Head First Object-Oriented Analysis and Design

Brett D. McLaughlin
Gary Pollice
David West

O'REILLY®

Beijing • Cambridge • Köln • Paris • Sebastopol • Taipei • Tokyo

Head First Object-Oriented Analysis and Design

by Brett D. McLaughlin, Gary Pollice, and David West

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Series Editor: Brett D. McLaughlin

Editor: Mary O'Brien

Cover Designer: Mike Kohnke, Edie Freedman

OO: Brett D. McLaughlin

A: David West

D: Gary Pollice

Page Viewer: Dean and Robbie McLaughlin

Printing History:

November 2006: First Edition.

ISBN-10: 0-596-00867-8

ISBN-13: 978-0-596-00867-3

[M]

5 good design = flexible software: academic supplement

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!

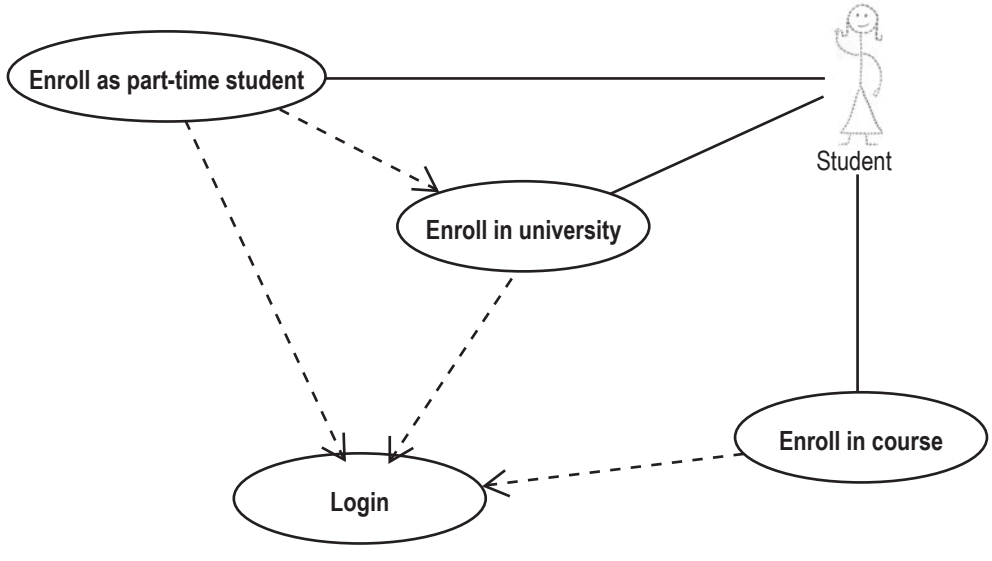
Short Answers

1. One UML element that adds to a model's understandability is called a stereotype. "A stereotype lets you create new kinds of building blocks similar to existing ones but specific to your problem." (*The Unified Modeling Language User Guide*, by Grady Booch, James Rumbaugh, and Ivar Jacobson, Addison-Wesley)

A stereotype is rendered in UML diagrams as a name enclosed by French quotation marks, called guillemets (« »). Stereotypes may be used on any UML modeling element. In the following diagrams, you are asked to place the appropriate stereotypes on the elements where they belong. You may need to look at a UML reference to learn what some of the notation means. (It's good to have to learn some things on your own!).

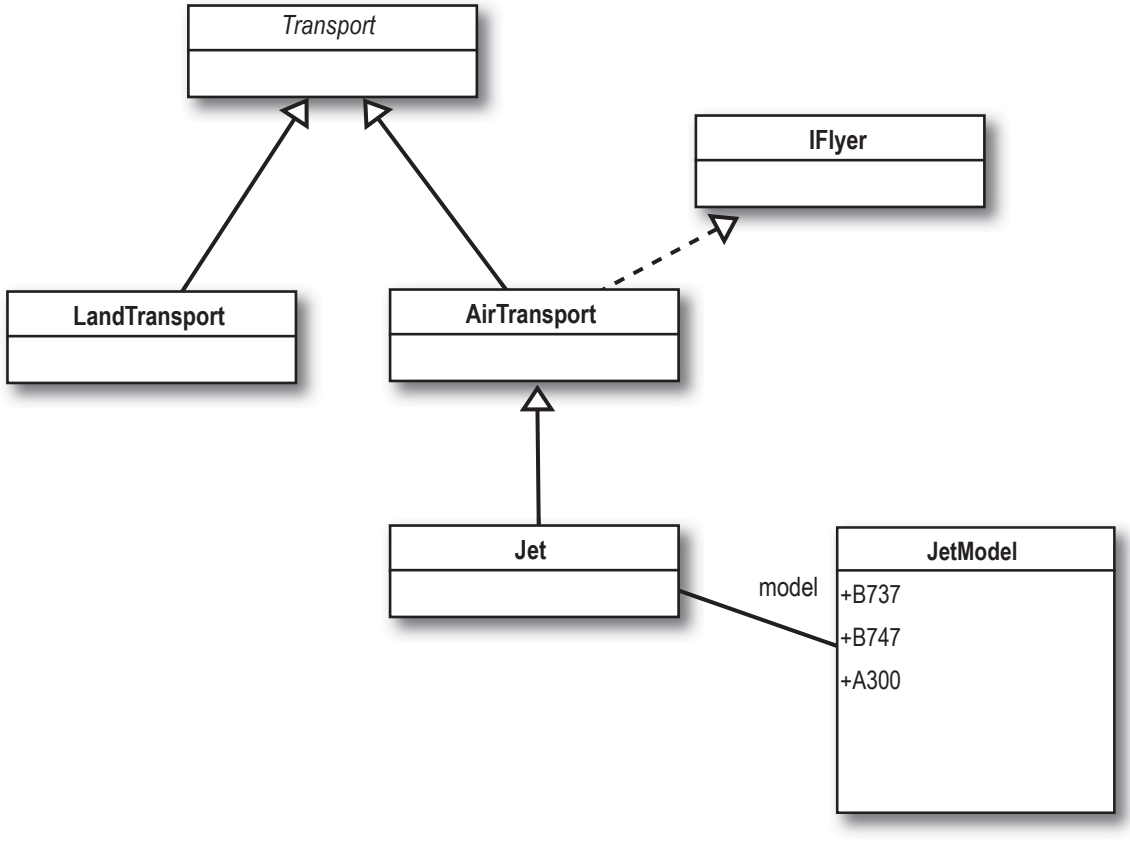
A. Show stereotypes for the «`include`» and «`extend`» relationships in the following use case diagram.

Yes, we know we said that you shouldn't use them, but some people do, and you should know what they look like.



Short Answers

1. B. Show stereotypes for «`interface`» and «`enumeration`» in the following diagram.



Short Answers

2. Why don't you need a stereotype, «`abstract`», for abstract classes?

   Would it be wrong to use one?

3. What is the difference between an interface and an abstract class? How do you decide which is the best one to use? (We're not talking about any specific language or implementation but about general object-oriented concepts. After all, by the time you graduate, Java will be so yesterday.)

Short Answers

4. Look at the class diagram on page 199. Does showing the properties of the class with associations make the model clearer or not? Why?

5. The name of the association, such as “spec”, as shown in the diagram on page 199 is called the **role name**. Why do you think it is called that? Do you think it’s a good description of what the name stands for?

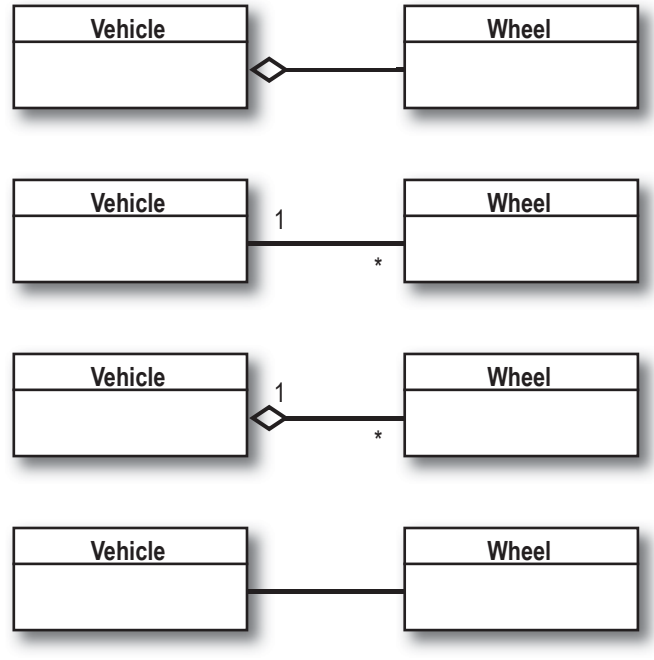
Short Answers

6. In the diagram 1B earlier, the Jet is derived from AirTransport. Does this mean that the Jet "is an" AirTransport? Is this always the case with generalization (inheritance)?

Hint: the answer may surprise you. We'll get to it later, but this will get you thinking about these deep philosophical—and practical—issues.

7. We introduced the notation for aggregation in this chapter. The following four associations between a Vehicle class and a Wheel class use different UML notations.

What is the difference in meaning for each? Which do you prefer and why? Does it really matter?



Short Answers

8. Create a UML model of your current OOA&D course. Include the people, assignments, books, and anything else that is appropriate for your class and institution. Is this analysis or design?

9. Revise your model from question 8 so that it reflects the classes you might use in implementing a simulation of your OOA&D course in software. Use interfaces and abstract classes where appropriate.

Describe your design decisions (like why you used a particular interface, and so on). Is this analysis or design? Do all of the classes have **behavior**, or are some of them simply data containers?

Short Answers

10. One popular practice for designing a system is called Test-Driven Development (or Design), abbreviated as TDD. Using this practice, you write tests first and then implement the code that makes the tests pass. Can you think of benefits of this approach? What about disadvantages? (References: *http://www.testdrive.ncom, http://www.methodsandtools.com/archive/archive.php?id=20, http://www.artima.com/intv/testdriven.html,* and many others.)

11. We talk about coupling and cohesion in this chapter. Do some research and come up with different definitions of coupling and cohesion. Cite your references.

Which do you think is the clearest and best? Why?

Warning, bad pun: You should be able to find a couple of cohesive definitions.

Short Answers

12. Look at the Instrument and InstrumentSpec classes on pages 204–205. Have we done a good job of abstracting common behavior? Will this work for other types of instruments, like flutes, drums, kazoos, maracas, and the ever-popular washboard? Why or why not? How would you make the design better? (Don't look ahead!)

13. Can you describe the relationship between cohesion and coupling? Okay, we dare you to do it and make it understandable to your parents!

Short Answers

14. Now for a patterns exercise. Look at the final version of Rick's application in this chapter. It's pretty flexible, if we do say so. Now the instrument spec is responsible for determining if it matches another instrument spec. Pretty cool, huh? But is it flexible enough? What might be likely to change, and how would you adapt the software to deal with it?

15. One metric that is used to evaluate the "goodness" of a design is called the Depth of Inheritance Tree (DIT) metric. If you represent your class hierarchy as a tree (yes, I know there could be a forest, but let's assume for now that there's just one tree), DIT is the length of the maximum path from a node to the root of the tree. In other words, it is a measure of how many ancestor classes any given class has. (See, it works for a forest, too!) What is better, a high value of DIT or a low value, and why?

Short Answers

16. What is the DIT value for each class in the diagram for question 1B?

17. Create a model that describes all of the people in your academic institution. Describe the model as a class hierarchy with the root class being Person. Try to keep the maximum value of DIT to a minimum, while keeping the different type of people distinct (you can use "helper" classes to do this—think delegation). Produce a UML class diagram of your model.

Short Answers

18. Another metric, originally proposed by Chidamber and Kemerer is the Lack of Cohesion of Methods. (Reference: Chidamber, S.R. and Kemerer, C.F., "Towards a metrics suite for object-oriented design, *Proceedings of 6th ACM Conference on Object Oriented Program, Systems, Languages and Applications (OOPSLA)*, Phoenix, AZ, pp. 197-211)

In order to calculate the metric as it was originally proposed:

Consider a Class C_1 with methods M_1, M_2, ... , M_n. Let { I_i }= set of instance variables used by method M_i. There are n such sets {I_1}, ... , {I_n}. LCOM = the number of disjoint sets formed by the intersection of the n sets." (Taken from: Etzkorn, Davis, and Li, *http://www.cs.uah.edu/tech-reports/TR-UAH-CS-1997-02.pdf*).

Take the final version of Rick's application from this chapter and calculate LCOM for each class.

19. Calculate LCOM for the final version of Doug's dog door application from the previous chapter.

Short Answers

20. Java allows single inheritance, while C++ allows multiple inheritance. What are the benefits and disadvantages of each? Justify your answers.

21. Can you use interfaces in Java to approximate multiple inheritance? How? What can't you do with interfaces that you can with multiple inheritance?

Short Answers

22. C++ does not have an interface in the language. How can you achieve the same results as the interface in Java?

23. We talk about encapsulating what varies in your application. A complementary principle is to **abstract commonality**. What do you think this means? How does this relate to encapsulating what varies?

Short Answers

24. How does applying **encapsulate what varies** naturally lead to delegation?

25. Repeat the "Sharpen your pencil" exercise on page 244 for the final version of Doug's application in the previous chapter.

Short Answers

26. We used a Map class to implement dynamic properties for Rick's application. This was a good choice for Java, but not all languages have such a simple way of handling this. What other choices can you think of? Explain their advantages and disadvantages?

If you did question 9 earlier, implement the classes from your model.

Add behavior to some of the classes, such as gradeExam for the instructor or staff, and impllement the appropriate methods.

Write tests for your classes and try to get 100% coverage.

An alternative to programming exercise 1: Implement the software using TDD.

Write a short paper evaluating the experience and quality of the code. Is this a practice you might consider using? Why or why not?

If you answered question 14 earlier, you hopefully realized that there might be more than one way to match instruments. For example, what if someone came in and said, "My daughter wants to learn to play a stringed instrument. It doesn't matter if it's a banjo, mandolin, or guitar. I just want it to cost less than $100.00 since she'll probably want to build a model rocket next week!"

In fact, there are all sorts of searching algorithms you could choose to fit different needs. The Strategy pattern (go get your *Head First Design Patterns* book now) could help you handle these situations.

Modify the code to add a Strategy pattern and implement two different strategies. Make sure you test your code.

Note: This achieves some of the same functionality as we ended up with at the end of this chapter. Is this better or worse? Justify your conclusions.

Long Exercise 1

This is a research project. We'll be looking at delegation a lot as a way of designing flexible software. One principle that good designers use is to prefer delegation to inheritance.

Research this issue and explain why delegation is preferred (if, in fact you agree with the statement).

Give examples of how delegation makes your software more flexible than inheritance. Use UML diagrams in your answer.

Long Exercise 2

There are several versions of LCOM. Read *http://www.cs.uah.edu/tech-reports/TR-UAH-CS-1997-02.pdf* and summarize the different ways to calculate LCOM.

Which do you think are the most meaningful and why?

Calculate LCOM for some application (Rick's?) using each of these methods.

Long Exercise 3

There is a concept called **mixin classes**. Several (object-oriented) programming languages support mixin classes. Research the topic of mixin classes and write a 3–4 page report on your findings.

Describe the concept and the benefits it provides. Provide an example of a mixin class for at least two such languages. Cite your references.