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

Object-Oriented Analysis and Design, Part 2

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

Java Solutions

Brewing a good cup of Java takes careful design, even in C++.

This is the second of a two-part series on a problem I use to teach object-oriented design. The problem is simple, strong on "design" issues, and illustrates a work situation in which circumstances change regularly. It provides a touch point for discussions of designs in even large systems.

In the previous article (*CUJ*, May 1998), I said that in communicating a design, we provide the following:

- the name of the key components in the system
- the main responsibility of each
- a drawing of the key communication paths between each component
- a list of the services provided

The first article showed a simple design for a bank, and also for the first part of the coffee-machine problem. Here is what happened last time.

The Coffee Machine Problem Recapped

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 on their lunch and coffee breaks. 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 got together and decided there would be a coin slot coin return, coin return lever, and four buttons: black, white, black with sugar, white with sugar. You designed a suitable machine using objects. I showed the typical solution I get at this point. It 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.

We installed the machines. Then Arnold visited us, and gave us an extra requirement:

"Add bouillon, at twenty-five cents."

We added the extra button, and designed the second machine. I was not happy, because the second machine had a radically different

allocation of responsibilities. (I am hoping, in all of this, that your own, personal designs worked out better than the typical ones I see.)

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 if enough money has been put in; instructs Mixer what to mix.
- Mixer & Dispensers. Same as before.

I was still not happy with the design, because the front panel had become a "mainframe" object. It knew everything about the machine. This typically leads to a maintenance nightmare. But I could not argue against it better than that, so we proceeded with the design.

Arnold visited us again, and added another requirement:

"I want to use company badges to directly debit the cost of coffee purchases from the employees' paychecks. Since there already are badge readers, this should be a simple change."

We added a badge reader and link to payroll. We made the design change. It turned out not to be too bad. Our responsibility allocation was basically sound, so we had to change only the cash box so it could report "sufficient credit available," instead of "coins deposited."

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 & Dispensers. Same as before.

We pick up the story at this point...

Arnold Gets Competition

People are starting to buy Starbucks lattes instead of Arnold's coffees. So Arnold wants the machine modified just slightly, so that he can create a "drink of the week." He wants to be able to add new drinks and change prices any time, to match his competition. He wants to be able to add espresso, cappuccino, hot chocolate, latte, choco-latte, steamed milk - in short, anything he can mix together.

We add a couple more buttons, a milk steamer and dispenser, and some more powder dispensers. We conclude that our cash box design is pretty safe as it is, and discuss how to meet Arnold's request.

The first, and typical suggestion at this point is to beef-up the mainframe object so that it knows everything. I put my foot down, finally, and ask for a really good, robust design, so that we or Arnold can change the products and prices at will, without tearing the machine apart any further. This works best if you cover up the diagrams that follow and do your design on your own. Ready, set, Go.

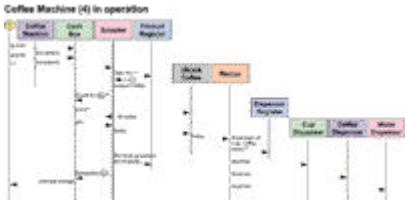

Figure 1: The Coffee Machine, design 4

The design we come up with at this point bears no resemblance to our original design. It is, I am happy to see, robust with respect to change, *and* it is a much more reasonable "model of the world." For the first time, we see the term "product" show up in the design, as well as "recipe" and "ingredient." The responsibilities are quite evenly distributed. Each component has a single primary purpose in life; we

have avoided piling responsibilities together. The names of the components match the responsibilities.

COFFEE MACHINE (4) DESIGN:

- Coffee Machine. Knows how the machine is put together; handles input.
- Cash Box. Knows how much credit is available; handles money and direct debit.
- Selector. Knows products and selection; coordinates payment and drink making.
- ProductRegister. Knows what products are available.
- Product. Knows its own price and recipe.
- Recipe. Tells dispensers to dispense ingredients in sequence.
- DispenserRegister. Acts as a librarian for the dispensers; controls nothing.
- Dispenser. Controls dispensing; tracks amount it has left.
- Ingredient. Knows its name only.

Figure 1 shows the new design. Figure 2 shows an interaction diagram for it. The code appears in Listings 1 and 2. Listing 3 shows sample input for the test program, and the output is in Listing 4.

When Arnold wants to add a product, we add a new Product to the Product Register. Should Arnold change the recipe or price of a product, we have exactly one, localized change to make. In other words, we have reduced the trajectory of change, as described in all the advertising on object technology.

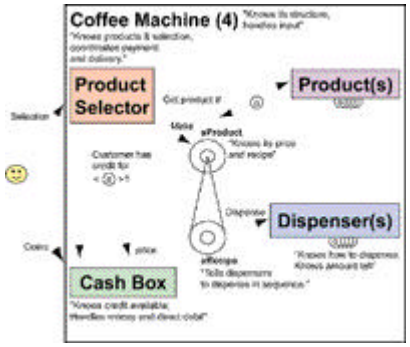

Figure 2: Interaction diagram for design 4

For the first time, we have a component that is not static. The selected Product rolls around the system, sharing its knowledge with the fixed components. For many newcomers to Object-Oriented (OO) design, such an object is non-intuitive. However, it is a common sort of OO design, as it provides the localization of changes we have been searching for.

There are two other changes from the previous designs. First, there is no mixer. Some of you will have noticed in the earlier designs that the mixer was doing very little, and with the new design, it vanishes entirely. The other change is that Ingredient is its own type of object. In the first designs, we used strings and enumerated types to identify a product or an ingredient. One of the things I have learned over the years is that a string identifier or fancy data type in a program almost always will come into its own over the life of the project, and so these days I turn them into objects right from the start. Had Arnold's designers done that, they would have spotted the Product object a lot earlier. I also suspect that Ingredient is going to grow in capabilities over time.

Evaluating the Quality of a Design

We saw in the first article that a design consists of:

- components,
- responsibilities,

- interaction channels,
- services

But what makes a design good? How might we have detected the flaws in the earlier designs, or gotten to that fourth design faster? My answer is in the form of a principle:

Discussing the quality of an OO design is discussing the futures it naturally supports.

Let us first settle on one important point: each of the four designs was fine in its time. Each was object-oriented, and each worked. There was no intrinsic fault in any. What differs in them is that they support different futures. In the first design, we did not think that the future might call for prices varying by product. Hence, the design did not easily support that. In the first two designs, we did not think of direct debit or credit cards.

The third design is constructed so that most of the system is indifferent to whether cash or credit is used, supporting both futures naturally. The fourth design is constructed so that most of the system is indifferent to the actual prices and recipes. It naturally supports a future of changing recipes and prices.

At OOPSLA '97, there was a workshop on OO Design Quality. In that workshop, we found several dozen dimensions of quality or "goodness." One of those was simplicity. At our table, we found people disagreeing strongly as to which design was simpler, and then found there were at least six different ways to look even just at what makes a design simpler. However, there was consensus there, as in most places I visit, that a good OO design supports change well.

It is good to find consensus on even one point. But does that mean we can't discuss the quality of a design until we have planned the future?

E-e-yuck! I don't have a real answer yet, but it does appear that capturing the abstractions in the domain actually helps with handling futures. Designs 1-3 of our coffee machine did not capture the abstractions, and design 4 does a better job.

I regularly question the futures and check the trajectory of change in designs, and in fact that is how my class and I found design 4 in the first place. We took a student's design and asked: "How many components do we have to change to add 'mocha' at $1.25?" The answer was "two." So I asked: "Is it possible to create a design in which only one component has to change?"

We found that two responsibilities were affected, those of price and recipe. They were in separate objects at the time. To create a single point of change, we merged the two responsibilities and asked whether their merger meant anything. Someone called out: "Product!" And the room gave an audible sigh of relief, as they recognized a better fit of the abstractions to the problem (see test 2, below). It does seem that recognizing and creating the abstractions would have led us to the design that protects the futures quicker.

Six Tests for Evaluating a Design

In general, I use and have watched people use six tests to help evaluate a design. They are

1. Data Connectedness
2. Abstraction
3. Responsibility Alignment
4. Data Variations
5. Evolution

6. Communications Patterns

The Data Connectedness test checks whether you actually can traverse the network of collaborations to gather all the information you need to deliver the services. (I have seen a major system design fail this test!) The Abstraction test checks whether the name of the object conveys its abstractions that is, whether the abstraction has a natural meaning and use in the language of the domain experts. Very many objects do not do well in this test. Although the test is subjective, my experience is that everyone gets a sort of "ahh" feeling when you improve the abstraction, as with the introduction of the Product object, above.

The Responsibility Alignment test checks whether the name, main responsibility statement, data, and functions align. During design evolution, usually the functions grow well past what is required by their names or primary responsibilities. Sometimes that is the time to split the object; sometimes it is time to rethink what abstraction you really have in front of you. The Data Variations test checks that the design naturally handles all the sorts and shapes of data the system will encounter. The Evolution test is the one I already described. It asks, what changes are likely in the business rules, technology, services, etc., and how does the design handle them? How many components have to change? Finally, the Communications Patterns test checks for oddly shaped run-time communications patterns. The designer particularly looks for cycles, but sometimes other odd shapes show up. Sometimes nothing is necessarily wrong with any one shape, but you may get suspicious and discover something out of alignment.

The experienced designers I compare notes with pay closest attention to the Abstraction and Responsibility Alignment tests. One even said that the Responsibility Alignment test covers all the rest. When I showed this coffee machine problem to him, he immediately criticized the first three designs. He said: "There are no abstractions here, just machine parts; and the 'front panel' doesn't even say what it is good for, it just says *where* it is." So, in the absence of a known future, it seems that the Abstraction and Responsibility Alignment tests may help

predict the robustness of the design. I am not ready to assert this too strongly yet, not based on the evidence of a mere coffee machine problem. But I am ready to nominate it.

Postscript

The coffee machine problem is handy. I have used it to teach basic OO design in many situations: to college freshmen in a two-hour challenge situation, to business people with whom I have to spend several days doing OO business modeling, and to experienced non-OO programmers learning OO object design. I am also using it to compare OO design approaches. I still encounter new solutions. I hope you found it challenging, too. If you have a different and "better" solution, feel free to write me at **arc@acm.org**. o

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