C/C++ Users Journal™
Advanced Solutions for C/C++ Programmers

Object-Oriented Analysis and Design, Part 1

by Alistair Cockburn,
with C++ code by Chuck Allison

Object-Oriented design is easy - once you learn how to identify the right objects.

Introduction

"If you have done an object-oriented (OO) design, how do you describe it to another person? What do you talk about or show, to convey your design?" I have asked this of experienced designers for several years now. I usually get back a very long silence, ending with a shrug of the shoulders. This is a tough question, not covered in classes and books.

This two-article series presents a problem I use both to teach and test OO design. It is a simple but rich problem, strong on "design," minimizing language, tool, and even inheritance concerns. The problem represents a realistic work situation, where circumstances change regularly. It provides a good touch point for discussions of even fairly subtle designs in even very large systems. This problem was successfully solved by six first-year computer science students with no OO background. It was less successfully addressed by business people, commercial programmers learning OO, and experienced OO programmers.

If you do a great job on this problem, congratulate yourself. If you are taken by surprise at the developments, you are in good company. What

I hope you get from this series is:

- how to talk about designs,
- a bit of how to evaluate and improve designs,
- and a sample problem against which to test your favorite design technique.

Warm-Up: What Is "Design?"

To learn what design is, I have been asking my students for years now, before we start on any OO lectures, to sketch a simple design of an everyday system and present it to the class on two overhead transparencies. No talking is to accompany the transparencies. The class then tries to understand, from the transparencies, the design of the system. At the end the class discusses which of the teams' presentations best captured the notion of design.

The system I ask them to design, and am now asking you to design, is a small, one-branch bank. Sketch the design for the whole thing, not just the computer part. Take no more than 15 minutes. Work with someone else, if possible. Capture the design on a single sheet of paper. (Your pencil lets you put much more on a page than the fat transparency pens I give the students.) You should be able to show this piece of paper to pretty much anyone on the street, and they would recognize the bank's basic design. On your marks, get set. Go.

The acclaimed winners in my classes consistently present four things:

1. The name of the key components, or things, in the system
2. The purpose, basic function, or main responsibility of each thing
3. A drawing of the key communication paths between them
4. A list of the services provided

I have adopted these four things as essential to communicating design. I see these things used to describe the design of significant parts of large companies, of large OO systems, and also of code-level solutions to hard programming problems (see "In search of methodology" [2]).

Nowadays, I insist upon these items in program or system design documentation (along with email records of design arguments and resolutions). The set of four design items tells us why an UML/OMT-style drawing is not satisfying to experienced designers. A UML/OMT drawing gives the components, but says neither what their purpose is nor what their main communication channels are.

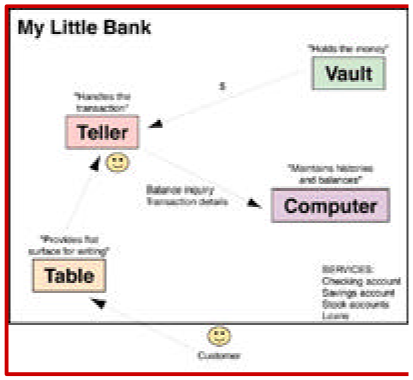

Figure 1: Sample bank design

Of the four design elements, the brief "responsibility statement" is the most important. It tells an experienced person what will be included and what will be excluded from the data and behavior for the component. Ward Cunningham, Kent Beck, and Rebecca Wirfs-Brock have been championing the responsibility statement for years [1, 4]. The drawing of the communication channels summarizes the communication patterns and gives a quick view of the traffic patterns and collaborations. It gives the viewer a rapid understanding of the system's behavior. Trygve Reenskaug's "role models" are an articulate and precise way to describe communication channels [3]. Rebecca Wirfs-Brock's "contracts" were an earlier attempt to capture them [4]. The communication channels drawing expands into interaction diagrams, which demonstrate that the components, holding to their responsibilities, really can work together properly to deliver the services.

I gave you the bank exercise just to introduce the four key items needed for good design. The problem I use for instruction in this article is the design of a coffee machine. In discussing and comparing designs in this article, I shall focus on Component Names, Component Main Responsibilities, and Intercomponent Communication Channels. The services list will come to you through the requirements statement. Expect me to ask, "Which component knows the price of the coffee? Which object knows how to make the drink?" I shall show, and ask you to show, a few interaction diagrams to demonstrate that the system really works. Although you may use inheritance in solving the problem, I am going to spend less time on inheritance. If you get the basic design wrong, no inheritance structure will fix it. If your basic design is strong, you can play with the inheritance structure to make it better.

Figure 1 shows a sample solution to the bank design question that uses the four elements. Note that the Customer is not a component of the bank and has no responsibilities. (There is no way, at system design time, to distinguish a customer from a robber; that has to show up in the operation of the system.) The components of the bank are:

- The table. It provides a flat surface for writing and filling out forms.
- The teller. This person is responsible for coordinating the actions involved in the transaction, such as getting money from the vault and entering the transaction into the computer.
- The vault. Holds the money.
- The computer. Knows the current state and history of the customers' accounts.

Now let's go on to the main problem.

The Coffee Machine Problem

You and I are contractors who just won a bid to design a custom *coffee vending machine for the employees of Acme Fijet Works to use. Arnold, the owner of Acme Fijet Works, like the common software designer, eschews standard solutions. He wants his own, custom design. He is, however, a cheapskate. Arnold tells us he wants a simple machine. All he wants is a machine that serves coffee for 35 cents, with or without sugar and creamer. That's all. He expects us to be able to put this little machine together quickly and for little cost. We get together and decide there will be a coin slot and coin return, coin return button, and four other buttons: black, white, black with sugar, and white with sugar.*

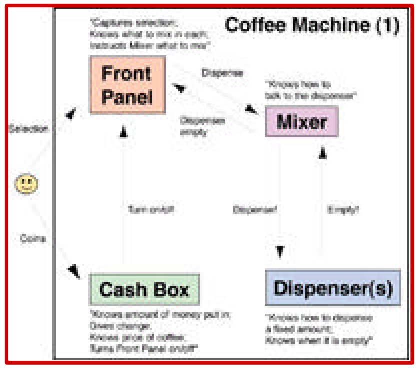
Coffee Machine (1)
"Captures selection; Knows what to mix in each; Instructs Mixer what to mix"
Front Panel
Dispense
"Knows how to talk to the dispenser"
Mixer
Dispenser empty
Selection
Coins
Turn on/off
Dispense!
Empty!
Cash Box
"Knows amount of money put in; Gives change; Knows price of coffee; Turns Front Panel on/off"
Dispenser(s)
"Knows how to dispense a fixed amount; Knows when it is empty"

Figure 2: Design of the coffee machine (1)

Now design the machine using objects. What are the components, what are their responsibilities, how do they work together to deliver the simple service: give coffee for 35 cents? This process will work best if you cover up the diagram in Figure 2 and sketch your own design before reading further. Go.

Which component knows the price of the drink? Which component knows how to make the drink?

Test your design against Kim's test scenarios:

- Kim puts in a quarter and then selects a coffee.
- Kim puts two quarters in and then selects a coffee.
- Kim puts in a quarter, then pushes the coin return lever.
- Kim puts in two quarters, then walks away from the machine and forgets to come back.

- Kim buys two coffees, white with sugar. The sugar dispenser runs out of sugar after the first.

Figure 2 shows the basic design that students commonly give me at this point, after we have straightened out the standard bugs. Figure 3 shows the interaction diagram. The code in Listing 1 shows sample C++ declarations for the classes involved (a full implementation is delayed for a better design). For space reasons, I omit details of making change and handling insufficient change. Note this is not a good design; it is a typical one. I hope you do better. Try to say why your design is "better" than this one. The answer will be clear by the time we get done.

The typical coffee machine design submitted to me has four main components:

COFFEE MACHINE (1) DESIGN:

- Cash Box. Knows amount of money put in; gives change; knows price of coffee; turns Front Panel on and off.

- Front Panel. Captures selection; knows what to mix in each; instructs Mixer what to mix.

- Mixer. Knows how to talk to the dispensers.

- Dispensers (cup, coffee powder, sugar, creamer, water). Knows how to dispense a fixed amount; knows when it is empty.

Arnold Visits

After five machines are installed and have been operating for a while, Arnold comes along and says, "I would like to add bouillon, at twenty-five cents. Change the design." We add one more button for bouillon, and one more container for bouillon powder. How else do you change your design?

Below is the typical design I get at this stage of the story (see also Figures 4 and 5). Notice how the division of responsibilities is fundamentally different from the first solution.

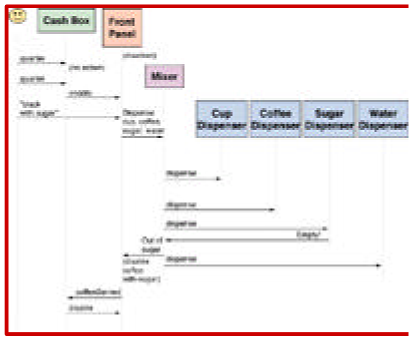

Figure 3: Interaction diagram showing machine (1) serving coffee, running out of sugar

COFFEE MACHINE (2) DESIGN:

- Cash Box. Knows amount of money put in; gives change.
- Front Panel. Captures selection; knows price of selections, materials needed for each; asks Cash Box how much money was put in; instructs Mixer what to mix.
- Mixer. Same as before.
- Dispensers. Same as before.

Snap Analysis of Designs 1 and 2

According to popular report, we are using object-orientation to reduce the impact of changes. Notice that we have completely revamped the machine. Arghh! Evidently, the first design was not a really good design. The key error was that the cash box knew the price of the coffee. That means that asking for different prices for different drinks raised havoc with the design. The new design is better in this regard. Since the front panel knows the selections and the price, we can change prices at will, and only change one component.

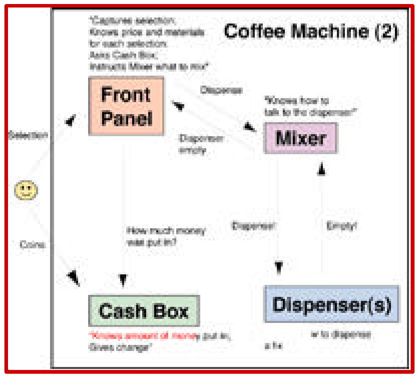
"Captures selection; Knows price and materials for each selection; Asks Cash Box; Instructs Mixer what to mix"
Coffee Machine (2)
Front Panel
Dispense
"Knows how to talk to the dispenser"
Mixer
Selection
Dispenser empty
Coins
How much money was put in?
Dispense!
Empty!
Cash Box
Dispenser(s)
"Knows amount of money put in; Gives change"

Figure 4: Design of coffee machine (2)

I voice a worry at this point: the front panel looks awfully smart. It hardly deserves to be called a "front panel." I call this the "mainframe" approach to design. One of the objects in the system has all the smarts. Almost any change to the system can be accomplished by changing the mainframe object. You may have come up with a slightly different mainframe approach. You keep the front panel dumb; all it does is register a selection. You add a fifth object, which in a spark of inspiration, you call the Controller. The Controller knows everything the second design's front panel knows, only does not physically have the buttons. The front panel tells the Controller the selection, the Controller talks to the cash box, the Controller tells the mixer what to dispense.

Although the trajectory of change in the mainframe approach involves only one object, people soon become terrified of touching it. Any oversight in the mainframe object (even a typo!) means potential damage to many modules, with endless testing and unpredictable bugs. Those readers who have done system maintenance or legacy system replacement will recognize that almost every large system ends up with such a module. They will affirm what sort of a nightmare it becomes.

Although I voice the worry, I cannot quantify it, and so we leave the design the way it is, with either the front panel or the controller being the mainframe object.

Arnold Visits Again

Arnold comes back a while later with a brilliant idea. He has heard that some companies use their company badges to directly debit the *cost of coffee purchases from their employees' paychecks. Since his employees already have badges, he thinks this should be a simple change.*

We add a badge reader and link to payroll. I ask you to make the design change. Ready, set, Go.

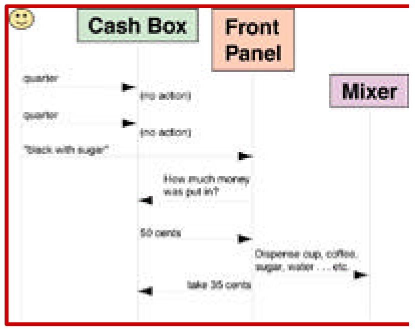

Figure 5: First part of the interaction diagram for design (2)

How does your new design change the system? You should find that this is not as bad as it sounds. To make the result clean and robust, we need only a change in attitude. It is no longer possible to ask the cash box for the amount of money put in. None was put in. So we borrow the inquiry style used by credit cards. When you go to pay for a meal, the restaurant does not ask American Express or Visa, "How much money does this person have?" The restaurant asks, "Can this person accept a charge of the following amount?" And the answer comes back yes or no.

So now, the cash box answers only one question from the front panel: does the customer have <amount> of credit? We become indifferent to whether the customer pays by cash or direct debit. Further, payroll can shut the credit down at a certain point for whatever reason. If your original design already worked this way, give yourself ten extra credit points. Even though it would have cost nothing to have designed this way from the start, the design was not obvious at the start. You might have arrived at the design if you had enquired after how the design requirements might evolve in the future. The responsibility, "knows how much money is put in," is sensitive to assumptions and to a technology. "Knows whether there is sufficient credit" works the same for cash, but is less specific about assumptions and technology. It is therefore more robust. Design (3) (Figure 6) changes very little from design (2). The new responsibilities are:

COFFEE MACHINE (3) DESIGN:

- Cash Box. Accepts cash or charge; answers whether a given amount of credit is available.
- Front Panel. Same as before, but only asks Cash Box if sufficient credit is available.
- Mixer and dispenser. Same as before.

The code in Listing 2 implements Design 3. Try to say why this design is or is not "better" - in arguable terms. Avoid arguments like, "because one should/shouldn't use objects." I have even found that "simpler" is an argument fraught with conflict. See what you come up with.

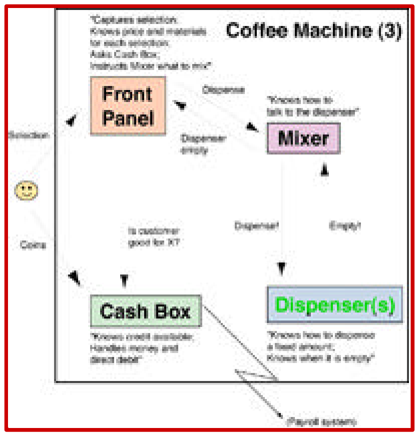
Coffee Machine (3)
"Captures selection; Knows price and materials for each selection; Asks Cash Box; Instructs Mixer what to mix"
Front Panel
Dispense
"Knows how to talk to the dispenser"
Mixer
Selection
Dispenser empty
Coins
Is customer good for X?
Dispense!
Empty!
Cash Box
Dispenser(s)
"Knows credit available; Handles money and direct debit"
"Knows how to dispense a fixed amount; Knows when it is empty"
(Payroll system)

Figure 6: Design (3), using credit instead of cash

I am still not happy with the design. We'll see why in the next part of the article. See if you can improve the design, and say why it is "better," while waiting. If you have a really good or different solution, send the responsibility statements and interaction channels to me at arc@acm.org. I'll publish some of the more interesting designs on my web page, http://members.aol.com/acockburn. o

References

1. Ward Cunningham, and Kent Beck. "A Laboratory for Teaching Object-Oriented Thinking," *ACM SIGPLAN* 24(10):1-7, 1989.

2. Alistair Cockburn. "In search of methodology," *Object*

Magazine, July, 1994, pp.52, 54-56, 76.

3. Trygve Reenskaug, with P. Wold, O. Lehne, et al. *Working with Objects* (Prentice-Hall, 1996).

4. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software* (Prentice-Hall, 1990).

About the Author

Alistair Cockburn, Consulting Fellow at Humans and Technology, is in Oslo this year as special advisor to the Central Bank of Norway. His recent book is Surviving OO Projects. Besides teaching OO design and project management, Alistair specializes in cognitively simple design techniques and asking "What is OO design quality?" He likes sitting underwater, dancing, and learning languages.

| Top | Search