from The Rational Edge: A favorable review of the O'Reilly Head First series book.

Brett D McLaughlin, Gary Pollice, David West

O'Reilly Media -- Head First Series, 2006
ISBN 0596008678
Paperback: 600 pages

No matter how great you think your software is, if it's not what someone wants, how great can it be? *Object Oriented Analysis & Design* puts O'Reilly's "Head First spin" on what it means to develop great software -- software that does exactly what your customers want. Through this user-centric view, readers are guided through the picture-book world of demanding customers and automatic dog doors to show why OOA&D really matters.

**The "Head First" way**

For anyone new to O'Reilly's Head First series, these are not your every-day bland technology books. Though the 600 page monster looks intimidating at first, every page averages three small paragraphs of text -- i.e., handwritten notes to help you understand the text. Plus, each page is dominated by pictures and illustrations to keep your brain churning. Sounds like a picture book, right? Well, the Head First editors took the saying, "A picture is worth a thousand words" quite literally, and they definitely found a good way to use pictures as a substitute for long, drawn out text.

The book actually asks you questions and provides space for you to write your answers. Also, the book has this exercise called Code Magnets, which is like fill in the blank ... but with code! The "magnets" are the word bank, and it is the reader's responsibility to fill in the blanks (don't worry, they give you the answers later).
Welcome to Objectville

*Head First OOA&D* sets you up in this imaginary world, Objectville, with imaginary teams and imaginary customers. I say imaginary, because you can quickly forget that the problems you encounter in this book are not real: The reader is thrown into four projects during the course of the book, all geared toward satisfying the customer’s needs.

**Rick's guitars**

Rick is a guitar aficionado and owner of a guitar shop, and he's hired your firm to create his guitar search tool. However, the reader enters this world in a sticky situation: the application isn't working! You work with members of your team to clean up the design and fix Rick's search tool, but immediately thereafter Rick changes his requirements. You and your team are able to satisfy his requirements, but the main goal of this chapter is to show you real-life issues with developing software and how focusing on what the customer wants and designing for reusability really pay off.

**Doug's dog doors**

After solving Rick's problems, you jump to a new project: Doug's Dog Doors. These are high-tech dog doors that let the dog out for you. The reader's first customers are pretty clear about what they want it to do, but the book shows us what happens when we don't gather requirements; i.e., there's a good chance the software isn't going to work the way the customer wants it to. In this example, the dog door doesn't close the way the customer wants, and other critters like bunnies end up back in the house along with the dog.

This endeavor shows the reader the importance of requirements gathering, and has them talk to the customer and figure out exactly how they are going to use the dog door. The authors then go on to show how to write use cases from requirements (but disguises the exercise as "just figuring out what the door does"). The section talks about what use cases are used for, and finally shows how to check requirements against use cases. Right when you think you're done, Doug gets more customers ... and they all have different requirements on how the dog door should work! This exercise readdresses how to gather requirements and create use cases from them.

At this point, you're convinced everyone is happy with Doug's Dog Doors, but the customers come back some new ideas, like a Bark Recognizer. This introduces the one constant in software analysis and design: change. You walk through updating requirements and use cases, get introduced to scenarios (a path through a use case), then go to the code and make the necessary changes.

However, there's another problem with Doug's Dog Doors; the bark recognizer opens the dog door when ANY dog barks. This point shows that software has to work in the real world, and not just in your test environment. You walk through identifying the problem, planning for a solution, and eventually updating the door use case to reflect these changes. This issue also shows the need for a new use case: storing the dog bark. The book walks you through a couple of different solutions, begins to address coupling and delegation, introduces textual analysis (analyzing use cases to figure out what to focus on), and UML class diagrams. Whew! There is a lot covered here, but it's well organized and flows well. The authors do a good job of getting a lot of content stuck in your head.
Rick's stringed instruments

That's right, Rick is selling more than guitars now! This section tests the previous design out, explains a bit more about UML (generalization and aggregation) and shows how convoluted things can get! With a bunch of ugly code on our hands and problems spewing up, the reader jumps into "OO Catastrophe!", Objectville's Favorite Quiz Show (No, I'm not making this up). It goes through four Jeopardy-style answers, like "This code construct has the dual role of defining behavior that applies to multiple types, and also being the preferred focus of classes that use those types". What is an INTERFACE? Bingo!

Anyway, after the game we get back to working on Rick's problem: adding support for a ton more instruments. After 24 pages we reach a much better design, which mainly uses a map to store properties rather than sub-classing for each type of instrument. Then the bureau de change pops up and introduces cohesiveness as a measure of ease-of-change. They make a point that a class should only have one reason to change, which is the big take-away from this section.

Gary's games

In project #3, Gary's Games has hired you to create a turn-based strategy game framework. You are provided a vision statement, but that's about it; it's your job to figure out exactly what the customer wants. The book introduces commonality and variability analysis to really figure out what features (set of requirements) the customer wants, and discusses use case diagrams in detail to help you begin to see the big picture of your system. It also introduces domain analysis, which is just a way to make sure your feature list still speaks the customer's language.

This is a much larger project, so the book tackles how to break up big problems into much smaller problems, by dividing functionality into packages that can be worked on independently. It also introduces "design patterns" and shows that they can emerge even in this stage by revealing that our package design is the Model-View_Controller pattern. In this way, a big problem Gary wants solved is turned into a bunch of smaller problems that the reader already knows how to solve.

So, now that everything is broken up, where do we start? This is where architecture comes in, which helps you tie together all the parts. However, you still need to start somewhere, so we start at the points that are architecturally significant -- the really important parts. The book provides guidance in how to pick what's significant by asking "Is this feature essential?", "What does the feature mean?", and "How do we build the feature?". These points all address "Managing Risk," which is at the heart of what architecture is all about.

The book takes a slight detour and discusses design principles in more detail. It guides the reader through principles such as the Open-Closed Principle (OCP), Don't Repeat Yourself (DRY), Single Responsibility Principle (SRP), Liskov Substitution Principle (LSP), Delegation, Composition, and Aggregation. The authors take their time on these and provide really good examples, so it's much less daunting than it seems.

Now that it's time to begin iterating on the design, the book describes two different options -- Use case or Feature-driven development -- and discusses when one approach is better than the other. You pick feature-driven development, select a feature, and begin building it ... making
design decisions as you go. It also explains the difference between programming by contract and defensive programming with a clever situation where your customers request that the framework code do the checking.

**Objectville RouteFinder**

Whoa! What happened? Why are we on a new project, when we didn't even finish the previous one? We were just beginning to care about the customer when we left Gary with a half finished framework! This caught me off guard, but I was getting pretty bored with the last project, so maybe it's for the best. We actually never finished the other two projects either. Anyway, in this new chapter we go through the OOA&D Project Life-cycle from start to finish and actually build something to completion.

Objectville RouteFinder will help people get directions around Objectville. All from a statement of work, we can derive the feature list and use case diagrams, break up the problem, and then pick a use case to start to iterate. For the specific use case we figure out the problem, write the use case, analyse it to figure out what we need to be thinking about, and do some preliminary design. After that we’re ready to implement. Then repeat the process for the other use case and BAM! Objectville now has a working RouteFinder.

**Leftovers: Top ten topics and Welcome to Objectville**

This section is devoted to the topics they didn't want to cram into the main text of the book. It covers IS-A/HAS-A relationships, use-case formatting, anti-patterns, CRC cards, metrics, sequence diagrams, state diagrams, unit testing, coding standards, and refactoring. Okay, so we're being introduced into Objectville at the end of the book? This is written like it belongs at the beginning, so it seems like a last-minute change. Anyway, this section talks about UML and class diagrams, inheritance, polymorphism, encapsulation ... things they should have shown you in the beginning.

**Check it out!**

*Head First OOA&D* offers a refreshing look at software development that really focuses on what's important: happy customers and maintainable/reusable designs. It is definitely geared at technical software beginners, but anyone interested in writing great software can get something from it. Don't let the high page count scare you away. It's a very quick read (took me approximately eight hours, and I'm usually very slow). They do such a good job at presenting the content that the information really stuck in my head.

Bottom line: If you want to make your customers (and yourself) happy, get your hands on this book.

© Copyright IBM Corporation 2007
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)