

Head First Object-Oriented Analysis and Design

Wouldn't it be dreamy
if there was an analysis and
design book that was more fun
than going to an HR benefits
meeting? It's probably nothing
but a fantasy...



Brett D. McLaughlin
Gary Pollice
David West

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Tokyo

Table of Contents (summary)

	Intro	xxiii
1	Great Software Begins Here: <i>well-designed apps rock</i>	1
2	Give Them What They Want: <i>gathering requirements</i>	55
3	I Love You, You're Perfect... Now Change: <i>requirements change</i>	111
4	Taking Your Software Into the Real World: <i>analysis</i>	145
5	Part 1: Nothing Ever Stays the Same: <i>good design</i>	197
	Interlude: OO CATASTROPHE	221
	Part 2: Give Your Software a 30-minute Workout: <i>flexible software</i>	233
6	"My Name is Art Vandelay": <i>solving really big problems</i>	279
7	Bringing Order to Chaos: <i>architecture</i>	323
8	Originality is Overrated: <i>design principles</i>	375
9	The Software is Still for the Customer: <i>iteration and testing</i>	423
10	Putting It All Together: <i>the ooa&d lifecycle</i>	483
	Appendix I: <i>leftovers</i>	557
	Appendix II: <i>welcome to objectville</i>	575

Table of Contents (the real thing)

Intro

Your brain on OOA&D. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing object-oriented analysis and design?

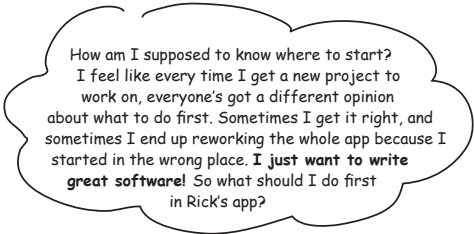
Who is this book for?	xxiv
We know what you're thinking	xxv
Metacognition	xxvii
Bend your brain into submission	xxix
Read Me	xxx
The Technical Team	xxxii
Acknowledgements	xxxiii

well-designed apps rock

1

Great Software Begins Here

So how do you *really* write great software? It's never easy trying to figure out **where to start**. Does the application actually **do what it's supposed to**? And what about things like duplicate code—that can't be good, can it? It's usually pretty hard to know **what you should work on first**, and still make sure you don't screw everything else up in the process. No worries here, though. By the time you're done with this chapter, you'll **know how to write great software**, and be well on your way to improving the way you develop applications forever. Finally, you'll understand why **OOD** is a four-letter word that your mother actually *wants* you to know about.



Rock and roll is forever!	2
Rick's shiny new application	3
What's the FIRST thing you'd change?	8
Great Software is...	10
Great software in 3 easy steps	13
Focus on functionality first	18
Test drive	23
Looking for problems	25
Analysis	26
Apply basic OO principles	31
Design once, design twice	36
How easy is it to change your applications?	38
Encapsulate what varies	41
Delegation	43
Great software at last (for now)	46
OOA&D is about writing great software	49
Bullet Points	50

gathering requirements

2 Give Them What They Want

Everybody loves a satisfied customer. You already know that the first step in writing great software is making sure it does what the customer wants it to. But how do you figure out **what a customer really wants**? And how do you make sure that the customer even *knows* what they really want? That's where **good requirements** come in, and in this chapter, you're going to learn how to **satisfy your customer** by making sure what you deliver is actually what they asked for. By the time you're done, all of your projects will be "satisfaction guaranteed," and you'll be well on your way to writing great software, every time.

Todd and Gina's Dog Door, version 2.0

Requirements List

Todd and Gina's Dog Door, version 2.0

What the Door Does

1. The tall

2. A b

3. On

4. The dog door opens.

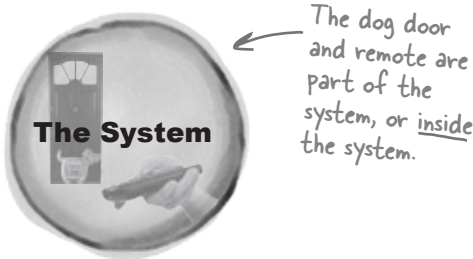
5. Fido goes outside.

6. Fido does his business.

7. Fido goes back inside.

8. The door shuts automatically.

You've got a new programming gig	56
Test drive	59
Incorrect usage (sort of)	61
What is a requirement?	62
Creating a requirements list	64
Plan for things going wrong	68
Alternate paths handle system problems	70
Introducing use cases	72
One use case, three parts	74
Check your requirements against your use cases	78
Your system must work in the real world	85
Getting to know the Happy Path	92
OOA&D Toolbox	106



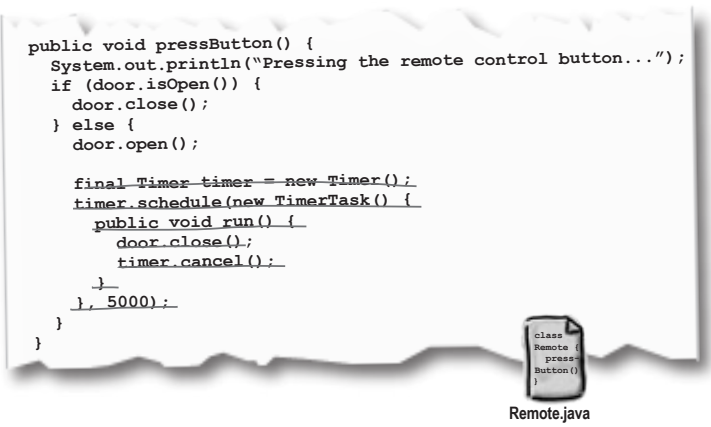
requirements change

3

I Love You, You're Perfect... Now Change

Think you've got just what the customer wanted? Not so fast... So you've talked to your customer, gathered requirements, written out your use cases, and delivered a killer application. It's time for a nice relaxing cocktail, right? Right... until your customer decides that they really wanted something different than what they told you. They love what you've done, really, but it's not quite good enough anymore. In the real world, requirements are always changing, and it's up to you to roll with these changes and keep your customer satisfied.

You're a hero!	112
You're a goat!	113
The one constant in software analysis & design	115
Original path? Alternate path? Who can tell?	120
Use cases have to make sense to you	122
Start to finish: a single scenario	124
Confessions of an Alternate Path	126
Finishing up the requirements list	130
Duplicate code is a bad idea	138
Final test drive	140
Write your own design principle	141
OOA&D Toolbox	142



analysis

4

Taking Your Software into the Real World

It’s time to graduate to real-world applications.

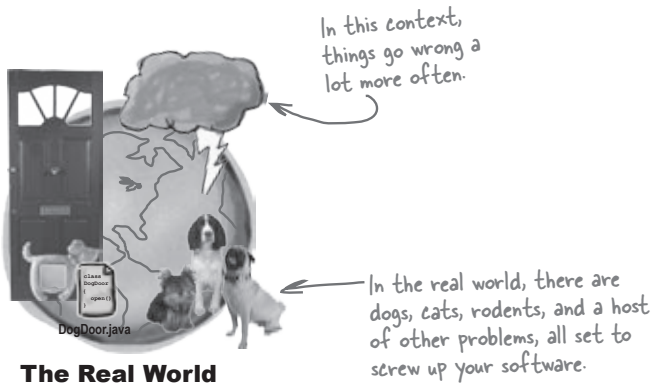
Your application has to do more than work on your own personal development machine, finely tuned and perfectly setup; your apps have to work when **real people use them**.

This chapter is all about making sure that your software works in a **real-world context**.

You’ll learn how **textual analysis** can take that use case you’ve been working on and turn it into classes and methods that you know are what your customers want. And when you’re done, you too can say: “I did it! My software is **ready for the real world!**”



One dog, two dog, three dog, four...	146
Your software has a context	147
Identify the problem	148
Plan a solution	149
A tale of two coders	156
Delegation Detour	160
The power of loosely coupled applications	162
Pay attention to the nouns in your use case	167
From good analysis to good classes...	180
Class diagrams dissected	182
Class diagrams aren’t everything	187
Bullet Points	191



5 (part 1)

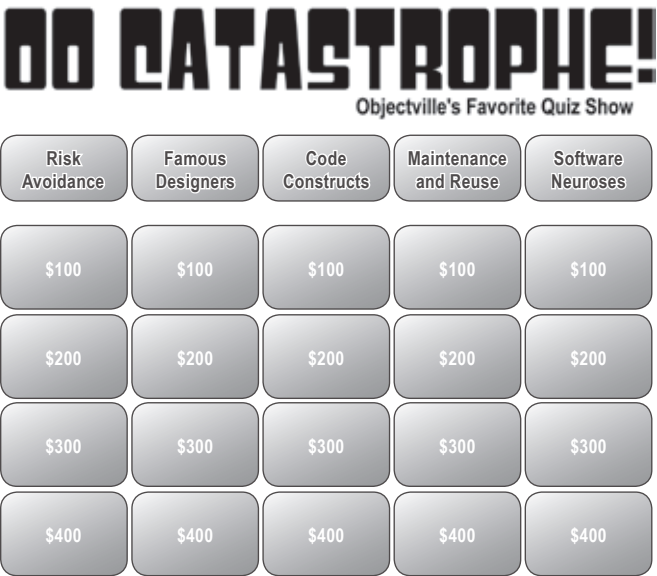
good design = flexible software

Nothing Ever Stays the Same

Change is inevitable. No matter how much you like your software right now, it's probably going to **change** tomorrow. And the harder you make it for your software to change, the more difficult it's going to be to respond to your **customer's changing needs**. In this chapter, we're going to revisit an old friend, try and improve an existing software project, and see how **small changes can turn into big problems**. In fact, we're going to uncover a problem so big that it will take a TWO-PART chapter to solve it!

Rick's Guitars is expanding	198
Abstract classes	201
Class diagrams dissected (again)	206
UML Cheat Sheet	207
Design problem tipoffs	213
3 steps to great software (revisited)	215

5 (interlude)



5

(part 2)

good design = flexible software

Give Your Software a 30-minute Workout

Ever wished you were just a bit more flexible?

When you run into problems making changes to your application, it probably means that your software needs to be **more flexible and resilient**. To help stretch your application out, you're going to do some analysis, a whole lot of design, and learn how OO principles can really **loosen up your application**. And for the grand finale, you'll see how **higher cohesion can really help your coupling**. Sound interesting? Turn the page, and let's get back to fixing that inflexible application.

Back to Rick's search tool	234
A closer look at the search() method	237
The benefits of analysis	238
Classes are about behavior	241
Death of a design (decision)	246
Turn bad design decisions into good ones	247
"Double encapsulation" in Rick's software	249
Never be afraid to make mistakes	255
Rick's flexible application	258
Test driving well-designed software	261
How easy is it to change Rick's software?	265
The Great Ease-of-Change Challenge	266
A cohesive class does one thing really well	269
The design/cohesion lifecycle	272
Great software is "good enough"	274
OOA&D Toolbox	276

solving really big problems

6

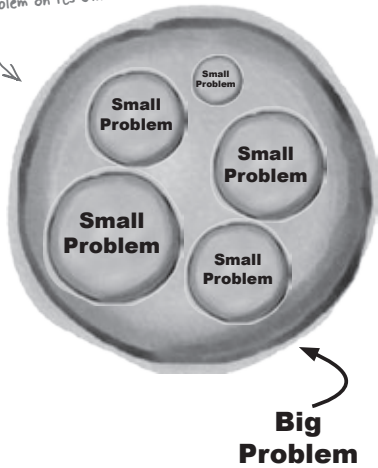
“My Name is Art Vandelay... I am an Architect”

It’s time to build something REALLY BIG. Are you ready?

You’ve got a ton of tools in your OOA&D toolbox, but how do you use those tools when you have to build something **really big**? Well, you may not realize it, but **you’ve got everything you need** to handle big problems. We’ll learn about some new tools, like **domain analysis** and **use case diagrams**, but even these new tools are based on things you already know about—like listening to the customer and understanding what you’re going to build before you start writing code. Get ready... it’s time to start playing the architect.

Solving big problems	280
It’s all in how you look at the big problem	281
Requirements and use cases are a good place to start...	286
Commonality and variability	287
Figure out the features	290
The difference between features and requirements	292
Use cases don’t always help you see the big picture	294
Use case diagrams	296
The Little Actor	301
Actors are people, too (well, not always)	302
Let’s do a little domain analysis	307
Divide and conquer	309
Don’t forget who the customer really is	313
What’s a design pattern?	315
The power of OOA&D (and a little common sense)	318
OOA&D Toolbox	320

This **BIG PROBLEM** is really just a collection of functionalities, where each piece of functionality is really a smaller problem on its own.



7

Bringing Order to Chaos

You have to start somewhere, but you better pick the *right* somewhere! You know how to break your application up into lots of small problems, but all that means is that you have **LOTS** of small problems. In this chapter, we're going to help you figure out **where to start**, and make sure that you don't waste any time working on the wrong things. It's time to take all those **little pieces** laying around your workspace, and figure out how to turn them into a **well-ordered, well-designed application**. Along the way, you'll learn about the all-important **3 Qs of architecture**, and how **Risk** is a lot more than just a cool war game from the '80s.

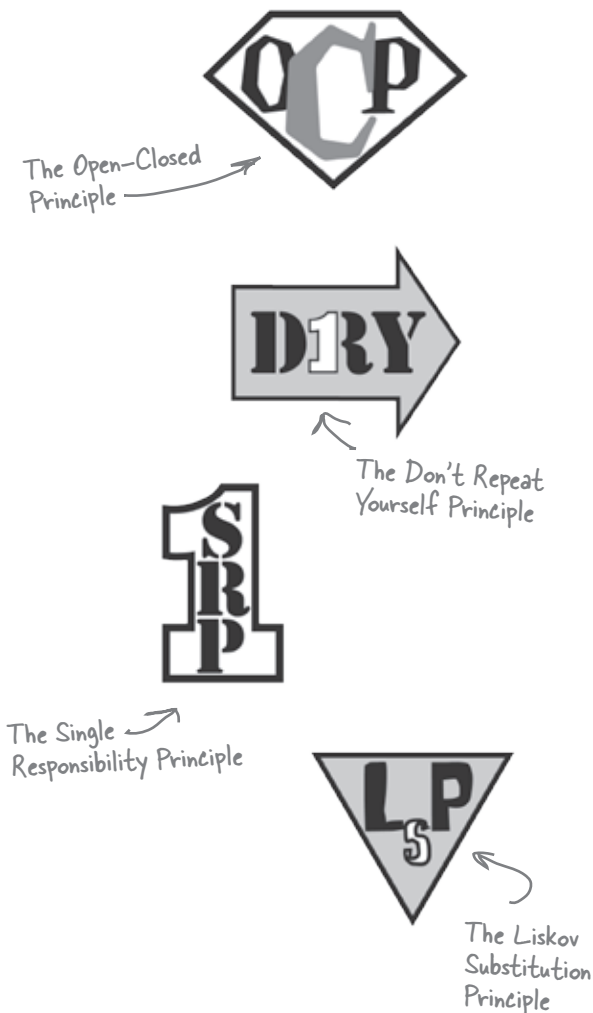


Feeling a little overwhelmed?	324
We need an architecture	326
Start with functionality	329
What's architecturally significant?	331
The three Qs of architecture	332
Reducing risk	338
Scenarios help reduce risk	341
Focus on one feature at a time	349
Architecture is your design structure	351
Commonality revisited	355
Commonality Analysis: the path to flexible software	361
What does it mean? Ask the customer	366
Reducing risk helps you write great software	371
Bullet Points	372

design principles

8 Originality is Overrated

Imitation is the sincerest form of not being stupid. There's nothing as satisfying as coming up with a completely new and original solution to a problem that's been troubling you for days—until you find out someone else **solved the same problem**, long before you did, and did an even better job than you did! In this chapter, we're going to look at some **design principles** that people have come up with over the years, and how they can make you a better programmer. Lay aside your thoughts of “doing it your way”; this chapter is about **doing it the smarter, faster way**.



Design principle roundup	376
The Open-Closed Principle (OCP)	377
The OCP, step-by-step	379
The Don't Repeat Yourself Principle (DRY)	382
DRY is about one requirement in one place	384
The Single Responsibility Principle (SRP)	390
Spotting multiple responsibilities	392
Going from multiple responsibilities to a single responsibility	395
The Liskov Substitution Principle (LSP)	400
Misusing subclassing: a case study in misusing inheritance	401
LSP reveals hidden problems with your inheritance structure	402
Subtypes must be substitutable for their base types	403
Violating the LSP makes for confusing code	404
Delegate functionality to another class	406
Use composition to assemble behaviors from other classes	408
Aggregation: composition, without the abrupt ending	412
Aggregation versus composition	413
Inheritance is just one option	414
Bullet Points	417
OOA&D Toolbox	418

iterating and testing

9

The Software is Still for the Customer

It's time to show the customer how much you really care.

Nagging bosses? Worried clients? Stakeholders that keep asking, "Will it be done on time?" No amount of well-designed code will please your customers; you've got to **show them something working**. And now that you've got a solid OO programming toolkit, it's time to learn how you can **prove to the customer** that your software works. In this chapter, we learn about two ways to **dive deeper** into your software's functionality, and give the customer that warm feeling in their chest that makes them say, **Yes, you're definitely the right developer for this job!**

Unit
type: String
properties: Map
id: int
name: String
weapons: Weapon *
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object
getId(): int
setName(String)
getName(): String
addWeapon(Weapon)
getWeapons(): Weapon *

All the properties that were common across units are represented as variables outside of the properties Map.

Sam figured that id would get set in the Unit constructor, so no need for a setId() method.

Each of the new properties gets its own set of methods.

Your toolbox is filling up	424
You write great software iteratively	426
Iterating deeper: two basic choices	427
Feature driven development	428
Use case driven development	429
Two approaches to development	430
Analysis of a feature	434
Writing test scenarios	437
Test driven development	440
Commonality Analysis (redux)	442
Emphasizing commonality	446
Emphasizing encapsulation	448
Match your tests to your design	452
Test cases dissected...	454
Prove yourself to the customer	460
We've been programming by contract	462
Programming by contract is about trust	463
Defensive programming	464
Break your apps into smaller chunks of functionality	473
Bullet Points	475
OOA&D Toolbox	478

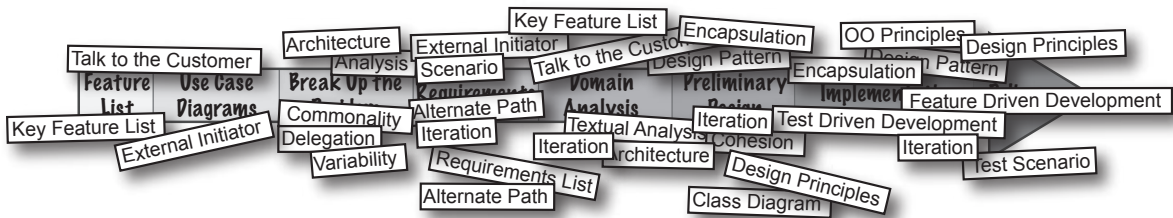
the ooa&d lifecycle

10

Putting It All Together

Are we there yet? We've been working on lots of individual ways to improve your software, but now it's time to **put it all together**. This is it, what you've been waiting for: we're going to take **everything** you've been learning, and show you how it's all really part of **a single process** that you can use over and over again to **write great software**.

Developing software, OOA&D style	484
The Objectville Subway problem	488
Objectville Subway Map	490
Feature lists	493
Use cases reflect usage, features reflect functionality	499
Now start to iterate	503
A closer look at representing a subway	505
To use a Line, or not to use a Line	514
Points of interest on the Objectville Subway (class)	520
Protecting your classes	523
Break time	531
Back to the requirements phase	533
Focus on code, then focus on customers	535
Iteration makes problems easier	539
What does a route look like?	544
Check out Objectville for yourself!	548
Iteration #3, anyone?	551
The journey's not over...	555




appendix i: leftovers



The Top Ten Topics (we didn't cover)

Believe it or not, there's still more. Yes, with over 550 pages under your belt, there are still things we couldn't cram in. Even though these last ten topics don't deserve more than a mention, we didn't want to let you out of Objectville without a little more information on each one of them. But hey, now you've got just a little bit more to talk about during commercials of CATASTROPHE... and who doesn't love some stimulating OOA&D talk every now and then?



Anti Patterns
Anti-patterns are the reverse of design patterns: they are common BAD solutions to problems. These dangerous pitfalls should be recognized and avoided.

#1. IS-A and HAS-A	558
#2. Use case formats	560
#3. Anti-patterns	563
#4. CRC cards	564
#5. Metrics	566
#6. Sequence diagrams	567
#7. State diagrams	568
#8. Unit testing	570
#9. Coding standards and readable code	572
#10. Refactoring	574

Be sure you write down things that this class does on its own, as well as things it collaborates with other classes on.

Class: DogDoor

Description: Represents the physical dog door. This provides an interface to the hardware that actually controls the door.

Responsibilities:

Name	Collaborator
Open the door	
Close the door	

There's no collaborator class for these.

appendix ii: welcome to objectville



Speaking the Language of OO

Get ready to take a trip to a foreign country. It's time to visit Objectville, a land where **objects do just what they're supposed to**, applications are all **well-encapsulated** (you'll find out exactly what that means shortly), and designs are easy to **reuse and extend**. But before we can get going, there are a few things you need to know first, and a little bit of **language skills** you're going to have to learn. Don't worry, though, it won't take long, and before you know it, you'll be speaking the language of OO like you've been living in the well-designed areas of Objectville for years.

UML and class diagrams	577
Inheritance	579
Polymorphism	581
Encapsulation	582
Bullet Points	586

