we decompose problems and get the web of objects, community, or network-like analogy that Alan Kay originally had in mind for OO.
  1. Composition, delegation, or subclass inheritance
  2. Mention that there are only two ways to do this (composition & inheritance — two topics we'll discuss more in detail below).

*Fix this diagram here → it's not “is a”, it's “played by” — and mention that this is something borrowed from a topic called “Role-oriented programming”, but that's actually what we're doing here*



■ **Is-a vs. has-a:** When objects refer to each other in an *is-a* context, that's called *classical inheritance*. When objects rely on each other in a *has-a* context, we are still applying inheritance, but in this case, it's typically either something called composition or delegation. Don't worry about this too much just yet. It'll be more important when we get to 35. Mapping Concepts & Designs to Object-Oriented Code - Rough Notes.

Why is this significant? Well, with these four capabilities, we make the four primary object-oriented concepts possible: abstraction, encapsulation, and polymorphism, and inheritance.

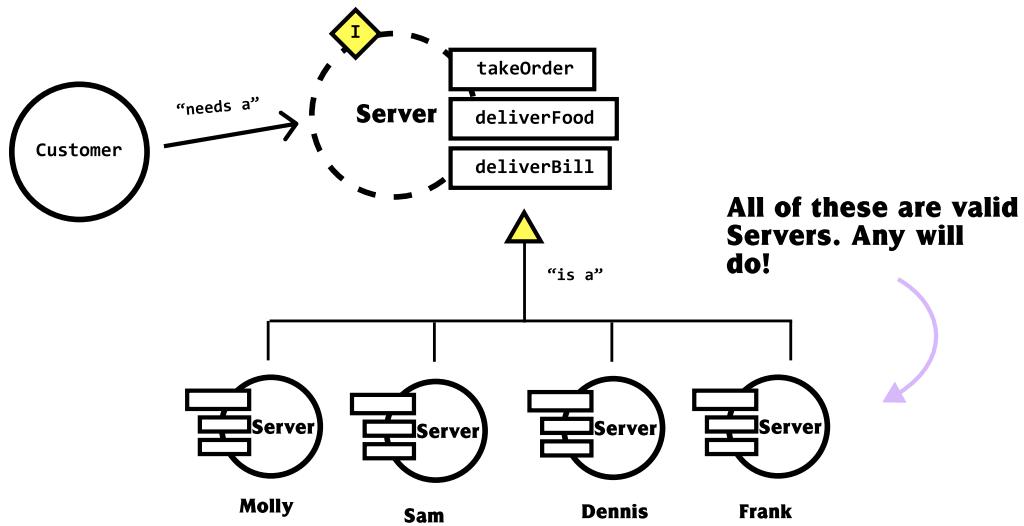
### Objects come and go, filling in roles

As discussed in Part III: Phronesis, features are the main thing we're interested in developing. When we use the object-oriented paradigm, we build features by connecting objects, each with unique behaviours, together. By developing collaborations, objects interact with each other in meaningful ways.

They say no object lives in isolation. That's true. Just like actors in a broadway performance, objects too, work with others, filling in their specific *roles* to play. Since multiple objects can fill the same *role*, when the show starts, it doesn't actually matter which object is on stage; only that it's an object that fills the intended *role*.

For example, if you're a manager at a local restaurant and you notice that you're understaffed in the *Server* department on a very busy evening, all you care about is getting *one* or more of \*\*your servers into the shop to help out. You don't really care if it's Molly, Sam, Dennis, or Frank. As long as someone with the role of *Server* appears, you're relieved.

**Fix this diagram here → it's not “is a”, it's “played by” — and mention that this is something borrowed from a topic called “Role-oriented programming”, but that's actually what we're doing here**



This is what it means for objects to come and go. It's more important for us to *contractualize* a set of *responsibilities* that our app deals with into a *role* (using an abstraction technology such as interfaces, abstract classes, or types) so that we can safely expect that when an object fills in to the job, it'll be equipped with the chops to do the work.

What we have just discussed here is the foundation for effective use of polymorphism.

■ **Objects & roles:** Objects play one or more sets of roles; roles contain a set of responsibilities.

## Object models violate real-world physics

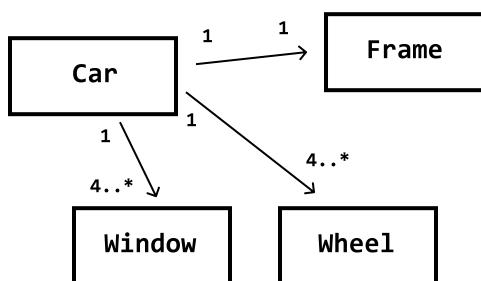
"At the heart of object-oriented software development, there is a violation of real-world physics. We have a license to reinvent the world, because modeling the real world in our machinery is not our goal." — Rebecca Wirfs-Brock

A common mistake in OO is to strive for realism. To try to make designs mirror reality. We shall see that this is not necessary, and in fact, often harmful to our designs.

Consider three different modelling scenarios:

1. Modelling a car
2. Modelling a car rental application
3. Modelling a multiplayer racing game

(1) Modelling a car is easy. Cars have doors, a frame, wheels, windows, and so on. These concerns can be mostly seen as mostly *domain* concepts. We can draw this relationship between concepts as a UML-like drawing.



If realism was our intention, why stop there? We could go further. We could model engines, carburetors, ignition sparks, if the car has 4-wheel-drive and more. Why not? Why don't we go further? **Because realism is not our intention. We only model what is 100% necessary to meet the application requirements.** Many developers get stuck here, modeling things into oblivion — an easy mistake if your work isn't guided by clear requirements.

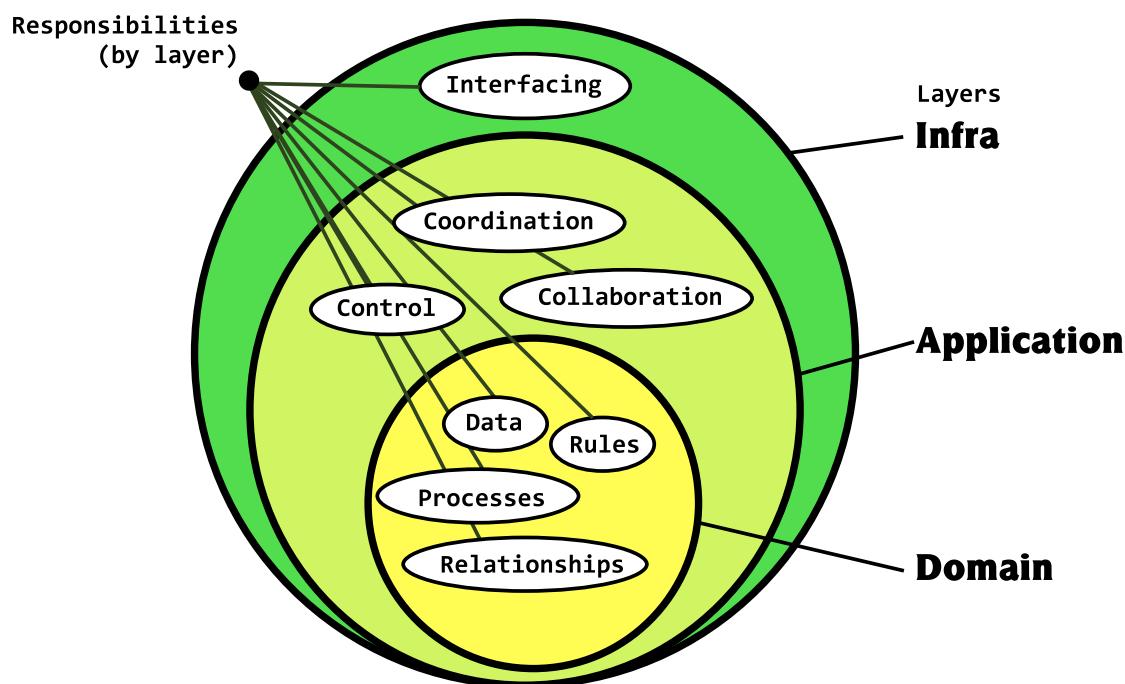
This makes sense when we think of what it takes to model what's necessary to build a (2) car rental application. We're more used to these types of problems. In 16. Learning the Domain, we saw that the Ubiquitous Language represents **rules, processes, data**, and the **relationships** between them. This is all *domain* layer \*\*stuff. If we studied DDD, we know which tactical patterns from we'd use to model these things (*Value Objects, Entities, Aggregates*). That's a good start.

Soon, we realize we must also invent objects that represent concepts only understood by application developers (us). We invent *infrastructural* things like databases, APIs, or other forms of networked communication and storage so that our app can **interface** with the real world.

It's not until we bring our awareness to the complexity of something like submitting a form that we recognize how many other types of objects we need to invent or integrate. Objects to handle user input, validate the form, formulate a request, send it, sanitize it, and route it to correct destination object to handle the request. Here, we see we must also invent *application layer* objects. These are objects that handle **control**, **collaboration** and **coordination** between each other to get things done. This is the technical aspect that pulls everything together like glue.

Now consider something outside most of our web development wheelhouses: (3) to model a multiplayer racing game. We still have the *domain* concepts that describe what a car is. We also still have common *application* layer concerns like handling what happens with user input. But now, we have the unique responsibilities of drawing the other racers on the screen, updating the map, showing speeds, knowing the placement of the racers, accepting input on the rematch screen when the game is over, and more.

Now we're *really* starting to get into technical territory. Computer graphics, web socket connections, dealing with ping, reconciliation, making efficient use of resources — all sorts of computer-y details!



You have to understand that we are *not* in the business of creating models that mimic the real-world. We are in the business of creating models that get things done. In this case, inventing or integrating objects that assume roles capable of handling the responsibilities that arise (such as computer graphics and multiplayer networking).

It's about inventing or integrating existing objects (from libraries and frameworks) that make the end result possible. Such objects often have nothing to do with the physical world.

“All models are wrong. Some are useful.” — Rebecca Wirfs-Brock

## Core design concepts

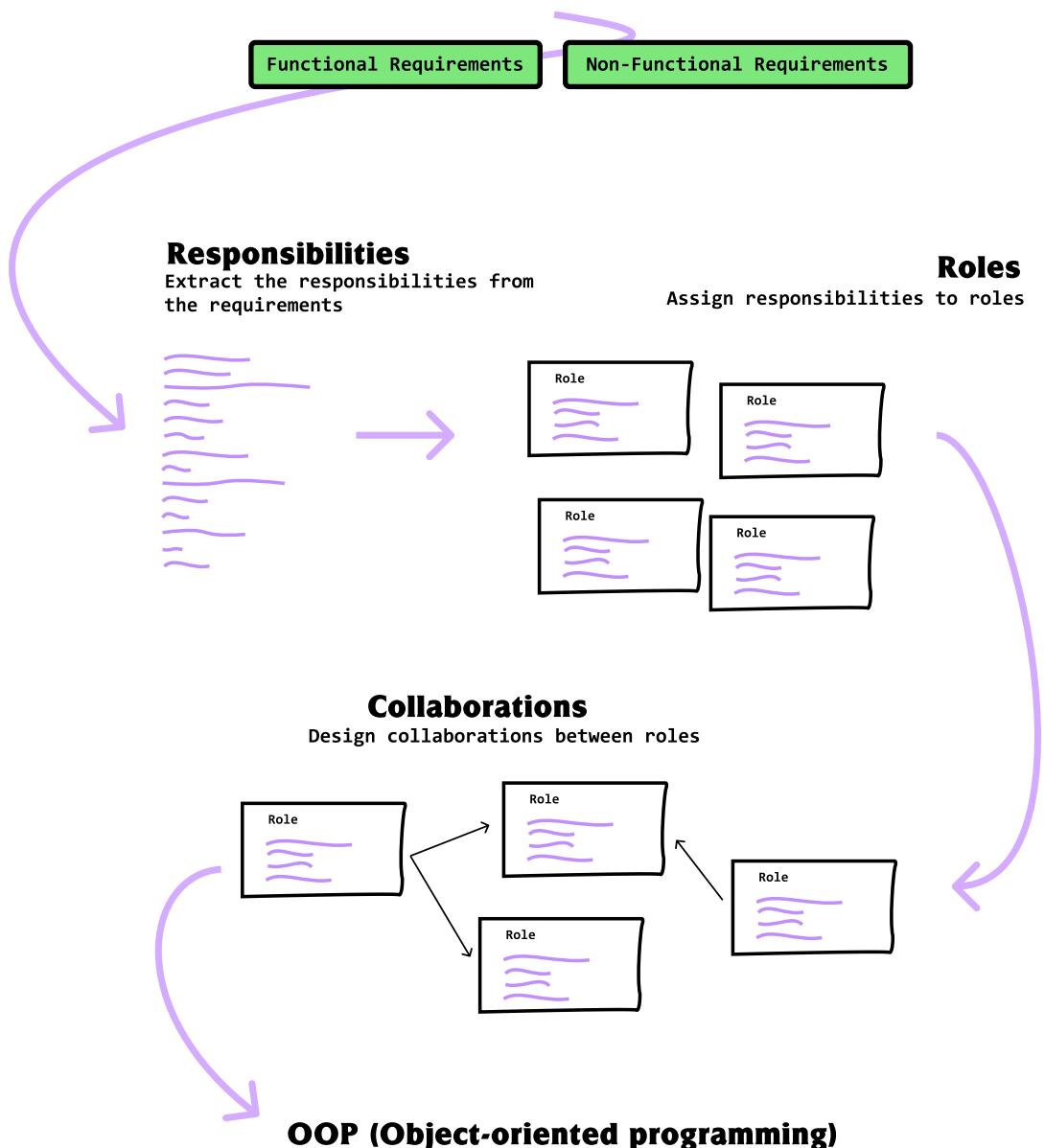
The core design concepts behind RDD keep us focused on behaviour — making sure that we never create objects out of thin air. There's always a reason for an invention. The concepts are:

- **Responsibilities**: an obligation to *do* (some task/behavior) or *know* something (some data)
- **Roles**: a collection of related responsibilities that can be used interchangeably
- and **Collaborations**: an interaction between objects or roles (or both)

Let's start with responsibilities.

# The Responsibility-Driven Design Process

## Start



## Responsibilities

Applications contain a number of functional and non-functional requirements, right? Well, to design objects that do the work behind these requirements dictate, we must break down all of the *responsibilities* within a requirement.

- **Requirement & responsibilities:** Requirements are features, and features con-

tain a number of different things that must be known (information) and performed (task/behaviour). These are obligations for doing or knowing are *Responsibilities*.

To demonstrate, let's imagine we've been asked to build a tiny, four-wheeled, house robot. What could be involved in the design of this 'lil guy?

One of the best tools to kick off design is a *Design Story*.

**Design story:** The house-bot gradually learns the layout of the house, although if you're in a certain room with him, you can say "house-bot". Then he'll look at you and light up. And then you can say, "this is the kitchen", he'll blink and then say "gotcha, this is the kitchen" and learn. He can also be told to go to certain rooms with commands like "go to the kitchen". He also doesn't bump into walls while moving.

Based on this design story, what are some of the *responsibilities* our house-bot would need to handle? What does he need to do? What does he need to know?

Without filling the page, here are a few that come to mind.

*Responsibilities (doing and knowing):*

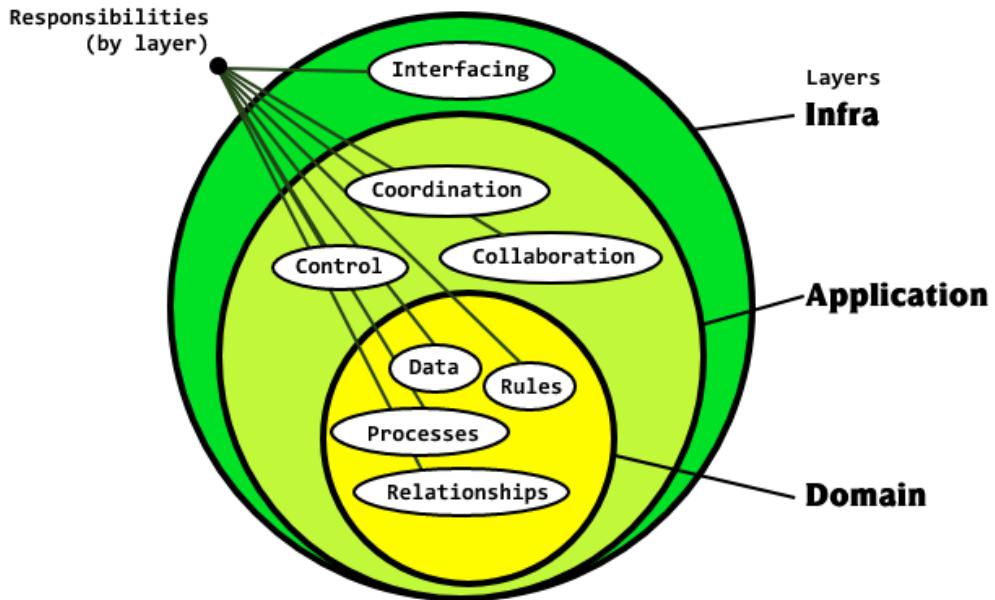
- Knowing how to get to a certain room
- Turning left and right
- Moving backwards / forwards
- Taking pictures

*First of all, let's know the stereotypes.*

*Second of all, let's know the layers of the layered architecture*

*Third of all, let's know what layer each stereotype belongs to (usually)*

- *Diagram to demonstrate*



- → Picture (Information Holder *guess*)
- → takePicture (Use Case/Controller *guess*)
- Composing a layout from pictures
- Knowing when we've moved into another room
- Creating a layout of a room
- Connecting room layouts to each other through halls
- Updating the layout of the house
- Listening for voice
- Detecting when a "house-bot" command has been said
- Blinking
- Playing back audio in response to a command
- Playing back audio that says he didn't understand the command
- Handling the right command

■ **How do you do this?:** In 34. The RDD Process By Example - Rough Notes, we learn "Techniques for finding responsibilities".

This is an initial list. It will grow. More responsibilities will appear as we continue to design and add tests. However, what we've done here is great because we've started to become explicit about what's actually required to make this thing work. The low-level details emerge. Suddenly, we're dealing with less obscurity and we can begin to wrap our heads around the work to be done.

As you can see, the features we need to build contain many *responsibilities*. Next, we need to do something with these *responsibilities* — divvy them up into *roles*!

## Roles

There are a number of different techniques we can use to find the *roles* that should take charge of these *responsibilities*. Among those techniques are to think about the *Use Cases*, the types of *domain*, *application*, and *infrastructure* objects necessary, and so on.

Thinking on this for about sixty seconds, here's what I come up with.

*Roles*:

- PathFinder
- Path
- CommandController
- RoomLayout
- Home
- Blinker
- Microphone
- Use case (generic)
- NameRoom (use case)
  - execute
- ChangeRoom (use case)
- Wheels
- Motor

In the earlier days of object design, it was advised to merely “look for the nouns” in the design story. That works, but it’s no longer known as a best practice due to the tendency to only identify *roles* that represent the *domain* and language used by domain experts in the real-world.

As we now know, object design is just as much about the invention of programmer-specific machinery as it is about user-facing or reminiscent machinery.

A superpower for discovering *roles* is to become aware of the stereotypical roles that tend to occur (*object stereotypes*). We will learn about these shortly.

■ **How do you assign responsibilities to roles?**: It can be a little bit of an art, but luckily, our phronimos developers give us some guiding questions and techniques we can ask and follow. We learn about these in “Techniques for assigning responsibilities to roles” in 34. The RDD Process By Example - Rough Notes.

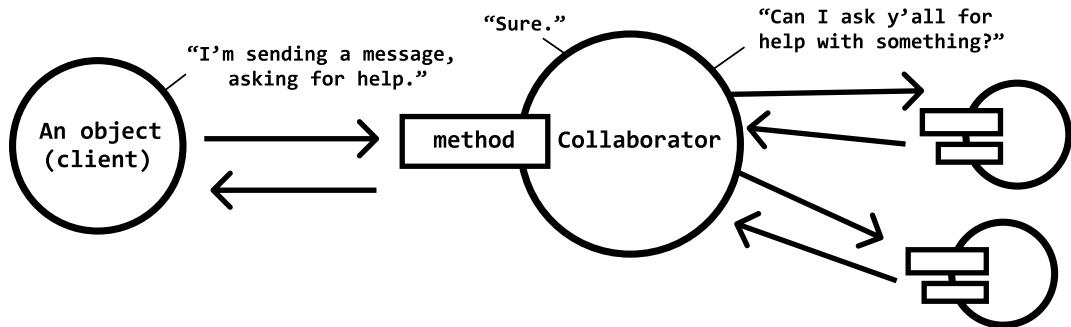
## Collaborations

Nothing interesting happens unless we design *collaboration* between *roles*. This is often the most challenging part of RDD because what we’re actually doing is discovering how we want to design *control* into our application (something that is often abstracted away from us by libraries and frameworks).

Our application gets smarter when objects collaborate. We can go on to design entire neighbourhoods of collaborating objects, and as the authors of *Object Design* say, collaborations “raise the IQ of the whole neighbourhood”.

When we design *collaboration* between *roles*, the one asking for help is the *client* and the one

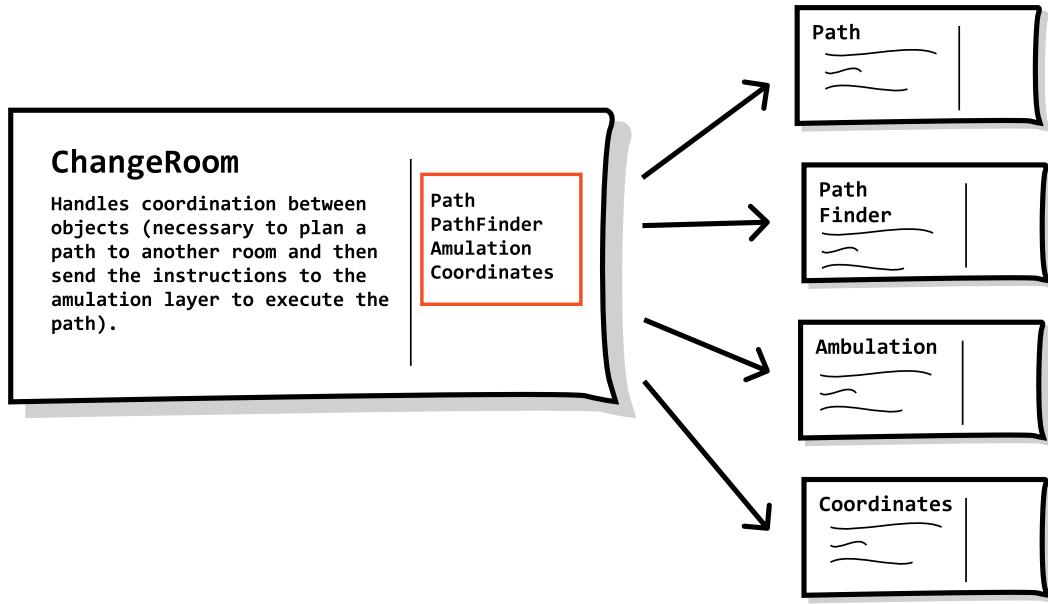
helping out, responding to the message is the *collaborator*. This is essential theory for *Mockist TDD* which we learn in 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - Rough Notes.



Like *roles* and *responsibilities*, there are many methods to discover *collaborations*. One of the methods we can use is to look at a responsibility for doing and ask “what *role* or object helps with this task?” and to look at responsibilities for knowing and ask “what *roles* or objects need to know this information”?

Thinking about the *Path* object candidate, we ask ourselves “what *role* might need to know this”?

Upon reflection, we realize that the *ChangeRoom* use case would likely need to know about *Paths*, for sure. Thinking through the pseudocode for the *UseCase* (like we did in 21. Understanding a Story ), if the robot was going to change rooms, the *ChangeRoom UseCase* would also likely need to handle coordination between a number of other objects that make changing rooms possible too. This, in and of itself — needing to handle coordination between objects — is a new *responsibility*. We just happen to know that this is typically something a *UseCase* does.



Sometimes new *responsibilities* emerge. Sometimes new *roles* emerge. That's because we're getting real about **how this could work**.

Now, are these collaborations definite? Are we 100% going to use these *roles*? Are we 100% going to configure them to work this way? Maybe. Maybe not. We can't know until we try our designs out for real with tests and scenarios.

But we're further along, and that's what this design process is all about.

■ **Finding collaborations:** This is often the hardest part of RDD. We discuss strategies for identifying collaborations via “Techniques for identifying collaborations” in 34. The RDD Process By Example - Rough Notes.

## Object stereotypes

One reason I still find LEGO so welcoming after all these years is the fact that we still start out with the same baseline elements: the brick, the baseplate, the tile, the mini-fig, etc.

Object stereotypes are similar. They're our starting pieces for object-oriented designs. We can rest easy, find *roles*, and possible *collaborations* a lot easier if we know that the majority of our objects tend to conform to one or more of the following object stereotypes:

1. **The Information Holder**
2. **The Structurer**
3. **The Service Provider**
4. **The Coordinator**
5. **The Controller**
6. **The Interfacer**

### I. The Information Holder

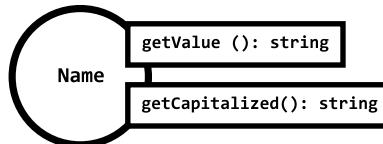
The *Information Holder* is perhaps the simplest stereotype we know of. Most similar to idea of a *Value Object* made popular in Domain-Driven Design, *Information Holders* are usually *domain* layer concerns, but they can also represent other things as well (such as *Data Transfer Objects*).

## 1 Information Holder

- Knows things, encapsulates data
- May provide computed or derived versions of the data it encapsulates

### Examples

- Entities
- Value objects
- Aggregates
- Data Transfer Objects
- Domain Events



```

class Name {
    getValue () {
        return this.name;
    }

    getCapitalized () {
        return capitalize(
            this.name
        );
    }
}
  
```

What's important to know is that the stereotypical responsibility of this *role* is to hold information. Anytime you notice responsibilities to hold info, consider designing that object candidate as an *Information Holder*.

## 2. The Structurer

*Structurers* are primarily about creating and maintaining relationships between objects. *Structurers* protect against invariants and can make it easy to find related objects.

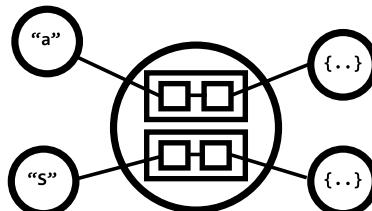
Examples of *Structurers* are *Aggregates* (from Domain-Driven Design), hash tables (or any mechanism responsible for caching), or objects that pool or handle connections.

## 2 Structurer

- Maintains the relationships between objects or values

### Examples

- Dictionary
- Hash tables
- Caches
- Entities, Value Objects



```

abstract class HashTable <T> {

    private data: {
        [key: string]: T
    };

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

    add (key: string, val: T) {
        if (this.exists(key)) return;

        this.data[key] = val;
    }
}
  
```

■ **Aggregation & composition:** An *aggregation* is an object that represents a concept as a whole and handles managing how its parts are accessed and changed. Composition is re-

lated but it adds an additional meaning to aggregation. Composition says that parts of the aggregation can't exist independently (ie: a *wheel* can't exist unless it belongs to a *car*).

### 3. The Service Provider

Service providers are objects that sit passive and contain useful, specialized \*\*computations.

**Stable service providers:** They typically sit dormant waiting for other objects to call them, but because they're often relied upon by so many objects, they may contain responsibilities that relatively *stable*, not changing often.

For example, utility classes like *StringUtil*, *DateUtil*, *NumberUtil* provide respective text, date, and number utilities and probably don't change dramatically (or at all for that matter). For this reason, we'd say they're stable. Just think about what would happen if we had to change the `JSON.stringify` method in JavaScript.

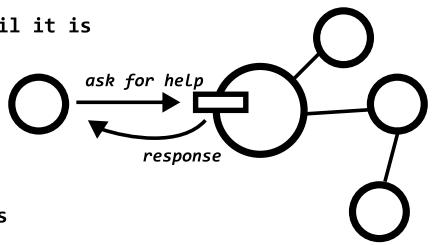
3

### Service Provider

- Does a service for other objects upon request
- Generally passive until it is activated by another object

#### Examples

- Domain Services
- Utility classes
- Configuration classes



```
class SyncService {  
    public determineItemsToSync (  
        desktopItems: Items,  
        cloudItems: Items  
    ): SyncTask {  
        // Determine new  
        // Determined changed  
        // Determined removed  
        ...  
    }  
}
```

**Pure fabrication:** Alternatively, service providers are also objects that live in the domain layer yet don't represent concepts from the problem domain. Instead, they're objects that are specifically invented to ensure other domain objects maintain low coupling, high cohesion, and encapsulation. In Domain-Driven Design, we call these types of classes *Domain Services*.

■ **Domain services:** We learn more about *Domain Services* in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS.

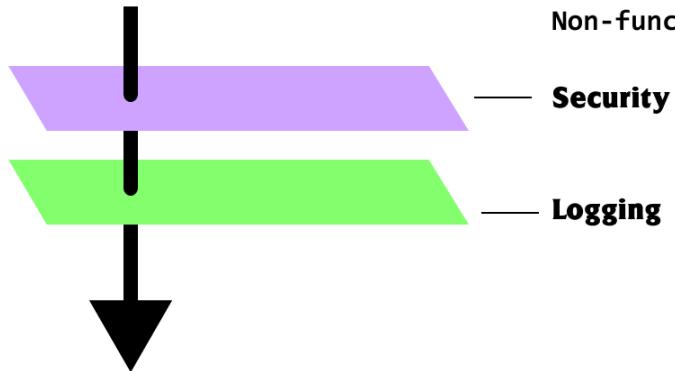
**Cross-cutting concern service providers:** Lastly, service providers are known to contain responsibilities that are commonly known as “cross-cutting concerns” like logging, security, or tracing (each of which are non-functional requirements). A concern = a functional or non-functional requirement. Non-functional requirements are the ones that are “cross-cut”. This means that *Use Cases* either **rely on** these service-providers OR we design the collaboration so that the service providers **wrap** the *Use Case* (hence the name “cross-cutting”).

## Core concern

Use case, functional requirement

## Cross-cutting concerns

Non-functional requirement



Either solution works, but there are tradeoffs to both.

## 4. The Coordinator

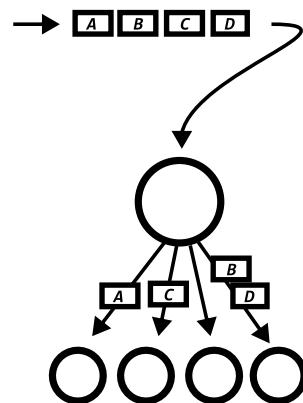
The *Coordinator* stereotype has one real purpose: to pass information to other objects so that they can do work. When we look at it like this, ***Coordinators are one of two objects most responsible for determining how control happens in our applications (the other is the Controller)***.

## 4 Coordinator

- Reacts to events and relays them to others
- Event-driven
- Helps decouple listening/detecting from processing
- Coordinator listens; others process

### Examples

- Event Listeners
- Express Routers
- Handling user input
- Web servers



```
class AppEvents implements EventsSubject {  
    private eventObservers: {[event: DomainEvent]: EventObserver[]} = {}  
  
    // Subscribe to a particular event  
    attach(observer: EventObserver, event: DomainEvent): void {  
        if (!this.eventObservers.hasOwnProperty(event)) {  
            this.eventObservers[event] = [];  
        }  
        if (this.eventObservers[event].includes(observer)) return;  
  
        this.eventObservers[event].push(observer);  
    }  
  
    // Notify all observers about an event.  
    notify(event: DomainEvent): void {  
        for (const observer of this.eventObservers[event]) {  
            observer.handleEvent(event);  
        }  
    }  
}
```

## 5. The Controller

We've dealt with this particular type of stereotype before, but we've been calling it the *Use Case* instead. In RDD, the *Controller* stereotype is the exact same thing as the *Use Case* pattern.

The stereotypical responsibility held by a *Controller* is to act as a higher-level (*application-layer* specifically), declaratively focused object that coordinates the behaviour of lower-level concerns.

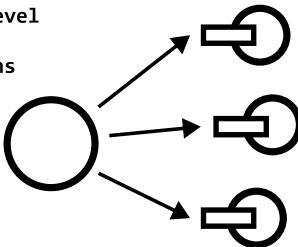
5

## Controller

- Controls and directs the actions of objects
- High level; delegates the low-level work to the appropriate objects
- Makes application-level decisions
- Makes objects interact

### Examples

- Use case  
(application service,  
interactors)



```
class PostComment {  
  async execute (  
    input: PostCommentInput  
  ) {  
    // Get post  
    const post = await this.repo  
      .getPost(input.postId);  
  
    // Create comment  
    const comment = Comment  
      .create(input.comment);  
  
    // Add comment to post  
    post.addComment(comment);  
  
    // Save post  
    await this.repo  
      .savePost(post);  
  }  
}
```

## 6. The Interfacer

The last of the stereotypes is the *Interfacer*. The *Interfacer* primary acts as a bridge between object neighborhoods (internal and external).

**Internal interfacer:** To cross internal object neighborhoods, we need to make the entry-point for our components as simple as possible. The object which sits at the front, relaying messages to objects behind the scenes is the *Interfacer*.

**External interfacer (gateway):** To cross external object neighborhoods (as is the case when we send messages out to remote APIs or to libraries which we don't own), the more common name for the object that encapsulates this access is called a *Gateway*, however — stereotypically, it's an *Interfacer*.

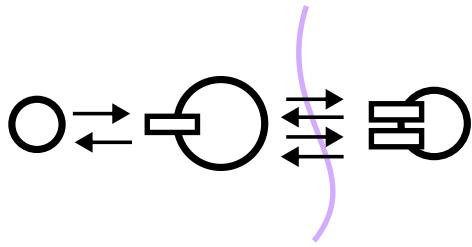
**UI-related interfacers:** *Interfacers* which listen to events from the UI and pass them to non-UI related parts of our applications are typically just called *Event Listeners* (in JavaScript, there is an *EventListener API*) whereas objects that format data for presentation in the UI are aptly named *Presenters*. Both are stereotypical *Interfacers*.

## 6 Interfacer

- Provide a means to communicate with other external parts of the system; be it external APIs, infrastructure, or end users
- Abstracts away some of the complexity of integrating with a more complex API
- Responsible for crossing boundaries

### Examples

- HTTP Server
- GraphQL Server
- Repositories
- Facades to external APIs  
(ex: StripeAPI, MailAPI)



```
class ReportingAPI {  
  
    async getTransactions (startDate: Date, endDate: Date) {  
  
        if (this.accessToken === '') {  
            await this.setAccessToken();  
        }  
  
        return axios({  
            method: 'GET',  
            url: `https://api.paypal.com/v1  
                  /reporting/transactions`,  
            params: {  
                'start_date': startDate,  
                'end_date': endDate  
                fields: 'all'  
            },  
            headers: {  
                "Content-Type": "application/json",  
                "Authorization": `Bearer ${this.accessToken}`  
            },  
        })  
    }  
}
```

**Todo → need to update the image here for interfacers because they actually do more — this is just *one* example.**

### Considerations on stereotypes

**Stereotypes make it easier for us to identify objects:** This should clear up a lot of headroom for you. Because we know the six general types of objects we're likely to encounter, we'll find it easier to invent object candidates.

**Stereotypes help us emphasize what matters about our objects:** We're constantly striving to ensure that our objects are cohesive and well-defined. If we discover a new object candidate, we can ask ourselves “what stereotype is this most like?” This question helps us determine if we should redistribute misplaced responsibilities to other object candidates to make them more cohesive and focused.

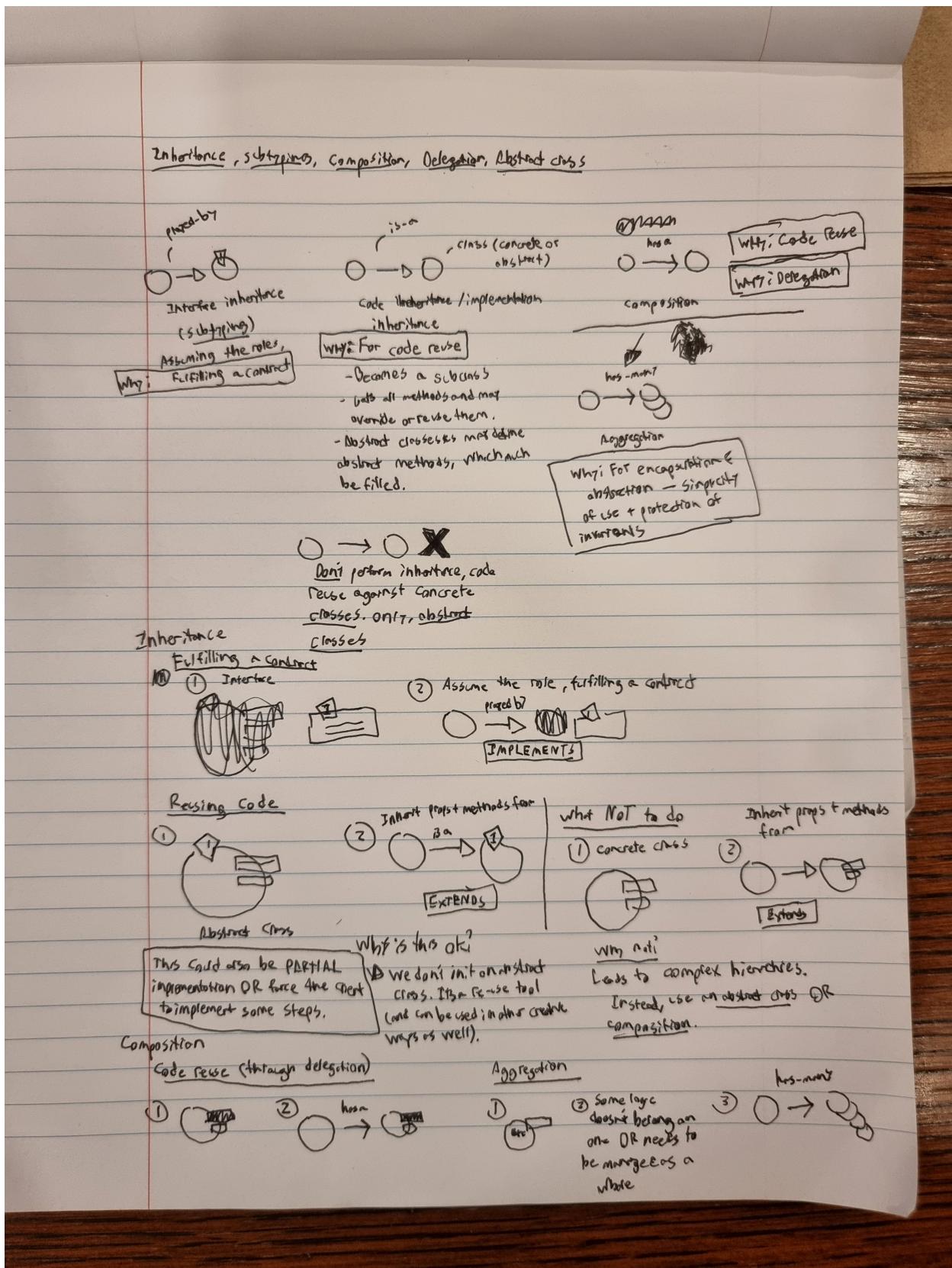
**A candidate object can belong to more than one stereotype:** Taking it the other direction, there are times when objects have responsibilities that float between multiple stereotypes. For example, consider the *Repository* pattern. If it were to also cache the objects it retrieves, we'd say it acts more like a combination of the *Interfacer* and *Structurer* stereotypes.

**Object stereotypes make understanding & communication simpler:** It's easier to talk

about design with stereotypes in mind. Once you're aware of object stereotypes, you'll start seeing them (or the lack of them) in the frameworks and libraries you're using. This presents a great opportunity for us to get what we need by creating it ourselves.

**Exercise:** Look through one of your projects and find the classes, functions, or components you've built in the past. Ask yourself which stereotype(s) they most belong to. If this is the first time you're hearing about object stereotypes, you'll never be able to see your objects the same way again.

## From design to code



So, about this section — I'm probably going to just allow it to completely adopt what

I wanted to talk about in *mapping OO*. I'm just going to cover those ideas here instead in as much of a crunch as possible. I even have a picture here I can use that really crunches as much of it down as possible.

Regardless of if the design is in our heads, on paper, on a whiteboard, CRC cards, or a doodle, we need to know how to make these designs real.

Let's shift our awareness to some of the important concepts around realizing our responsibility-driven designs.

Mention that we're going to focus on the *what* here, merely learning the concepts, and leave the *when, why* pragmatism to 35. Mapping Concepts & Designs to Object-Oriented Code - Rough Notes.

## Contracts

In the physical world, contracts are serious agreements made between two or more parties that dictate mutual obligations. When I rent an apartment, what's expected of me, the renter? What's expected of the landlord?

Object-oriented software heavily relies on contracts. What happens if an object expects a neighbour to know how to respond to a certain kind of message, but it doesn't? Done! This is a broken contract and it should be treated as an exception.

We want to limit this from happening as much as possible.

Contracts are an important part of the design and *realization* of objects. In OO, we think of the design of an object in three parts, we could say it's:

1. **The API (the responsibilities for doing/knowing):** The public interface, exposing the possible set of methods that can be called.
2. **Preconditions & postconditions (behaviour):** Given X (preconditions), when Y (behaviour), what Z (postconditions) can we expect?
3. **How (concretion):** The implementation of the behaviour.

Therefore, the contract is the first two parts of the design of an object,

■ **Contracts:** Contracts define the *API* and *behaviour* of a *role*.

## Contract technology

Alright, so we know what contracts are, but ... what are they?

To actually create contracts, we use contract technologies: interfaces, abstract classes, and if you're using a language with types like TypeScript, types!

We know what interfaces look like, but for demonstration purposes, take a look at the following one.

```
interface Storage<T> {  
    get (key: string): T;  
    set (data: T, key: string): void;  
}
```

**Understanding the API:** First part of the design — the API. What's the API here? The API is the total set of method signatures. It dictates what responsibilities for *doing* can be called and which for *knowing* can be called. The message signature, correct. The get and the set methods with their parameters and return types.

**Understanding the preconditions & postconditions:** What about the second part? the preconditions/postconditions? Well, we can *imply* as much behaviour as possible within the method signatures, but it's never going to be complete. We'll never be able to describe the preconditions and postconditions perfectly and explicitly this way.

For example, there's no part of this contract that says that:

1. Given, I haven't added anything to storage,
2. When: I perform get
3. Then: I get nothing back.

Nowhere is that expressed in the design of this interface. However, what this little realization *did* do is drive us to see that we should change the method signature to get (key: string) : T | Nothing to account for the scenario where nothing is returned in a more explicit manner.

this is why I advocate for unit testing (especially Given-When-Then style) for tests. They help us expose problems in our designs; in this case, the lack of detailed ability for our API to understand the implied preconditions/postconditions of our API.

**Contracts are for programmers. They exist to help us understand, as much as possible, what can be done, how to implement roles, and what to expect.**

It's not easy to t's hard for us to communicate the post-conditions and pre-conditions in the behaviour,

- example: how can you express the fact that if you're trying to send money to a friend and you don't have any money in your account, how can you design the public API to express that?
- this is why we make such special use of the Result type, because it gives us the opportunity to express these things (see 12. Errors and exceptions).
- the more we improve upon our contracts, the more robust, resilient, and less prone to bad potential designs they become.
  - All of this puts pressure on us as API designers to ensure that we can express as much of the conditions of use as possible within the confined set of things possible in the contract technology.
  - Later: Abstract classes give us more flexibility and allow us to define more rules.

■ **Contracts are for programmers:** They let us know what to expect, etc ... , etc.

## Concretions

- Concretions

### Concretion

Then we can talk about concretions are and what they mean — it's when we *fulfill* a contract.

## Contract technology vs. concretion technology

For the contract technology, we have interfaces, abstract classes, and types.

### Interfaces, Abstract classes, Types

→ There's a lot that can be said here around when to use which, what the implications of using each are, and so on. For now, it's enough to understand the logical grouping of these. These are *contract* tools, for separating some part of what should happen into the *what / declarative* so that we (or others) can implement the remainder of the *how / what* later → we will talk about when and how to use these in *Mapping Designs to Object-Oriented Code*

Later

- When to use an interface?
- When to use an abstract class?
- When to use just a class?

For the concretion technology, we have classes.

### Classes

→ inheritance (concrete class, abstract classes, subclass, composition)

- is a (inherit)
- has a (composition)
- subclasses, superclass, specialization, generalization

Good contracts inform both parties of what *should happen* and what both parties can expect, without actually having to implement any of the work just yet. Even just the structure, the expected shapes of responses from a RESTful API is a contract, because both sides (front end and backend devs) can do their work against a single agreement.

This keeps things lightweight, and for modelling, it's an incredibly versatile technique, especially when we're working with code that relies on infrastructure. In the *Mockist* version of TDD, we rely heavily on contracts to test behaviour using mocks (see 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - Rough Notes).

Eventually though, we must make good on the contract and implement it.

To implement a contract is to make a **concretion**. A real thing. An actual object that goes and does work.

In classical OO, the tool that we predominantly use to create objects is the class.

### Classes

Classes are blueprints for objects. All responsibilities for doing and knowing that we've assigned to a *role* and perhaps described within an *interface* (or some other contract technology), at last, it is in the class that we make those things real and concrete.

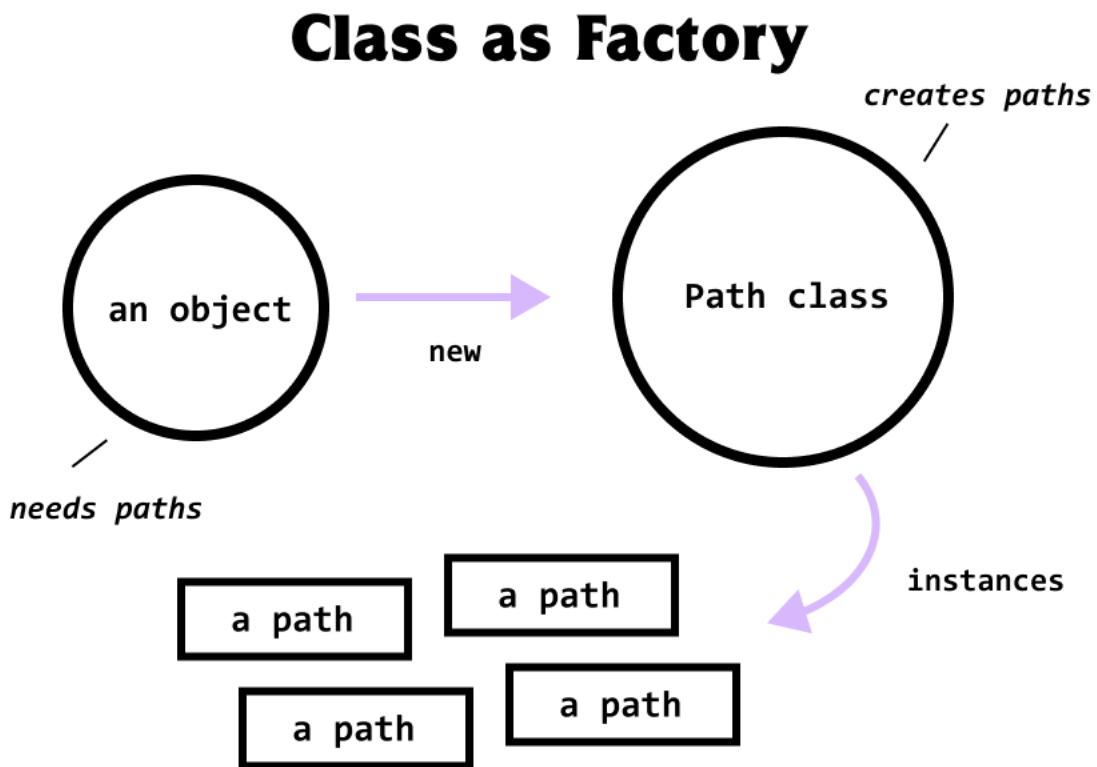
Most developers know this already. However, what most don't know is that classes have two roles:

1. As an **object factory** and
2. As a **collaborator** or independent provider, standing in for objects themselves.

### Classes as object factories

The first role that a class can play is that of a factory to create instances of objects. Here, we use the new keyword to create objects from classes.

The objects created share the same methods as the class, but now they're out in the wild and can interact with other dynamic objects.

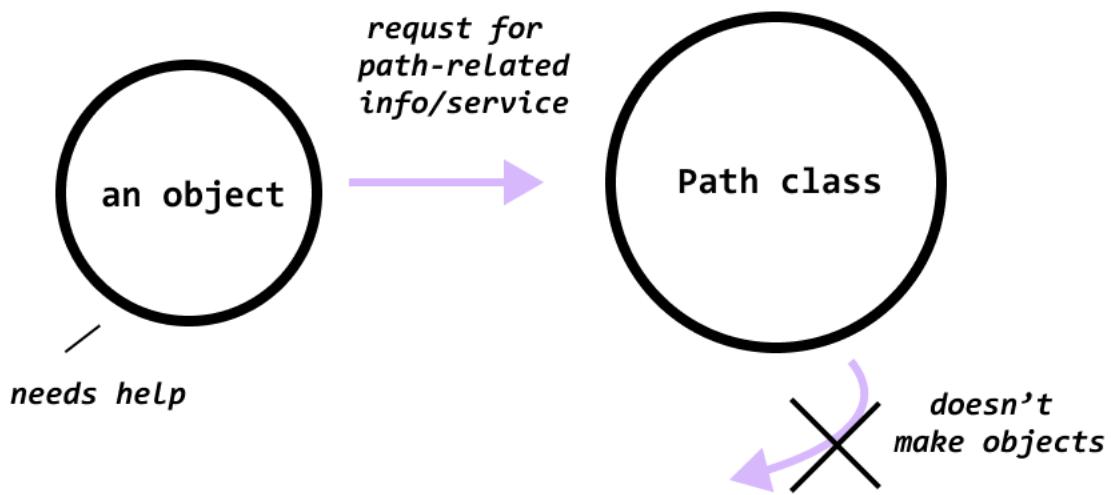


### Classes as collaborators

When we consider classes to be collaborators or independent providers, they shed the role of object factories.

- there only needs to be one of them
- utilities
- objects interact with them directly
- enumeration/etc

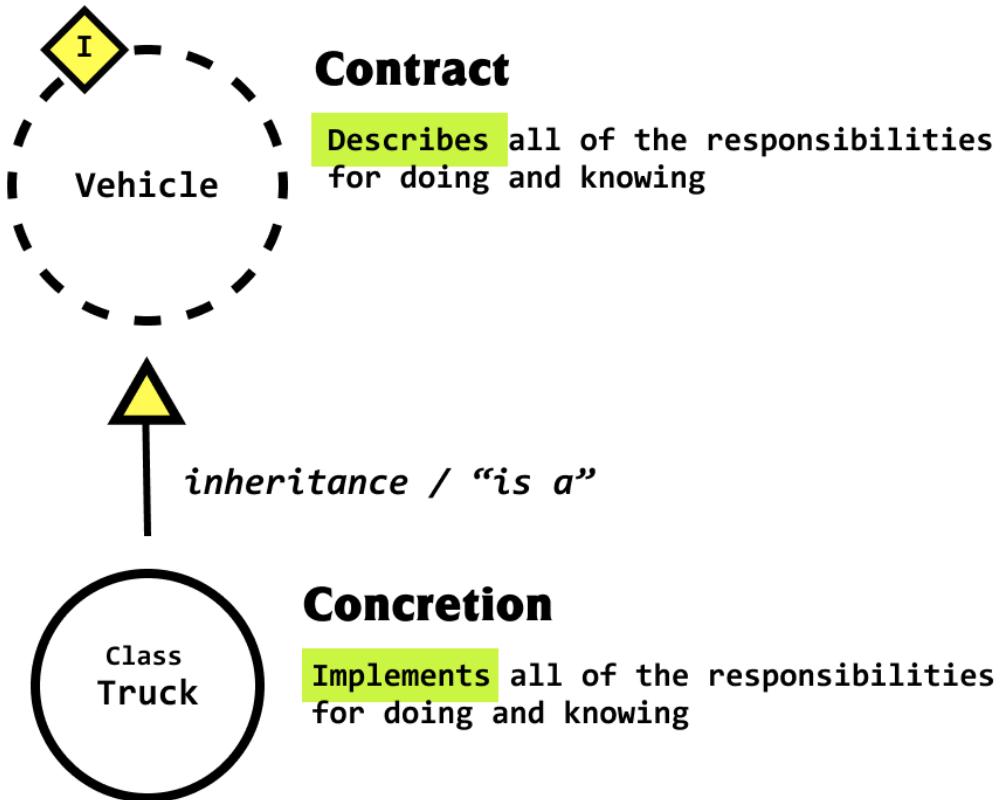
# Class as Collaborator



## Fulfilling a contract

*Fix this diagram here → it's not “is a”, it's “played by” — and mention that this is something borrowed from a topic called “Role-oriented programming”, but that's actually what we're doing here*

*If the vehicle thing really is an interface, then, yeah no — this is played by, slightly different. Mention that we call this “interface inheritance”*



- when a class implements an interface, we creates a concretion from a contract. This is an act of inheritance. we perform inheritance.
  - An instance *uses* another's responsibilities through collaboration. An instance *assumes* another's responsibilities through inheritance.
  - From an object-oriented *programming* point of view (the third of three parts of object design), **inheritance appears to be for code reuse**. How? You can use inheritance to prevent duplication by implementing an *abstract class* or *subclassed an already concrete class*. Neither of these techniques are what inheritance is actually about from a responsibility-driven design standpoint.
- \*\*Because you can use either **inheritance** or **composition** to perform reuse, popular practices advise us to opt for composition instead. However, from a responsibility-driven design point of view, **inheritance is not about reuse** is to opt for composition instead.
- Inheritance should not be confused with subtyping.[3][4] In some languages inheritance and subtyping agree,[a] whereas in others they differ; in general, subtyping establishes an *is-a* relationship, whereas inheritance only reuses implementation and establishes a syntactic relationship, not necessarily a semantic relationship (inheritance does not ensure behavioral subtyping). To distinguish these concepts, subtyping is sometimes referred to as *interface inheritance* (without acknowledging that the specialization of type

variables also induces a subtyping relation), whereas inheritance as defined here is known as *implementation inheritance* or *code inheritance*.<sup>[5]</sup> Still, inheritance is a commonly used mechanism for establishing subtype relationships.<sup>[6]</sup>

■ Inheritance is contrasted with object composition, where one object *contains* another object (or objects of one class contain objects of another class); see composition over inheritance. Composition implements a has-a relationship, in contrast to the is-a relationship of sub-typing.

```
interface Ine {  
    // methods here  
}  
  
// Implement the methods of Ine  
class Aelf implements Ine { ... }
```

- now, if we've done this right, from a responsibility-driven design point of view, we've filled in a role using an object — and that's great.
  - **ehhhhh, idk baout this** *This allows us to generically refer to interface in other parts of the codebase and leave the implementation details o*
- now, people tend to get antsy when we talk about inheritance, because you'll see multiple levels of inheritance hierarchies.
- this doesn't make sense with what we know of RDD; roles are filled in by objects, so we only ever inherit roles
- we don't inherit other objects → that is, we don't extend

```
class OneObject { ... }  
  
class Second extends OneObject { ... }
```

show a picture of what is OK compared to what's NOT ok — inheriting from a contract vs. extending another class.

**What is it that we need to show in order to get to object neighborhoods? Show how objects are created, how we implement contracts (inheritance), how we connect objects to each other (composition/delegation),**

## Composition

### Interfaces

Let's interface defines how a class should look like (as a bare minimum). Whether you implement this in a base class or in the lowest subclass doesn't matter.

- We said that the contract was the first two parts — the API and the preconditions & postconditions — that leaves the last part, the implementation.
- The implementation is said to be the **concretion**.
- Contracts are fulfilled once we **make them concrete** — we do so by implementing the interface

show a picture of a class implementing an interface

- this creates the relationship of a “is a” — now, the implemented class *is a* *e*
- The most common piece of contract technology is the interface.
- Objects implement interfaces, mandatorily implementing the methods and *hopefully* adhering to the preconditions and postconditions.
  - Designing explicit method signatures really explicit forces clients, developers implementing the classes, to *make sure* they handle these preconditions and post-condition situations.
- Now, do we always use an interface? When do we use an abstract class? When would a type suffice?
- I’ll partially answer the first now. No, we don’t always have to use an interface. We can model *roles* using *classes* right away instead. When’s the right time to do this?
  - That’s a part of a longer discussion we’ll leave for 35. Mapping Concepts & Designs to Object-Oriented Code - Rough Notes.
- But for now, the simplistic answer is to say to use an interface when we think we’re going to need multiple implementations of a particular *role*.

■ Mention that we’ll spend more time on mapping designs to code and some specific nuances about object-oriented programming and moreso in 35. Mapping Concepts & Designs to Object-Oriented Code - Rough Notes.

## Growing neighborhoods

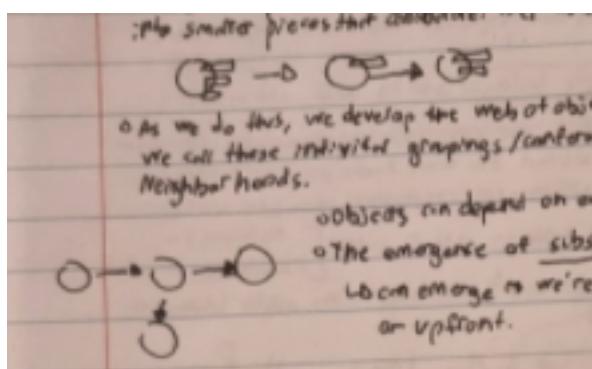
### Object neighbourhoods

- Everything else that was here

Mention that

- Alright so we have objects with roles, responsibilities, and collaborations nicely organized into object contracts. And when we implement a contract, we have a **concrete object** which is what actually performs behavior.
- With a mixture of domain, application (and infrastructure) specific objects, they’ll link together into neighborhoods to serve use cases.
- **Improving cohesion:** As we develop objects, you’ll find that often, roles can be split into smaller pieces that collaborate as well. This forces them to become more cohesive and focused. We appoint new objects to take some responsibilities off an object’s plate.

Show picture here



- As we do this, we watch the web of objects Alan Kay described emerge. We call these individual groupings/confederations of objects *Object Neighborhoods*.
  - \* Objects can depend on each other for specific behavior
  - \* If we zoom out and the capabilities get large enough, we see that **subsystems emerge**.
    - Subsystems can appear as we're designing upfront, or they can emerge later down the road as the system unfolds.
- Those subsystems interact with each other, but just like how if we zoom out on the map of our city and we see the state or province, we can zoom out on the neighborhood and see it as a component.

**This is just going to be about how as we begin to develop collaborations between related objects and move responsibilities *out* of objects and into other ones (why? → so that objects become more focused and cohesive → see 36. Better Objects with Object Calisthenics - Rough Notes for tips on how to do this), we start to see a formation of object neighbourhoods.**

*And then put a picture here to describe this phenomenon*

**What is an object neighbourhood? It really just is a federation of objects that work together to accomplish a feature or a some functionality.**

**Mention that this is what's great about OO — objects depend on each other for specific behaviour, but in combining those behaviours, what emerges is a more sophisticated behaviour overall.**

This is the emergence of *subsystems*, a smaller, functional part of solving the problem, that works

Then transition into the next section about using these subsystems.

## Components

Most of us are familiar with the idea of components, but *what is a component really?* In RDD, a component is essentially **an object neighborhood with a public API**.

It's something that encapsulates away all the internal details (and all the other neighboring objects that do the work behind the scenes) so that you can just use it.

React's Component is an example

*another example*

another example

**Show pictures of an object just chilling at front and there being a bunch of objects behind the scenes that handles much of the work, but is not known to the client.**

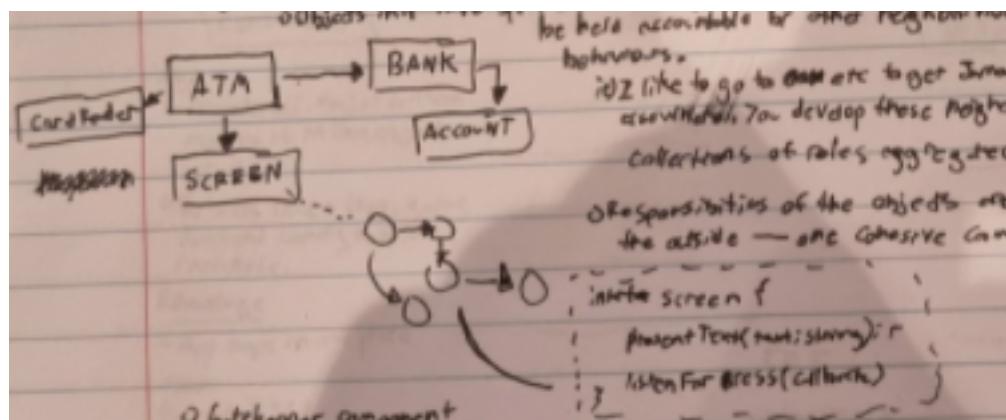
Gatekeeper component:

Think of it like *border patrol* or the *officers at the airport*, they'll let you in and

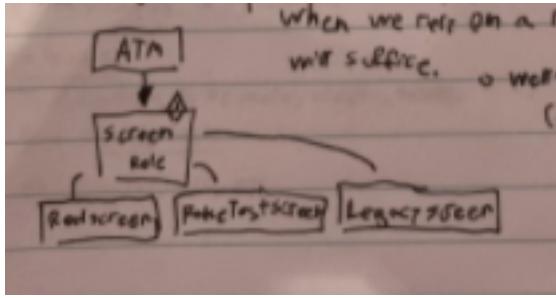
**Mention something about how components are pretty much the idea of subdomains in domain driven design. This is the full picture of software design, I'm tending to get at here in this way.**

And looks like this neighborhood here is devoted to the idea of handling the screen and then this one seems like it has to do with the bank. And perhaps this one is has to do with reading the card. And all of those neighborhoods need to work together. But at this level, having zoomed out, we can see that we've developed these big architectural components. So that's the idea of components.

- What is a component?
  - Components are any a confederation of objects (a neighbourhood) that work together to expose related functionality through a public API.
  - For example, in the context of an ATM application, we can think of it in larger-scale parts. There's the ATM itself, but what sorts of neighborhoods of objects will it need to interact with?
    - \* The Screen, the Cardreader, the Bank connection, the bank's connection to your Account.
    - \* Each of these are components because they encapsulate a lot of related functionality and expose publically *some* of that functionality — only what's necessary — to neighboring object neighborhoods.



- usually has a gatekeeper (interfacer)
  - components usually have a gatekeeper object which sits at the front like a guard, and translates messages to the correct objects within the neighborhood (**guess: what kind of stereotype is it probably?**)
  - perhaps there's a bouncer at the front
- React components, npm packages to things like AWS s3 are components too because they encapsulate functionality through a contract (the public API)
- Since they have a contract, they can be plugged into or swapped in/out for other components (be it testing, be it to swap a legacy one in for a new one, etc)



## Designing control between neighborhoods

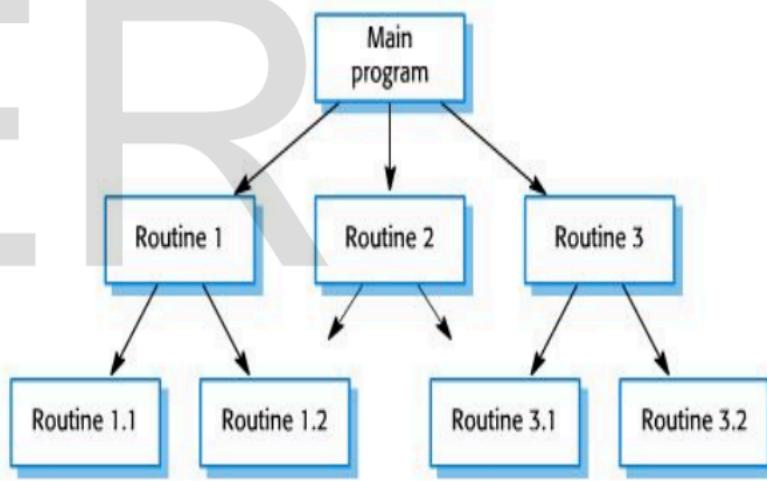
Mention that *so now we have these components that do their things in separate object neighborhoods, or they're these features, now what? How do we link them all together?* → the key to this is we must decide upon a control style. We can also talk of situations where we don't design components in isolation from each other, but we sort of just stack code on (centralized). Should probably show some code here.

### Control center

#### Control styles

- Let's say we give our program life. What object starts it up? What keeps it alive? Which objects are responsible for handling how our program flows?
  - Actually that's a good point. This tends to trip me up sometimes too.
  - It's a great question. So many of us are using frameworks or using libraries that handle startup for us and we just plug our code in. We need a better understanding of this phenomenon of control.
- The control center is said to be the objects that decide where controlling and coordinating happens. This means this is done by *Controllers* and *Coordinators*.

#### I. Centralized



Fig(3.1).Call Return Model

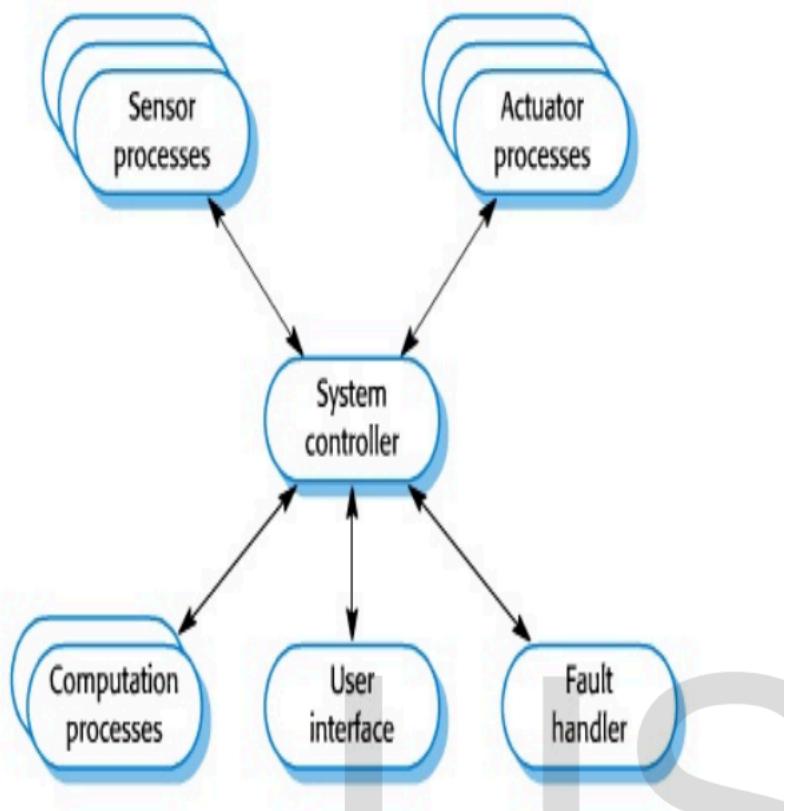


Fig (3.2). Manager Model

Manager model is applicable to concurrent systems. One system component controls the stopping, starting and co

ordination of other system processes. It can be implemented in sequential systems as a case system.

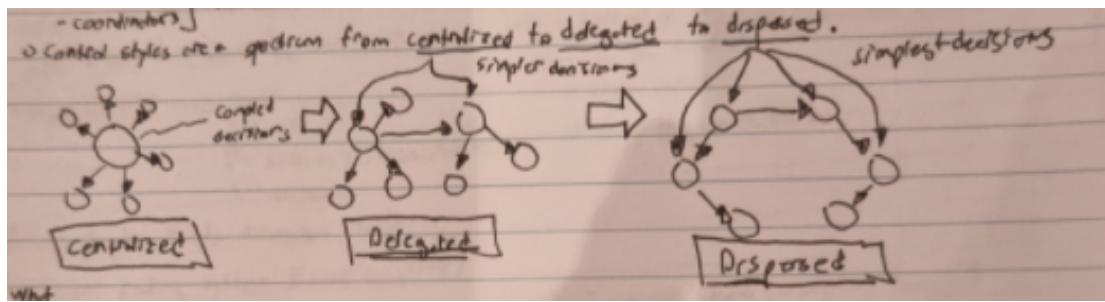
Figure 3.2 is an illustration of centralized management model for a concurrent system. It is often used in real time systems which do not have very tight constraints. The central controller manages the execution of a set of processes associated with sensors and actuators. The system controller process decides when processes should be started or stopped depending on system state variables. The controller usually loops continuously, polling sensors and other processes for events or state changes. For this reason, this model is called an event-loop model.

## **2. Delegated**

## **3. Dispersed**

- There are 3 variations of control styles: Centralized, Delegated, Dispersed

- Show a picture of each



- **Centralized:** All app logic lives in one place but that makes the control center complex, couples objects to each other inadvertently, and makes domain objects anemic. Good for small, simple tasks.
- **Delegated:** An event-driven sort of model which is easier to understand, change, and divide work between. The control center listens and waits for events or interrupts and passes execution off to neighbouring objects to perform specialized logic. This is the control style we use with the Clean, Hexagonal, or Ports and Adapters architecture in task based UIs (see Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS).
- **Dispersed:** There is no control center at all. Logic is completely split across individual objects — where each object is small and has as few dependencies as possible. There is no advantage to this approach and it should never be used.
- **Control styles are a spectrum.** A control style can be said to be more centralized or delegated than another.

### *Clustered Control Styles*

This control style is a variation of the centralized control style wherein control is factored among a group of objects whose actions are coordinated.[19] The main difference between a clustered and delegated control style is that in a clustered control style, the decision making objects are located within a control center whereas in a delegated control style they are mostly outside.[20]

*Remember to document this properly — when we use the State Pattern, we end up with a Clustered Control style.*

From RDD book, “The MessageBuilder must know a lot in order to handle a selection or timer event. Its correct responses are based on the state of the message being constructed as well as what has been already spoken. When an object seems burdened with complex decisions based on state, you can simplify its processing by distributing statespecific actions to other objects. The State pattern explicitly addresses moving decisions from an object into a number of smaller decision makers working directly on its behalf. If we adopt the State pattern, we’ll end up with a clustered control style. Each small decision maker will assume responsibility for responding to the events that the controller is handling given a particular state the controller is in, explaining the name State pattern.” — page 267

## Patterns

**Consider getting rid of ...**

## Architecture

**Consider getting rid of ... no...**

***Essentially, architecture is about NON-FUNCTIONAL REQUIREMENTS.***

***The only reason we use a layered architecture is to improve evaluation qualities.***

***The only reason we use a blackboard pattern is to improve execution qualities.***

***Know what qualities are required, know which patterns and styles get you there, and integrate***

In the first chapter of this book, we talked about *levels of design* and how design patterns are similar to architectural styles in the sense that if we expand them to the component level (from the class level), we find a lot of similarities.

Architectural styles are concerned mostly with:

1. **Component interaction:** How are components (or layers) allowed to interact?
2. **Control style:** How are decision-making responsibilities delegated *within* and *between* layers?

The whole thing that makes the layered architecture work is because we have made it very clear about what layers are allowed to talk to which layers. There is an explicit chain of command.

For example, in a layered architecture, with respect to component interaction, components within the same layer are allowed to talk to each other, but when it comes to reaching across layers, dependencies may only reach in one direction — from infrastructure to application to domain. Then with respect to control style, how are decision-making responsibilities delegated? We use the delegated control style and place the bulk of the logic within *Use Cases* — pools of logic which call upon lower level operations.

**Show a picture here to illustrate. Use the layered architecture and the blackboard pattern.**

It's through these two constraints that architectural styles like layered architectures, pipes-and-filters, and blackboard vary most.

If we wish to build testability, security, performance, maintainability, or portability into our applications, we can use combinations of architectural styles that support these characteristics.

■ We explore architectural styles in more detail within 8. Architectural Principles and use a layered architecture to build a web app in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS.

## Filling in the gaps

## Frameworks

[https://twitter.com/\\_developit/status/1537800286932193282](https://twitter.com/_developit/status/1537800286932193282)

<https://twitter.com/stemmlerjs/status/1537870333348401152>

Last design topic: frameworks. How do frameworks fit into design? This is a good question, since we don't always code everything from scratch.

First of all, let's get clear about what a framework is. **A framework is a collection of classes that we can tailor and extend to solve a particular problem; and most of the time, frameworks decide on the control style for you.**

That's right. Frameworks are typically responsible for booting up and running the code. The difference between a framework and a library is whether *we* get to define the control style or the collection of classes does. If we merely plug our own code in, it's a framework. If we still have to think about and design the way control flows throughout our application, it's a library.

Both frameworks and libraries are excellent for saving time and getting us productive faster, but be sure you're aware of the constraints on control style and if it will suit the requirements.

## Libraries/packages

Mention that when you're thinking of using a library or a framework (electron, etc, etc) you should ask the questions "what role does this thing play? what are the responsibilities owned by that role"? right? or you know what responsibilities you *need* (something for notifications, etc, etc) — then you have to ask yourself which components (from a library or framework) implement those roles.

You see, this way of thinking is going to serve you really well in this dance of coupling and cohesion.

## Exercises

- Stereotypes: tell readers to go and identify the types of stereotypes in their own projects or lack thereof — give examples
- Make readers list all the responsibilities for their capstone project and then assign them to roles (stereotype these roles). Then draw out how they collaborate? Make note of polymorphism (similar vs. general)
- OR
- Give them some design stories to start training them to think about responsibilities, do this exercise
  1. Video game
  2. Checkers, drag and drop, multiplayer
- Ah yes this is again related to the problem of awareness. Most devs struggle with designing these things because of a lack of clarity. Become crystal fucking clear about the responsibilities. Technical, app, domain, etc.
- the main capabilities for this chapter is gain the skill of awareness.

## Summary

- Summary attempt #1
  - The way of design (start talking about how we actually begin the process of designing objects) — this is just the title
  - Responsibilities, what they are, techniques to find them, and some design stories/examples of us actually finding them
    - \* *transition: after we've found responsibilities, we need to organize them into candidate roles*
  - Roles:
  - Stereotypes: *one thing that helps tremendously with translating into roles is object stereotypes* organizing responsibilities into roles using stereotypes
    - \* **How do we know what roles they should be?** → Object stereotypes
      - Describe what object stereotypes are
      - Describe what the different categories are
      - Show images for each of them
    - \* *transition: starting to have an idea of the types of roles, how do we actually make this a role — what do we do?*
  - Contracts vs. Concretions: *we have two options when we want to package together the responsibilities* → contracts or concretions
    - \* Contracts: *talk about what a contract is as an option for making this a reality*
      - why we need them, and why assigning responsibilities to roles IS DEFINING A CONTRACT, what the implications of contracts are
      - How to implement them (brief because we explore more in another chapter)
      - Interfaces, abstract classes, types
    - \* Concretions: *mention that we don't have to use contract technology (interfaces, abstract classes or types) and that we can use raw classes instead (concrete classes)*
      - Mention something brief about this (look to book)
      - We won't talk specifically about *when something* should be a contract vs. a concretion and when you should inherit from concretions/etc just yet. That's for mapping OO, I'd say.
  - Creating objects with classes
    - \* Classes and the fact that we create objects with them — consider putting this in the next section
  - Classes have two roles
    1. Object factory → to create instances of objects
    2. To act as an object itself (when only one is needed) — *static*
  - Collaborations and how they evolve into object neighborhoods
  - Then we talk about how to use control styles to cross , object neighborhoods
    - \* We also talk about needing control centers as a place to (what is a control center for? — it's where coordination and control happens
      - , and implications of each of these different types

## Missing

- Requirements → responsibilities → roles → (stereotypes) → collaborations → (peer collaborations) → neighborhoods/components → control styles

Really emphasize the importance of *contracts* and clean it up everywhere.

As I've covered in articles like "dependency inversion & injection", "the principles of OO", and sections in solidbook like "testing strategies" and "how to write object-oriented code (with tests)", contracts are **one of the**.

From →Summary

In this chapter, we:

- Learn how to think about design the RDD way using roles, responsibilities, collaborations, objects, and contracts
- Learn how to use the six object stereotypes to speed up the invention of well-defined objects
- Discuss the invention of domain and application-specific objects
- Learn about how object neighborhoods grow from small clusters to components with reusable public APIs
- Learn about control styles and how to intentionally choose a control style to control the flow of your application.
- Learn how to use design patterns to deal with emergent flexibility and maintainability challenges.
- Discuss how RDD can help us determine the appropriate architectural styles we need to meet system-level non-functional requirements.
- Learn how we can use object machinery and the ideas from RDD to speed up development time with frameworks, libraries, and custom code in an intentional way.
- According to Rebecca Wirfs-Brock, "object-oriented applications are composed of objects that come and go, assuming their roles and fulfilling their responsibilities".
  - Think of a job description. If roles, responsibilities, and collaborations represent the job description **\*\***(the *contract*), then objects represent the temporary consultants-for-hire — replaceable and substitutable. Each of which carries out the contract and adheres to the responsibilities outlined within it.
- In object design, domain objects represent concepts that domain experts, developers, and users can easily discover and discuss like User, MakePayment, and Email.
- Application (and infrastructural) objects, however, refer to objects **we must invent** to handle specific application functionality with respect to things like handling the user interface, talking to the outside world, or passing off control to other objects to do work. These are objects less trivially discovered; they require the intentional responsibility-driven design process.
- Object stereotypes make the object invention process easier. By knowing the six different object role stereotypes and their predefined responsibilities, we can rapidly recognize the shapes our object inventions should take. This also leads to familiar, more well-defined objects.

- As we invent and connect objects together, we come to create object neighborhoods — subsystems of objects working together. Since object oriented programs can grow quite large in size, we encapsulate the internal complexities of object neighborhoods by packaging them as components ready for public use. We then connect components together, setting up highways between these larger architectural pieces.
- Control styles have to do with how control flows between subsystems (components). Control styles are a spectrum between centralized, delegated and dispersed.
  - Centralized: Where a single object coordinates and controls the flow of the application in a call-return-like fashion
  - Delegated: Which represents an event-based, broadcast-like model, where we pass control to the appropriate subsystems upon receiving a message
  - Dispersed: Where there is no control style and we spread logic across the entire population of objects
- Though we may choose to use frameworks (and frameworks do not give us the ability to choose a control style), the modern object designer doesn't build everything from scratch. Object design is an act of filling in the gaps. We assign responsibilities to be handled by libraries, frameworks, and platforms and create object-inventions that links it all together.
- In the next chapter, we demonstrate the process of responsibility-driven design.

## Resources

- <https://docs.microsoft.com/en-us/archive/msdn-magazine/2008/august/patterns-in-practice-object-role-stereotypes>
- <http://www.wirfs-brock.com/blog/2021/09/13/design-and-reality/>
- <https://stackoverflow.com/questions/2399544/difference-between-inheritance-and-composition>
- Robert Greene, “The Laws of Human Nature”
- Mysteries of The Mind — <http://webhome.auburn.edu/~mitrege/ENGL2210/USNWR-mind.html#:text=According> to cognitive neuroscientists%2C we, goes beyond our conscious awareness.
- Object-oriented Programming in C# by Kurt Nørmark — <https://homes.cs.aau.dk/~normark/oop-09/pdf/all.pdf>
- [https://homes.cs.aau.dk/~normark/oop-csharp/html/notes/contracts\\_themes-contract-sect.html#contracts\\_contracts\\_title\\_1](https://homes.cs.aau.dk/~normark/oop-csharp/html/notes/contracts_themes-contract-sect.html#contracts_contracts_title_1)
- <https://www.ijser.org/researchpaper/Control-Models-in-Software-Engineering.pdf>
- <https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Architecture/ArchPatterns/CentralControl.html>
- [https://en.wikipedia.org/wiki/Inheritance\\_\(object-oriented\\_programming\)#Inheritance\\_vs\\_subtyping](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Inheritance_vs_subtyping)

- [https://en.wikipedia.org/wiki/Role-oriented\\_programming](https://en.wikipedia.org/wiki/Role-oriented_programming)
- <https://en.wikipedia.org/wiki/Subtyping>
- <https://stackoverflow.com/questions/2137201/composition-vs-delegation#:~:text=Composition> is about the relationships,(B refers to A.
- <https://stackoverflow.com/questions/21022012/difference-between-dependency-and-composition>
- “What Drives Design?” — Rebecca Wirfs Brock [2009] talk

## 34. The RDD Process By Example - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book.

■ A demonstration of how to use the responsibility-driven design method to design and build checkers game using React, mobx, and a drag-and-drop library. Starting from a design story, we identify themes, responsibilities, roles, and collaborations using the theory of object stereotypes and control styles.

You can find the game at @ <https://github.com/stemmlerjs/checkers>. Not complete. Many refactorings to be made to be more in alignment w/ the *Essentials* applied to *Frontend Architecture*.

But can help for reference.

### Introduction

#### Chapter goals

- Show the RDD process in depth
  - of how one goes from a *problem in words* to using design tools like CRC cards, and stereotyping objects, seeing how they can think about the design, and then ready themselves to start coding with TDD. I'll give them some problems they can work through and attempt. Show them how to **find objects, assign responsibilities, decide on collaborations, and get organized about a control style**. That last bit about control styles should translate nicely into architectural styles and patterns.
- Take readers through the process with tests and that chess game I built

“Responsibility-Driven Design is the most efficient path to designing great software”. — John Vlissides (IBM TJ Watson Research)

#### How to find objects

#### How to find responsibilities

#### How to identify collaborations

## **How do you test code? When does that happen? At which point in the process?**

- Imagine that you have a design candidate planned out with *doing* and *knowing*.
- Now imagine that you think “hey, the best way to test this is to write the tests first” which is good.
- But then you literally just translate each doing and knowing to a test.
  - At this point, that’s not BDD anymore.
- So when you’re converting a design candidate:
  1. **Choose only the doings**
  2. **For each doing, triangulate them with scenarios.**
- This is how you implement a design candidate. That’s the way to do it. OR you can just skip all the ceremony and just remember that BDD is a thing you should be doing.

## **If you’re not sure where to start**

- Start with the responsibilities and assign them to roles
- Program by wishful thinking / GOOS / acceptance test / assume things work/ outside in test → this can jump start your identification of the boundaries
  - This will force you to do double loop TDD, where you jump in a loop and then start testing other shit.
- Walking Skeleton e2e test to get all the main components connected, then go in to your acceptance/feature/application level tests.

## **If you get stuck**

I got stuck, so let’s think of everything I did to overcome this.

## **If it’s hard to test**

- Think back to the use cases — figure out what the actual use cases are, then set up the various scenarios using the Given-When-Then style tests.
  - Lots of techniques we can use here to drive the design.
  - This is the GOOS approach, to use mock objects to do shit. Its great because it actually does help.

## **Resources**

- <http://www.wirfs-brock.com/blog/2021/09/13/design-and-reality/>
- Introduction
  - Here, I’ll probably say something about some real-world experience I had where I had come across some cool shit?
    - \* Maybe talk about the robot that I built in university
  - Not sure just yet, but anyway, we’ll segue into the conversation that I’m going to show you how the responsibility-driven design process works in practice
- What I’ll build:
  - Here, I talk about what it is that I’ve tasked myself to build. It’s the checkers game.

- \* what checkers is
  - explain what checkers is
- \* why I chose it
  - Because most people know checkers, but it's got a lot of complexity to handle somehow. like what? → logic, board, movements, organizing front-end logic, how to do this utilizing a tool like React, but also how to do this in a testable way
    - *This should entice the reader, seeing the complexity here.*
  - There's an opportunity to learn a lot of different things by building this.

- **Analysis**

- *The RDD formula for finding and accessing candidates*
  - \* Write a brief design story and explain what is important about the app
  - \* Identify major themes that define the central concerns of the app
  - \* Search for objects that surround and support each theme. Draw on existing resources for inspiration: descriptions of your system's behaviour, architecture, performance, structure.
  - \* Check that these candidates represent key concepts or things that represent your software's view of the world outside its borders.
  - \* Look for candidates that represent additional mechanisms and machinery
  - \* Name, describe, and characterize each candidate
  - \* Organize candidates. Look for natural ways to divide your application into neighborhoods — clusters of objects that are working on a common problem.
  - \* Check for their appropriateness. Test whether they represent reasonable abstractions
  - \* Defend each candidate's reason for inclusion
  - \* When discovery slows, move on to modeling responsibilities and collaborations.
- **Start with looking for candidate roles and objects, figure out how they'll be realized as interfaces or classes later.**
- Step 1: Starting with a design story and themes
  - \* So let's imagine this is the analysis phase — draw that out
  - \* What is the first thing I did?
    - I drew out the rules for the game, the use cases, what we expect to see (pictures), and how it should behave.
    - In doing this, I identified a few different general themes
    - **Rules** → the game has a lot of rules around stopping, starting, jumping pieces, knowing when the game is over
    - **UI, Dragging and dropping** → this is going to be a UI app, not a console app, so I'll need something capable of handling drag and drop
    - **Console** → there's a console on the side of the screen that should render all of the moves that were made, it should say whose turn it is and maintain a log of all the moves that have happened.
  - \* **Non-functional**
    - **Testable** → I must be able to test the application's logic and rules while I'm developing it, and I'll also want to be able to test the UI compo-

nents themselves, but separately from the logic. I don't want to need to start the app to test the logic. I want to be able to do this from the console so that tests can run really fast.

- **Maintainable** → the app needs to be divided into component neighborhoods, modularized, easy to find the features/sections of code I'm looking for. things must be aptly named according to what we learned about in Part II: Humans & Code's 8. Naming things.
- **Flexible, extendable** → in case we want to improve upon this and put it on the internet as a multiplayer checkers game, the code must be written in a way where that's possible

- \* I'm going to start by modeling checkers concepts, rules, and things like that.
- \* So the key themes are:

- Game logic
- Game presentation / board
- Drag and drop / handling interaction
- Converting game actions into messages that can be shown in the logger/console on the screen at the same time

- \* More

- Show the drawing of the board
- So in summary, what I've done is
  - i. Design story
    - i. Rules
  - 2. Architecture drawing (design)
  - 3. Non-functional requirements
  - 4. UI drawing

- Step 2: Identify as many objects and roles as possible

- \* Candidates represent

- the use cases
- things affected by or connected to the application (other software, physical machinery, hardware, etc)
- info that flows through the software
- decision making, control, coordination activities
- structures and groups of objects
- representations of real-world things that the pp needs to know about

- \* Use these as *perspectives* on the *themes*.

- Step 3: Stereotype objects and roles to gain a better understanding of what they're for

- \*

## Show techniques for this

- **Design phase**

- Finding responsibilities

- \* **List out what I've done**

- I took a look at the use cases, the drawing, and the non-functional requirements to begin to find many requirements.

- Theme (drawing)
    - Draw the board (black and white squares)
    - Draw pieces
    - Draw pieces at position
    - Draw white piece
    - Draw black piece
    - Draw container
  - Game
    - Starting the game
    - Moving a piece
    - Preventing invalid moves
    - Drawing valid pieces to move on the board when a piece is being dragged
    - Handle what happens when a piece is dropped
    - Dragging animation
    - Managing know if the game is over
    - Managing knowing whose turn it is
    - Knowing when a player must continue their turn and keep jumping pieces
    - Knowing all of the valid moves that a particular piece is allowed to make
    - Knowing that regular pieces can't move backward, only forward
- \*

### List out other ways to find responsibilities (from the RDD book)

## 35. Mapping Concepts & Designs to Object-Oriented Code - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book. This hasn't been reviewed for correctness and quality.

■ The four principles of object-oriented programming (abstraction, inheritance, encapsulation, and polymorphism) give us the ability to perform a number of useful techniques that can be used to make code more testable, flexible, and maintainable. Here, we learn how to translate object-oriented designs into object-oriented code using these techniques.

When you make the transition from candidates to an implementation specification, you will create abstract and concrete classes that implement your objects' responsibilities. An abstract class provides a partial implementation. It leaves its subclasses with the obligation to implement specific responsibilities. A concrete class provides a complete implementation. You are also likely to specify interfaces for responsibilities that can be implemented by different classes. An interface specifies method signatures without specifying an implementation. It defines the vocabulary clients use to invoke responsibilities, regardless of their implementation.

Abstract and concrete classes are the building blocks we use to specify an implementation.

Declaring interfaces is one means to make it more flexible and extensible.

Implicit in the declaration of an interface is the design idea that a single role can be carried out by several types of objects, regardless of their implementation. **When you suspect that a role might be played by different kinds of objects, declare an interface.** Do so even if you intend to implement a number of classes belonging to the same inheritance hierarchy. This makes it clear that your abstract and concrete classes are just one possible implementation and that others may be declared in the future without being constrained to a specific ancestry. It makes clients who use objects that support this interface more flexible, too. They needn't know about specific classes in order to use objects that share a common role declared in an interface.

## Introduction

- How are we going to do this section? Well, so far, we've maneuvered all the way through all the essential RDD topics that we need to talk about. Now we need to see how they map to object-oriented code. We've seen how to create designs, but we need to see how those designs look in code. Real code. So what's going to be the best way to do that? Use a real example? Well, we're going to need real examples to demonstrate for the most part, but what's really more important to identify here is the **mapping** that happens between the two.
  - Design → **mapping** → code
  - The mapping tells us *when, why, how*
  - If we think back to Human-Centered Design, this is about essentially that entire picture where we have the *GULF OF EVALUATION*;
  - when should you use interfaces? when should you use abstract classes? when should you use static classes?
  - We need some answers.

### *Old introduction that used to be in the previous chapter instead*

Here's the problem. As a beginner, our initial focus is to merely make the code work; to do what we've been asked to do. In the front of our minds are the imperative concepts used to make the code whirl, twirl, spin, and flash. New languages, frameworks, tools, technologies emerge and we're constantly vetting newer ones to see what capabilities they afford. This is the first camp. We all start out here.

Eventually, we experience pain. The pain of recognizing that our code ain't so hot. Brittleness, inflexibility, and confusion (among many issues) ensues. If we're using an object-oriented programming language or a framework like Angular, it can feel like we want to give it up for good.

*"OOP is trash!"*, as \*\*I've heard exclaimed many times before.

The problem, however, is not with object oriented programming. The problem has to do with design. Namely, the lack of it.

Design exists within the space between the *what* and the *how*. We've touched on this before, but let's return to it from an object-oriented lenses.

The *what* is declarative, essential, high-level, and easily communicable, where the *how* is

imperative, accidental, low-level, and involves technical details.

**The beginner approach to design is to design purely with the how in the foreground, rarely considering the high-level.**

On the front-end, this means doing things *the React way*, on the backend, the *Ruby on Rails way*, or — most relevant to this chapter’s discussion — **when using an object-oriented programming language, to first think in terms of interfaces, classes, and inheritance hierarchies.**

**This is too long; we can probably cut some of this. perhaps start here**

Such pre-described ways of solving problems have nothing to do with the essential complexity, and pull us deeper into the accidental complexity. As we’ve explored in Part III: Phronesis, we know that better approach is to be use case (or feature) driven; to uncover the essential complexity first, and focus on the accidental later.

In this chapter, we dive deeper into RDD (Responsibility-Driven Design): a better way to build to OO software. Building on top of our use case driven approach to software design, we’ll learn how the primary RDD concepts can be used to translate use cases into high-level designs of interacting objects.

But to realize our designs, we’ll still need to map the high-level to the low-level; to turn our designs into real OO code. To learn how this works is another goal of this chapter.

**This is really wordy.**

There are a lot of reasons why object technology is key for design, but in my opinion, one of the main ones is that OO is fundamentally designed to allow us to separate the how from the why. The principle responsible for this is called **abstraction**. And it’s foundational to all the others.

In mapping our high-level responsibility-driven designs to low-level OO code, we finally begin to understand the true utility of abstraction, inheritance, polymorphism, encapsulation and other traditionally misused object-oriented design techniques. No more silly designs, flying by the seat of your pants, five-level inheritance hierarchies, and not understanding when to use generics.

Ready? Let’s get into it.

## Chapter goals

- Learning outcomes:
  - You’ll learn conceptually, how object design works using rules, responsibilities and collaborations
  - You’ll learn other essential object machinery such as object stereotypes & control styles
  - Why abstraction is key principle of object-oriented programming
  - You know how to map your design candidates to code
    - \* You know when to use abstractions instead of classes
    - \* You know how to use abstractions to share common behavior (inheritance)

- \* You know how to use abstractions to enable dynamic behavior (polymorphism)
- \* You know how to use abstractions to simplify integration and usage (encapsulation)
- composition
- generics, generalization, dependencies,
- Other questions?
  - When to use abstract classes / interfaces?
  - How to map roles into objects
  - How to map responsibilities into objects
  - How to map collaborations into objects
  - What are the main principles of object-oriented programming?

## The Four Principles of Object-Oriented Programming

### 33. Demystifying The Fundamental Concepts of Classic OO

#### **Principle #1 — Abstraction**

- What is it?
  - It's the ability to define the implementation separately from the concretion.
  - What do you mean? Well, normally, in
- Why does this matter?
  - Because we
- How does it work?
  - **Mapping roles**
    - \* You want to map roles? Let's say you have a design already sketched out of roles, you have two options for how to map your designs to code because in oo languages, there are two things that roles can map to: abstractions or concretions.
  - **Concretions (classes)**
    - \* Explain what the class is
      - It has 2 responsibilities
    - \* Show that picture of the class
  - **Abstractions (interfaces, abstract classes, types)**

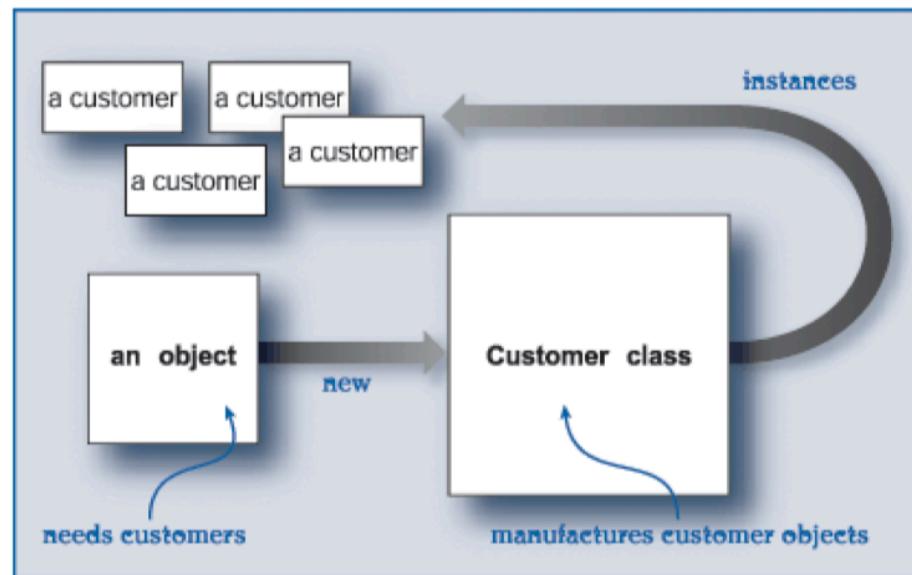
■ **Rule:** Think about the role design first, then consider if it needs to be a concretion or an abstraction.

#### **Mapping roles**

#### **Classes**

- **I need to talk about classes first because they're the initial building block really**
- talk about what they are, how they're the atomic block, show how easy it is to create them, but explain that they're not actually the most natural building block to start with. explain that there are two types of classes: concretions and abstractions.

- Next section → explain that abstractions, they act as contracts, and define the behaviors of a *potential concretion* — that is, they define the *role*.
- what's an analogy for a class?
  - blueprint still probably. I put this there earlier.
- what are the two responsibilities of a class?
  1. it's a factory for manufacturing objects (**we create objects from classes**)
    1. it's the place that we define implement the roles, responsibilities, collaborations by defining the connections/referernces to other objectzs
    1. we create objects with references and they in turn create other ones and create this network of objects just out there in the field living, dying, getting re-created, and so on until the first object (the one that will represent starting up the program) decides it's time to end.
  2. to create an object, we use the new keyword here to do this
  3. when we do this, we create *instances* of objects
  4. **picture of an object asking for help to create an instance**



5. one object is asking another one, hey can you create this object please?
  1. it's always like this, even when it doesn't appear to be. sometimes, in the public execution or global scope, it may not appear like that's happening. but it actually is.
    - maybe put an example of the **window** object asking another API or something to do something. this would make sense to JavaScript developers
    - Instances:

- \* The shape and behavior encoded in the class is going to dictate the shape and behavior of the object instances that get created from the class
- Constructors
  - \* Using the `new` keyword and the factory pattern
- **[Previous]: Creating objects from classes**

A class is like a blueprint for an object. We don't send blueprints out into the wild; we use blueprints to create deployable inventions. So consider objects to be the *real thing*. If objects are to real physical book copies that you can pick up and hold onto, classes are to the manuscripts from which they're created.

```
class Person {} // an empty class
```

This distinction is paramount. When we think of behavior, we have to remember that classes aren't the things that behave — that'd be objects.

**Show a picture here of classes and objects — blueprints & the real thing.**

Classes are merely the concept used to define the *runtime behavior* of an object. Classes also define how objects — *instances of a class* — come into existence.

```
class Person {
    // We don't have to put anything here yet
}

let personOne = new Person();
let personTwo = new Person();

console.log(personOne === personTwo); // False. Instances of classes
// are unique.
```

Above, we create a class — and the interesting thing to note is that we don't need to define it with anything other than a name; we can go ahead and create instances of it using the `new` keyword.

If we *did* want to define some sort behavior that an instance of a class (object) could perform, we can do so using methods.

```
class Person {
    // We don't have to put anything here yet

    public function doSomething (): void {
        // Put behavior here
    }
}
```

When we call methods, we call them against instances of the class — not the class itself.

```

class Person {
    // We don't have to put anything here yet

    public doSomething (): void {
        // Put behavior here
    }
}

Person.doSomething(); // not valid because we are referring directly
                     // to the class, not an object instance

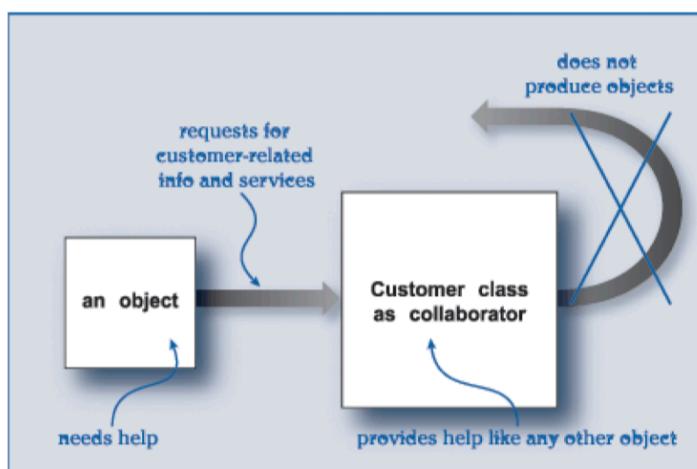
let person = new Person();
person.doSomething(); // valid since real behavior happens
                     // against the object

```

2. They act as *independent providers*

1. This means that they sometimes serve other objects in their neighborhood.
  - i. and they don't really do anything other than that.
2. Ah → so this gives meaning to the idea of static methods, sometimes we have classes with all static methods like Util classes (which arguably can sometimes not be the best design, however), this is the mental model you should have for this.

**Figure 1-6. A class can also act as an object when the application needs only one of its kind.**



– Classes vs. class instances

Here's the *first* place where things can get confusing. There exists a way to locate methods (behavior) on the class *itself* rather than the class *instance*. Such methods are called *static methods*.

```

class Person {

    public static doSomething (): void {
        // Put behavior that belongs to the class here
    }
}

Person.doSomething(); // valid since method exists on the
                     // class itself, not an instance

let person = new Person();
person.doSomething(); // not valid since the method exists on the
                     // class, not a class instance

```

When we declare a method as `static`, it can only be called by going through the class itself. Never an object instance.

Dang, doesn't this break our nice conceptual model of *objects being the things that interact with each other, not classes*? Yeah, it kinda does.

**Show a picture here of static behavior happening on the class itself — maybe make it easier to depict.**

If we were to just use static methods, taking it to the extreme, we could proceed to have a web of *classes* interacting with each other; and you may not know why that's bad yet, but it is.

We'll get into concrete rules, but you generally want to limit your usage of *static* methods. While there are some very valid reasons for using them, it is possible to usually *forgo* them. While I'll demonstrate a particularly useful example momentarily, understand that we should be wary of static usage. It fundamentally breaks our conceptual model of OO.

- [Previous content section] **Constructor & static factory method**

Every class has a special type of method (whether we define it or not) and that's the *constructor*.

Every time a client uses the `new` keyword, the constructor is called before the object is created.

```

class Person {
    constructor () {
        // Set up the object
    }
}

let person = new Person();

```

The constructor gives us ample opportunity to perform object set-up logic — requesting that certain arguments be passed in. If we don't get the correct arguments, we can

throw an exception — refusing to create the object if something isn't right.

This is one way to enforce object creation logic.

```
class Person {  
    constructor (name: string) {  
        if (!name) throw new Error('Name not provided');  
        // ... continue setup  
    }  
}  
  
let person = new Person(); // "Error: Name not provided"
```

### Static factory method

There is a cleaner way to define object-creation logic; as we know from 12. Errors and exceptions, throwing exceptions isn't the best thing to do. Also, as of the time of writing this, when using the `new` keyword, there is no way for a client to deduce that it may be thrown an exception instead of the object. Astute developers will recognize that can lead to a lot of unnecessary defensive coding capable of cluttering your codebase.

Here is one scenario where I think *static* methods are absolutely indispensable: to **define object creation**, never to perform object behavior.

**Show a picture with the static method on a class saying “for defining object-creation only”**

The pattern is called a *static factory method*.

```
class Person {  
    constructor (name: string) {  
        // Set up the object  
    }  
  
    public static create (name: string): Person | Nothing {  
        if (!name) return '' as Nothing;  
        return new Person(name);  
    }  
}  
  
let personOrNothing: Person | Nothing = Person  
    .create('Khalil') // the result is either the object or a  
                    // nothing type
```

Notice that we call the `create()` method on the `Person` class (statically) and not on an instance of the class? Wait, *how would you create an instance of a class if you need an instance of the class to create it anyway?* This sort of chicken-and-the-egg problem is why I believe we sometimes need static methods — to help us break free of this conundrum. Let us make that a rule.

**Rule — Only use static methods for object creation logic:** You may encounter

times where it is easier to put logic in static methods, but as a general rule of thumb, try not to. OO is supposed to be about how objects interact. By giving static methods a distinct policy for valid usage, we reduce the possibility of abusing it. Logic can usually be placed cohesively within a class instead.

However, do take note of the fact that object creation using the `new` keyword is still possible from outside of the class.

```
let person = new Person('name'); // We can still do this.
```

Geez. Now there are *two ways* to create objects. Two separate code paths to do the same thing. That'll confuse people, won't it? Yeah. Hmm, is there a way to force clients to use the *static factory method* and simultaneously restrict the use of the `new` keyword at the same time? Yes, there is.

## Scope

So first, let us investigate why it is that we can call `new` on the `Person` object.

Well, that's because the default *scope* (visibility) of a constructor is `public`. There are others as well like `private` and `protected`.

To limit the potential surface area for mistakes, we constrain the two possible ways to create an object using this class to one. We do that by making the constructor `private`. With a `private` constructor, clients are forced to use the static factory method to create objects.

```
class Person {  
    private constructor (name: string) { // Make the constructor private  
    }  
  
    public static create (name: string): Person | Nothing {  
        if (!name) return '' as Nothing;  
        return new Person(name);  
    }  
}  
  
let person = new Person(); // You can't do this anymore!  
                        // The constructor can only be called  
                        // from within the class. Not outside.
```

Great! Constrain the options. Make things simpler to use.

## Interfaces & abstract classes

after having talked about classes, then I can talk about interfaces and what they're about

- The idea of a **role** can be mapped to an *abstraction*
  - a role is — in itself — an abstract concept. it's not a real tangible concept.
  - you can't *create* an **animal** or a **mail carrier** really — there's some *specificity* required there.

- some roles express that abstraction, but when they don't, this is when they can be nicely mapped to something in object-oriented programming languages
- What is an abstraction really?
  - An abstraction is anything non-concrete. It:
    - \* Can be used to express the blueprint for an object but it cannot be instantiated
    - \* You cannot use the new keyword
- How do you create abstractions in OO?
  - Two ways to create abstractions:
    - \* Interfaces
    - \* Abstract classes

**Type systems as an abstraction tool:** We should also note that we can use types to implement abstractions if we use a language with a type system (such as TypeScript).

- Well then what's the point? Why would you want this in the first place if you can't
  - Abstraction opens up the door to **encapsulation, inheritance, and polymorphism.**

## Mapping responsibilities

Responsibility for knowing & doing

## Constructors

**yeah, we're going to talk about how to set up objects**

## State

Pull from the other chapter. There's stuff on state.

## Methods & messages

## Generalization (generics)

## Mapping collaborations

## Composition

## Dependencies

- Certain objects need other objects to perform their job — like *Use Cases*, as we've seen in 21. Understanding a Story .
- There are 3 different ways to get access to a dependency
  - Instantiate itself
  - Get from a known location (service locator or inject directly from file)

- Inject upon construction (or through a method)
- **Show an example of each**

### **Technique 1: Instantiate**

```
import { Logger } from '.../../shared/logging/logger';

class Foo {
    public someMethod (): void {
        logger = new Logger(); // Instantiate a logger when needed
        logger.debug(...);
    }
}
```

### **Technique 2: Use a known location**

```
// Pull an already created instance of a logger in from a
// module
import { logger } from '.../../shared'

class Foo {
    public someMethod (): void {
        logger.debug(...);
    }
}
```

### **Technique 3: Inject through constructor (or method)**

#### *Method*

- Perhaps the most trivial example of this is one where we pass an object in through a method parameter
- Not great, it clutters the parameter list — if we needed to do this everytime we needed to utilize a dependency, the parameter lists for all of our methods would be tarnished by dependencies.

```
class Foo {

    public someMethod (logger: Logger): void {
        logger.debug(...)
    }
}
```

There's a better way, and this is the ideal way most of the time, at least for dealing with *infrastructure dependencies*. That is to inject the dependency via the constructor.

```
class Foo {
    private logger: Logger;

    // An instance of Logger is passed in as an argument
    // to the constructor
```

```
constructor (logger: Logger) {  
    this.logger = logger;  
}  
  
public someMethod (): void {  
    this.logger.debug(...)  
}  
}
```

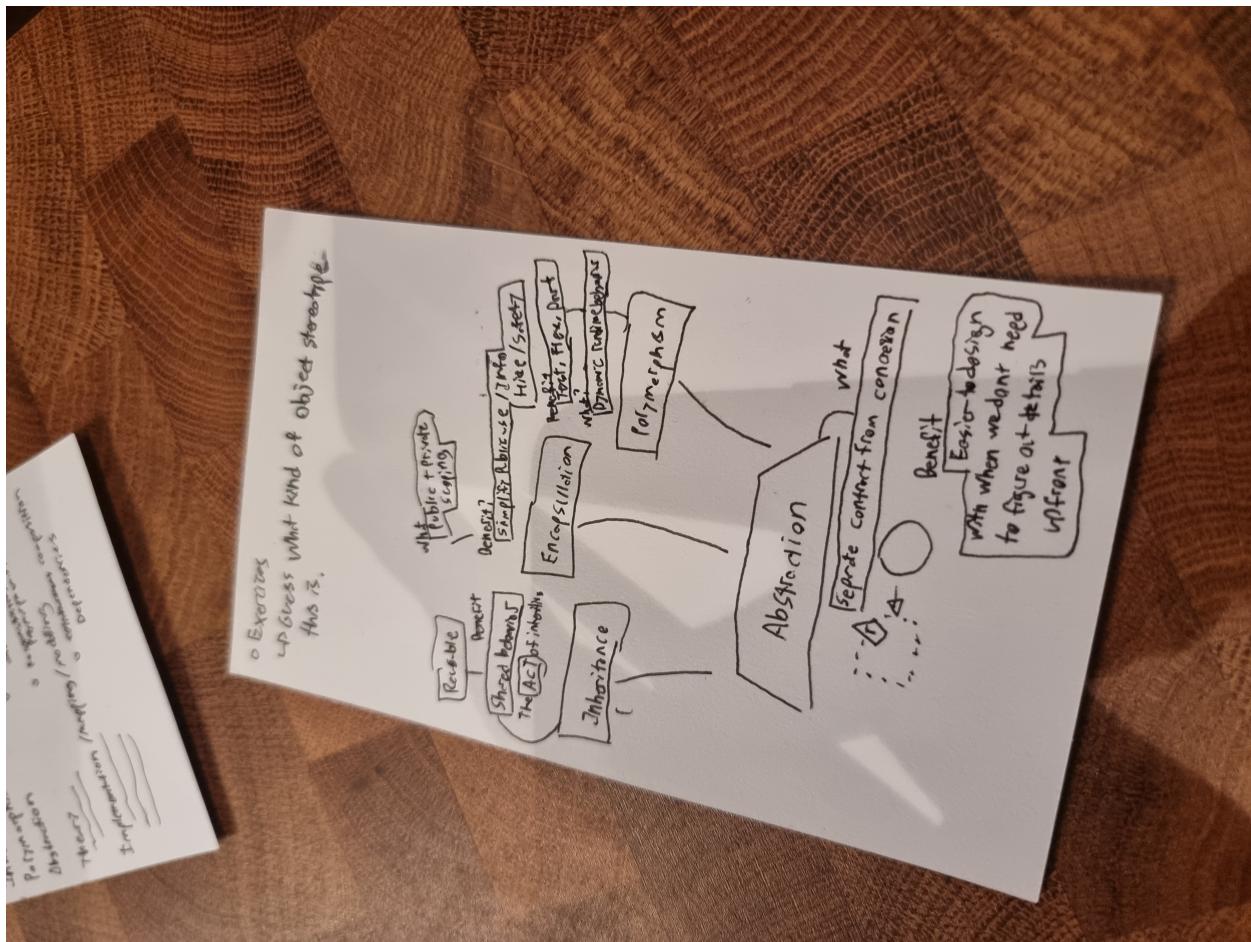
This contractualizes all of the dependency references that the object needs to operate at runtime, whereas with method injection, we'd likely run into issues later trying to locate or creating one-off instances of dependencies each time a client wishes to call a particular method.

Message passing (calling methods) is like interacting with a public API — we want to make it as easy to use as possible.

## Object-oriented programming principles

- These are the principles of object-oriented programming — not design. The principles of object-oriented design are
  - Be use-case driven — start with the functional and non-functional requirements
  - *I actually laid this out already in the previous chapter.*
-

## Why is it important to know this stuff?



### Abstraction

**Exercise here**

### Inheritance

**Exercise here**

### Encapsulation

**Exercise here**

### Polymorphism

**Exercise here**

**Force developers to do you know? Also, think about Promise; what would the promise class look like? Anyway, that's what's happening here.**

"Object-oriented applications are composed of objects that come and go, assuming their roles and fulfilling their responsibilities."

This notion of *roles*, *responsibilities* (and *collaborations*) is entirely around the idea of *roles* and *responsibilities*, it — giving us the ability to create applications from them; because as the original inventor of RDD, Rebecca Wirs-Brock says:

**This quote makes me think of someone working a night-shift. Think about a guard at an underground parking lot just chilling there. What are his responsibilities? To let people in, to check to see if etc, etc, etc.**

**He's not there all night, so when his shift is over, he's swapped out for a new employee to come in and carry out the same responsibilities.**

**On this small scale, we're seeing an example of polymorphism.**

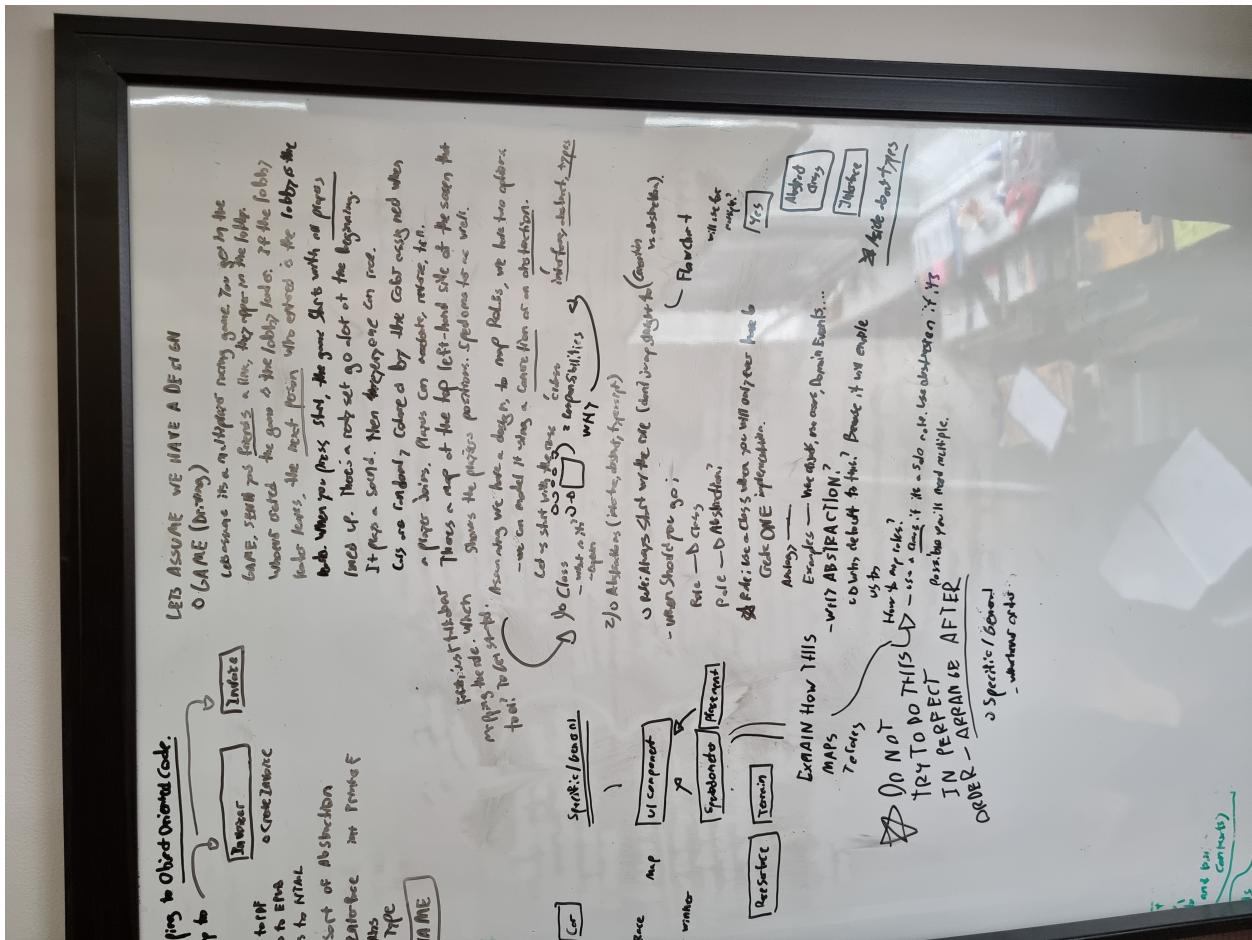
**Maybe employeeTwo does his job a little differently — maybe he's been there for 10 years so he knows everyone that comes in and out and he doesn't need to use a database to know who's allowed in and who isn't. The responsibility is the same — because it's a part of the role — but the behavior is different. This is the essence of polymorphism.**

## Summary

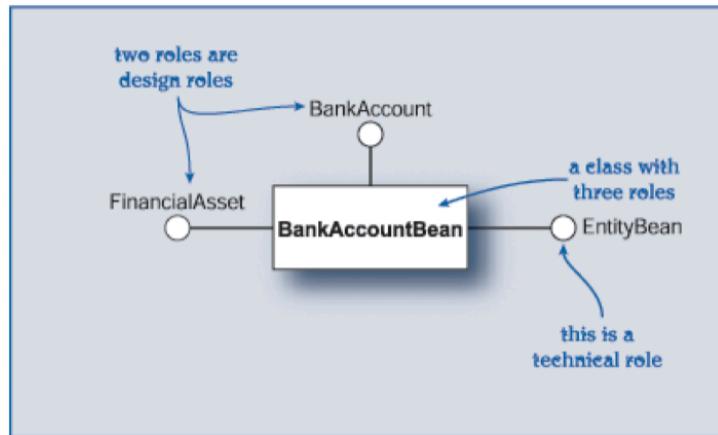
- **The Object design concepts from RDD** must come first because they describe the way we can think about designing with objects. They give us a repeatable design method to think in terms of Alan Kay's object machinery model.
- **Why use object machinery?** Because not only do we understand and communicate effectively with the concept of objects, but object machinery provides us with the adequate design capabilities to solve the problems and do so in a way that developers can work with. Essentially, it makes *design* possible. Our priorities for design are to solve our numbers problems but also to do so in ways devs can work with.
- **What are those design capabilities?** They are the four principles of object-oriented programming: Abstraction, Inheritance, Encapsulation, and Polymorphism.
- **Abstraction** → Separate contract from concretion; focus on the why and how separately; declarative before imperative.
  - **Abstraction** is the ability to separate concrete from abstract; to develop the *what* in the abstract and develop the *how* in the concrete; to do both separately allows for a variety of design techniques
- **Inheritance** → Share behavior (composition is a form of inheritance)
  - **Inheritance** is the act of either implementing an abstraction — creating a concretion (preferred) or subclassing an existing concretion (not preferred)
- **Encapsulation** → Simplify developer experience with thin, understandable, readable public APIs; safeguard object integrity
  - **Encapsulation** is the ability to make certain responsibilities (for doing and knowing) public and certain ones private within a concretion
- **Polymorphism** → Enable dynamic behavior & testability, flexibility, portability & more.
  - **Polymorphism** plays on the fact that since we can separate abstract from concrete, we can *refer* to abstractions rather than concretions in our classes and in our algorithms. And since there is a one to many relationship between abstractions and concretions (a single abstraction may have  $N$  number concre-

tions), that means that if we refer to an abstraction in our production code, then we have the option to provide any valid implementation (concretion) of the abstraction at runtime. This allows for pluggable, dynamic behavior. It's the foundation for portability, flexibility, and testability.

- To map to roles, we use **fill in the blanks here**
- To map to responsibilities, we use **fill in the blanks here**
- To map collaborations, we use **fill in the blanks here**



- Roles
- Responsibilities
  - Classes can have more than one role

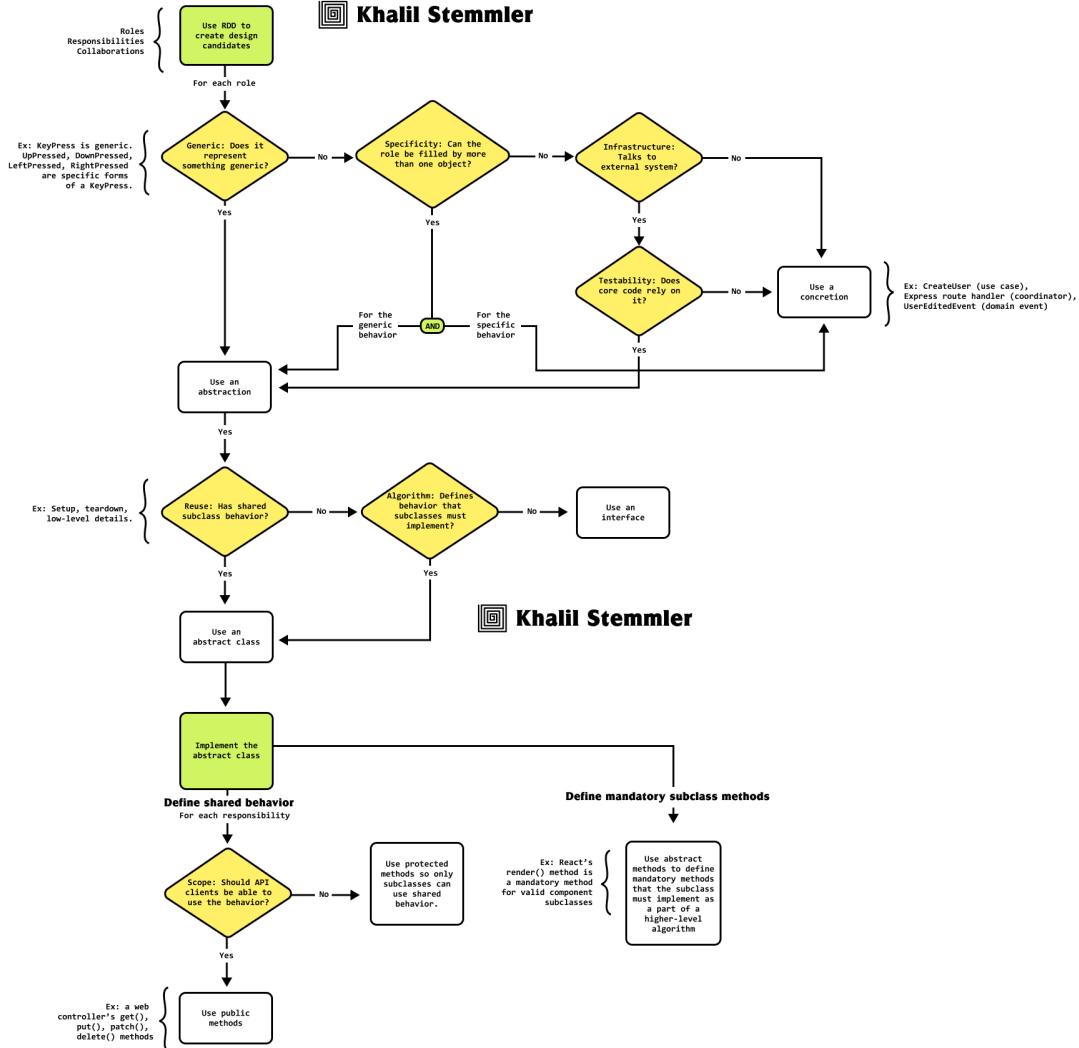


To sharpen your role modeling skills, go to your favorite class libraries and reverse-engineer some interfaces and classes into one or more roles, each with clusters of responsibilities.

- Collaborations
- Abstraction
  - Is-a
  - Genericism & specificity
  - Algorithmic (high-level)
  - Inversion (testability, portability, flexibility)
- Then explain the rest, but don't need to carve out specific sections for them.

## Mapping Object-Oriented Designs to Abstractions & Concretions

Khalil Stemmler



### 33. Demystifying The Fundamental Concepts of Classic OO

Yeah, we need to talk about abstraction, encapsulation, inheritance (and composition) and polymorphism here first before we get into the Responsibility-Driven Design method

What I'm attempting to do here is to undo much of the nonsense you've been taught about OO.

Dog-Animal examples are terrible because that's not real life. Of course, if you

## 36. Better Objects with Object Calisthenics - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book. This hasn't been reviewed for correctness and quality.

■ In design, less is more. Object calisthenics are a set of rules which can be used as a shortcut to Simple Design. Use object calisthenics to simplify your designs during the refactor phase of the TDD process.

### Chapter goals

In this chapter, we:

- **Todo:** walk through all of the things that we've learned thus far — study them.

### Design

Just because you *can* doesn't mean you should.

- What are some of the problems we currently face?
  - We haven't really talked about design yet — we've just done methodical work.
    - \* focused on cleaning up duplication
    - \* discussed mechanical transformations for mostly algorithmic problems (TPP)
  - In object-oriented design, we often need to solve problems that are more about structure and organization than they are about algorithms.
    - \* To do this, we need to make effective design decisions using the concepts explored in 32. A Refresher on OO Basics.
    - \* But how?
    - \* Let's consult the steps we've taken so far:
- The first step was to understand the conceptual model of object-oriented programming. We did this in 23. Programming Paradigms.
- The second was to cultivate a workflow in which it is *safe* to make design decisions. With TDD, this is possible. We learned about the 28. Test-Driven Development Workflow and the refactor step that exists for us to take chances; and if it doesn't work, we can revert back to the last commit.
- Third, in the last chapter, we made ourselves aware of all of the tools in the toolbox and what they can be used to do.
- Now what?
  - Now, we exercise the art of subtraction.
- In this chapter, we constrain some of the possible set of things that we can do.
  - Given the tools, there are a lot of wonky things we can do
    - \* For example → we know that we probably shouldn't be using static methods liberally — why? because objects (instances) are meant to talk to each other, and the classes are just the *blueprint* in a way. Two *blueprints* shouldn't be talking to each other.
    - That's just one example, but there are *tons*.
    - here, we start with a few hard, fast rules that you **will find** challenging at first

to follow, but will work wonders for your designs in terms of balancing coupling and cohesion.

## Object calisthenics

- What is it?
  - Object calisthenics is about cutting back on all of the things you can do. It's a small list of rules that, if followed, leads us to more maintainable object-oriented code.
- Where did it come from?
  - They were programming exercises and original 9 rules. They were invented by Jeff Bay in The ThoughtWorks Anthology.
- What does it mean?
  - Why calisthenics? Calisthenics this is a type of exercise that you can do by just solely using your body weight. this style of working out keeps things extremely simple. you're not fussing with tools, weights, machines, and so on — you're just using what you currently already have.

## The rules

The rules are as follows

1. Only one level of indentation per method
2. Don't use the ELSE keyword
3. Wrap all primitives and strings
4. First class collections
5. One dot per line
6. Don't abbreviate
7. Keep all entities small
8. No classes with more than two instance variables
9. No getters, setters, or properties

### I — Only one level of indentation per method

- What's the problem?
  - We've all seen methods with layers of conditionals, nested for-loops, and so on.
  - Not only is this code hard to read, but it tends to clutter methods, making them responsible for too much behavior.
- Here's an example of a method with three levels of indentation.

```
let groupsData = [
  [{ name: 'Bill' }, { name: 'Sam' }],
  [{ name: 'Susan' }, { name: 'Heather' }, { name: 'Jason' }]
]

class FriendGroups {
  printAllGroups (groups) {
    // 0
    for (let i = 0; i < groups.length; i++) {
```

```

        let groupString = "";
        // 1
        for (let j = 0; j < groups[i].length; j++) {
            // 2
            if (j !== groups[i].length - 1) {
                groupString = groupString + groups[i][j].name + ", "
            } else {
                groupString = groupString + groups[i][j].name;
            }
        }
        console.log(groupMessage);
    }
}

let fg = new FriendGroups();
fg.printAllGroups(groupsData);    // Bill, Sam
                                // Susan, Heather, Jason

```

## What to do?

The key idea of this rule is to reduce the amount of conditional and iteration trees by breaking the nested functionality out into its own methods.

```

let groupsData = [
    [{ name: 'Bill' }, { name: 'Sam' }],
    [{ name: 'Susan' }, { name: 'Heather' }, { name: 'Jason' }]
]

// Start
class FriendGroups {
    private isLastFriendInGroup(index, group) {
        return index === group.length - 1
    }

    private createGroupString (group) {
        let groupString = "";

        // 0
        for (let j = 0; j < group.length; j++) {
            // 1
            if (!this.isLastFriendInGroup(j, group)) {
                groupString = groupString + group[j].name + ", ";
            } else {
                groupString = groupString + group[j].name
            }
        }
    }
}

```

```

        return groupString;
    }

    public printAllGroups (groups) {
        // 0
        for (let i = 0; i < groups.length; i++) {
            // 1
            let group = groups[i];
            console.log(this.createGroupString(group));
        }
    }
}

let fg = new FriendGroups();
fg.printAllGroups(groupsData);    // Bill, Sam
                                // Susan, Heather, Jason

```

This applies to conditionals as well! If you can inline or simplify nested conditionals, do so.

```

// Example #1 - Not ideal
if (conditionOne) {
    if (conditionTwo) {
        ...
    }
}

// Example #2 - Good (one level)
if (conditionOne && conditionTwo) {
    ...
}

// Example #3 - Good (one level)
let hasConditionOneAndTwo = conditionOne && conditionTwo;
if (hasConditionOneAndTwo) {
    ...
}

```

**■ Refactorings:** There are a number of different refactoring techniques like *Extract Method*, *Decompose Conditional*, and *Consolidate Conditional Expression* that formalize how to do just this. You don't need to know the names of the refactorings. Just keep note of the indentation-level. We discuss different refactoring techniques in 37. Refactoring - Rough Notes.

## Benefits

It can be hard to figure out the correct amount of functionality that one method should have. This rule gives us a good heuristic. If you practice it, your code should have the following benefits:

- Your methods should be more focused. Instead of doing several things, they'll be focused on doing one thing particularly well.
- Smaller methods are easier to understand and reuse.
- For public methods with high-level steps involved, they become more readable.

**From an RDD perspective: (methods) This also brings awareness to something else — your public methods are your responsibilities for doing and knowing while your private methods are the substeps of your private methods**

## 2 — Don't use the ELSE keyword

- Most of the time we see the Else keyword, it's not fun to have to deal with. The reason is again because it creates another branch for us to mentally compute.

```
// Example with nested, nested if else statements
if (condition) {
    if (condition) {

        } else {

    }
} else {
    if (condition) {

        } else {

    }
}
```

- Draw a tree here with two branches and two branches coming out of it

### What to do?

- What should we do?
  - Return early. Fail fast.

```
if (!condition) return;
if (!nextCondition) return;
if (!anotherCondition) {
    // Sometimes you'll need to do things here within the block
    return;
}
```

```
public login(username: string, password: string): void {
    if (userService.isValid(username, password)) {
        redirect("homepage");
    } else {
        showError("error", "Bad credentials");
        redirect("login");
```

```
}
```

There are two things you can do: take an **optimistic** or **defensive** approach.

**The optimistic approach:** Remove the `else` and rely on the early solution. The code should still work exactly the same but without creating an additional branch.

### **Use a different example for this**

```
public login(username: string, password: string): void {  
    // Success case returns early  
    if (userService.isValid(username, password)) {  
        return redirect("homepage");  
    }  
  
    showError("error", "Bad credentials");  
    redirect("login");  
}
```

**The defensive approach:** Implement *failure branches* early in the method. This is better. It helps us think through all of the things that could go wrong and put them up closer towards the top of the method.

```
public login(username: string, password: string): void {  
    // Failure case(s) return early  
    if (!userService.isValid(username, password)) {  
        showError("error", "Bad credentials");  
        return redirect("login");  
    }  
  
    if (!someOtherFailureCondition) {  
        // Return early as well  
    }  
  
    return redirect("homepage");  
}
```

## **Benefits**

- 

**Promotes a main execution lane with few special cases.** • Suggests polymorphism to handle complex conditional cases, making the code more explicit (for example, using the State Pattern). • Try using the Null Object pattern to express that a result has no value.

Also, it is worth mentioning that Object Oriented Programming gives us powerful features, such as **polymorphism**. Last but not least, the **Null Object**, **State** and **Strategy** patterns may help you as well!

For instance, instead of using `if/else` to determine an action based on a status (e.g. RUNNING, WAITING, etc.), prefer the State pattern as it is used to encapsulate varying behavior for the same routine based on an object's state object:

[https://en.wikipedia.org/wiki/State\\_pattern](https://en.wikipedia.org/wiki/State_pattern) example — check it out

- why?
  - **No conditional trees.** Instead, we've got a sequential list of conditions to evaluate. This is a lot easier to hold in the brain than a tree of conditionals.

## From an RDD perspective:

### 3 — Wrap all primitives and strings

- Explain what the problem is
  - If we're practicing Domain-Driven Design, a string or number is never just a string or number; it always represents something else, like a Name, Money, or a Task. Yet, we often use primitives like this in our solutions anyway.
- Show an example

```
// Show an example where we calculate something based on the
```
- Explain the solution
  - Instead, to avoid this *Primitive Obsession* code smell, this rule dictates that we **encapsulate** the primitive to its own class. This is critical to Domain-Driven Design since we use Value Objects to properly encapsulate primitive state and behavior together.
- Show the improved example

```
// Show a value object class here (money?)
```
- What to do:
  - Always wrap primitives that you pass around or that have behavior (this makes it a domain object)
  - No primitive return types — (number, boolean, string, etc)
  - No primitives in method arguments (except constructors)
  - Primitives are only allowed as private instance variables (class members)
- Benefits
  - Primitives turn into meaningful and explicit types which express the domain
  - Better encapsulation of state and related behavior circumvents the Anemic Domain Model anti-pattern and creates a single place for changes related to the underlying primitive to belong.

### 4 — First class collections

- Explain what the problem is

- Just as we said that we should encapsulate primitives, we encapsulate a **collection (or a list)** of primitives as well.
- Show an example of the problem
  - There's a lot of times we'll write behavior against collections of primitives — this will happen all over the place, and often, it gets repeated — how about the blog filtering logic that I use for headers and shit?

```
// Write some code that shows them all together
let filteredElements = elements
  .filter((e) => e.something);

let filteredElements = elements.onlyFilteredElements();

// Prefer this over that
let element: Element[];
let elements: Elements;
```

- Explaining the solution
  - We're treating collections of primitives just as if they're primitives as well. The idea is to encapsulate any behavior we may need to perform against a collection of them.
    - \* This can get very specific.
      - Domain-specific behavior: Perhaps show an HTML example or something here.
      - Filtering behavior
- Show the improved example

### The concrete collection:

### The stable collection abstraction:

```
abstract class Collection<T> {
  protected items: T[];

  constructor (items: T[]) {
    this.items = items;
  }
}

class Task {
  private description: string;
  private type: TaskType;

  constructor (description: string, type: string) {
    this.type = type;
  }
}
```

```

class Tasks extends Collection<Task> {
    constructor (items: Task[]) {
        super(items ? items : []);
    }

    // Specific behaviour
    getTasksByType(type: TaskType): Task[] {
        return this.items.getType() === type;
    }
}

```

- What to do (rules)
  - If a class contains a list (a collection), it should contain no other class member variables. Make that class a *collection* class.
  - You may have to do this a few times — in which case, feel free to refactor to an abstract class that you can call upon.
- Benefits
  - The same benefits of encapsulating primitives — behavior has a proper home
  - We improve cohesion and reduce decoupling by removing all of the filtering, searching, and collection logic that tends to find ad-hoc find itself spread across a codebase. It's a better separation of concerns.

## 5 — One dot per line

- What is it?
  - Essentially, don't method chain.
  - When you do: Person.Hand.Wave() you're coupling the ...
    - \* Instead, just do Person.greet().
- Why?
  - It seems like it's not a big deal, but we're breaking the essence of OO's conceptual model.
  - We're increasing coupling between classes by doing this.
    - \* How? → Assume that the class *calling* only knows what it knows about the class it's sending a message to based on that method's *public interface*. Nothing else.
    - The class calling shouldn't know the internal details of the class it's talking to.
    - It's a violation of The Law of Demeter and the fact that “you should only talk to neighbours”.
- What to do instead?
  - Don't method chain.
  - This rule doesn't necessarily apply when we're using the *Builder pattern* or other APIs which are specifically designed for this kind of shit like
- Why is this good?
  - Reduces coupling between classes
  - Keeps API usage very simple and declarative — don't expect clients to need to

method chain, so design APIs return exactly what is necessary

## 6 — Don't abbreviate

- What
  - We've already discussed this in depth in 8. Naming things, but it's one of the original object calisthenic rules.
  - Show an example
  - Why does this happen anyway? Writing the same name over and over usually happens when we're missing a concept or some abstraction.
  - If you find yourself needing to do this, it probably means something is wrong and needs to be diagnosed.
- Why?
  - Abbreviations can be confusing.
  - If we have to abbreviate more than once, that's duplication. We know why that's bad. We can refactor it out by creating the correct abstraction.
- Benefits

## 7 — Keep all entities small

For this rule, we

- Explain what the problem is
  - Without an attempt to decouple concerns, classes can blow up into large many-responsibility files that are long, hard to read, and hard to maintain.
- Explain the solution
  - The idea here is to impose some constraints on packaging, class sizes, and methods. Here are some rules which will influence your design:
    - \* No more than 10 files per package\*
    - \* No more than 50-100 lines per class
    - \* No more than 5-10 lines per method
    - \* No more than 2 arguments per method

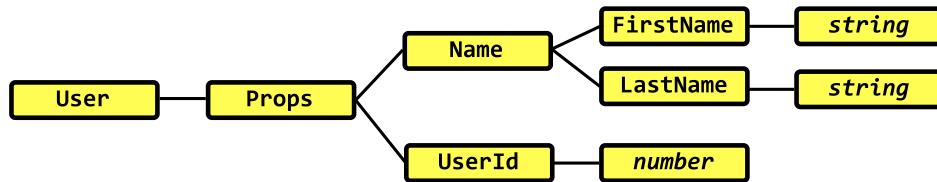
\*Ex: in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, we use *Use Case* folders containing use cases, Gherkins feature files, response objects and so more.

- Benefits
  - By making classes smaller, we tunnel-vision them; they remain focused on performing something specific.
  - By making classes smaller, we also make them easier to understand and reuse.
  - By colocating together related classes, we create a cohesive, well-defined package/namespace of classes (see 6. Organizing things).

## 8 — No classes with more than two instance variables

- Explain what the problem is

- There's a tendency to just continue to add more and more instance variables to a class over time. When we do this, we run the risk **lowering cohesion** and **reducing encapsulation**.
- This may be the hardest object calisthenic rule to implement.
- Explain the solution
  - Lets idealize two different types of classes: actuators and orchestrators.
    - \* **Orchestrators do what? → coordinators & controllers**
    - \* **Actuators do what? → domain-layer stuff, all the actual work**
  - **The actuator class:** This type of class is purely responsible for managing something that represents *state* — a single instance variable. → **Actually, actuators are pretty much all of your domain layer classes and services that do the work. These are anything that aren't orchestrators (coordinators & controllers).**
    - \* For example, take a look at the relationship structure of a User class.



- \* The User class is responsible for both the handling the behavior of any user behavior but also for encapsulating the User state — managing the way it's accessed and mutated.
- \* Notice that for this rule to work, we rely on Rule 3: Wrap All Primitives and Strings.
- **The orchestrator class:** The other type of class coordinates the interaction between classes. The best example of these classes are your *Controller* and *Use Case* classes. They contain little to no state other than what's passed between their collaborators. Admittedly, this is probably the hardest object calisthenic rule to follow because as we know from 16. Learning the Domain, there are often more than one or two collaborators for a use case. Therefore, that's one of the very few times I'll break that rule. However, if I'm doing something more design-based I don't always follow it. Why? While it's applicable in a lot of situations, it isn't always. If the essential complexity of the use case dictates the use of a particular collaborator (or three), then I'll have to inject those. However, should you need to add additional instance variables, ask yourself if they can be consolidated into a single cohesive class or object. Hey, I said it was hard!

- Benefits
  - Yet again, anytime we make our classes more focused, we strive to improve the cohesion of our classes.

- We encapsulate away details that could live in a more focused class.

## 9 — No getters, setters or properties

- Explain what the problem is
  - The last rule is about encapsulation.
  - In OO, when we create class properties, most languages default to making them public.
    - \* This is easily fixed by making them private.

- Show an example

```
class Game {
    private score: number;

    constructor () {
        this.score = 0;
    }

    ...
}
```

- This is good, but what often happens is that developers create getter/setter methods for every single property regardless of it makes sense.

```
class Game {
    private score: number;

    constructor () {
        this.score = 0;
    }

    public setScore(score: number): void {
        this.score = score;
    }

    public getScore(): number {
        return score;
    }
}

// Setting the score from outside the class
game.setScore(game.getScore() + ENEMY_DESTROYED_POINTS);
```

- In this example, we're deciding how to update the score *outside* of the scope of the class. The problem here is a lack of encapsulation.
- The **Tell, Don't Ask** design principle dictates that we should never be telling out classes *how* to execute the functionality that they own. It's apparently that the logic

for how to update the score of the Game is something that should be known to the Game object, not to other classes outside of it.

- This leads to duplication (since clients have to implement the score logic)
  - And it leads to stateful bugs (since there are no invariants around how score can change, if class A depends on the statefulness and it changes because of something class B did, class A may not work with the Game class since class A shouldn't and cannot know everything about how class B interacts with Game.)
- Ultimately, clients using the API of the Game class shouldn't have to know specifically how to update score.
  - Instead, focus on **telling** the Game to do things. Let classes out the *how* themselves.

```
type Action = 'Enemy_Destroyed' | 'Life_Lost' | 'Acquired_Coin'

class Game {
    private score: number;

    constructor () {
        this.score = 0;
    }

    public updateScoreByAction(action: Action): void {
        let delta = 0 ;

        switch (action) {
            case 'Enemy_Destroyed':
                delta = 3;
                break;
            case 'Life_Lost':
                delta = -5;
            break;
            case 'Acquired_Coin':
                delta = 1;
                break;
            default:
        }

        this.score += delta;
    }

    public getScore(): number {
        return score;
    }
}

// Correct amount of encapsulation
game.updateScoreByAction('Enemy_Destroyed');
```

- The `getScore` method is OK, but make a habit of only exposing setters for properties that are actually necessary to query for outside of the score of the class.
- Setters, however — basic, default setters. Do not allow them.
- And make all properties private by default.
- Benefits
  - Better encapsulation: Prevent a lot of state-related bugs this way.
  - More succinct, useful client APIs (methods)

## Exercises

### Summary

### Moving forward

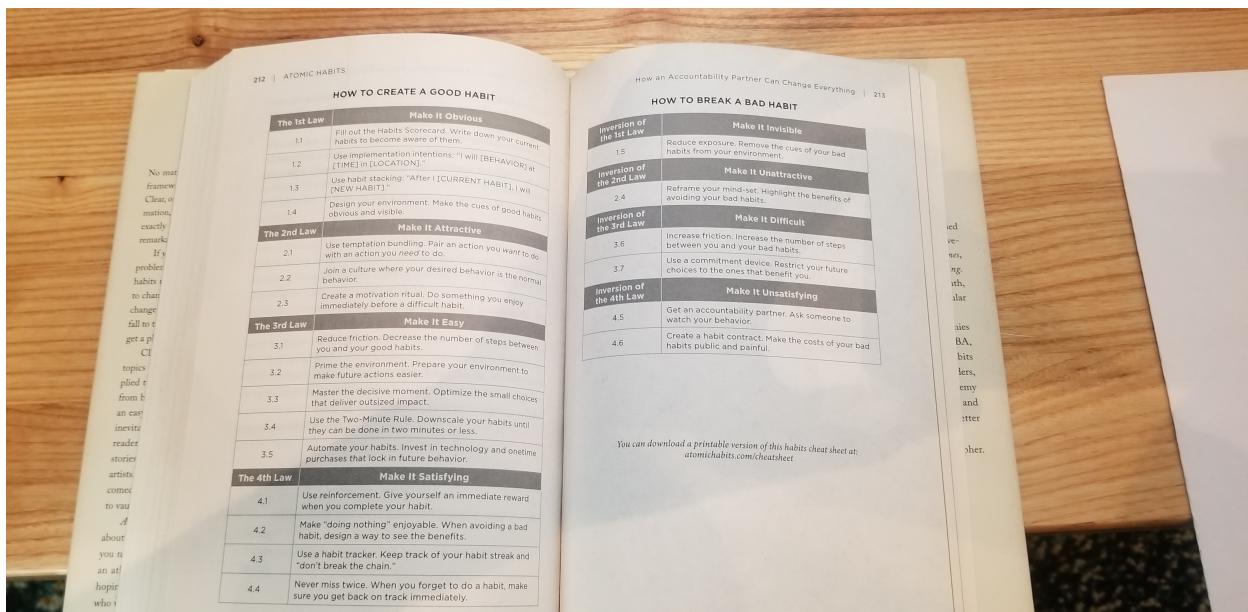
let's

### Next objective

#### *Object calisthenics*

We have now covered most of the rules. In the next chapters we will be covering higher level concepts. Make sure you can apply TDD effectively, including:

- Red, green, refactor.
- Writing the test starting from the assertion.
- Using transformation priority premise to keep your code simple.
- Using object calisthenics to help you with design. You should feel ready to walk now. :)



## References

### Articles

- Object Calisthenics by William Durand

## 37. Refactoring - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book. This hasn't been reviewed for correctness and quality.

■ Refactoring is a way to improve the design of existing code without changing the behavior. Not something to be treated as a seasonal activity, it should be performed constantly and most importantly — safely — within the context of a TDD workflow. Using tests, the RDD philosophy, and shorthand rules to inform our design flaws, we can improve design while optimizing code for simplicity and maintainability.

### Chapter goals

In this chapter, we learn:

- About the goals of refactoring and the various types of refactorings that can be made
- Refactoring principles
- The main categories of refactorings
- The most common refactorings you'll likely need to use and how to perform them

### About refactoring

- ■ What is refactoring?
  - As you should know by now, refactoring is an act (verb) that we perform in the middle of the TDD loop. To refactor is to restructure code by **applying a series of refactorings**. Each of those **refactorings** changes the structure, however — it does not change the observable behavior of the code.
  - So that means that refactoring is both a verb (to refactor) and a noun (a refactoring) where the noun is an individual change that we've made to the code that doesn't change the behavior.

### What refactoring is not

- ■ What *isn't* refactoring?
  - We should also be very clear on the fact that refactoring *is not* something that is done once every couple of months or after a large number of stories have been completed. Refactoring isn't something we do way later on in the future once we realize our designs are pretty bad. Refactoring is *how we prevent bad design*.
  - Refactoring existing, non-tested code — Can we refactor if we haven't done TDD? What's the first step?
    - \* No. If we haven't written any tests for our code yet, we can't safely refactor.

### When should we refactor?

- ■ When to refactor
  - When we break duplication + object calisthenic rules

## Types of refactoring operations

- In refactoring, you will perform five different operations to classes, methods, functions, variables, and so on.
  - Rename – Change the name of classes, methods, variables....
  - Extract – Extract a class (or methods or variables...), creating a new abstraction.
  - Inline – The inverse of extract – inline a method (or variable), deconstructing an abstraction.
  - Move – Move a class (or methods or variables...) to some other place in the codebase.
  - Safe delete – Delete code and its usages in the code base. — **Make it my own**

## Refactoring principles

### Humans first, design second

**First, the intent, let's get that right — you should intend to make code simpler for people to read. Design is secondary**

- 1 - Prioritize humans (readability)
  - And when I say that we want to improve the design, that's true — yes. We want to apply everything we've learned up until this point including:
    - \* The Rule of Three
    - \* Object Calisthenics
  - These two things can objectively tell us when we've broken a rule and we need to fix our designs with a refactoring, yes. And that's great, we need some level of that as a baseline for design.
  - However, you should **always prioritize humans before design**. This means prioritizing for readability, maintainability, and discoverability. Think back to the developer use cases from Part II: Humans & Code. If someone were to look at this, would they be able to figure out how it works? Do they know where the entry-point is?
    - \* Thinking back to our North Start of design — Simple Design, refactoring within the context TDD will give us the ability to fix coupling and cohesion problems, yes, but most importantly, it will enable us to **have as minimal elements as possible, and consistently re-name things to be more expressive and sensible as we move things around, invent, and tear down abstractions**.
  - Why do we prioritize humans instead of design?
    - \* 80% of the improvement in refactoring is going to come from improving how easy your code is to understand. If humans don't understand it and can't change it, very little we do to make our designs technical correct or impressive really matters in the long run.

### Stay in the green

- 2 - Stay in the green
  - Don't slip up and get clumsy. If you break a test (or two, or three) while refactoring, tread very lightly. We don't want to get too far ahead with a new design

in the red that we're banking on working, only to figure out much later that we can't get our existing tests to pass.

- Stay in the green by making very small refactorings. As you'll learn momentarily, there are a number of different refactorings, and they can all be performed in small, atomic steps.

## Use the method template

- How to refactor methods? Method template

## A good method template

- We should generally follow a template (check for data, etc, etc,etc)
  - \*

## precondition checks

- Check for null arguments

Some languages allow a client to pass null as an argument even if the type of a parameter has been explicitly declared. So in the example in listing 1.15, the provided argument for bar may be null, even though it's typed as Bar. Trying to call doSomething() on bar would then cause a NullPointerException to be thrown. This is why you always have to check for null, or preferably, let a compiler or static analyzer warn you against potential NullPointerExceptions. The fictional programming language used in this book by default does not allow null to be passed as an argument. In examples where we want to allow it, we explicitly have to declare it using a question mark (?) after the type declaration of a method parameter. This also works for property types and return types:

```
class Foo { private string? foo; private function someOtherMethod(Bar? bar): Baz? { // ... } }
```

- There are good, clean ways to do this

- the Guard class (you can check all kinds of things)

- Use this instead of if statements

- Replace primitive with object refactoring

```
final class EmailAddress
{
    private string emailAddress;
    public function __construct(string emailAddress)
    {
        Assertion.email(emailAddress);
        this.emailAddress = emailAddress;
    }
}
public function sendConfirmationEmail(
    EmailAddress emailAddress
): void {
```

```
// no need to validate emailAddress anymore  
}
```

- \* Failure scenarios
  - Do these earliest on
- \* happy path
  - do this last — most of the time, our code is just about dealing with the failure paths
- \* postcondition checks
  - safety checks — nice to have
- \* return void or return the type
  - Only query methods return an object
  - Command methods return nothing (void)

## Categories of refactorings

- Composing methods

### **■ Explain what this category of refactorings is all about**

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand—and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

**■ Examples:** show hands-on examples of one or two refactorings in this section — preferably ones from the following section

- Inline + extract methods/variables
- Moving features between objects

### **■ Explain what this category of refactorings is all about**

Even if you have distributed functionality among different classes in a less-than-perfect way, there's still hope.

These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

- Organizing data

### **■ Explain what this category of refactorings is all about**

These refactoring techniques help with data handling, replacing primitives with rich class functionality.

Another important result is untangling of class associations, which makes classes more portable and reusable.

Other refactorings include:

- Self Encapsulate Field — If the problem is that you have direct access to private fields in a class, the solution is to getters and setters for the

Replace Data Value with Object

Change Value to Reference

Change Reference to Value

Replace Array with Object

Duplicate Observed Data

Change Unidirectional Association to Bidirectional

Change Bidirectional Association to Unidirectional

Replace Magic Number with Symbolic Constant

Encapsulate Field

Encapsulate Collection

Replace Type Code with Class

Replace Type Code with Subclasses

Replace Type Code with State/Strategy

Replace Subclass with Fields

- Simplifying conditional expressions

#### ***■ Explain what this category of refactorings is all about***

**Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.**

Decompose Conditional Consolidate Conditional Expression Consolidate Duplicate Conditional Fragments Remove Control Flag Replace Nested Conditional with Guard Clauses Replace Conditional with Polymorphism Introduce Null Object Introduce Assertion

- Simplifying method calls

**These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.**

#### **Rename method**

Rename Method Add Parameter Remove Parameter Separate Query from Modifier Parameterize Method Replace Parameter with Explicit Methods Preserve Whole Object Replace Parameter with Method Call Introduce Parameter Object Remove Setting Method Hide Method Replace Constructor with Factory Method Replace Error Code with Exception Replace Exception with Test

- Dealing with generalizations

**Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.**

- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

Let's take a look at each of them and the most common refactorings.

### A plan for applying the most common refactorings

- Fowler's intro to the first set of refactorings

I'm starting the catalog with a set of refactorings that I consider the most useful to learn first. Probably the most common refactoring I do is extracting code into a function (Extract Function (106)) or a variable (Extract Variable (119)). Since refactoring is all about change, it's no surprise that I also frequently use the inverses of those two (Inline Function (115) and Inline Variable (123)). Extraction is all about giving names, and I often need to change the names as I learn. Change Function Declaration (124) changes names of functions; I also use that refactoring to add or remove a function's arguments. For variables, I use Rename Variable (137), which relies on Encapsulate Variable (132). When changing function arguments, I often find it useful to combine a common clump of arguments into a single object with Introduce Parameter Object (140). Forming and naming functions are essential lowlevel refactorings—but, once created, it's necessary to group functions into higherlevel modules. I use Combine Functions into Class (144) to group functions, together with the data they operate on, into a class. Another path I take is to combine them into a transform (Combine Functions into Transform (149)), which is particularly handy with readonly data. At a step further in scale, I can often form these modules into distinct processing phases using Split Phase (154).

### Fowler recommends these

#### Start with cleanup

- Clean up formatting
- Remove
  - Remove comments, dead code, unused fields or variables

## Extract things

- 4 - Move things
  - Take a method or an abstraction and break it into something simpler
  - object → a method
- 4 - **Make the steps explicit & declarative** — Keeping high-level methods as declarative and close to the essential complexity as possible
  - As we've learned, features are made up of a number of sub-steps. Those substeps are methods.
  - At the highest level of abstraction, we'll have our tests. These describe work to be done.
  - The next level down, we have the implementation.
    - \* checkThis, doThat, nextStep, nextStep()
  - You want to strive for always having one layer of code that looks sort of like this if possible.
    - \* You will eventually have to deal with low-level stuff, but that should be abstracted away in private methods or other objects where that work is cohesive (makes sense based on the responsibility of the object).
  - How?
    - \* Refactoring
      - → extract nested conditionals to private methods
      - → follow the method template
      - Move method or move field – move to a more appropriate class or source file
      - Pull up – in object-oriented programming (OOP), move to a superclass
      - Push down – in OOP, move to a subclass
- **Move:**
  - Extract method
  - Extract variable
  - Inline method (inverse)
  - Inline variable (inverse)

## Rename things

- 2 - **Improve names** —
  - Rename method, field, variable, rename field – changing the name into a new one that better reveals its purpose
- **Name:**
  - Change method/function declaration
  - Rename variable
  - Encapsulate variable
  - Introduce parameter object

## Create abstractions

- How do you know when you need to create a new abstraction? (this is the emergent design approach) our entire Agile approach is emergent, though in Learning the Domain, we can do *some* BDUF to wrap our heads around the magnitude of the domain and we can merely ask great questions about the Use Case/Story before we build it. Most of the time, we don't care about having a big requirements document, a massive class diagram or sequence diagram (though those can help). The tests are the single source of truth for if we are building the right thing. Both functional and non-functional.

**Nope — the answer to this is roles, responsibilities, and collaborations.**

- 5 - Create necessary abstractions
- When your classes are unnecessarily coupled → see if you can uncouple them with an abstraction
- When your classes are not cohesive → takeOff() example → put those functions, properties, etc, into a new class that makes more sense.
- Show the example of refactoring the guitar props out to a guitar spec (encapsulate what varies)
- **Create abstractions**
  - Combine functions into class
  - Combine functions into transform
  - Split phase → declarative

## How to master refactoring

- How should we continue to learn these? — **you can study them, but eventually, once you improve your understanding of coupling, cohesion, and can ascertain what readable code looks like, you won't need to have them memorized.** I do recommend either taking a look at Fowler's catalog of refactorings or checking out the website refactoring guru for more inspiration

## Exercises

## Summary

## References

- [https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896bodb96d9d](https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896bodb96d9d)
- https://sourcemaking.com/refactoring/refactorings
- https://www.cloudzero.com/blog/refactoring-techniques
- https://blog.kylegalbraith.com/2019/06/30/the-three-most-common-refactoring-opportunities-you-are-likely-to-encounter/
- https://www.stepsize.com/blog/the-ultimate-engineers-guide-to-refactoring
- https://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/
- https://www.industriallogic.com/img/blog/2005/09/smellstorefactorings.pdf

## Books

- Refactoring: Improving the Design of Existing Code — by Martin Fowler, with Kent Beck

## Websites

- Refactoring Guru — Refactoring Techniques

## 38. Detecting Code Smells & Anti-Patterns - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book.

■ Code smells are early indicators of *potentially* poor object-oriented design. If we know how to spot code smells, we can make better design decisions by understanding and managing the tradeoffs involved in the occasional breaking of rules, principles, and the philosophy of DDD.

### Chapter goals

In this chapter, we will aim to:

- Understand the difference between signs of complexity, code smells, and anti-patterns
- Understand the five different categories of code smells
- Learn how the rules of object calisthenics helps prevent code smells
- Learn how you can use refactoring techniques to clean code smells
- Identify the seven most common code smells and the refactoring techniques that can be used to clean them up

### Understanding complexity, code smells, and anti-patterns

#### Symptoms of complexity

#### Rely on the stuff that I wrote in Complexity & The World of Software Design

Some refer to these as *design smells*.

#### Code smells

#### Rely on the stuff that I also wrote earlier at the top

#### Also paraphrase some of the best highlights from Agile Technical Practices

#### Anti-patterns

#### I can use a lot of what I already wrote in the notes at the top here

Say that you can't really *detect* this and then fix it. You just have to know about these ahead of time. You just have to know what's wrong and not do these things.

## **The five categories of code smells**

here, we'll just describe each of the categories, and for each, we'll include a small blurb on what exactly the code smell is.

I can find the answers/ideas for each of these in both:

- Agile technical practices
- [https://en.wikipedia.org/wiki/Code\\_smell](https://en.wikipedia.org/wiki/Code_smell)
- <https://refactoring.guru/refactoring/smells>

### **Bloaters**

### **Object-orientation abusers**

### **Change preventers**

### **Dispensables**

### **Couplers**

## **Using object calisthenics to prevent code smells**

here, I just want to say that object calisthenic rules are what helps us prevent a number of code smells that occur — not all of them, but a fair amount.

### **Demonstration**

Then, I should also show one example of

### **Violation consequences**

Object calisthenics is a set of rules that help us avoid some of the code smells. If we decide to break the rules, we have consequences that materialize as code smells. In design, following rules is not always the best path; sometimes we need to bend or break rules. This is fine as long as we are conscious about it, namely the consequences and the benefits. Software design should be a series of decisions. Our worst enemy is accidental complexity<sup>77</sup> due to unconscious design decisions.

Object calisthenics violation	Code smells consequence
Only one level of indentation per method	Long Method
Don't use the ELSE keyword	Long Method / Duplicated Code
Wrap all primitives and strings	Primitive Obsession / Duplicated Code / Shotgun Surgery
First class collections	Divergent Change / Large Class
One dot per line	Message Chains
Don't abbreviate	NA
Keep all entities small	Large Class / Long Method / Long Parameter List
No classes with more than two instance variables	Large Class
No getters/setters/properties	NA
All classes must have state, no static methods, no utility classes	Lazy Class / Middle Man / Feature Envy

## Use refactoring techniques to fix code smells

let's say that we've broken some of the rules or we've identified some code smells. What do we do next? refactor of course. but what refactoring do we use?

Well, if we were to look at Fowler's book (or refactoring guru), there are a large number of refactorings.

one bloater, one change preventer, one disposable, coupler, object abuser

### Data clumps (bloater)

#### Switch statements (object-orientation abuser)

Show an example of some code

Show the list of potential refactorings

### Shotgun surgery (change preventer)

### Comments (disposable)

### Inappropriate intimacy (coupler)

## The five most common code smells

Say that if you prioritize solving these code smells predominantly in your code first, you should be in a good place.

See numbers 1 to 5 in here <https://medium.com/geekculture/5-most-common-code-smells-that-you-should-avoid-86ae41cb1dc7>

*Great demonstrations of 5, 6, 7 in Agile technical practices*

## **1 - Duplication**

## **2 - Long method**

## **3 - Large class**

## **4 - Long parameter list**

## **5 - Primitive obsession**

## **6 - Feature envy**

## **7 - Message chains**

### **Summary**

### **Exercises**

#### **Tic Tac Toe (with code smells)**

Created by Pedro Santos (in TypeScript).

### **References**

### **Articles**

- <https://apiumhub.com/tech-blog-barcelona/code-smells/>
- <https://medium.com/geekculture/5-most-common-code-smells-that-you-should-avoid-86ae41cb1dc7>
- [https://en.wikipedia.org/wiki/Code\\_smell](https://en.wikipedia.org/wiki/Code_smell)

### **Books**

- Refactoring
- Agile Technical Practices Distilled

### **Web**

- <https://github.com/stemmlerjs/CodeSmells>
- <https://refactoring.guru/refactoring/smells>

## 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book.

■ Classic (inside-out) TDD can be used to write fast executing unit tests against core code. Up until now, that's what all of our algorithmic or OO design problems have been focused on. However, real world problems involve APIs, databases, and other forms of infrastructure. To keep code pure, we must master the art of dependency inversion. In this chapter, we learn to detect and decouple from obvious and subtle forms of infrastructure most commonly known to bleed into core code.

"At last a book, suffused with code, that exposes the deep symbiosis between TDD and OOD. The authors, pioneers in test-driven development, have packed it with principles, practices, heuristics, and (best of all) anecdotes drawn from their decades of professional experience. Every software craftsman will want to pore over the chapters of worked examples and study the advanced testing and design principles. This one's a keeper." — Robert Martin on *Growing Object-Oriented Software Guided by Tests*

Really, this is the Humble Object Pattern

<https://martinfowler.com/bliki/HumbleObject.html>

Incredible read below

Responsibility Driven Design with Mock Objects

<https://twitter.com/GGalezowski/status/1513426282557628418>

### Chapter goals

- Revisit the ideas of core and infrastructure code and discuss how we test them
- Deepen our understanding of dependency inversion, the most important technique for decoupling core from infrastructure code
- Discuss five of the most common core-infrastructure code coupling scenarios and how to fix them
- Demonstrate how to decouple a feature from infrastructure code

### Where are we so far?

In 24. An Object-Oriented Architecture, we saw that Dependency Inversion was key to implementing a *Layered Architecture*. A layered architecture separates core code from infrastructure code.

I may want to rely on the usage of the "How to Test Code Coupled to APIs and Databases" in this post <https://khalilstemmler.com/articles/test-driven-development/how-to-test-code-coupled-to-apis-or-databases/>

### Core code is pure, infrastructure is not

Core code is pure, infrastructure is not

- What does *pure* mean?
  - Means it doesn't touch any sort of infrastructure
  - No network, filesystem, I/O

## Test core code with unit tests

We test core code with unit tests

- *Reminder:* Unit testing principles
  - **According to Michael Feathers, a test is not a unit test if:**
    1. It talks to the database
    2. It communicates across the network
    3. It touches the file system
    4. It can't run at the same time as any of your other unit tests
    5. You have to do special things to your environment (such as editing config files) to run it.
- Very simply, if it breaks any of these rules, then the code is not core code and it cannot be unit tested.

## Dependency inversion revisited

### Dependency inversion (know it deeply)

- Let's discuss what it is again
- It's one of the foundational SOLID principles
- In object-oriented programming, *abstractions* are either interfaces or abstract classes.
- For now, we'll mostly be using interfaces
- Dependency inversion separates dependencies from each other and allows you to keep core code un-reliant on infrastructure code.
  - Why?
    - \* For testing purposes.
    - \* This is the foundation for creating *test doubles*, which is something we will be doing later.
- Show an image of what dependency inversion looks like
  - CreateUser (core) → UserRepo (infra)
  - Fix it with this
  - CreateUser (core) → IUserRepo (abstraction) ← UserRepo (infra)
- That is the simplest way I can describe dependency inversion, but it's so useful and important to testing that I wanted to re-introduce it again.

## Common core-infrastructure coupling scenarios

### Databases: Dealing with persistence

### Databases — Talking to a database

- Very common when you're using mysql, an ORM, or anything that relies on a persistence technology of any sort, even a key-value storage like Redis. If code pull some infrastructure like this along for the ride, we can't unit test it. It'll make our code slow.

```

// Directly importing infra dependency
import { models } from '../sequelize/models'

class UserService {

  constructor () {
    this.models = models; // assigning it here
  }

  async function createUser (userDetails) {
    // Validation logic
    ...

    // Application logic
    const userAlreadyExists = await models.User.findOne({
      where: { email: userDetails.email }
    });

    if (userAlreadyExists) {
      // return an error
    }

    ...

    // Persistence logic
    await models.User.create(userDetails); //
  }

}

```

How to solve it?

- First, move the import to the constructor
- **Wrap it all behind an abstraction**
- What is the *essential* persistence behavior? creating a user, finding one by email. Yes, let's move that.

```

interface IUserRepo {
  createUser (user): Promise<void>;
  findUserByEmail (userEmail): Promise<User | Nothing>;
}

```

## Network: Communicating over the network using an API

### Network — Communicating over the network or using an API

```

// Directly importing infra dependency
import { stripe } from '../'

async function billCustomer (billingDetails) {
  // Validation logic
  ...

  // Application logic
  ...

  const charge = await stripe.charges.create({  //
    amount: billingDetails.amountInCents,
    currency: 'cad',
    source: 'tok_mastercard',
    description: 'Book purchase',
  });
}

...
}

```

How to solve it?

- **Wrap the abstraction with an adapter**

```

interface BillingAPI {
  chargeCustomer (): Promise<void> {}
}

```

- *this is the*

**Statefulness: Relying on existing test state**

**Sequential tests — Relying on existing test state**

The best example of this is setting up game state for a *Chess* or *Checkers* game.

You have to load the initial state of the world.

How might you do that?

Where does that initial state come from?

How is the initial state modelled? Is it a big old *state* object?

Where do we get that data? From a *database*? Do we need to rely on one?

**Solution #1: Modelling state as an object**

But there's something unfortunate about this.

We miss the *temporal nature* of

This is especially important if we want to test that *certain behaviours* occur after *events*.

Therefore, when you recognize this, consider modelling state as events.

## Solution #2: Modelling state as a collection of events

The solution: Events.

Data, behaviour, namespaces.

State is data.

Moreso, state is *current* data.

That implies time.

That implies data *changes* over time.

How does data change?

Behaviour.

But what about the time part?

Events.

So then, our goal is to model state in a way that documents how it changed over time.

This will enable us to apply behaviour again.

**Therefore, to deal with existing test state**

```
describe('starting a game from previous events', () => {
  let game: Game;
  let pieces: Pieces;
  let turn: Turn;
  let board: Board;

  beforeEach(() => {
    pieces = new Pieces();
    board = new Board(pieces, new EventObserver());
    game = new Game(board);
  });

  it('replays the events and starts the game from that state', () => {
    let gameResult = Game.createFromEvents(board, [
      new PieceMovedEvent('R1', [1, 4]),
      new PieceMovedEvent('W3', [0, 3]),
      new PieceMovedEvent('R10', [7, 4])
    ]);

    let game = gameResult.getValue();
    let r1Moves = game.getAvailableMovesForPiece('R1').data as Movement[];
    let r1TargetPosition = r1Moves[0].getTo();

    expect(gameResult.isSuccess).toBeTruthy();
    expect(game.getCurrentTurn().getColor()).toEqual('white');
    expect(r1Moves.length).toEqual(1);
  });
});
```

```

    expect (r1TargetPosition).toEqual([2, 3]);
}

});
}

```

## System APIs: Using the system clock

- Using system APIs — Using the system clock

```

let task: Result<Task>;
let clock = new FakeClock(DateUtil.createDate(2021, 8, 11))

it('needs a title, time range, url, and day', () => {

  task = Task.create({
    title: new Title('Clean bathroom'),
    time: MilitaryTimeRange.create('12:00 - 13:00').getValue() as MilitaryTimeRange,
    date: TaskDate.create(1, 1, 2021).getValue() as TaskDate,
    url: new URL('https://notion.co/hi'),
    createdAt: clock.getCurrentDateTime().toISOString()
  })

  expect(task.isSuccess).toBeTruthy();
  expect(task.getValue()).toBeDefined();
});

```

```

export interface Clock {
  getCurrentDateTime(): Date;
}

```

```

// testUtils/fakeClock.ts

import { Clock } from "../services/calendar/clock";

export class FakeClock implements Clock {
  private dateTime: Date;

  constructor (dateTime: Date) {
    this.dateTime = dateTime;
  }
  getCurrentDateTime (): Date {
    return this.dateTime
  }
}

```

## Environment: Relying directly on environment variables

- Environment — Relying directly on `process.env`

## Demonstration

- *Maybe (or just walk through it to be honest)* Example — refactor this by first writing the acceptance tests to make sure that it works, and then decouple it so that we
  - Talk about how we could acceptance (unit) test the feature for the essential complexity, and then integration test dependencies themselves

## Examples

- Example — look at the following code and determine if it's core or infra
- Example — look at the following code and determine if it's core or infra
- Example — is this core or infra
- Example — is this core or infra

## Summary

### Picture? Where we are now

**I should make sure to apply dependency inversion as one of the techniques in unit testing for the picture**

## Resources

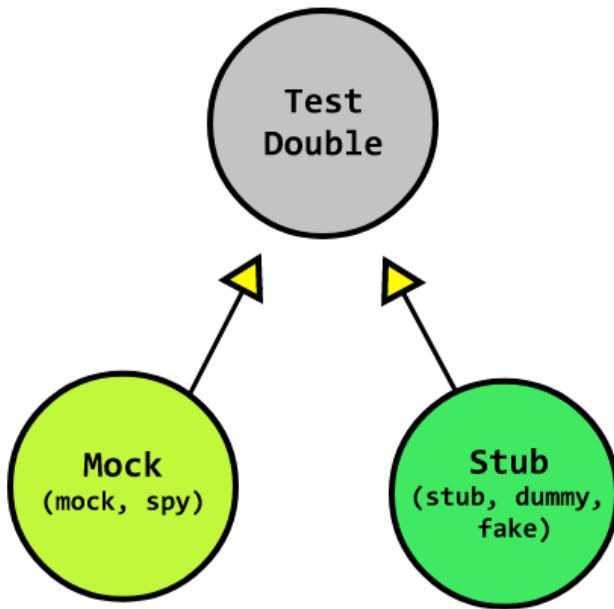
### Articles

- How to Test Code Coupled to APIs and Databases

## 40. Using Test Doubles - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book.

■ Pure functions are easily tested; inputs and outputs. For core code that relies on infrastructure or state<sup>\*\*</sup>, however, a more rigorous approach is required. In this chapter, we learn about test doubles: objects that stand in for production ones during testing. Test doubles are key to the Mockist school of TDD; with them, we can gain the ability to simulate and test a variety of scenarios involving state and infrastructure dependencies.



## Chapter goals

- Understand why we need test doubles to verify behavior against components that rely on infrastructure dependencies with unit tests

### What's missing in our feature verification?

#### Verifying state

#### Verifying behavior

#### Classic verifies state, mockist verifies behavior

In 29. Getting Started with Classic Test-Driven Development, I said that that what makes “Classic TDD *classic* is the absence of mocking”. That’s only partially correct.

Another way to think about it is that what separates classic TDD from mockist TDD is the way that we go about verifying that the SUT works correctly.

#### Expand on how classic TDD is state, mockist is behavior against the collaborators

- What’s the problem? (classic vs. mockist TDD)?
  - Now that we know about how to isolate core from infrastructure, that’s good, but that’s not
  - **Come back to the nature of a feature and what there is to verify → it’s not just about the result. It’s about the behavior as well.**
  - We can verify based on **state**, the end-result, but that’s not all that happens in a use case.

- For example, creating a user, yes we save the user in a database, but we may also need to make a request to an API, charge the customer, and also send an email at the end.
  - \* How can we test that we've done these things?
- Our testing at this point is no longer just about testing **state**, the **end result**, it's about testing **behavior** — and what happens during the test.
- This is the nature of the difference between the *Classic* and *Mockist* versions of TDD.
  - \* Classic → State
  - \* Mockist → Behavior
- **Reference: Classical vs. mockist style testing**
  - \* Classical → state verification
    - **This style of testing uses state verification:** which means that we determine whether the exercised method worked correctly by examining the state of the SUT and its collaborators after the method was exercised. As we'll see, mock objects enable a different approach to verification.
  - \* Mockist -> behavior verification

## Test doubles

What is a test double?

- What are they?
  - **A test double is a generic term to refer to any object that stands in for a production one during testing**
- What purpose do they serve?
  - Separate the SUT from collaborators necessary to run it
  - What is a SUT?
    - \* The SUT (system under test) is merely the current part of the system that is being tested.
    - \* **Show an image pointing to the current SUT**
  - What is a collaborator? What are some examples of collaborator?
    - \* Collaborators are often, but not always, *infrastructural dependencies*. When they *are*, we typically want to sub them in for test doubles in order to test the SUT using unit tests.
    - \* As we've learned, most use cases rely on dependencies → databases, external APIs, etc. And we need to call upon them during the execution of a use case.
    - \* Example: CreateUser → UserRepo, EmailService
- Why do we need test doubles? Why separate SUT from dependencies?
  - Dependencies, as we've learned, are usually infrastructural in nature
    - \* If we're using the *Clean Architecture*, we can apply the dependency inversion principle by making sure that the *Use Cases* never directly refer to an infrastructural dependency.
      - Instead, they rely on abstractions (interfaces) to dependencies
      - **Note: In the clean architecture, this is referred to as the adapter layer**
  - Test doubles let us separate core from infrastructure. This lets us run the SUT

- (typically the use case) to verify that we get the correct result
- Test doubles also give us control over the preconditions (given) \*\*in the Given-When-Then.
    - \* This enables us to verify that our SUT behaves in the ways we expect in certain scenarios.
      - Ex: *MockUserRepo* doesn't have the user created in it already? → we should attempt to save a new user vs. *MockUserRepo* already has it in there → then we should *not* attempt to save a new user, the request should end early.
  - What are the different categories of test doubles?
    - **Originating from the idea of a stunt double (like in action movies), there are two general categories of testing objects that we use in testing: mocks and stubs.**
    - **Show a list here, just so that we can see it at a glance**
      - \* *Fowler's list* →
      - \* **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
      - \* **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
      - \* **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
      - \* **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
      - \* **Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive. **Oh shit, so this really is a mock — using libraries to create mock objects. Yeah. I have an article where I demonstrate this.**
      - \* <https://khalilstemmler.com/articles/test-driven-development/how-to-mock-typescript/>
        - ts-auto-mock

## Behavior (commands & queries)

Behavior: Commands or queries

- Explain commands and queries again, and

## Guidelines

- Guidelines
  - Use mocks for commands, stubs for queries
  - Do not perform assertions on queries; assert the result instead

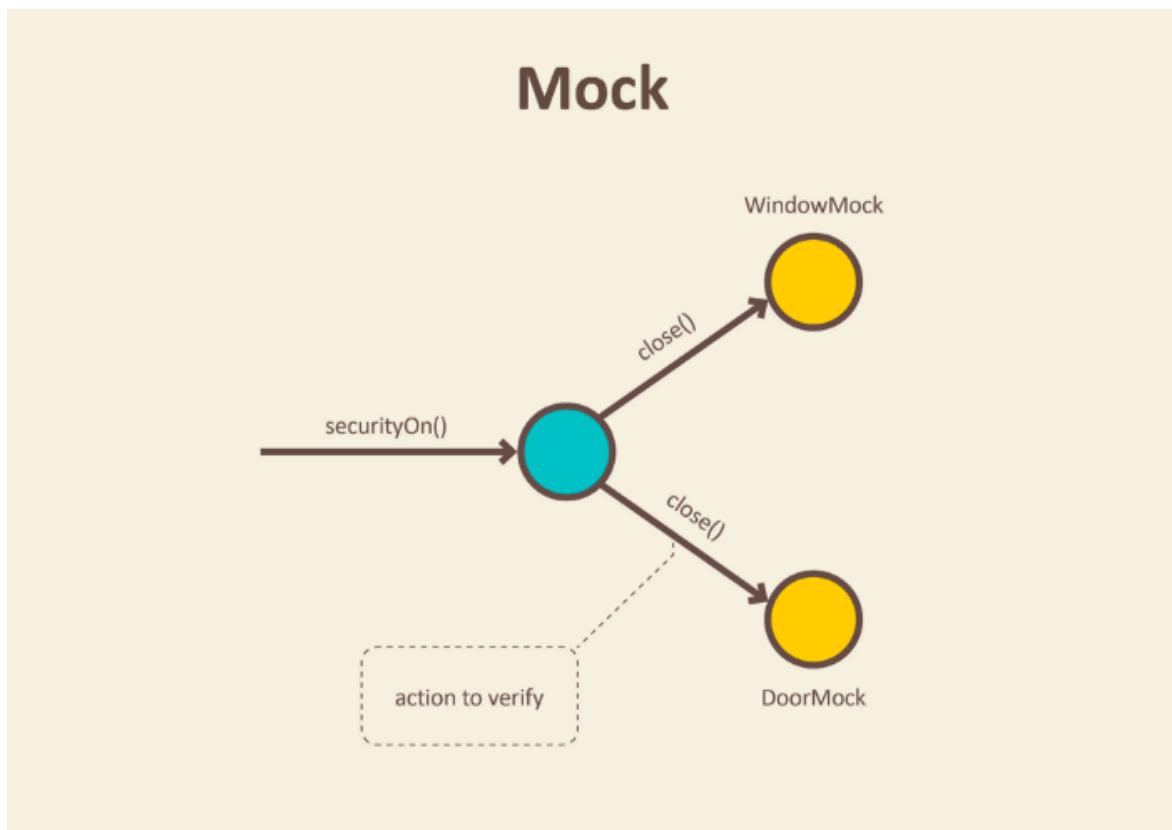
## Mocks

Explain what mocks are, what their purpose is, when we use them, and demonstrate how we use tools to create them.

Do this for each (what, why, when, how)?

Include diagrams for each

- What is a mock?
  - We use a mock to stand in and assert against command-like operations (outgoing interactions or state changes against dependencies). We pass mocks in to the system under test (SUT) and later check during the *assert* phase of a test that the correct calls to change the dependencies' state were made.
  - Mock objects encourage testing based on behavior verification
- Why use a mock?



We use mocks when we don't want to invoke production code or when there is no easy way to verify that intended code was executed. There is no return value and no easy way to check system state change. An example can be a functionality that calls e-mail sending service.

We don't want to send e-mails each time we run a test. Moreover, it is not easy to verify in tests that a right email was sent. Only thing we can do is to verify the outputs of the functionality that is exercised in our test. In other words, verify that e-mail sending service was called.

- When would we use this?
- How to use 'em? Example?
  - We may want to verify that save was called in the case where we *should* save a new user. And we also want to verify that save was *not called* in the scenario where we *shouldn't* save.
  - This also goes for using external APIs. as well.
- Your tooling should support creating a mock with merely an interface (the contract)

## Spies

**Explain what spies are, explain why we'd use a spy instead of a mock, example of when we'd use them, and demonstrate lightly how this works.**

**These are exactly the same thing as traditional mocks except that we *hand-roll* spies manually, whereas with traditional mocks, mocking libraries like ts-auto-mock help you create mocks in a single line or two (I highly recommend this package over basic mocking with Jest). You can see an example of a hand-rolled spy here.**

## Stubs

**Explain what stubs are, why we use them, and what this looks like.**

**A stub is a dependency that we can configure to return different values in query-like scenarios. This generally takes some effort to do.**

## Fakes

**Explain what fakes are, why we use them, and what this looks like.**

**A fake is practically the same thing as a stub. They only differ in the sense that we create a fake to sub in for a dependency that doesn't exist yet. This can be done very quickly.**

## Dummies

**Explain what dummies are, why we use them, and what this looks like.**

**These don't do anything. Nor are they used during a test. These are objects that are just used to fill up parameter lists so that a constructor, function, or method will execute. They can often be null or empty-string.**

## Exercises

**See agile technical practices for inspiration on this bit here**

## Demonstration

**We could demonstrate that this makes complete sense for when we're doing acceptance testing. yes, we have to test that the result we get back works (sometimes), but**

**more importantly, we want to test that the correct behavior was performed against the dependencies**

Scenario: Soemthing

Given We did this thing

When this other thing happens

# Only a test double can verify these things!

Then the system should save it to the database

And the system should send an email

And the system should charge the customer

**Do we see how vital the mockist approach is for testing use cases/acceptance testing?**

## Summary

Show where we are

Use the pictures again —

## Testing objects in a layered architecture

- Remember that all this work of separating code into layers is because we'd like to do some *whitebox testing* instead of just *blackbox* (end to end) testing. This applies to the front-end as well.
- As we continue on, specifically, in the Hexagonal Architecture chapter, we're going to learn about all of the different objects that exist in the layered architecture in detail. So far, the one we've been primarily focused on is the *Use Case*.
- We know how to test a use case, but we're going to have to discuss the other ones as well, repos, value objects, event handlers, controllers, etc.
- These things need to be tested as well, some of them relying on the usage of test doubles.
- In that later chapter, we'll learn how to test each object — using different types of test doubles for each.

## References

### Articles

- <https://martinfowler.com/articles/mocksArentStubs.html>
- <https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da>

## 41. Testing Legacy Code - Rough Notes

■ **Note:** These are my rough notes. This is just here for reference readers before I switch to the new outline / structure of the book.

■ Most of the code you'll encounter in the wild is existing, brownfield code. Much of it is untested. To safely refactor, add, change, or remove features, we must first establish a basis of tests that enable us to be productive without worry of breaking something. We learn how

to introduce tests to previously untested code using techniques such as *seams*, *Characterization tests*, and *Golden Master* tests.

## Part VI: Design Patterns

■ WIP: This section is still being worked on. ■

## Part VII: Design Principles

■ WIP: This section is still being worked on. ■

### Coupling, cohesion & connascense

#### 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.
**/

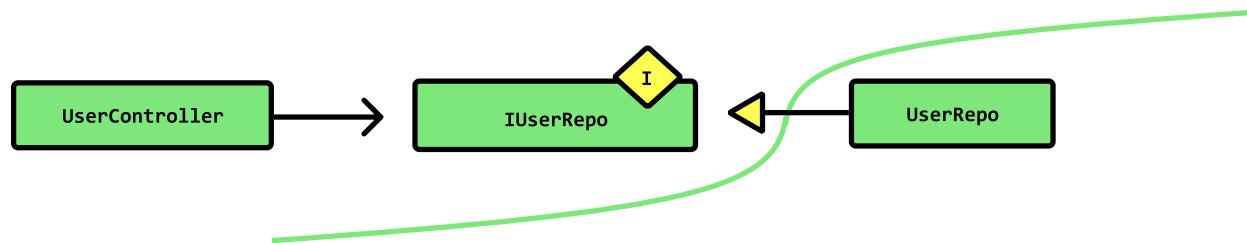

class UserController {
    private userRepo: IUserRepo; // like here

    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: [] });
})

```

## 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 implementations, 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.*

## Recollection

- Recall that in 24. An Object-Oriented Architecture, dependency inversion is the key technique used to create a layered architecture. We use layered architectures to separate core code from infrastructure code. A separation of core code from infrastructure code introduces testing options which we can be used to compose a testing strategy — writing unit tests for core code and integration tests for infrastructure.
- Recall that in 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - coming soon, there exists a number of ways that infrastructure can sneakily find itself into core code — through the network, I/O usage, and system APIs. Be cognizant. These concerns can make core code untestable, and turn fast existing unit tests into slow-running and potentially fallible integration tests.

## GRASP

### The Four Elements of Simple Design

#### Chapter goals

##### The four elements of simple design

Readability is what we want, but it's the result, not the

- represents the highest level of principles
  - not really a rigid set of rules — this is something you can print off and keep in your workstation

For the experts, the idea of *Simple Design* reigns supreme. It means

- Runs all the tests
- No duplication
- Expresses design ideas in the system
- Minimizes the # of elements

Order matters

#### Runs all the tests

#### No duplication

- one of the first code smells that we'll seek to remove
- every time we remove duplication, we're focused to create an abstraction
  - it's here that we create new abstractions to more *clearly* express what we're doing
- so it goes test → then duplication

#### Maximizes clarity

J. B. Rainsberger writes, "...usually names tend to go through four stages: nonsense, accurate but-vague, precise, and then meaningful or intention-revealing, as Kent Beck described it.

I focus my efforts on gradually moving names to the right of this spectrum. Instead of agonizing over the choice of a name, I simply pick a name, confident that I always look for opportunities to move names towards becoming meaningful.

Many people try to come up with a great name all at once. This is hard and rarely works well. The problem is that naming is design: it is picking the correct place for each thing and creating the right abstraction. Doing that perfectly the first time is unlikely.”

## Minimizes the # of elements

The best way to think about how to write clean code is the idea of Simple de

- we want coding conventions for Simple Design: expressive, readable, etc
- the practices will take us a long way
  - TDD → Refactor early + design simple abstractions
  - Domain-Driven Design (the metaphor) → name things better, express ideas, learn how to model the workflow using the domain and turning it into code

So then, how does this work really?

- [Principle 1]: runs all the tests
  - Most people know of *unit tests*
    - \* Generally speaking, these are tests that run really fast and run on the *inside* layers of your code.
  - At the very outer boundary of your code, it's possible to write what's called *Acceptance Tests* which tests the features and run just as fast
    - \* We want to have other tests too (integration/contract tests, and maybe E2E tests as well), but these tests they:
      - are written using the language of the domain.
      - read really well
      - If they're hard to write, we know that there's a problem with the design — so we take care
      - These are also tests that are written at a level of abstraction that **says that the features work**
- [Principle #2]: Expresses the features in the system
  - \* These are the features decided upon by the customer.
  - \* And if we write our tests well, then we can express the *functional requirements* which also represent the *essential complexity* of the application at a high level.
  - \* As we develop code using TDD, the idea is continue refactoring it as we go along.

## Other simple design definitions

### Intention

- Runs the tests
- Intention-revealing
- No duplication
- Fewest elements possible

## **Acceptance Test-Driven**

- Acceptance (unit) tests measure completion
- Self-documenting
- DRY
- YAGNI

## **Coupling & cohesion**

- Runs the tests
- Maximizes cohesion
- Minimizes coupling
- Fewest elements possible

## **Minimal**

- In Object-Oriented Programming, we have a lot of tools at our disposal — this increases the likelihood of design clumsiness
- Remember: The goal of the developer is testability, flexibility, maintainability — or more succinctly, something called Simple Design
- Simple Design is about tests, clarity, coupling & cohesion
- There are many different ways for us to express this idea of Simple Design

## **Diving into design**

It's time for us to start thinking about design

**Explain why we're moving over to design now based on where we currently are**

**Explain why TDD alone isn't enough — that yes, it's good that we can solve things using the TPP, but in oo, there are a lot of different tools at our disposal. Less is more.**

**Explain how simplicity is the key and that coupling and cohesion, again, are the two best measures of how we're doing against fighting complexity — testable, flexible, maintainable are the side-effects.**

**Explain why we're not starting with going over everything object-oriented programming, instead I'd like you to start with what you currently know**

**For each rule, explain how it helps coupling, cohesion — and evidently, testability, flexibility, maintainability.**

**For each rule, consider a drawing that helps to illustrate exactly what's happening based on our understanding of the conceptual model of object-oriented programming**

## **Shifting from technique to design**

- We spent the last little while learning design in TDD
- But in TDD, we need more than just technique
- Design is important too

- Why? → because
- What is design in the context of object-oriented programming?
  - Overall, the goals are the same — testable, flexible, maintainable. Tests help us with the *testable* part and having a declarative layer of code (our tests) that can act as a manual to how the rest of the code works, does a fair amount of good for *Maintainability* as well.
  - However, we have to look a little bit higher — that's too low-level.
    - \* **Insert a metaphor here about trying to fix something or get good at it by chipping away at it or not fully**
  - Yes, we have the TPP, yes we have some good techniques, but we need a better understanding of what design is
  - It is testability, maintainability, and flexibility
  - It is the balance of coupling and cohesion
  - It is simple design
    1. Passes its tests
    2. Minimizes duplication
    3. Maximizes clarity
    4. Has fewer elements
  - You could also say it
    - \* Passes the tests
    - \* Maximizes high cohesion
    - \* Maximizes loose coupling
- Why is this so important?
  - Because in OO, we have a lot of tools at our disposal. Abstractions, concretions, object creation, decorators, dependencies, scoping, and so on.
  - At a surface level, without a higher plane — a **value system** — it's still possible for us to produce messy designs that are hard to *change* and hard to *understand*, even with TDD.
  - We get into the ideas of code smells
- In OO, design is the way that we balance coupling and cohesion
- What is the north star of design?
  - Simple design
    1. Passes all tests
    2. Clear, expressive and consistent
    3. Duplicates no behavior or configuration
    4. Minimal methods, classes and modules
  - Simple design
    1. Passes tests
    2. Reveals intention → readable to humans
    3. No duplication
    4. Fewest elements

What's the challenge?

- OO has a lot of tools

- Designs can get out of hand quickly if we don't know what we're doing

What are we going to learn in this section?

**Explain that we're still going to be using tests to drive design. There's no doubt about that. However, the TDD workflow is just the engine. Within that TDD engine, we now have a safe and reliable way to take calculated design risks.**

**On the implementation side, there are patterns, techniques, — and pushing more towards the ascetic striving for good design, we have rules, principles, and values**

**It's our intention to:**

- learn what constitutes good object-oriented design first in this section of the book through rules like 36. Better Objects with Object Calisthenics - coming soon.
- then also through techniques like 37. Refactoring - coming soon we'll see different ways that code can transform to be more readable and maintainable
- Through detecting 38. Detecting Code Smells & Anti-Patterns - coming soon, we'll avoid stepping down well-known roads that lead to bad design
- In learning how to 39. Keeping Unit Tests Pure with Mockist TDD and The Art of Purity & Inversion - coming soon, we master the art of dependency inversion and practice separating core code from infrastructure code
- And finally in this chapter, we
- We start with some good rules in 36. Better Objects with Object Calisthenics - coming soon. A way to Explain what it is
- Then, o

## Coupling & cohesion is the north star of design

- We need a north star of design — we need something to understand — a principle
- I'll say it again, and several more times throughout the book — the overarching values of how we'll strive towards good design are coupling and cohesion.
  - We value loose coupling and strong cohesion
  - We devalue tight coupling and low cohesion
- But as we've also learned, values are hard to put into practice, so we can start with *rules* first to push us along the path. Then we adopt Principles, which are closer to values and can be applied in many more contexts. The principles get us close to the values. Eventually, though practice, you can develop your own principles stemming from the values, and subsequently, your own rules.
  - *Pictures of learning → rules (phronimos) → principles (phronimos) → values (ubiquitous) → principles (own) → rules (own)*
- In this chapter, we're primarily going to focus on a set of rules to which will get us closer the principles we learn about in Part VII: Design Principles.

## Summary

- Object

**Conclusion**

**Resources**

## **Programming by Wishful Thinking**

### **The Least Principles**

Principle of Least Effort

Principle of Least Astonishment (Surprise)

Law of Demeter (Principle of Least Knowledge)

### **Principle of Least Effort**

**Principle of Least Astonishment (Surprise)**

**Law of Demeter (Principle of Least Knowledge)**

### **Design by Contract (DBC)**

Chapter 21 in The Pragmatic Programmer is really good for this!!!

### **Separation of Concerns**

#### **CQS (Command Query Separation)**

**YAGNI**

**KISS (Keep It Simple, Silly)**

**DRY, WET, Rule of Three**

**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 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 {
      // 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

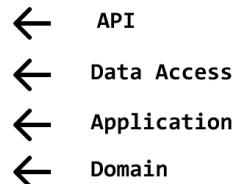
by Khalil Stemmler | @stemmlerjs

### Single Responsibility (Role)

Admin   Member   Guest



### Separation of Concerns



### 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*

**Strive for loose coupling between objects that interact**

**Principle of Least Resistance**

**Tell, Don't Ask**

**No And's, Or's, or But's**

## Part VIII: Architecture Essentials

### 8. Architectural Principles

Contracts

Component principles

Conway's Law

The Dependency Rule

Boundaries

Cross-cutting concern

The Principles of Economics

Separation of Concerns

## **Contracts**

High-level policy vs. low-level policy. Declarative vs. imperative. Specification vs. implementation. Abstraction vs. concretion. BDD vs. TDD. Driving design based on behavior vs. focusing on implementation details. What are we talking about here? There's a philosophical idea about software design underpinning all of these discussions. The idea is that **software design should be contractual**. That is, we succeed when we use contracts — when we decide what we should build, and can describe it clearly — before we build it.

## **Component principles**

Reuse-Release Equivalence Principle

Common closure principle (CCP)

The Common Reuse Principle (CRP)

Stable Dependency Principle

Volatile Components

### **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.

### **Stable Dependency Principle**

### **Volatile Components**

**Conway's Law**

**The Dependency Rule**

**Boundaries**

**Cross-cutting concern**

**The Principles of Economics**

The Principle of Opportunity Cost

The Principle of Last Responsible Moment

**The Principle of Opportunity Cost**

**The Principle of Last Responsible Moment**

**Separation of Concerns**

## **9. Architectural Styles**

Structural

Message-based

Distributed

**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**

Layered (n-tier) architecture  
Event sourcing  
Microkernels  
Microservices  
Space-based

### **Layered (n-tier) 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/>

### Event sourcing

#### Microkernels

#### Microservices

#### Space-based

## Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS

### II. Building a Real-World DDD app

About this chapter

Chapter goals

Domain-Driven Design

The Project: DDDForum — a Hackernews-inspired forum app

How to plan a new project

Building DDDForum

Where to go from here?

Resources

References

### 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 in 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 contains a number of tactical patterns which we can use in 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.

■ **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

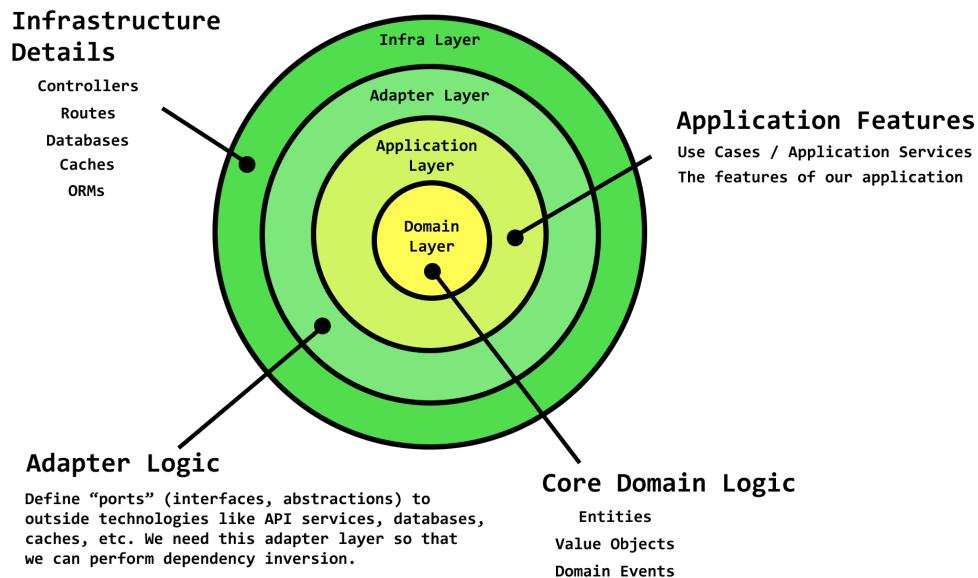
Once the common language is established and agreed upon, we use those words as our classes, use cases and types in the code.

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.

"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.

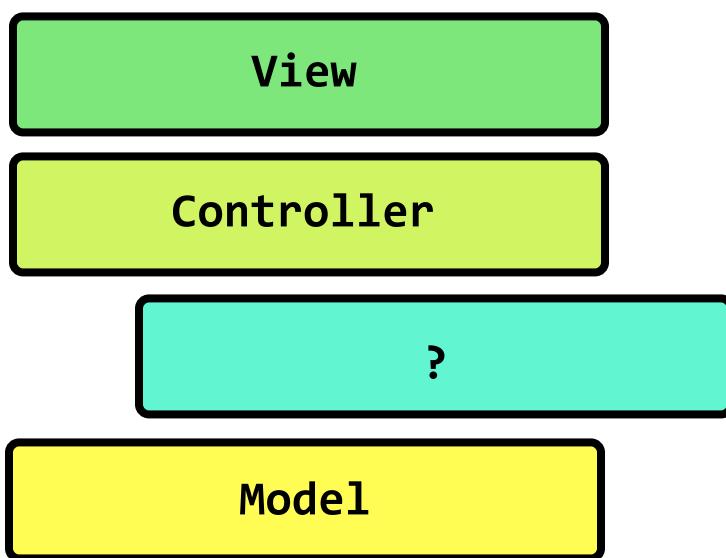
Let's say we extracted all of that access-control logic to some middleware functions.

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**.

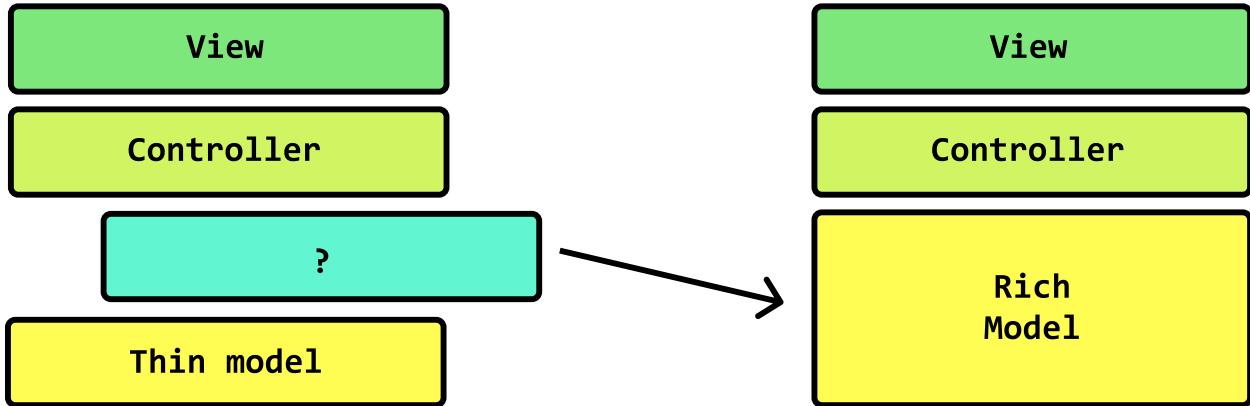


The missing layer in MVC.

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*.

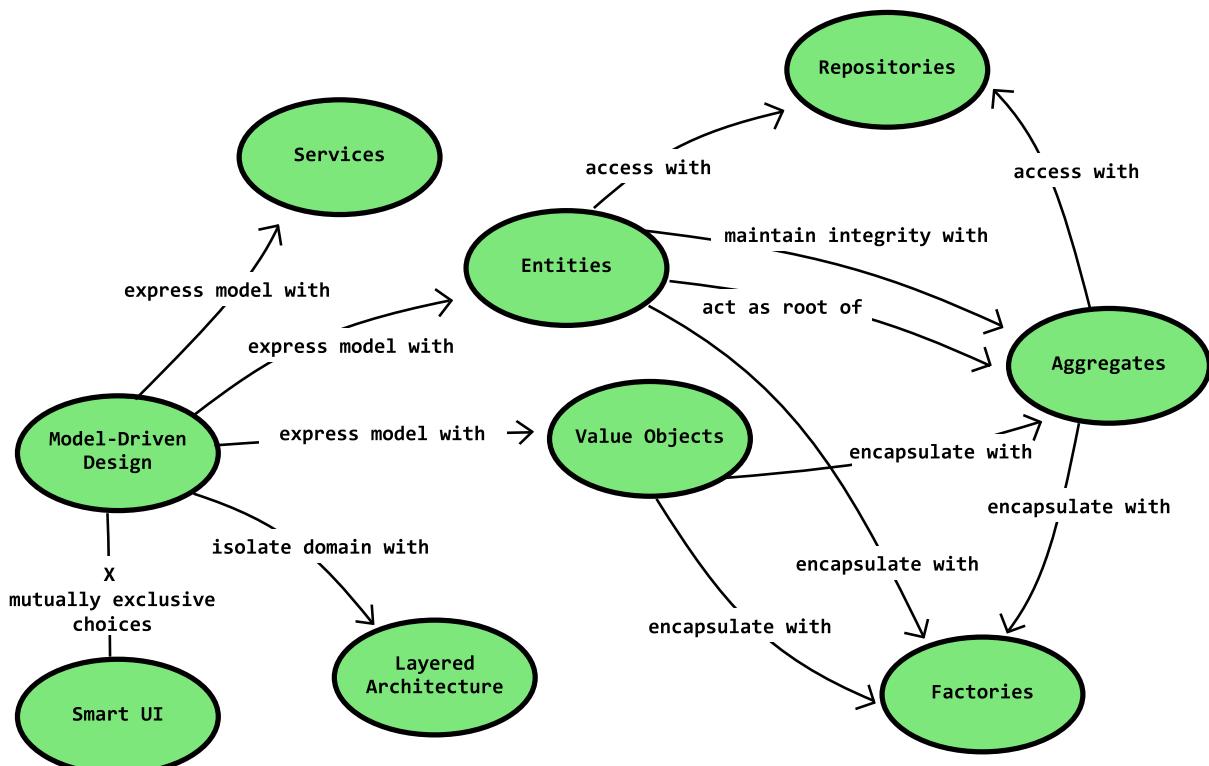
**Transaction script:** The simplest way to organize domain logic as possible. By using simple *if* and *else* control statements, we can express domain logic easily. This is perfect for simple applications without a huge amount of domain logic that can do without a lot of time spent on architecture.

*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.



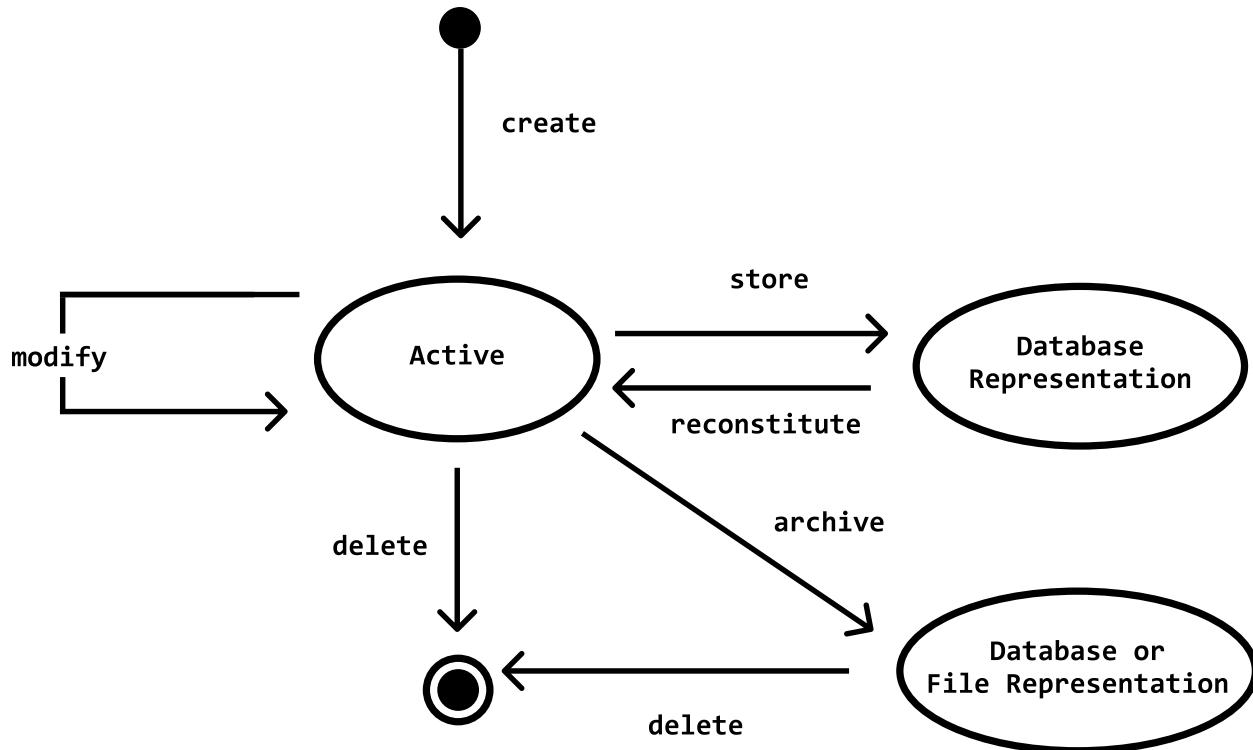
## Entities

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:

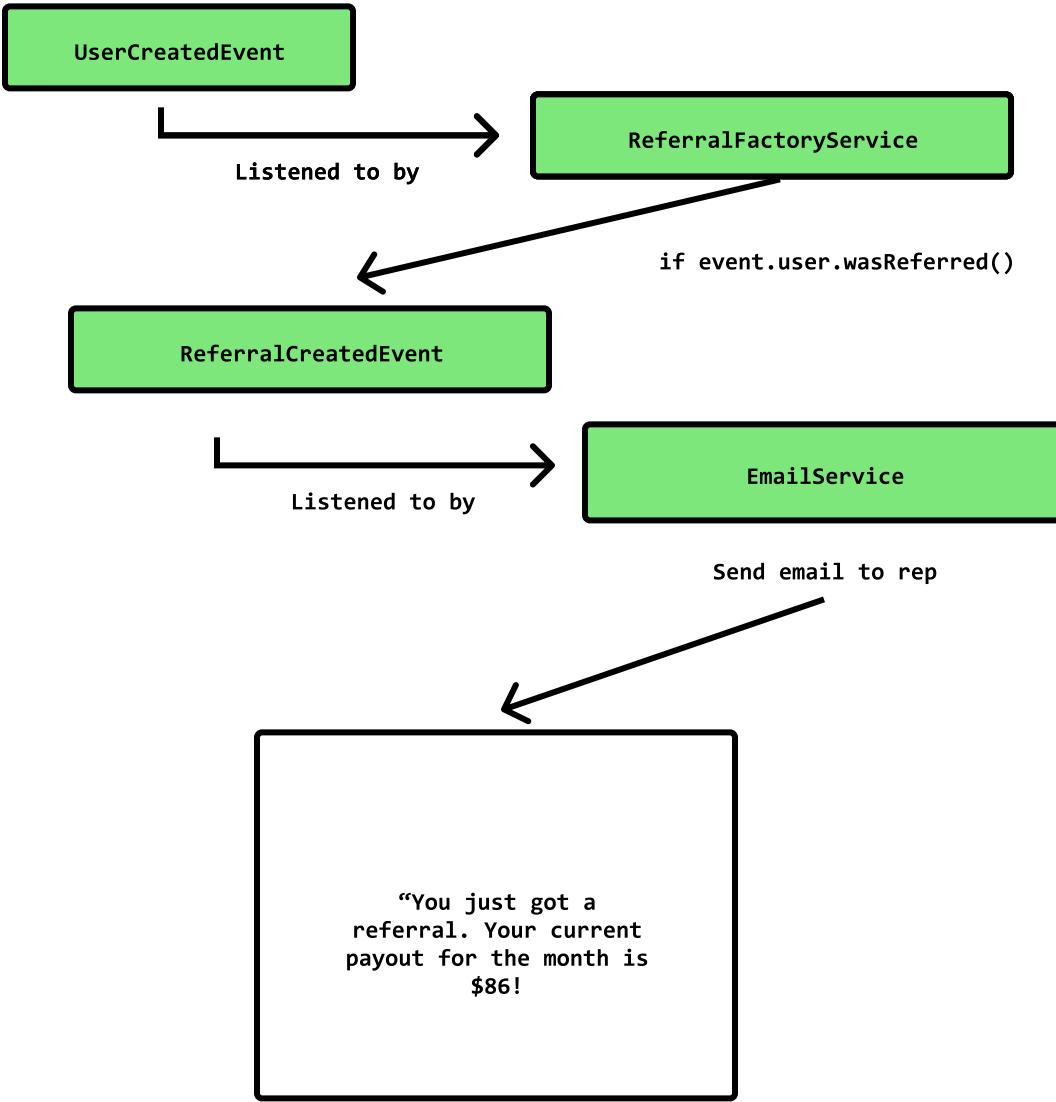
    // 1. Recording a referral (if one was made)
    if (user.referred_by_referral_code) {
      // calculate payouts
      // .. there could be a lot more logic here
      await Referral.create({
        code: this.req.body.referralCode,
        user_id: user.user_id
      });
    }

    // 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 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?

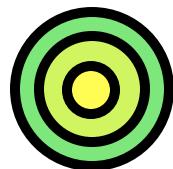
The *problem space* is the entirety of the things that a business is faced with solving.

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.

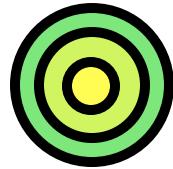
Each of these concerns are *subdomains*; decomposed logical slices of the entire problem domain.



**Billing**  
*subdomain*



**Trading**  
*subdomain*



**Catalog**  
*subdomain*

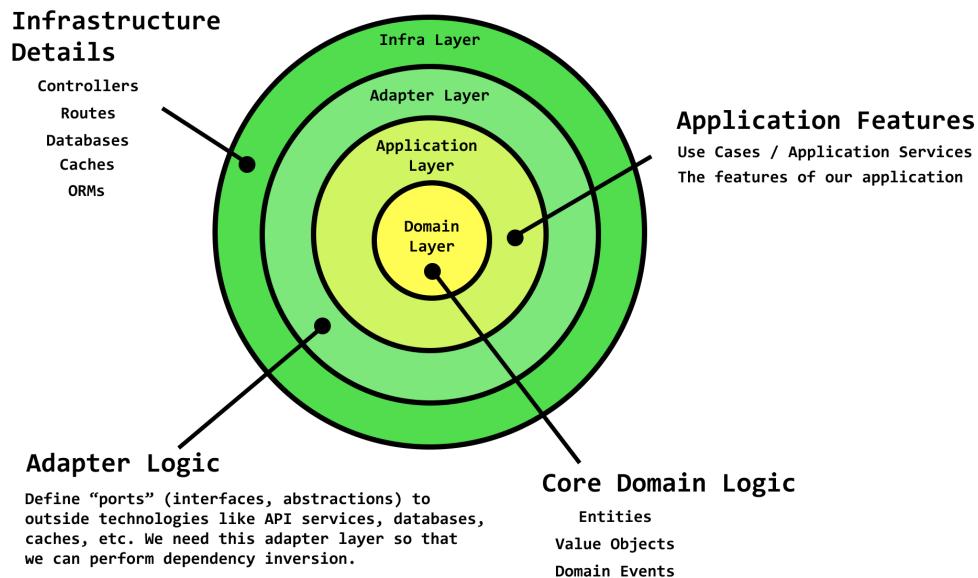


**Users**  
*subdomain*

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.

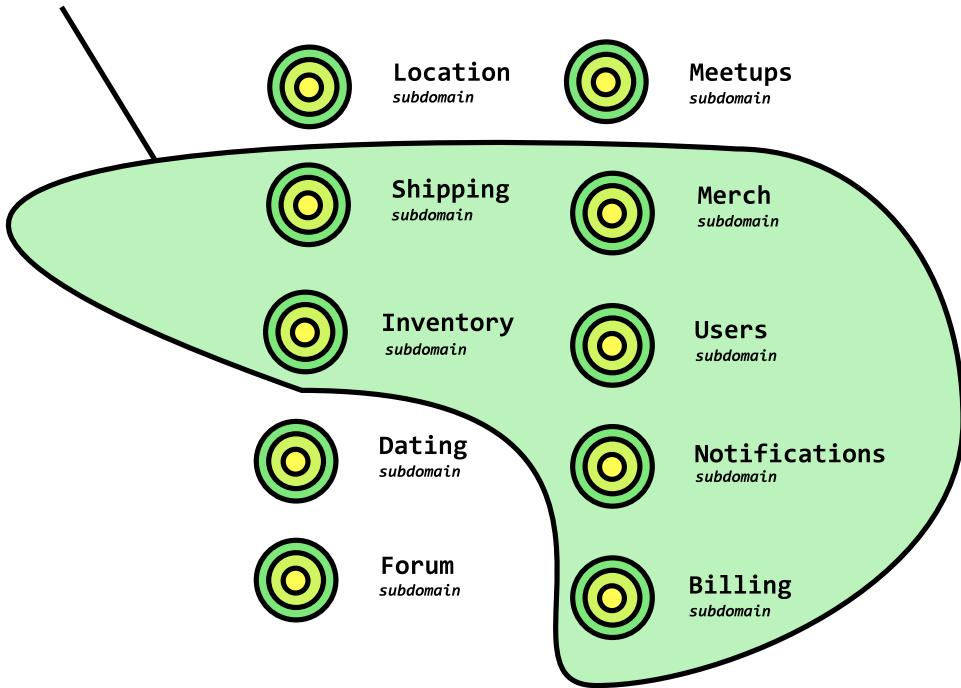


All subdomains in the DDD-apps enterprise.

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.

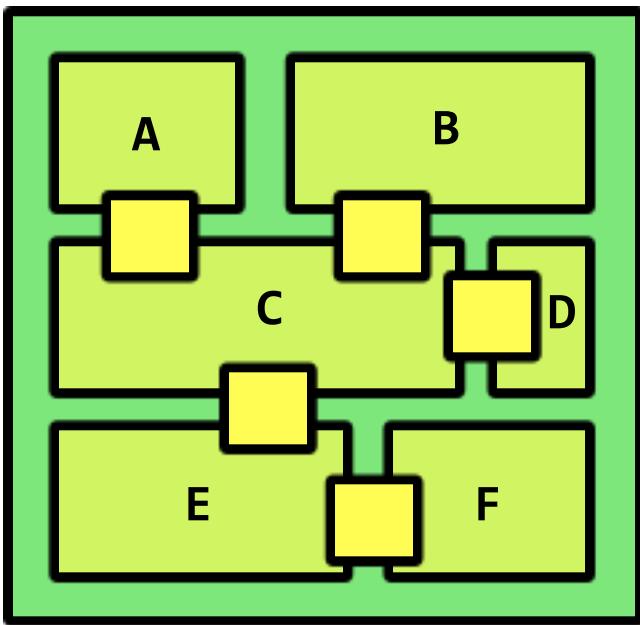
If we were to start building an application as a monolith, at the *folder* level, we could enforce those *Subdomain* boundaries by using the **package by module** principle.

```

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.

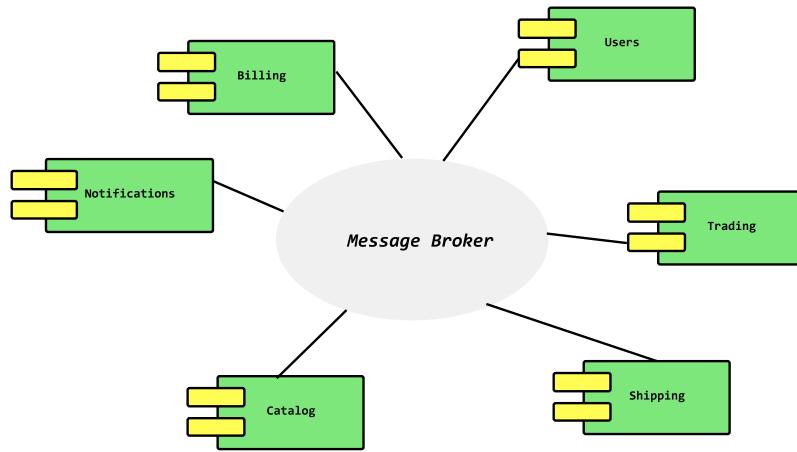
If we modeled **one of the DDD-apps** like DDDMerch as a *Modular Monolith*, what do we do when we want to also deploy another one like DDDDating or DDDForum?

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.

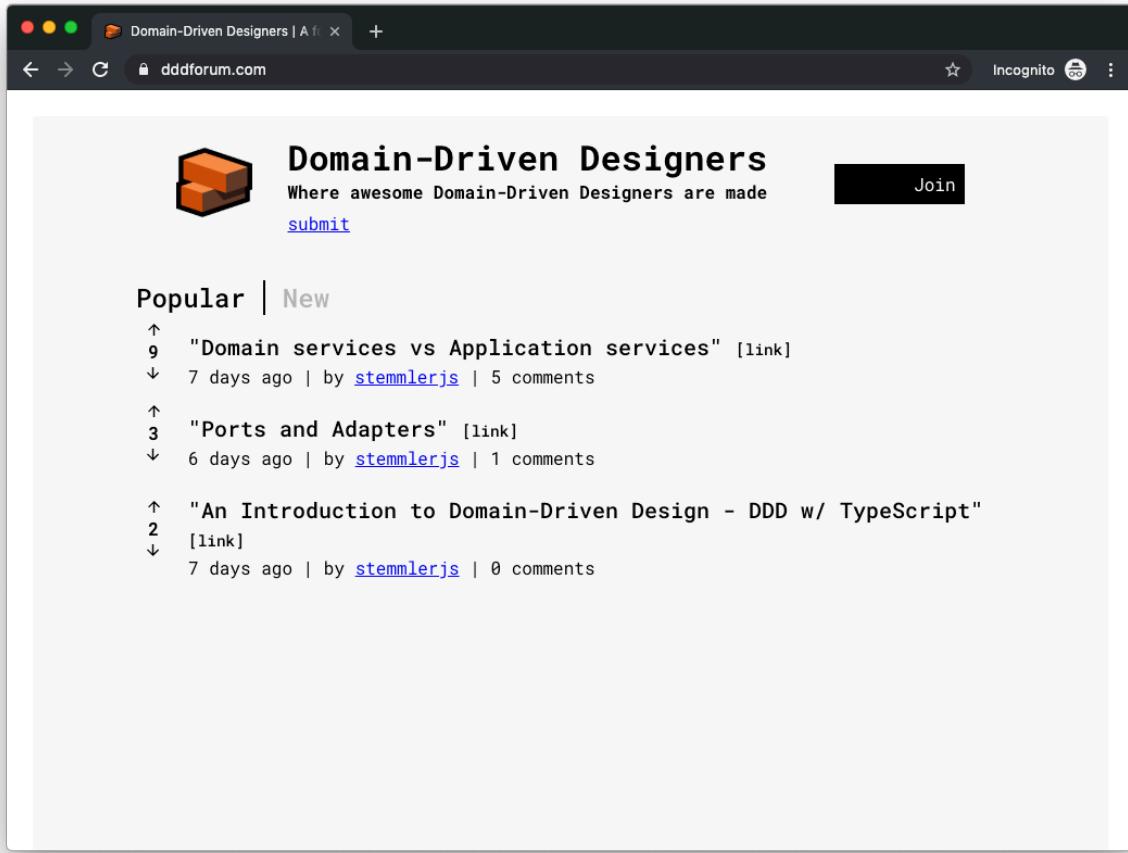
■ **Note:** If we can utilize our existing architecture to build new applications *on top of it with minimal-to-zero* modification necessary, that means we're applying the Open-Closed principle *architecturally*. That right there is a beautiful thing.

## The Project: DDDForum — a Hackernews-inspired forum app

### About the project

DDDForum.com is the sweetest forum-based website on the block (not really). It was created so that members could chat about Domain-Driven Design and learn from others in the community.

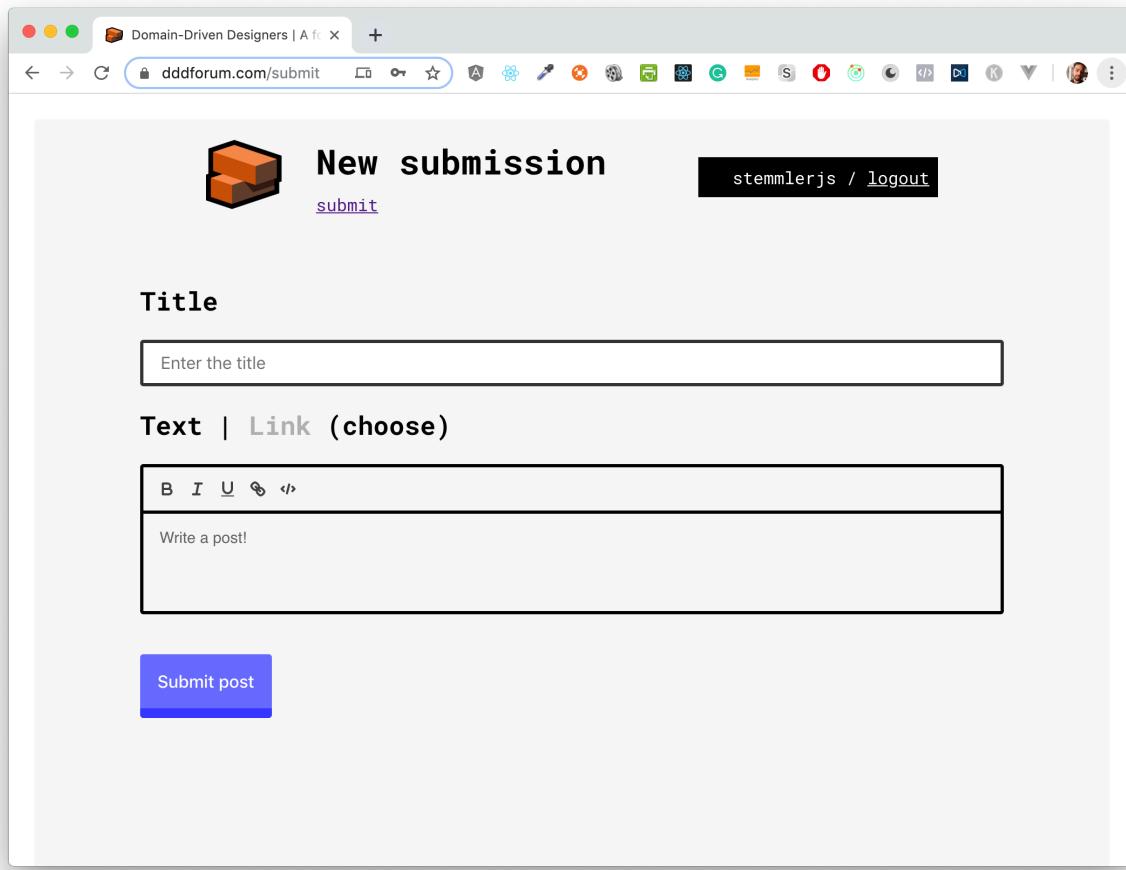
Members can post links, ask questions, share what they're working on, and spark conversations about anything DDD related.



DDDForum.com - a hackernews-inspired forum website to chat about DDD.

If you're familiar with Hackernews, DDDForum.com is pretty much a clone. For those not familiar with Hackernews, check it out first.

Once a user is signed up, they can create a *post* (also sometimes called a submission).



Submission page - a submission can be either text or a link.

In order to submit a post, it must contain a title and either a *text* (for posts where the member just wants to start a conversation) or a *link* (in order to promote or start a conversation on a piece of content).

When a post gets created, the post starts with an *automatic upvote* from the member it belongs to (hey, you posted it - it's most likely you're a fan of your own content too).

The designer decided it was a good idea to improve UX by ensuring that when a member creates a new post, it starts with a positive score of 1. It was an empathetic design decision that originated from trying to make users feel better knowing their post has value *initially*, rather than starting from 0.

Back to all discussions

stemmllerjs / logout

"An Introduction to Domain-Driven Design - DDD w/ TypeScript"

9 days ago | by stemmllerjs | 0 comments

Click to visit the link at [khalilstemmller.com](http://khalilstemmller.com)

Leave a comment

B I U ↲ ↴

Post your reply

Post comment

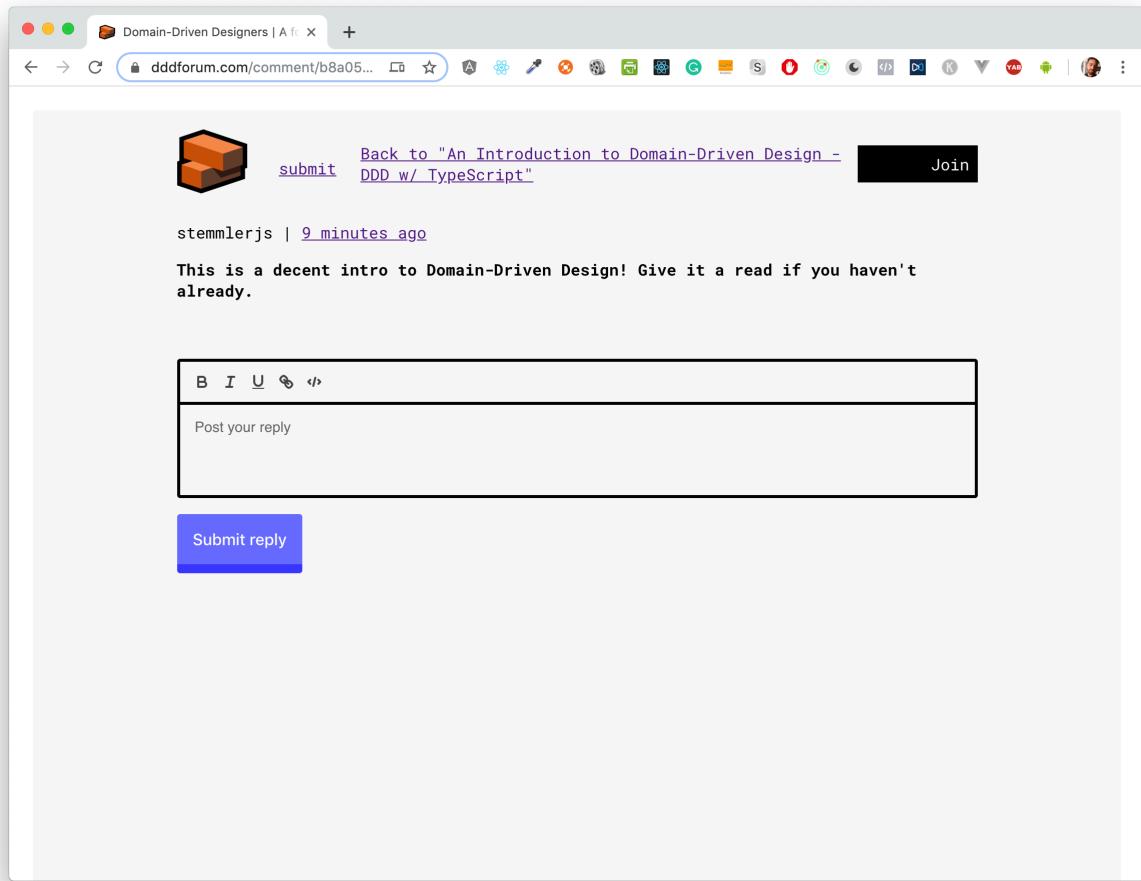
DDDForum.com submitted link post

Any member from the community (including the member who created the post) can cast a *vote*, post a *comment*, and cast *votes on comments* as well.

A screenshot of a web browser window displaying a comment section. The URL in the address bar is [dddforum.com/discuss/132594...](http://dddforum.com/discuss/132594...). The main content is a post titled "Driven Design - DDD w/ TypeScript" by [stemmlerjs](#), posted 9 days ago. A blue button at the top of the post says "Click to visit the link at [khalilstemmller.com](http://khalilstemmller.com)". Below the post is a "Leave a comment" section with a text area and a "Post comment" button. A reply to the post is shown in a box, posted by [stemmlerjs](#) a few seconds ago. The reply text is: "This is a decent intro to Domain-Driven Design! Give it a read if you haven't already." A "reply" button is located below the reply text.

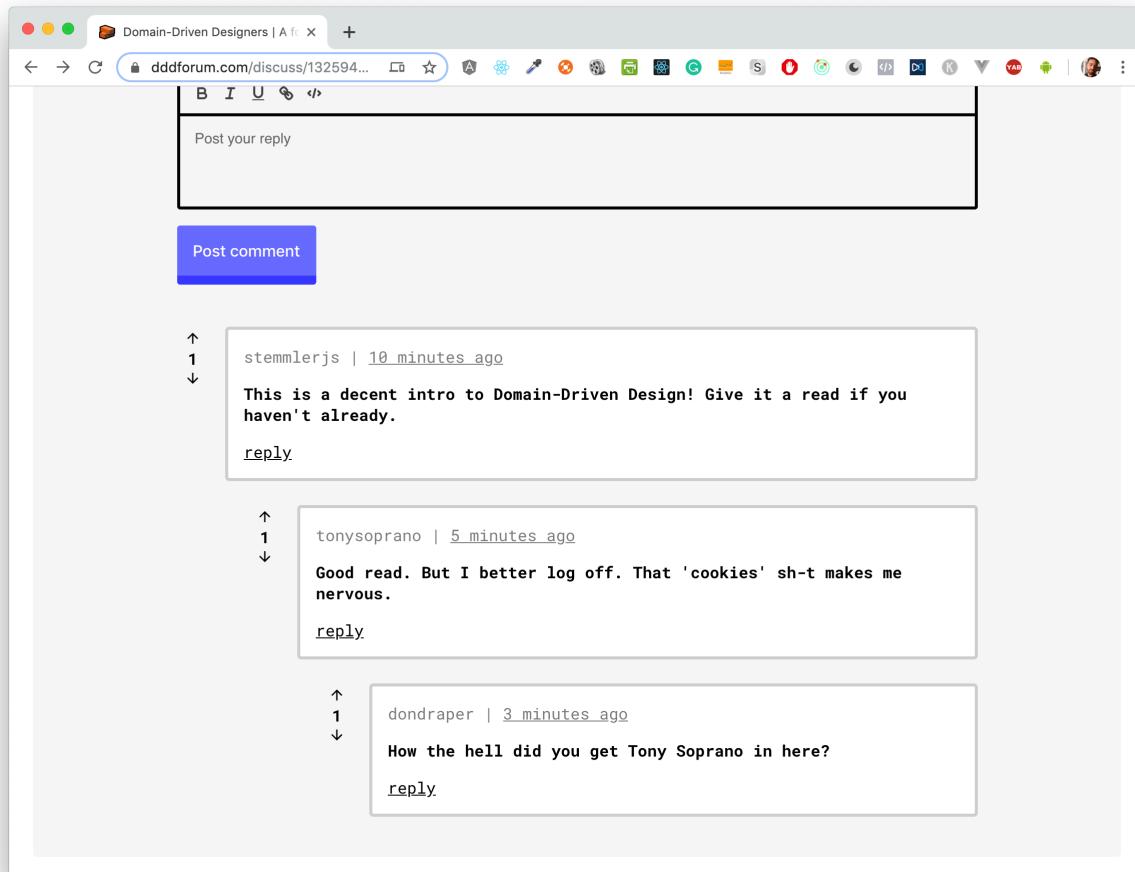
Post with a comment on it.

Not only can you write a comment against a post, but you can also post a *reply* to a comment. Clicking the “Reply” button takes you to the thread page for that comment where you can write your reply.



Thread page for a comment on the “Introduction to DDD post”. Use this page to post replies to comments.

Comments can be nested in - so when you reply to a comment, and someone replies to your reply, it creates a nested thread of replies.



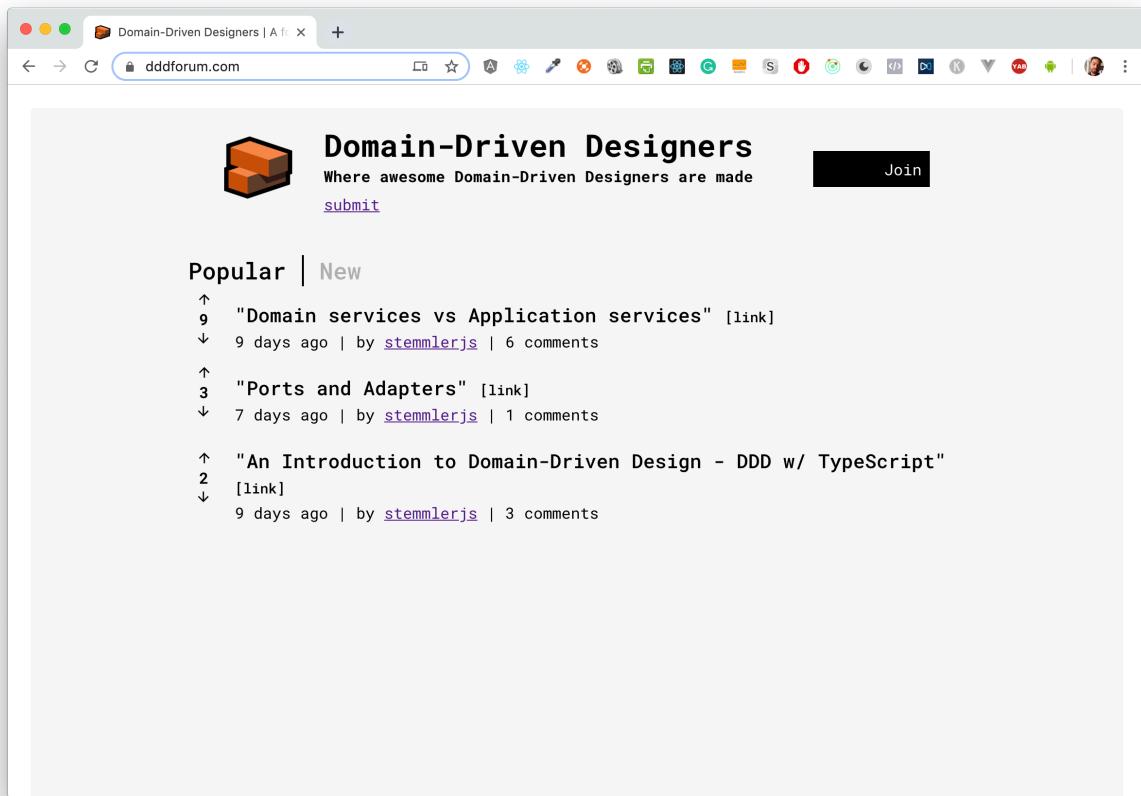
Nested replies on the post.

Most importantly, every post has a score to it. The score is calculated by the following simple equation:

$$(\# \text{ of post } \textit{upvotes} + \# \text{ of } * \text{comment } \textit{upvotes}) - (\# \text{ of post } \textit{downvotes} + \# \text{ of comment } \textit{downvotes})$$

**Only comment upvotes from *other members*, not the original author of the comment, are included.**

For now, the post with the highest score is shown at the front of DDDForum.com in the popular section.



The post with the most points is shown at the top of the popular page.

Anonymous visitors to DDDForum.com can view everything that members can view, but they need to become a member in order to submit posts, comments, and cast votes.

## The code

■ **View the code:** You can see the code in its completion here on GitHub via [@stemmlerjs/ddd-forum](#). I suggest you fork it and take a look around.

## 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.

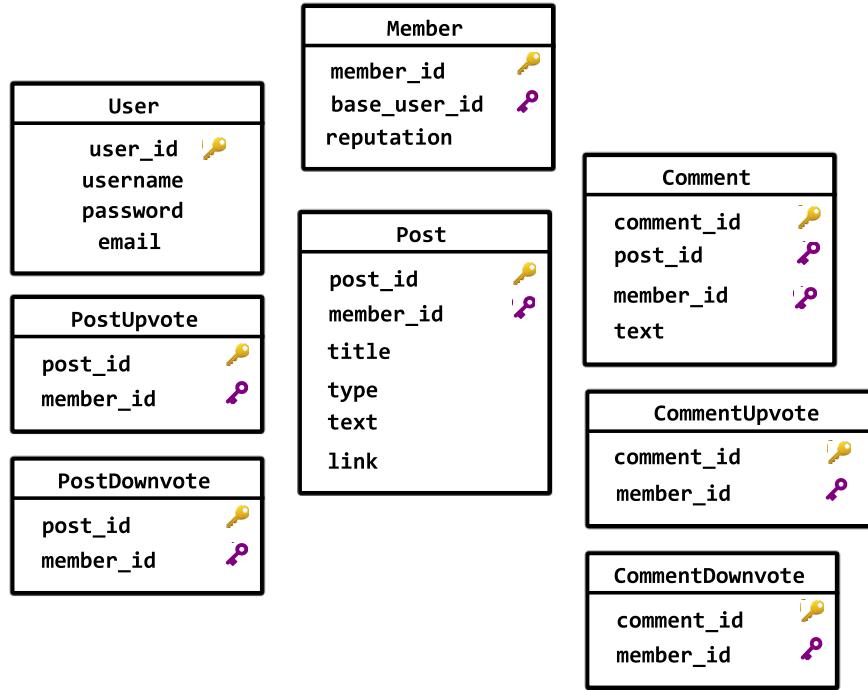
## Imperative design

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.

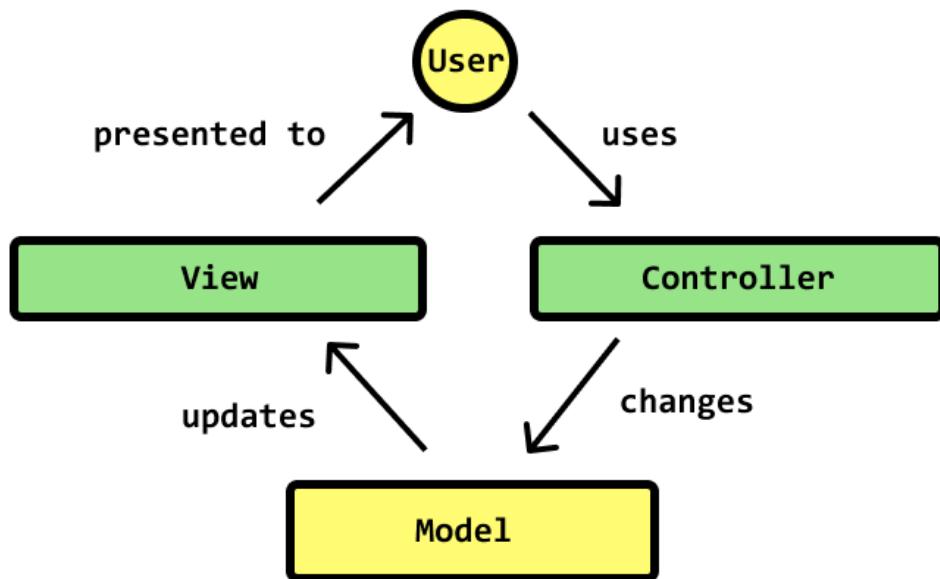
Doing design **UI-first** means that we start by creating all of the wireframes for the system. Then, we discover the API calls we need based on the functional requirements of the components. Additionally, the shape of the database is determined based on fields and relationships present in the UI.

**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.

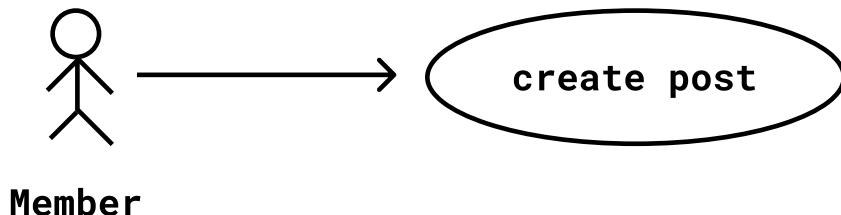
## Use cases & actors

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:

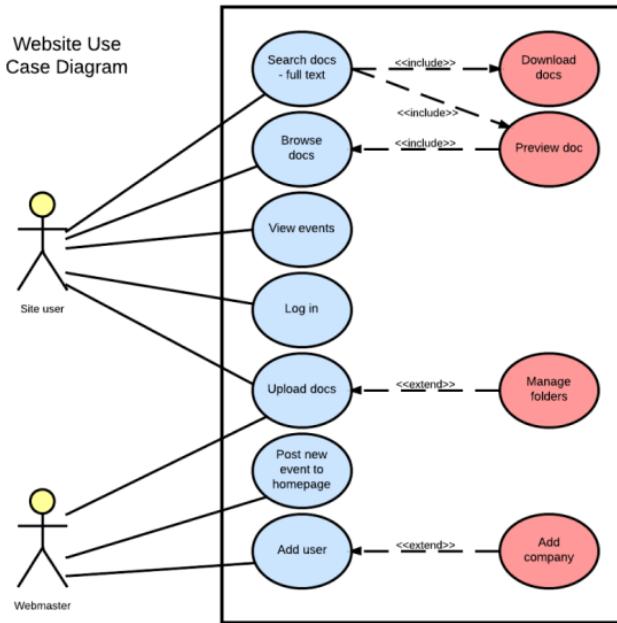
- postVinyl (details: VinylDetails): void
- getVinylDetails (vinylId: VinylId): VinylDetails
- makeOffer (tradeItems: TradeItems[]): void
- etc

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.

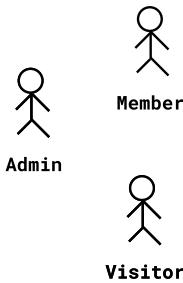


Member



Visitor

There's another type of role in this domain that could also be important, and that's the Admin role.



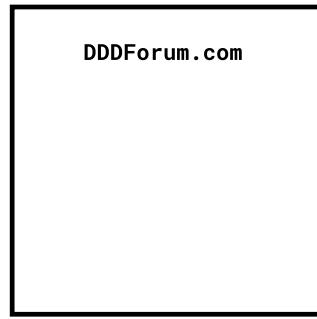
## 2 — Identifying the actor goals

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.

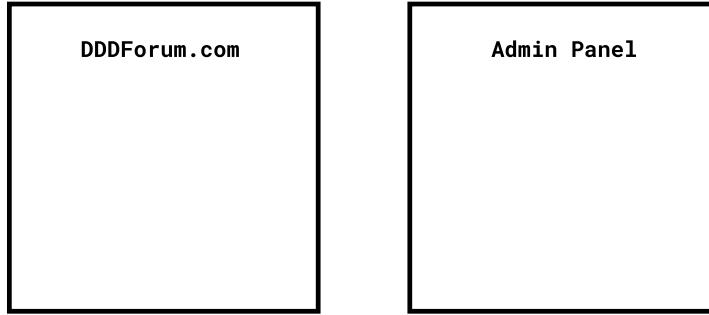
For Admin users, their goal is to **ensure that the DDDForum community is welcoming, helpful, and respectful**. To do that, they need the ability to **moderate users and content posted within the site**.

## 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.

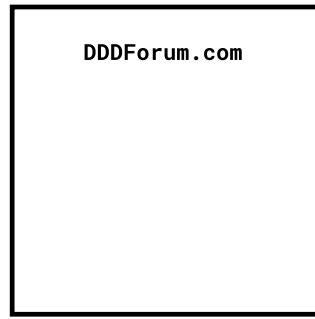


For Admin users, the *admin panel* could exist to serve their moderation needs.

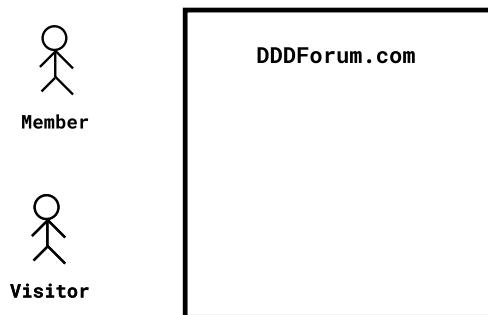


To contain the scope of our discussion in this chapter, we're just going to focus on DDDForum.com for Members and Visitors, leaving out the admin panel and Admins for now.

- We might explore including the Admin Panel in this chapter in a future release of this book.



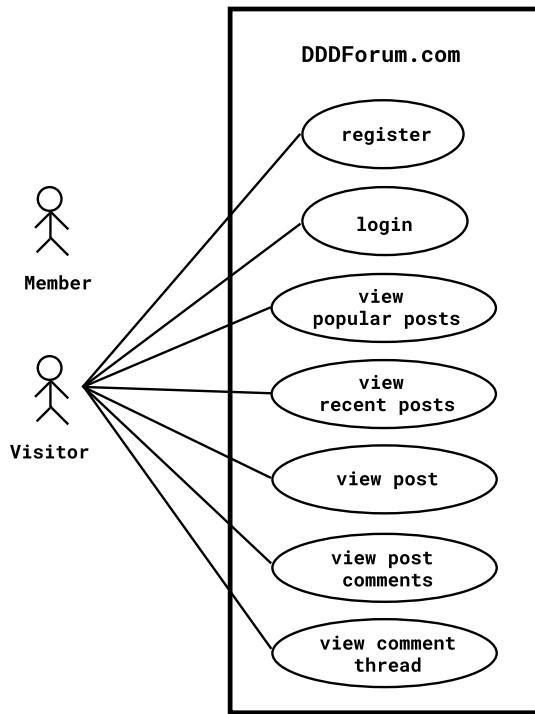
OK, so the DDDForum.com is a system that is going to serve the needs of Members and Visitors.



What can they do within the system? Let's think of a few use cases.

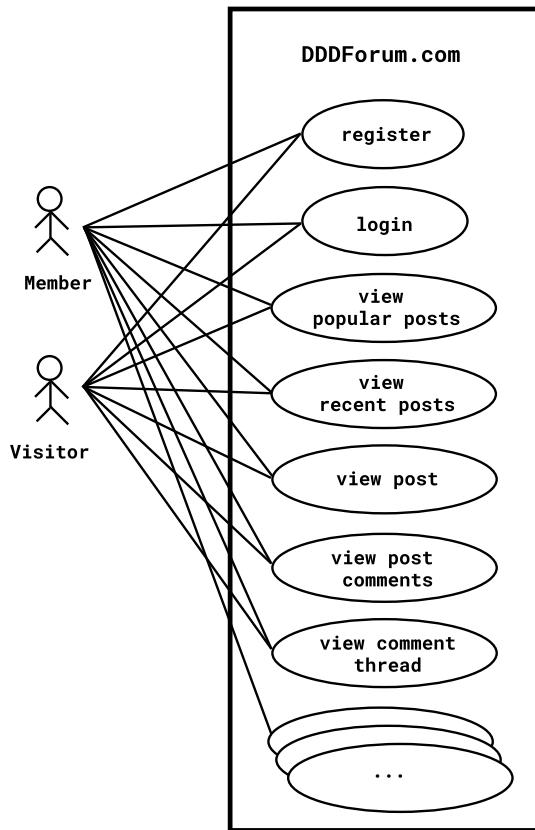
#### 4 — Identifying the use cases for each role

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.

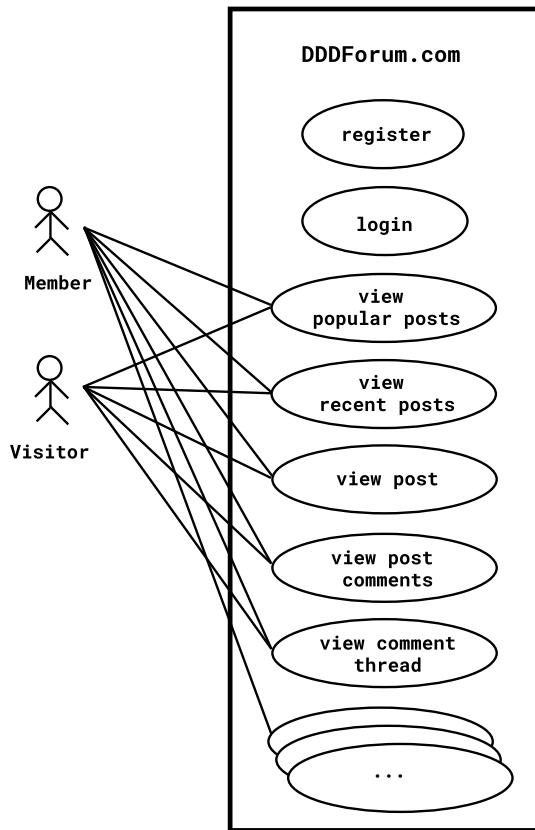


Remember that each of these use cases are either a **command** or a **query**. Try to identify which are which.

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.



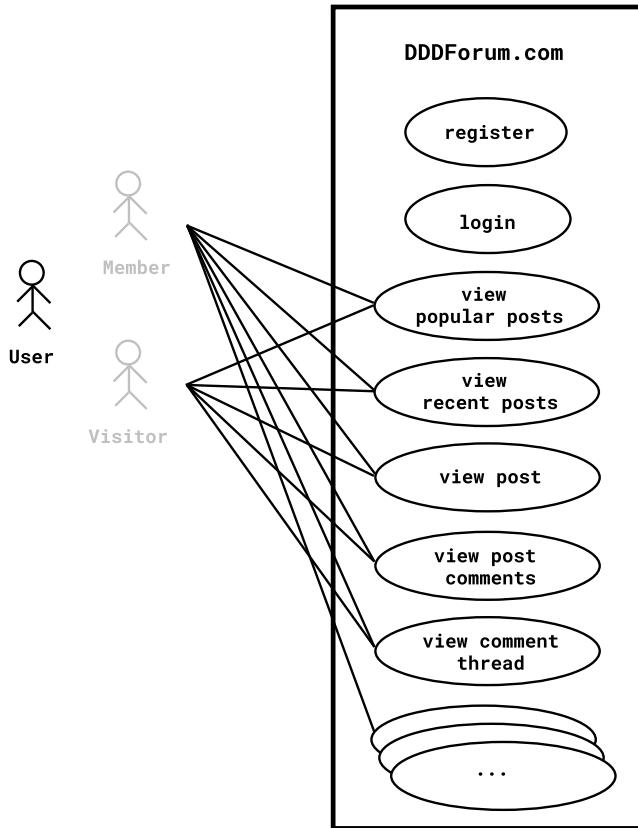
Why did I do that? Well, it's *complicated* and long-winded, but it has to do with *role, boundaries*, and, most importantly- *Conway's Law*. The detour is worthwhile (trust me).

### **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 modelling efforts are 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.

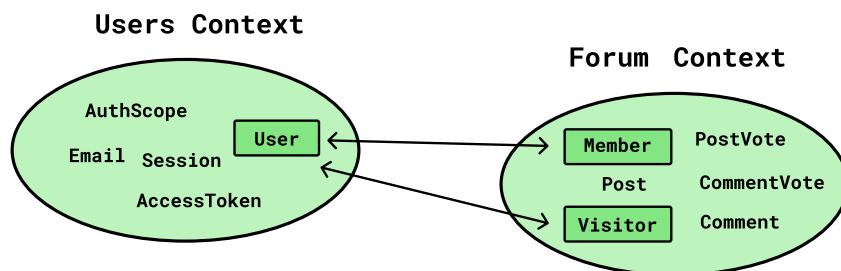
Normally, you'd work collaboratively with domain experts to identify the common language.

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 Cognito, 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:

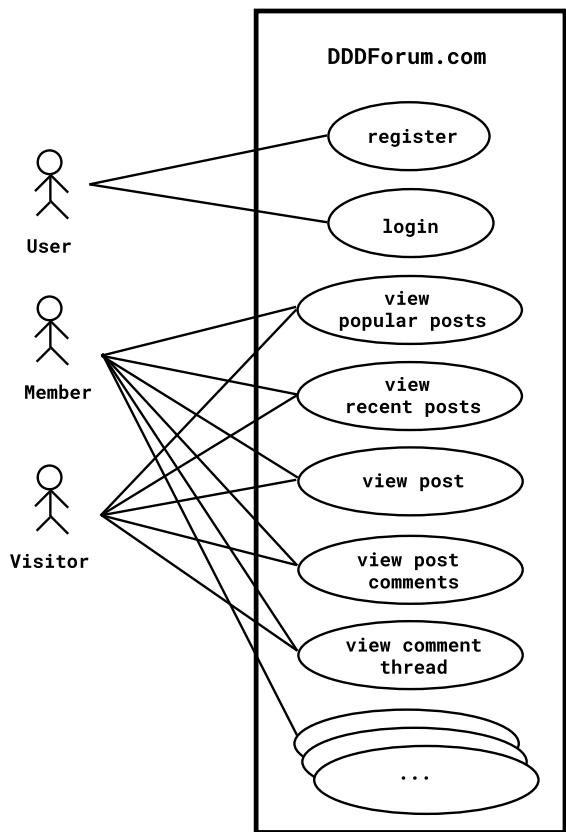
- A billing system: Customer, Subscriber, Accountant, Treasurer, Employee
- A blogging system: Editor, Reviewer, Guest, Author
- A recruitment platform: JobSeeker, Employer, Interviewer, Recruiter
- A vinyl-trading application: Trader, Admin
- An email marketing company: Contact, Recipient, Sender, ListOwner

Get the point? **Role** matters. When identifying **actors**, name them based on their *role*.

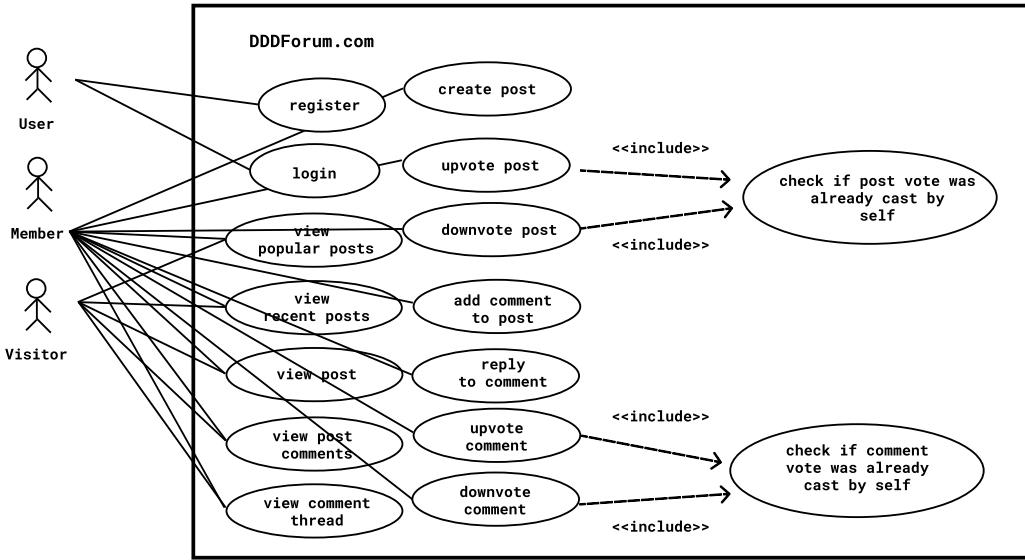
Because role dictates responsibility, sometimes when we uncover a responsibility (like log in and register), we have to ask ourselves if we've assigned it to the correct role. In this case, it's good that we were specific about Members and Visitors because it enabled us to understand the responsibilities 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*

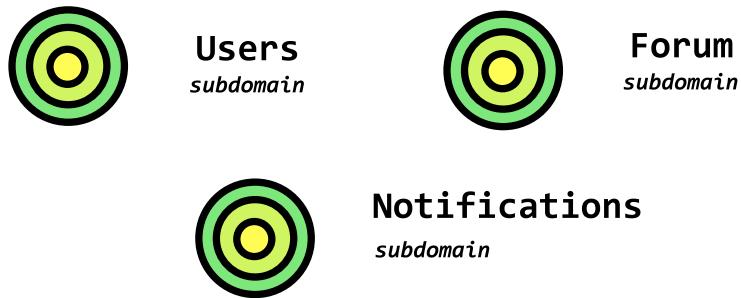


Forum  
*subdomain*

Users (generic) and Forum (core) subdomains for DDDForum.com

We might also have *one more* subdomain for notifications.

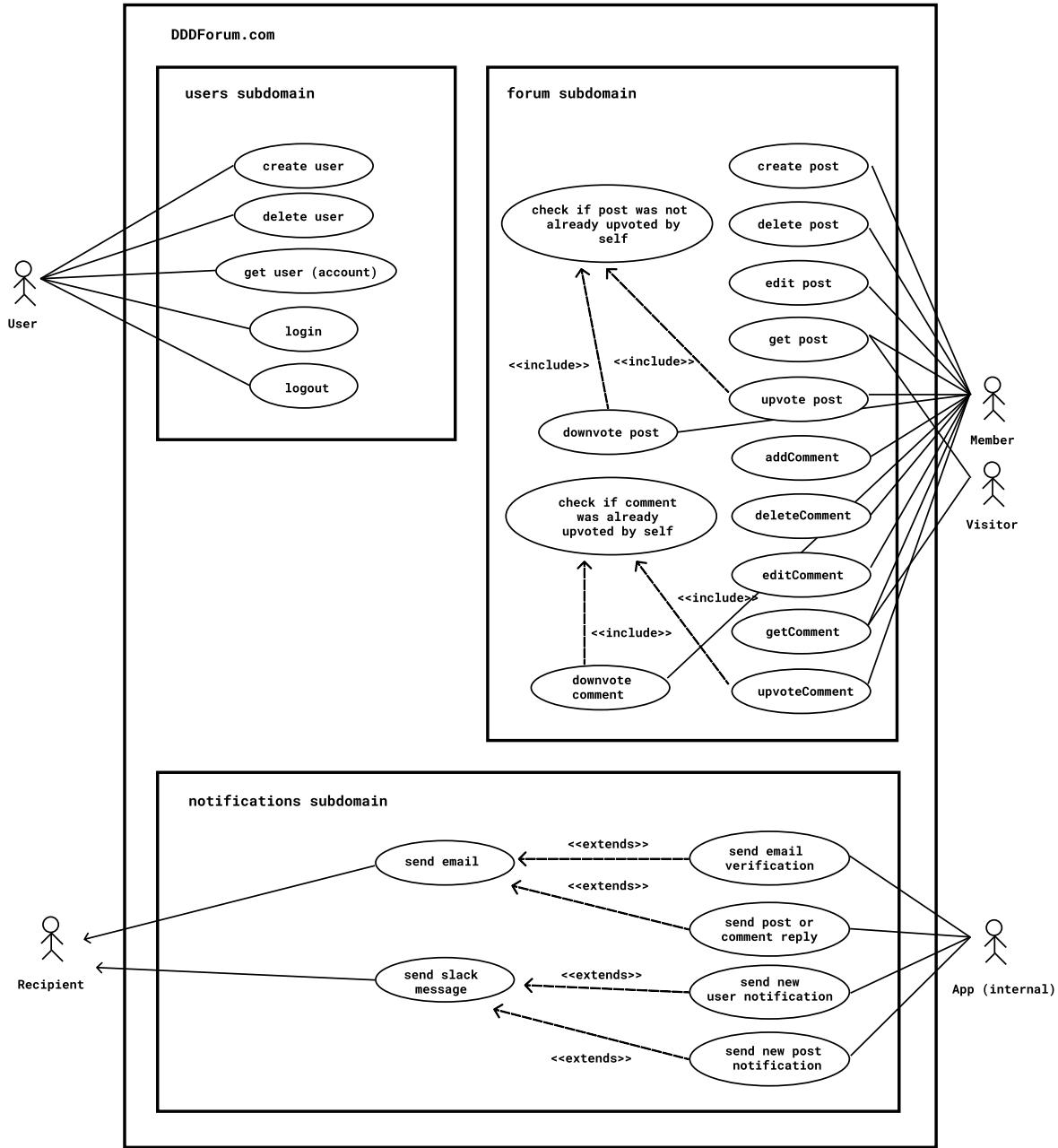
You see, if we're going to be *sending emails* to do *email verification* or if we're going to be sending notifications to members if someone replies to their comment, we need another *supporting* subdomain to decouple the concerns of everything related to emails, notifications, etc.



Users (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*.

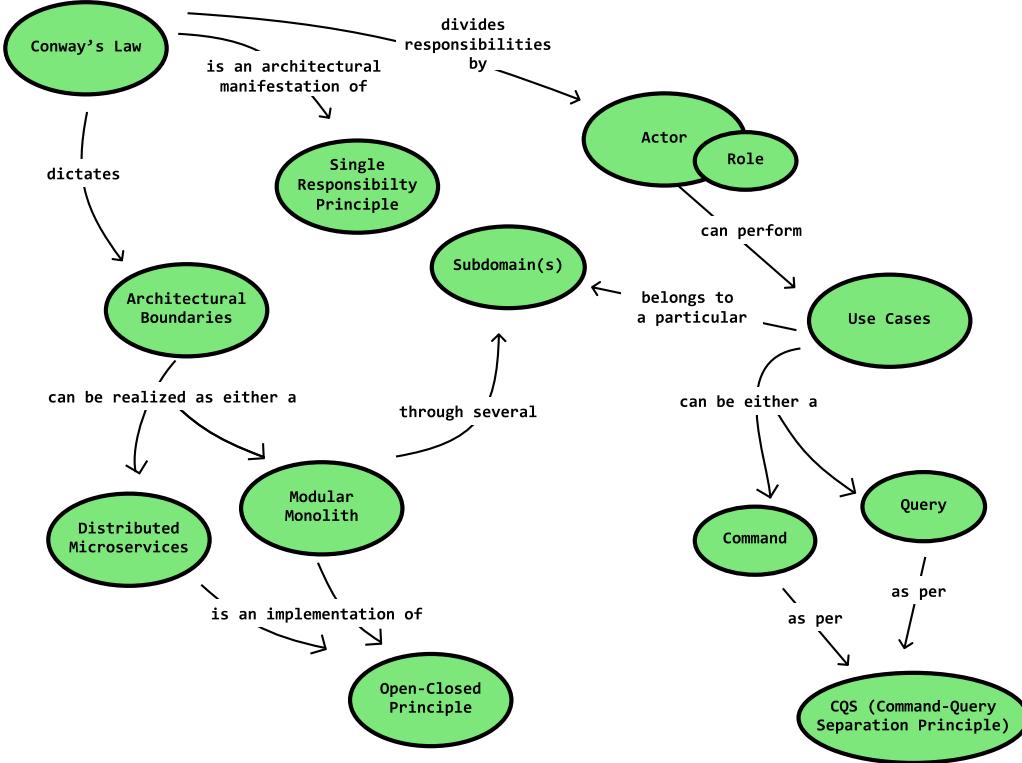
**SRP (Single Responsibility Principle):** Conway somehow discovered that if we allow each group to have their system, we constrain the possible surface area of required changes from one group rippling into another one as side effects, thus satisfying the **single responsibility principle** (before it 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.

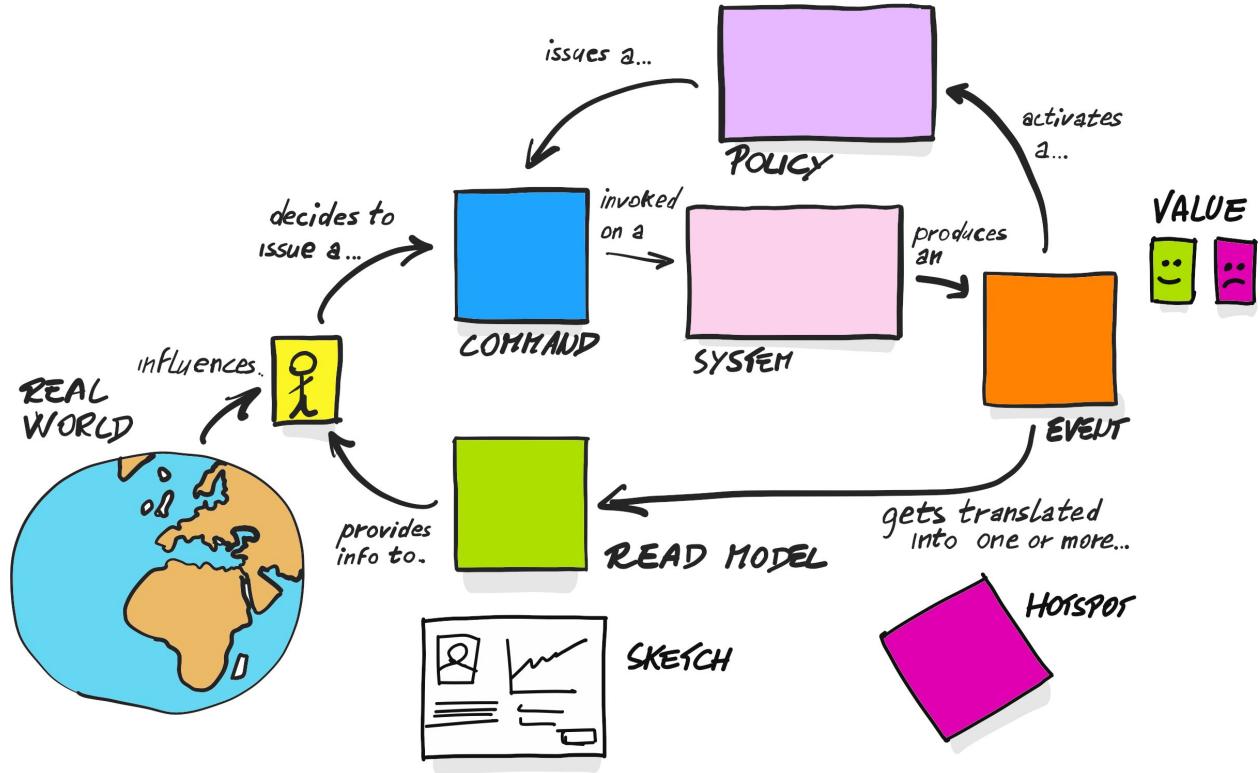
## Why we need event storming

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.

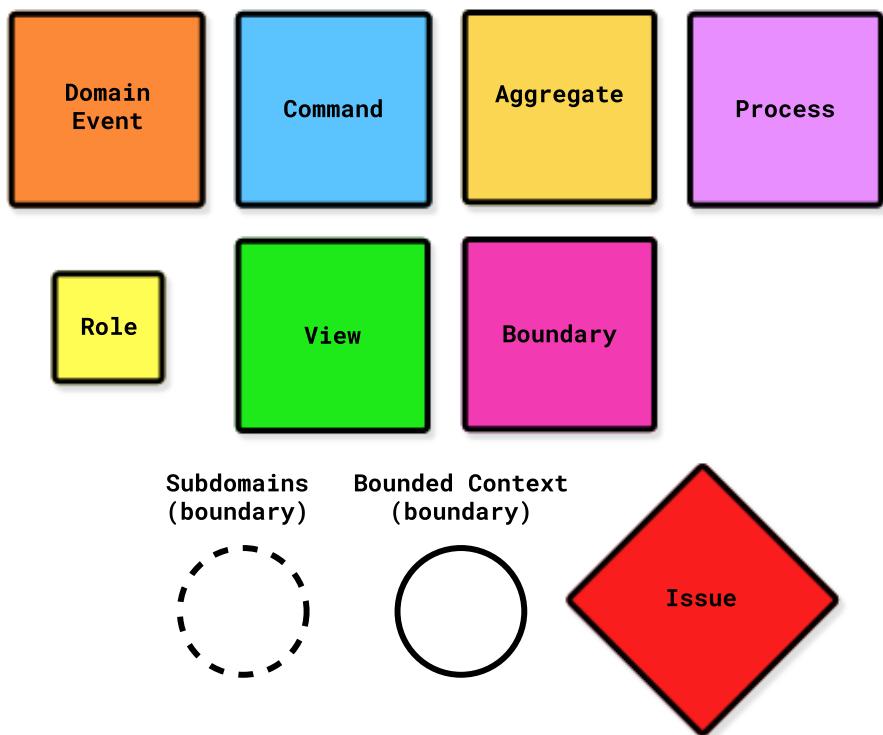
### How to conduct an event storming session

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

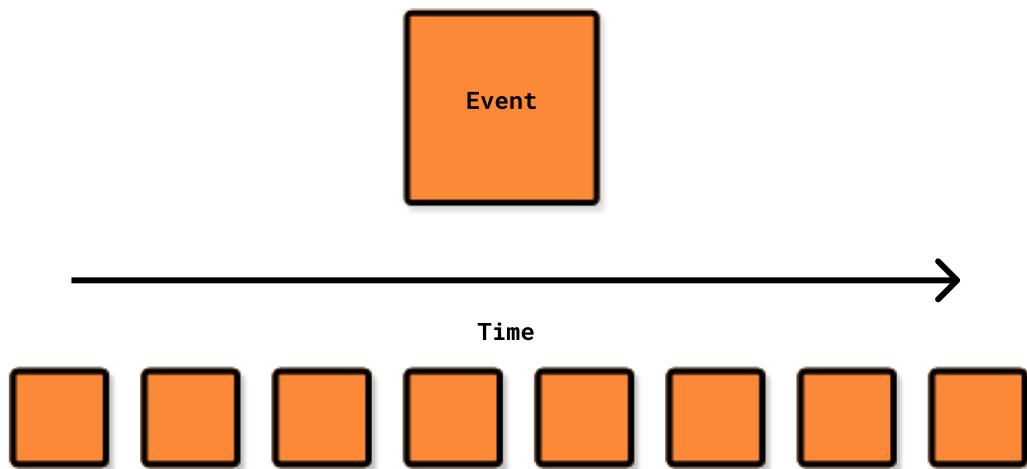
Before we even get started, Vaughn Vernon recommends we use a legend so that the event storming constructs and their color schemes are well understood by all participants.

- 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:

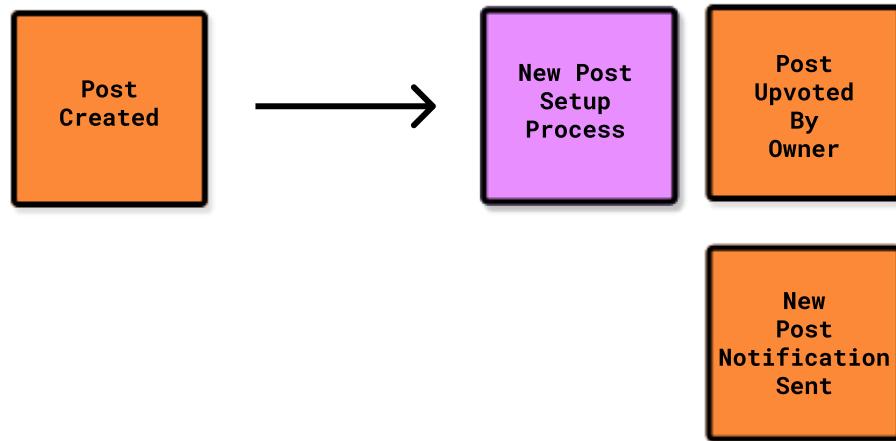
- **Domain events are *past-tense verbs*.** For subsequent steps, it's important to follow this convention. The challenge here is that domain experts often think of business processes as *tasks* rather than events, so we may have to warmly nurse them to get in the habit of using the past-tense format 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.

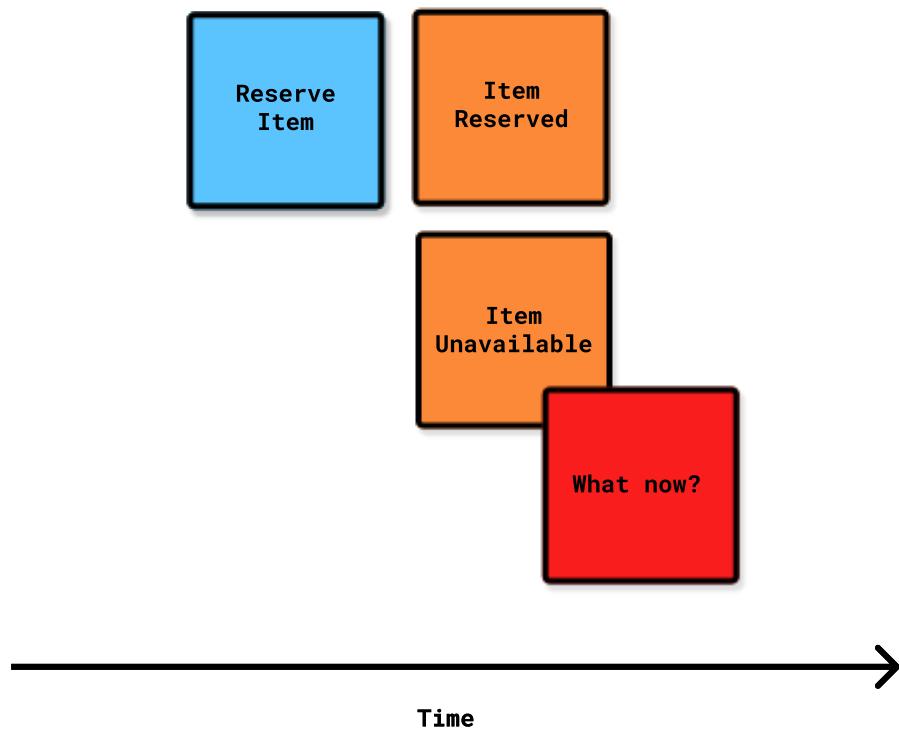
- **It's possible for domain events to be the result of something happening in another system that floats into this one.** In this case, we can still document the domain event.
- **When the outcome of a domain event is a process, document it with a purple sticky.** A process is any scenario where there 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.

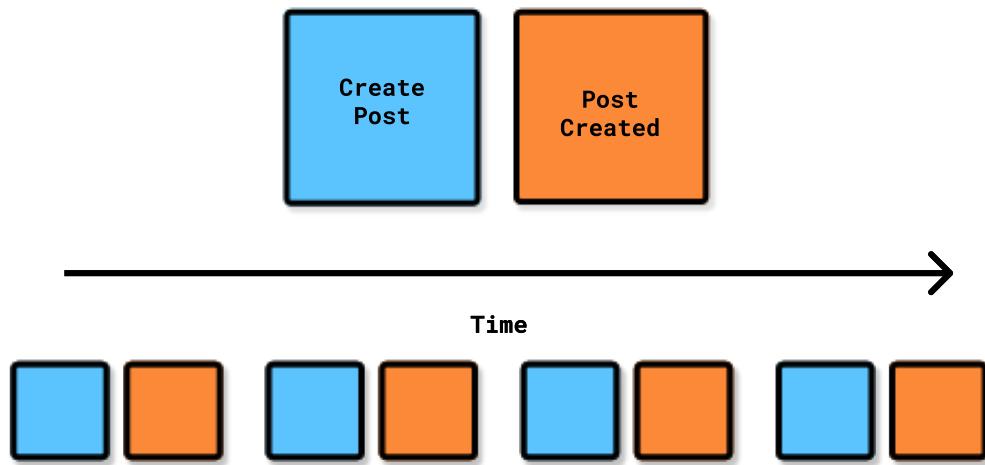


An example of encountering a meaningful alternative domain event that we would like to have a way to handle, but we don't have all the answers for at the moment.

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.



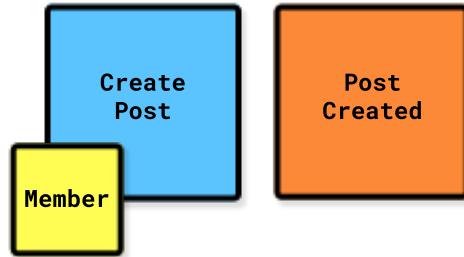
The *command* is exactly the same thing as commands from our UML use case diagrams and the CQS principle. You don't need to have done a UML use case diagram or anything in order to do this step. You can go through each domain event, and place the name of the command that creates the domain event, *before* 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.



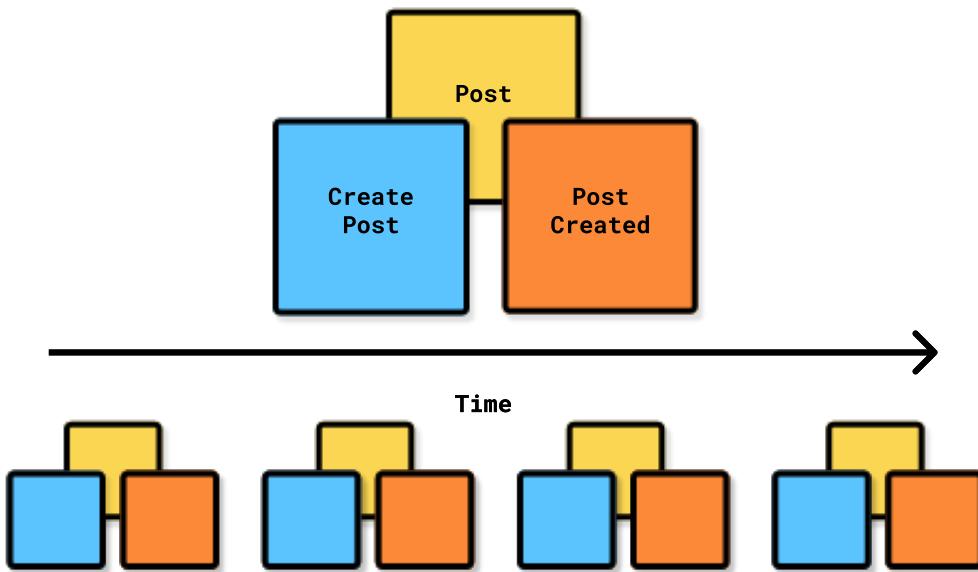
Example of documenting the *Actor/Role* for a command.

- 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* are 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 value of event-based design approaches like event storming is that we focus on the *essential complexity* first. **The behavior is more important than what we're calling the data models that represent that behavior.**

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).

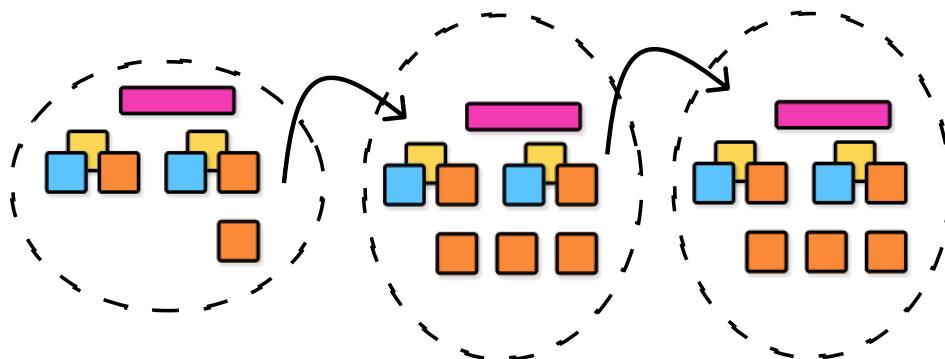
For this step, here are a few notes:

- **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 discover 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.



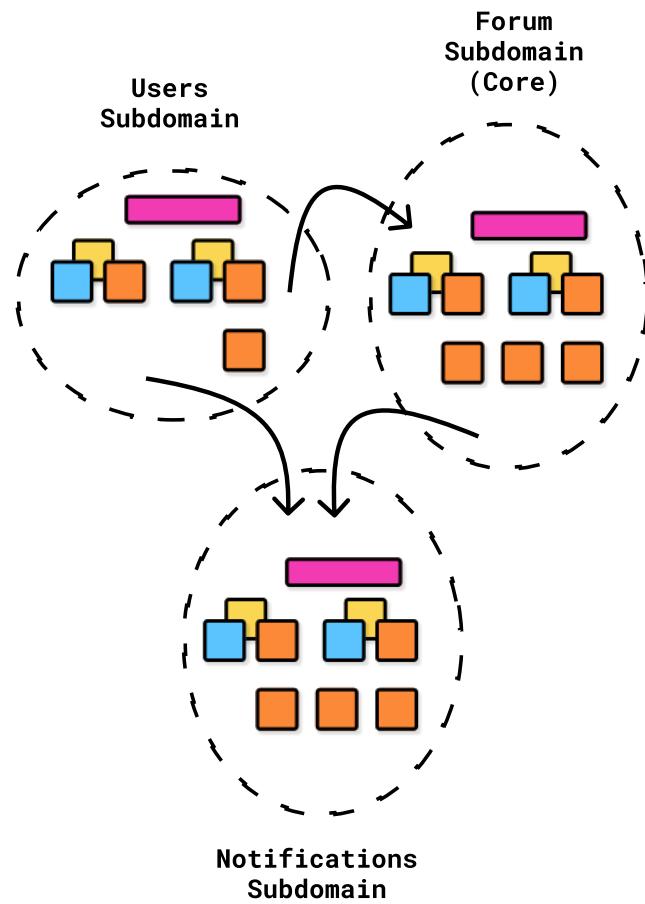
This step is asking you to apply **Conway's Law**.

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.**

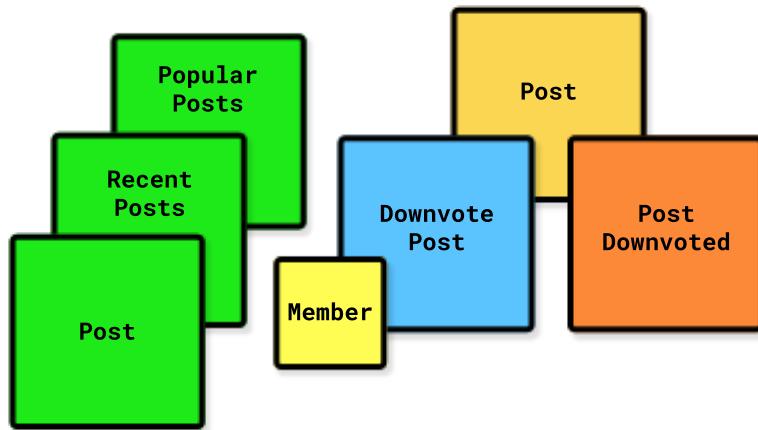
When we apply this step to DDDForum.com, we should end up with a similar diagram to the one we created when applying Conway's Law to our Use Case diagrams.



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*.



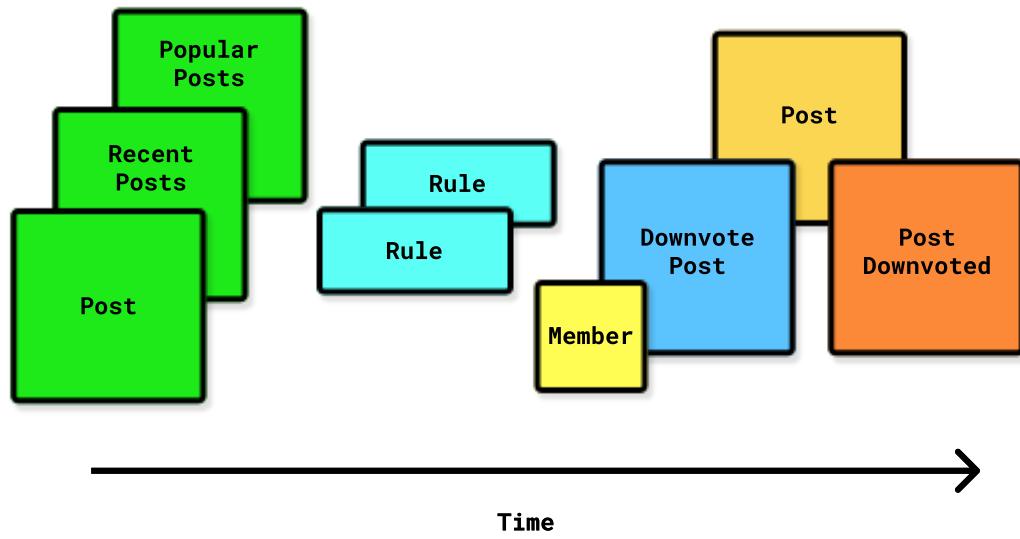
- If there are important roles, make sure to list them here.
  - (Optional) Create wireframes or mockups to illustrate what the views might look like. You want to do whatever is going to be most useful for you and everyone else to understand what's necessary for the model.
- 

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.

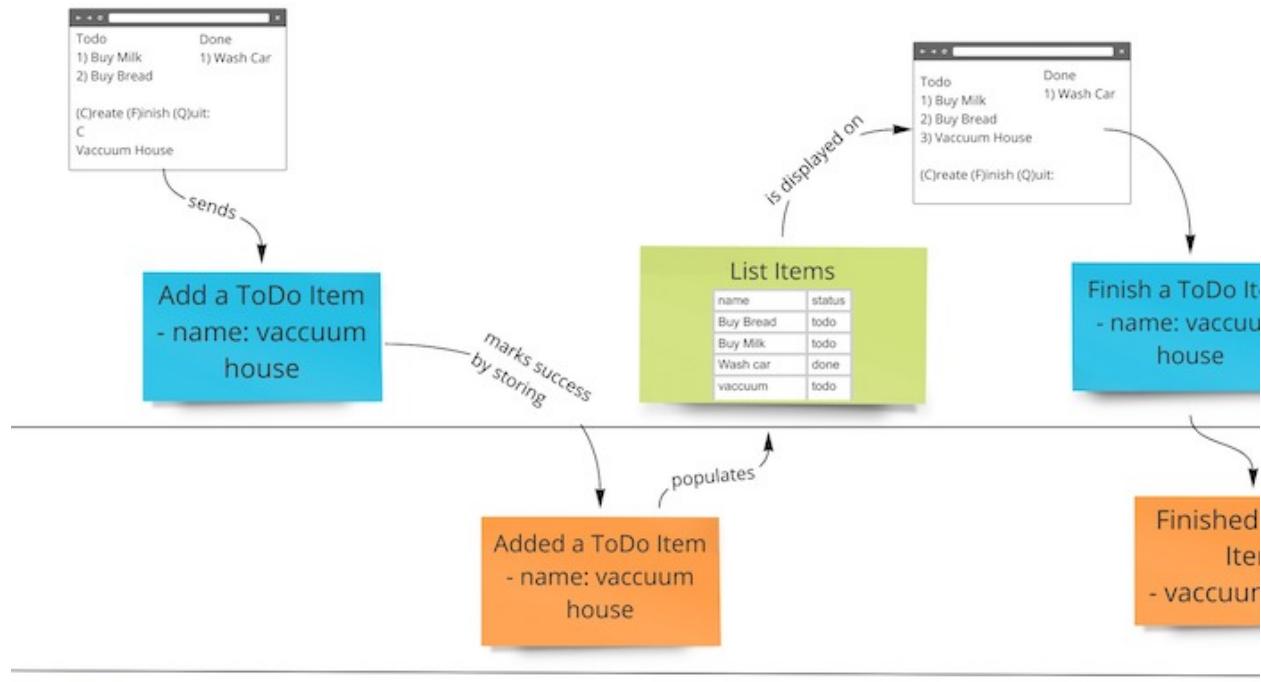
These rules are what we'll end up testing using examples when we do BDD-style acceptance testing using the **given-when-then** format.



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](http://eventmodeling.org).

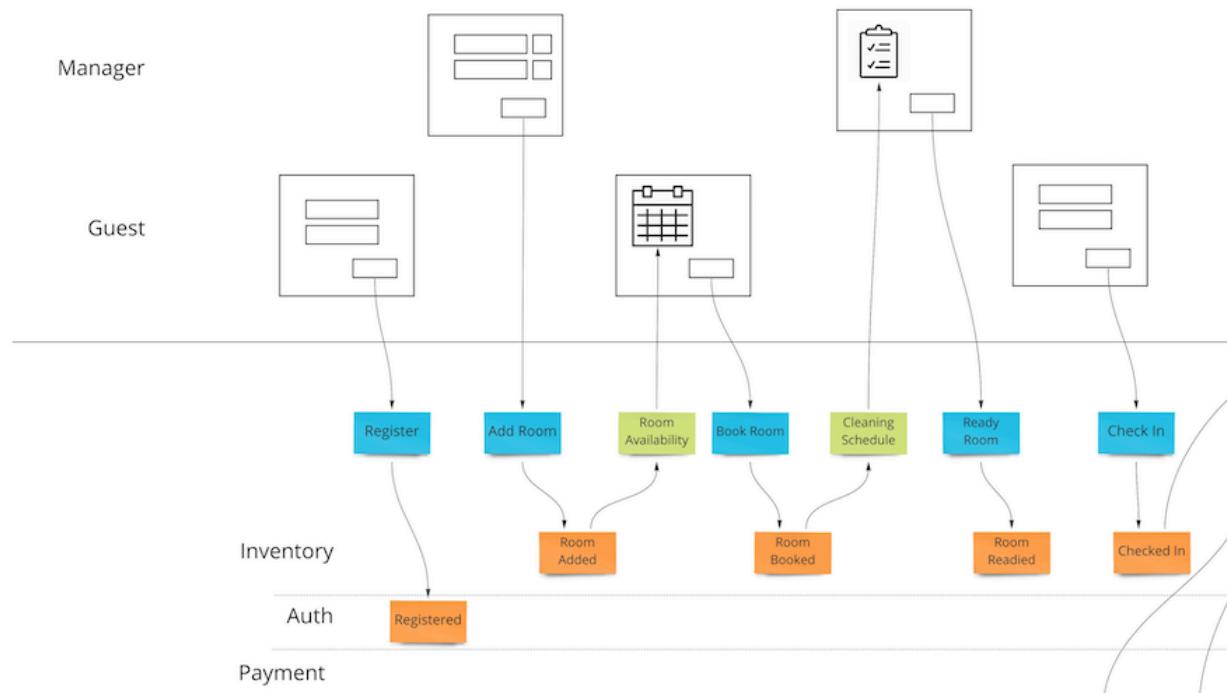
There's not much *new* about *Event Modeling*. It's more of a **formalization** of all the knowledge we've acquired about building event-driven systems.

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](http://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](http://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 patterns

That shouldn't be a surprise to you at this point in the chapter. Using Domain-Driven Design patterns 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.

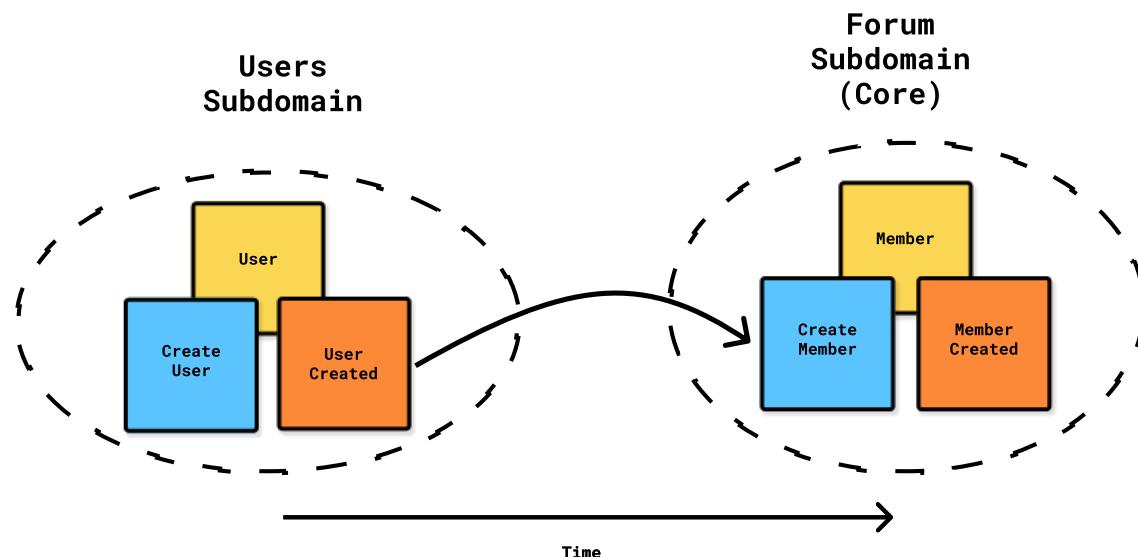
Because DDDForum.com is relatively simple, and because I don't want to expand too far past the scope of this section, we won't be implementing Event Sourcing today. All of what we'll learn how to do in DDD will be applicable when we finally get to Event Sourcing. And these are the basics. Let's start 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,

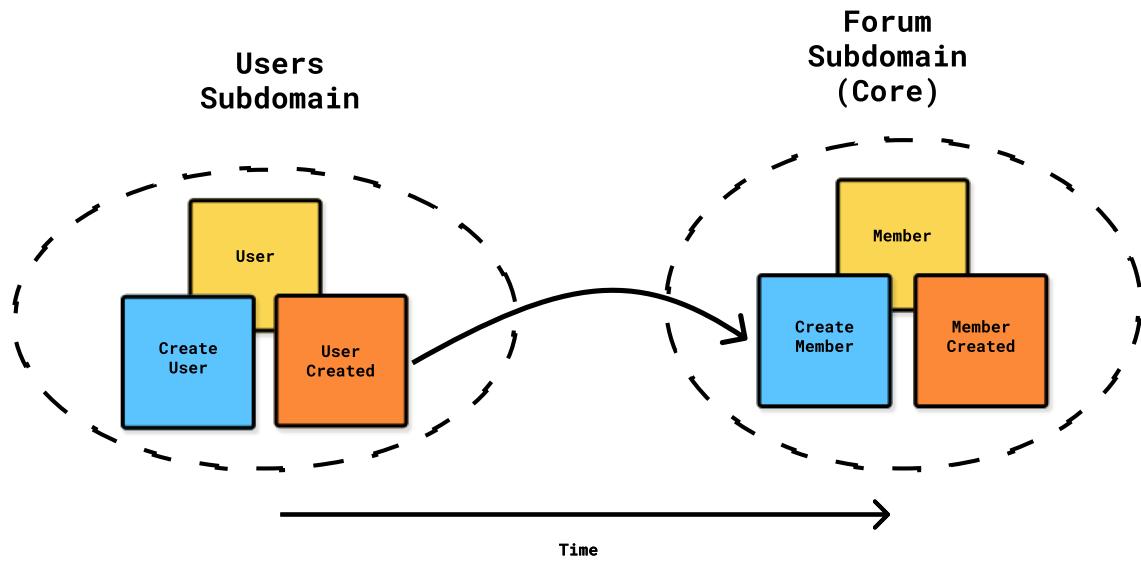
Why not use Node.js *EventEmitter*?

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
  }
}
```

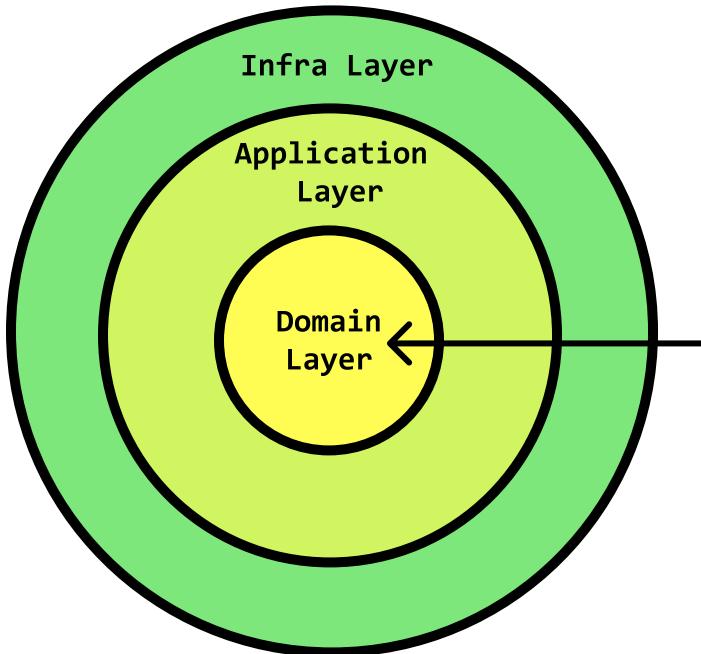
Nonetheless, because we're using a *Hexagonal Architecture*, the *type of API* that we use doesn't affect how our command works. It just affects *what calls our command*. Let's not forget that GraphQL and RESTful APIs, while we can spend a lot of time on their design, are simply API styles that belong 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.

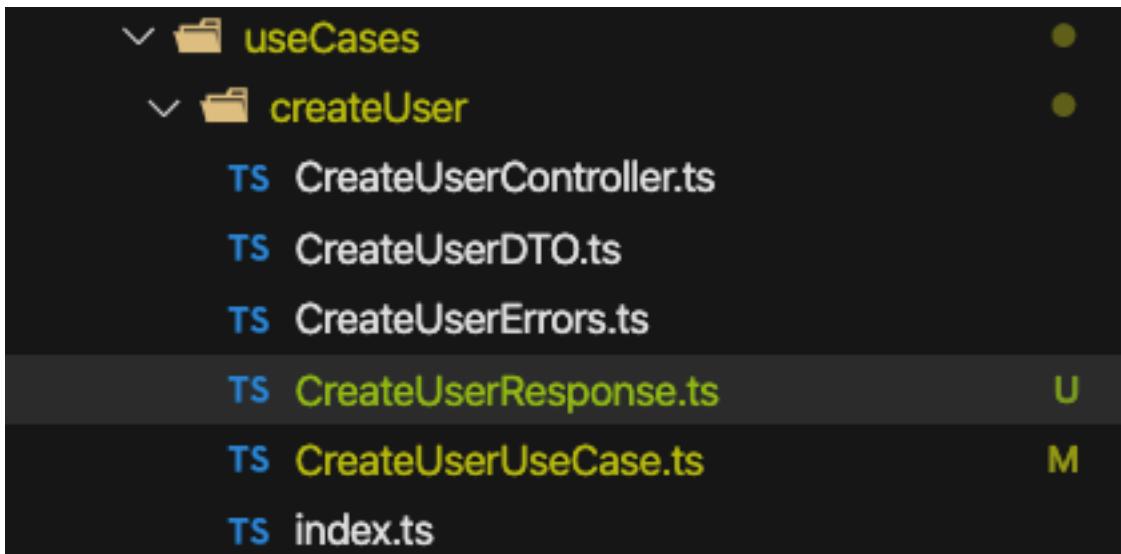
Read those first, then continue.

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
}

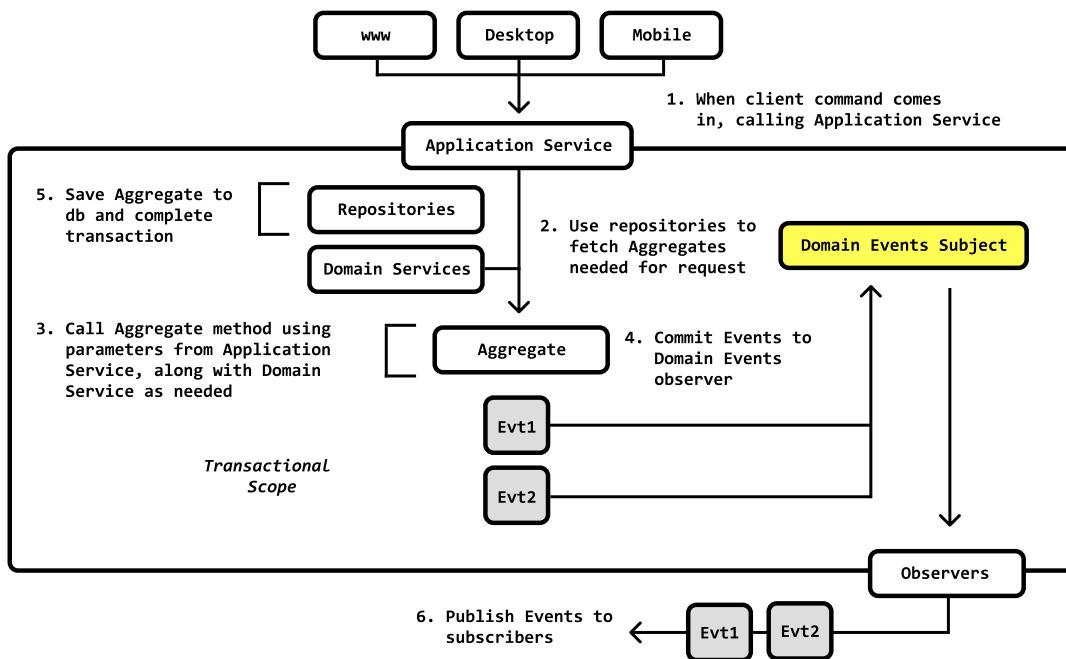
```

**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.

- I. 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

                /**
                 *
                 * @param error
                 */
                return res.status(400).json(error)
            }
        } catch (error) {
            console.error(`Error executing CreateUserUseCase: ${error.message}`)
            return res.status(500).json({ message: "Internal Server Error" })
        }
    }
}
```

```

    * 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 *Inter-*

*face 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)
}

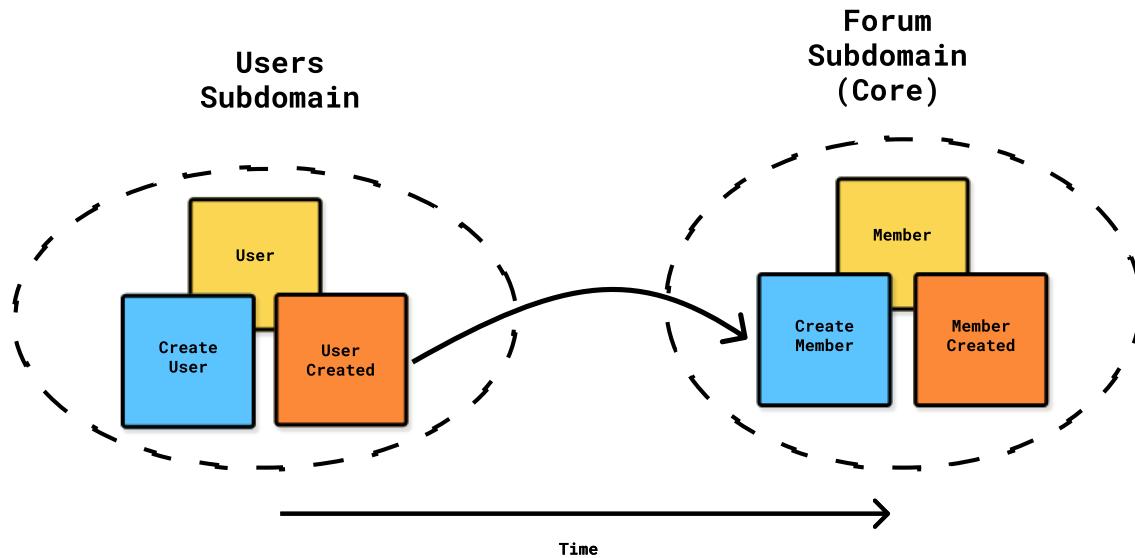
member = memberOrError.getValue()

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
```

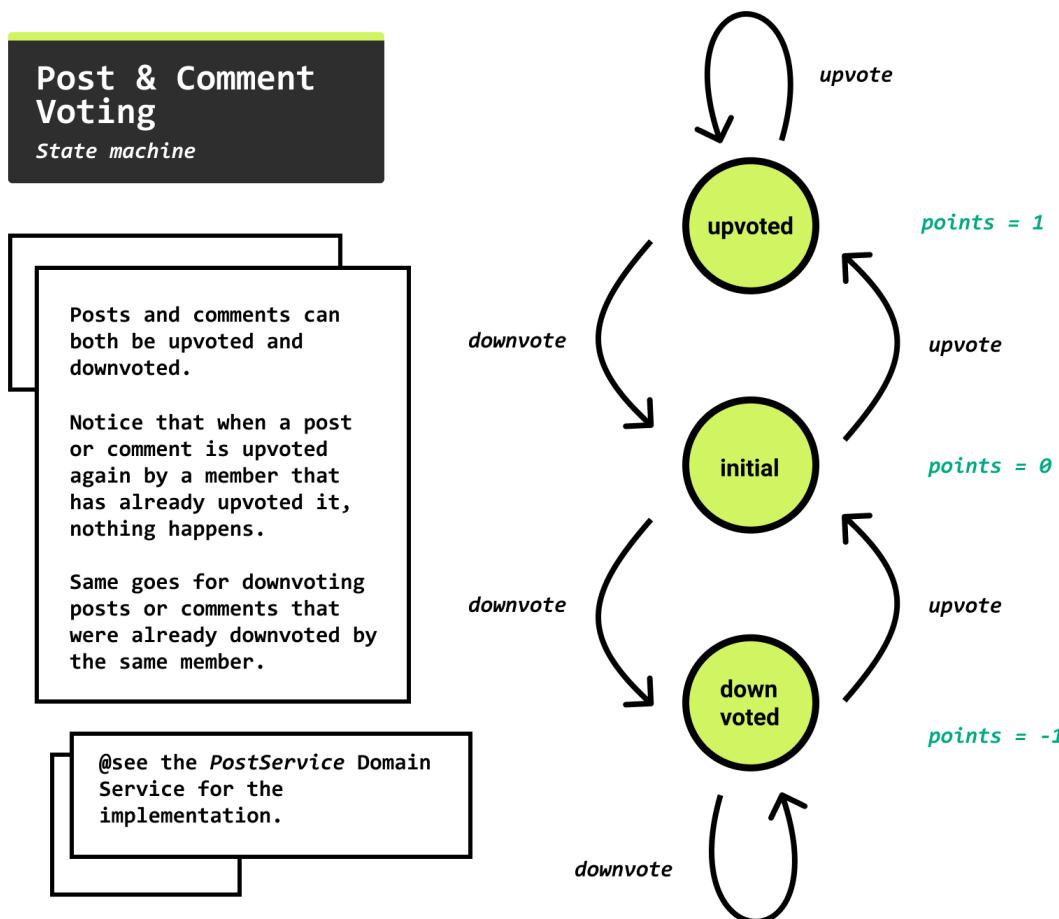
```
# ...
```

```
# 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.getNewItems()`.

```

// 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.getNewItems()) {
        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;

        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>;
}

```

```
getPostById (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](http://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](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/>.

## Context Mapping a Modular Monolith

In progress

### Shared Kernel pattern for shared infrastructure

- This is the *Shared Kernel* pattern. We call it this whenever we share a database schema and when subdomains are mutually dependent on each other. We'll also be sharing the base utilities and core domain object base classes throughout the project, but in a real-world micro-service deployment, it's often the case that teams' bounded contexts aren't even written using the same language.

## Partnership pattern for subdomain communication

- How will subdomains communicate with each other?
  - Domain events are the primary way that subdomains will communicate with each other within the context of our modular monolith. This means that we'll need some sort of messaging infrastructure, whether it be a queue, a bus, some publish-subscribe technology, etc in order to allow subdomains to subscribe to messages that are of particular interest to them.
  - **Show an image of the subdomains all subscribing.**
  - but in Part IX: Building Web Applications with Domain-Driven Design, Hexagonal Architecture and CQRS, we'll use an in-memory object capable of notifying other subdomains
- However, there is also the occasion where we'll need to utilize a service or a use case from another subdomain in the middle of a use case.
  - For example, **Use an example from understanding a story, or actually figure out what I'd do in ddd-forum to create a user.**
  - In a microservice deployment, this type of thing is very hard to get correct. We call that a distributed transaction, and it can be quite an engineering endeavor.
  - Because we're starting out with a monolith which shares the same database, we can thread a database transaction through use cases from neighbouring subdomains and commit the entire transaction at one time.

## Open-Host Service pattern

for communicating between subdomains

- What kind of context mapping pattern is this?
  - You may find that one subdomain tends to always be *used by others*, but never uses any other subdomains. A good example of this is the *Users* or *Identity and Access Management* subdomain.

## Anticorruption Layer for communicating with external models

- Relationships to other systems?
  - If we rely on external services like Stripe or PayPal for our system to work, we need to integrate with it. The pattern we'd use here is called an *Anticorruption Layer*. It's when a client (us) needs to use a model from another system (that doesn't particularly care about how we use their model — ie: Open/Host service), and we don't want those concerns to float into *our* model. We want to keep them entirely separate. Therefore, what we do is create couple the integration to an infrastructure adapter

**Show an image here of a relationship with our system to another one**

You can read more about the different context mapping patterns [here](#).

## Patterns

Use Case

Repository  
Domain model  
Transaction Script  
Domain Event  
Transactional Outbox  
Unit of Work  
Aggregates  
Commands  
Queries  
Value objects  
Entities  
Event handlers  
Clock (time)

## **Use Case**

**Repository**  
**Domain model**  
**Transaction Script**  
**Domain Event**  
**Transactional Outbox**

See here for now.

**Unit of Work**  
**Aggregates**  
**Commands**  
**Queries**  
**Value objects**  
**Entities**  
**Event handlers**  
**Clock (time)**

# Part X: Advanced Test-Driven Development

Test against behaviour, not implementation

Use Case Testing

End-To-End Testing Principles

Building an End-to-End Testing Rig

Stable End-to-End tests with the Page Object Pattern

Testing Managed vs. Unmanaged Dependencies

Integration Testing Input Adapters: GraphQL, REST, CLI

Unit Testing Principles

Acceptance Testing Principles

Dealing with Legacy Code

## **Test against behaviour, not implementation**

### **Use Case Testing**

### **End-To-End Testing Principles**

#### **Notes**

- End to end
  - Starting to believe that we should just really treat it as if we are a user and that we're doing
- Database cleanup?
  - Cypress io says not to bother with database cleanup after every test — instead, every now and then, just replace the test database.
- Using test fixtures hidden behind a non-production endpoint
  - You can also expose a backdoor that is only visible in test environments, never production — that backdoor allows you to set up test fixtures. These fixtures are hard-coded, it's just that the HTTP request is a command to load them into state.

#### **Resources**

<https://twitter.com/stemmlerjs/status/1427272766177386496>

<https://twitter.com/stemmlerjs/status/1427279582093262855>

<https://twitter.com/pscheit/status/1427316713893335044>

<https://twitter.com/stemmlerjs/status/1427314268530257921>

<https://kentcdodds.com/blog/static-vs-unit-vs-integration-vs-e2e-tests>

## Articles

- <https://martinfowler.com/bliki/PageObject.html>
- <https://www.cypress.io/blog/2019/01/03/stop-using-page-objects-and-start-using-app-actions/>
- <https://medium.com/reactbrasil/deep-diving-pageobject-pattern-and-using-it-with-cypress-e6ob9d7dod91>
- <https://www.toolsqa.com/cypress/page-object-pattern-in-cypress/>

## Building an End-to-End Testing Rig

### Stable End-to-End tests with the Page Object Pattern

Notes:

When we write tests, we want to make sure that they're not going to break often. One problem this exposes in end-to-end tests is that, to write these end-to-end tests, we need to simulate browser events, clicks, keypresses, as well as to be able to tell if components are on the screen or not, we might need to know implementation details like `className` or `id` attributes.

- This all leads to flakiness in tests because implementation details change often.
- “Encapsulate what varies” is a design principle - and architecturally, we want to do the same dependency inversion technique. Slap a contract inbetween a relationship where we need to rely on something, but we can’t be sure that it will always be stable. That’s how we stabilize relationships. We use contracts.
- Well, the way to do this on the front-end is to use a pattern called the Page Object pattern, or more generally speaking, it’s just to create something similar to that of what we do with the Clean Architecture and the
- Test specification: We write the test as declarative as possible (preferably using Gherkins)
- Test implementation: Something translates our spec into code that can realize that and make assertions.
- Page object (abstractions): We define abstractions to pages and items
- HTML
- So therefore, our test implementations execute page objects (an abstract contract), rather than across volatile HTML components.
- We can increase the stability of our HTML components by making a standard for how elements are organized in views. Using a component library works well here.
- Think about it this way. In your page objects, you define “how” it will know what is on the screen in the most declarative, standard, way possible, watch the tests fail, and then you write the code on the screen to make it pass. Eventually, once you spot duplication 3x, you refactor into cleaner abstractions for putting elements and components on a page in a way that can be traversed or found by our page objects.

- This is how to do TDD on the front-end. You start with the tests, then write the code to make the tests pass. This only really works if:
  1. You have the screens designed already (doesn't work well for exploratory coding)
  2. You start with the tests (I mean it - you have to think about "declaration of what you want" **first** and "implementation of making it work" **second**). Focusing on the behavior, not the implementation, is the way to have non-fragile end-to-end tests.

## **Testing Managed vs. Unmanaged Dependencies**

### **Integration Testing Input Adapters: GraphQL, REST, CLI**

### **Unit Testing Principles**

### **Acceptance Testing Principles**

### **Dealing with Legacy Code**

- Resources
  - Adding Unit Tests to Legacy Code  
<https://newsignature.com/articles/adding-unit-tests-to-legacy-code/>
  - What is Legacy Code: 8 Tips for Working Effectively with Legacy Code  
<https://www.perforce.com/blog/qac/8-tips-working-legacy-code>

## **Part XI: Above and Beyond**

DevOps

Testing Cloud Infrastructure

### **DevOps**

### **Testing Cloud Infrastructure**

### **Conclusion**