Writing Maintainable Code IV: Modular Code

Lincoln Stoller, Ph.D.

1/1/94

Overview

This article presents a simple method for designing modular 4D software based on the perspective of the problem domain. This is the last in a series of four articles.

A Simple Modular Design Method

In the previous article in this series [Stoller, 1993] I began my discussion of modular code. The I explained the keys to designing maintainable software. This involves first predicting which system components will, and which will not change, followed by dividing your software into modules along the lines of the permanent components. The first step is to consider the system from the user's perspectives — what is called viewing the system from the "problem domain."

In this article I describe a simple, seat-of-the-pants method for designing and evaluating modu designs for 4D applications. I provide a method that makes sense on the first reading and is simple enough to stick in the mind. This won't make you an object-oriented design expert, but i will give you the skill to base your projects on an effective, modular code base.

A design method helps to organize problem domain specifications into computer implementable structures. This requires choices that have a strong impact on the application's performance. A good design method enables you to see far enough ahead to know how current choices will affect subsequent alternatives.

An increasing number of books and articles present general, complex methods for creating modular code. The authors seem to be on a quest for the ultimate, all-encompassing theory. I've found their methods to be as ungainly and implausible as 17th century flying machines. The problem lies not just in finding a powerful method, but in presenting one simple enough to beco second nature. Where others offer complex, muscle-bound methods to overcome Goliath design problems, I offer you a slingshot.

My method has five steps:

- Break the specifications into tasks and components;
- Group the components into modules;
- Identify where data are stored;
- Enumerate the information shared between modules;
- Rate the design in terms of the module's independence.

Step Zero: Begin with specifications

Before beginning any design you need specifications. These should be written in the user's term and should reflect how the user gets the job done. Avoid making decisions about data storage, files, fields, procedures, or what the interface will look like. Leave out technical computer details. The specs should make sense to users without translation or explanation. If part of the specifications are based on the layout of computer screens or reports, rewrite them in problem-oriented terms. The specifications should express the original problem in the terms of the "problem domain."

Step One: Enumerate tasks and components

Begin by listing your application's general tasks directly from the specifications. Tasks are actions that the user takes to get things done. Tasks usually fall within the three categories of inputting data, processing data, and reporting. Extract the general tasks by rewriting the specifications in an abbreviated, list format. Be as brief as possible. If something needs to be explained try to do it by enumerating the terms involved and grouping them together. Don't feel constrained to listing tasks in a single column.

Each general task should have a readily identifiable name: it's a linguistic truth that central concepts are assigned their own labels. Of course it would be the users who would be aware of th appropriate terminology, which is why you need to write the specifications in the user's langua If your tasks don't have a simple title, it is either because they are not central, or you need to solicit further explanation from the user.

After listing the general tasks the next step is to resolve them into components. Components are either particular aspects of a task, or intermediate steps to completing a task. Components are still elements of the "problem domain," so don't drift off into the realm of computer algorithms relational data storage. Avoid writing a solution-centric description until after you have established a modular design.

Tasks and components express different levels of detail. For example, an application that handl vendors has the task of entering new vendors. This task has components that include specifying address, creating a vendor account, and listing vendor products. It isn't necessary to enumerate the elements of an address, account, or product list unless these elements are processed in an unusual manner. Make sure to list all the items hidden under a single task. The task of reportin for example, is composed of producing many reports, and they should all be listed. The followin two examples illustrate this step.

Example 1.1: Checkbook Accounting

The checkbook accounting example was introduced in the previous article and has the following specification:

> **Specification**
>
> **The user deposits and withdraws money; the bank provides various services (stop payment, wire transfer, prints checks, pays interest) for which it charges a maintenance fee; the bank supplies statements and confirmation documents; there is information particular to the bank account itself (account number, bank address, hours, bank manager, etc.).**

In Figure 1 I've rewritten this specification in terms of six general tasks and 20 components.

TASKS	COMPONENTS
Deposit	add, modify, delete
Withdraw	add, modify, delete
	mark withdrawal as covered
	print checks
Special Bank Services	stop payment
	wire transfer
	add interest on balance
	insufficient funds charge
	account maintenance fee
	check printing charge
Review Past History	locate specific items
	sort items
Print Reports	running balance report
	print list of items
Set Up Account	modify account information
	print account information

Figure 1: Preliminary design for checkbook accounting.

Example 1.2: Job Costing

Job costing involves managing jobs for customers according to the following specification:

Specification

A job consists of numerous items or services entered and billed to a customer at different times; a customer is the representative for whom the job is performed. The task of a job costing application is to enter and store customers, jobs, and items ordered on jobs. It must track debts, send out invoices, receive payments, prepare customer statements, and income and expense reports.

This, along with some minor embellishment, yields the tasks and components shown in Figure 2.

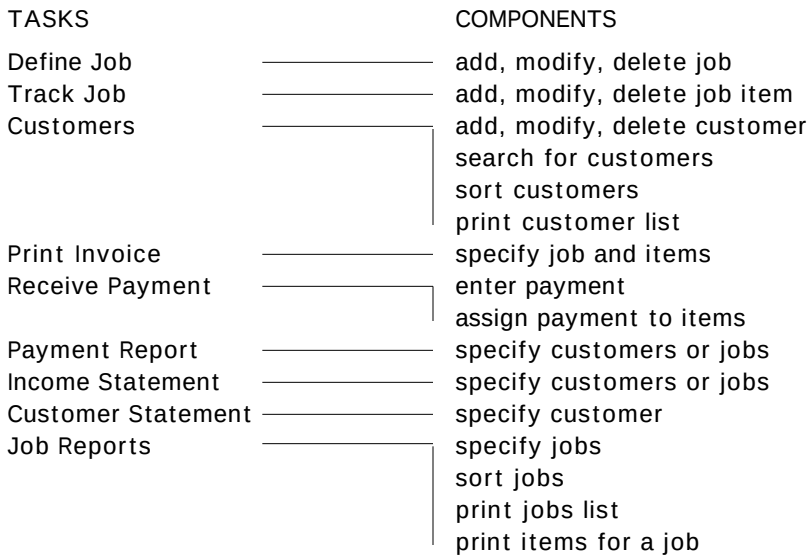

Figure 2: preliminary design for job costing.

Step Two: Group the components into modules

The purpose of breaking out components is to see what is involved in each task and to relate the elements to each other. You can then gather related tasks and components into modules representing related actions.

Determining how to group tasks and components is a judgment call. After completing the fifth st of this method you'll be able to evaluate your design and compare it with alternatives. But at th stage you have to let the components group themselves. Avoid grouping components whose similarity is only functional, such as sorting jobs and sorting customers, and focus on the objec being acted upon. Group components that perform related services on a common object or for a common objective. Following the terminology used in object-oriented design the components that compose the module are called "methods."

Example 2.1: Checkbook Accounting

At first glance the modules in the check book example are deposits & withdrawals, special services, reports, and the account. However, examining the components more carefully you see that deposits and withdrawals share the common objective of updating the account balance. The special services also update the account balance by adding or removing funds. Interest and wire transfers are both deposits; check printing, insufficient funds charge, and maintenance fees are special kinds of withdrawal. Gather these related components into a single “transactions” modu

The component labeled “stop payment” is an unusual service. It is not a deposit or withdrawal b instead acts to stop a preexisting withdrawal. The scope of this action is limited to the transaction module. In a more complex system this would become part of a submodule comprised of modifications to transactions — a module within a module — but this example is too simple to justify the extra effort. In conventional object-oriented terminology this submodule is called a “private method.”

Another common objective is producing reports: the running balance report, the list of items report, and the account information report. Define the second module as the reporting module. you will discover, all three of these reports can be absorbed into other modules so that the reporting module can be eliminated entirely. But that comes later.

Which components remain outside the scope of transactions and reporting? All that remains is t account itself, whose functions include initializing and modifying account information. So label the last module “account.” The three modules are illustrated in Figure 3.

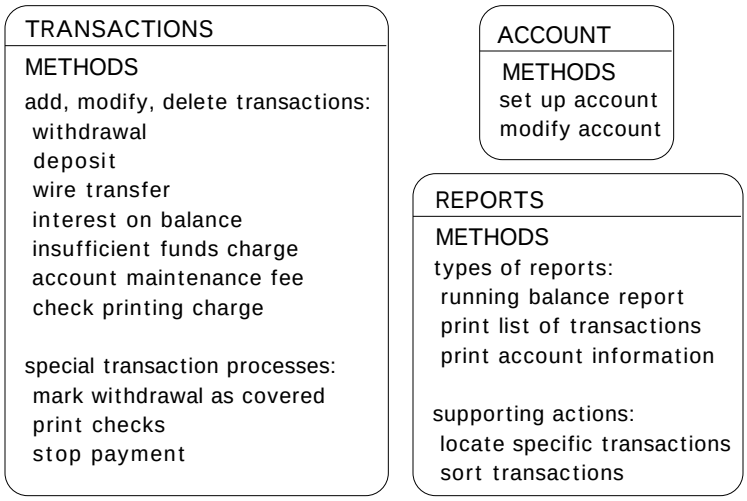

Figure 3: Preliminary checkbook accounting modules.

Example 2.2: Job Costing

The objectives of job costing are managing jobs, customers, payments, and producing reports. Yo can see this by going down the list of components in Figure 2 and identifying the object of each action. This grouping yields jobs, customers, payments, and reports modules.

I'd like to take a different approach from the previous example by considering reports accordin to the data they report on. The job, customer, and payment reports concern jobs, customers, and payments respectively. The customer statement concerns all jobs for a given customer, and the income statement concerns a selection of either customers or jobs. Using these distinctions defi the printing of jobs, customers, and payments reports to be methods of the jobs, customers, and payments modules. Printing the customer statement and the customer income statements are methods of the customer module. The jobs income statement is a method of the jobs module. Thes job costing modules are shown in Figure 4.

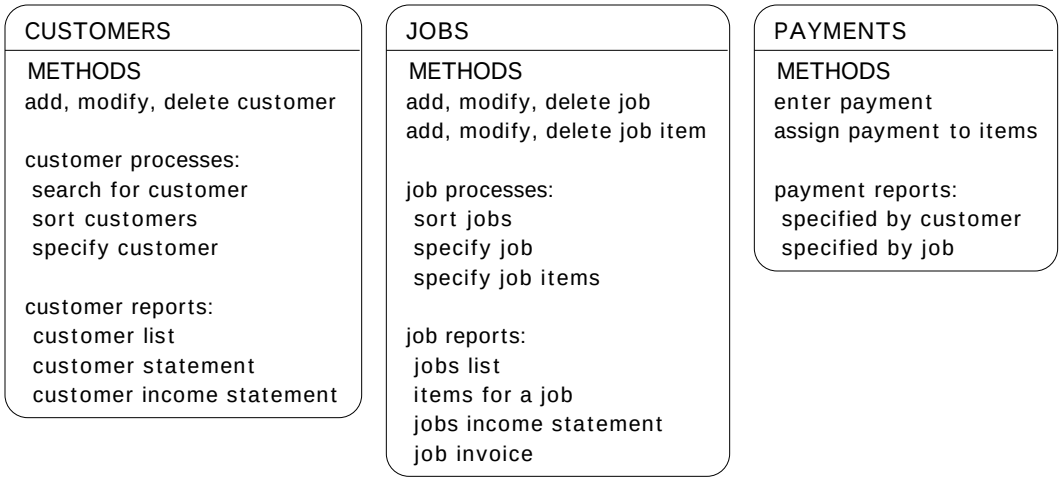

Figure 4: Job costing modules.

Associating particular reports with the data contained in the report makes the statement that t reports have less to do with each other and more to do with the information they display. It also says the evolution of reporting functions will be determined more by changes in the modules they're associated with, and less with features common to reporting. This is a clear judgment ca that requires knowing more about the system than I have presented here. It is also an example o an important design decision you might have to make without adequate information!

Step Three: Identify stored data

To judge design efficiency you need to know how the data, stored in the data file, relate to the modules. Since the modules represent all application functions all data are referred to in one o more of the modules. To establish a relationship between the data and the modules, create data storage areas for each module. Describe the information used or acted on by each method. In Figure 3, for instance, adding a transaction in the Transactions module acts on transaction data Sorting transactions in the Reports module also acts on transaction data.

Modify the diagrams shown in Figures 3 and 4 by drawing lines under the lists of methods and creating data areas. List all data involved in each of the module's methods in that module's data area. Because both Transaction and Report modules act on transaction data, write "transactions" in the data area of both modules, as shown in Figure 5. It's easy to identify the data used by reading the module's methods: the transactions and account modules use transaction and accoun information respectively, the report module uses both transaction and account information.

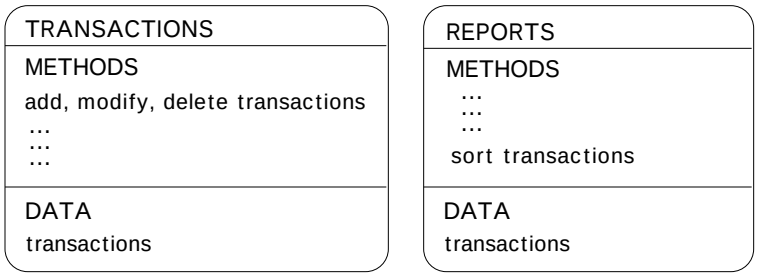

Figure 5: Transactions and Reports modules including data.

Next normalize the data areas. I'm using the term "normalize" loosely: I simply mean that a particular data item can only appear in one data area. When the same datum appears in multiple modules, such as in Figure 5, make a choice and assign it to the module that makes the most frequent reference to the datum, or to which the datum is most closely related. Remove all multiple references to the same datum.

Let's return to first principles. You want to design applications whose independent structural elements match the independent elements in the problem domain. Information that is independe in the problem domain should be stored and handled independently. Since the whole discussion based in the problem domain only talk about data in problem-domain terms; don't worry about file structures, normal forms, many-to-many relationships or any other programming level concepts.

Modeling the independence of data in the problem domain may sound straightforward, but it's difficult in practice. The difficulties don't occur here in the design phase where we can create and move data around with a wave of the hand. They arise in the programming phase where the modular design is translated into relational file structures. That's when you have to worry abou things like file structures, key fields, normal forms, and parent-child relationships. I discusse a method for deriving a relational 4D file structure in a previous article [Stoller, 1992].

Example 3.1: Checkbook Accounting

In most cases it's easy to determine which modules should contain which data. In some cases yo have to break large data concepts into smaller components and assign the data components to possibly different modules. Modules that act on data they don't contain will access it by messag passing, as discussed in the next section. Figure 6 shows the module structure for the cash accounting example.

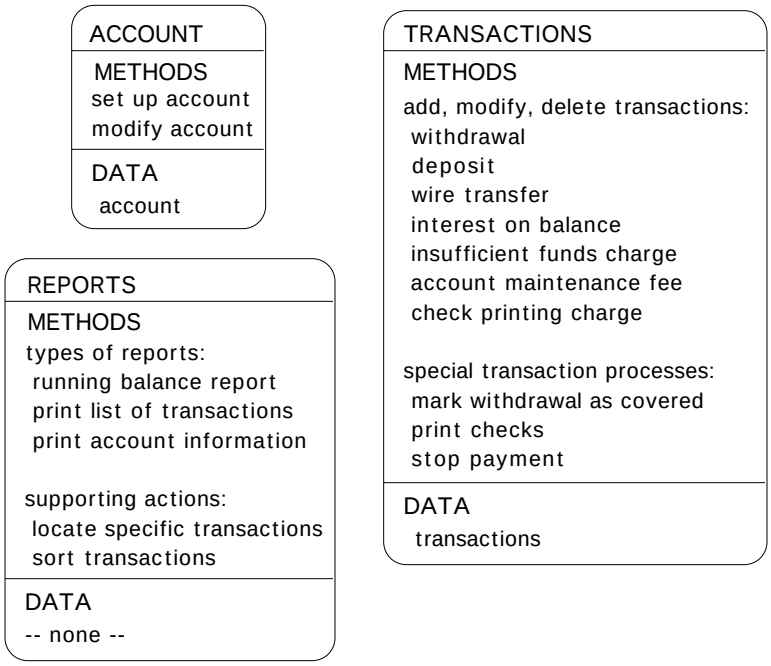

Figure 6: Structure for cash accounting modules.

Example 3.2: Job Costing

The data stored with the Customers and Payments modules are simply customers and payments respectively. The Jobs module stores both job and job item data.

Step Four: Enumerate module interactions (messages)

When a method acts on data outside their own module's data area, the information must be passe from another module. The movement of data, or the passing of data-related information, is calle messaging. Using messaging, one module can retrieve or affect data stored in other modules. A method can also send messages that trigger methods in other modules.

In programming terms, methods are procedures while modules are families of procedures. When one method "triggers" another, it means roughly that one procedure calls another. But in order avoid getting bogged down in the syntax of programming or the featureless landscape of compute code, it is better to use this new language of modules, methods, and data areas.

In the same manner, a message is just some directive, or variable, set up in one procedure to be read and acted upon by another. But this description underplays the importance of messaging i determining the quality of design and the maintainability of code. Messages, by their nature, a links between otherwise independent things. The more messages supported, the less independen and less modular the design. A design consisting of modules that don't relay any messages is a

design composed of independent applications — a design with modules whose every action requir interaction is one where the whole concept of modularity is a fiction. Messaging is simple in practice, effected by passing variables and accessing files, but it has important design implications that deserve careful attention.

Incorporate messaging in the module diagrams by drawing lines connecting each method with th data in other modules that it uses or affects. Every method relying on data in another module ge a line from the method to the data area. Don't be concerned with the direction the data flows — whether it is being read from or written to — the lines don't need arrow heads. In the case of transactions and reports, for example, draw a line connecting the sort transactions method with the transactions data.

Example 4.1: Checkbook Accounting

To draw the messaging lines for the module structure in Figure 6 you need to know that the account balance is maintained as a separate item stored with the account data. Consequently, every transaction affects account data when it changes the account's balance. Methods in the reporting module also draw on account data when they display this balance. There are four reporting methods that draw on transactions data: the running balance report, the list of transactions report, and the methods of locating and sorting transactions. This is enough information to draw all the messaging lines shown in Figure 7.

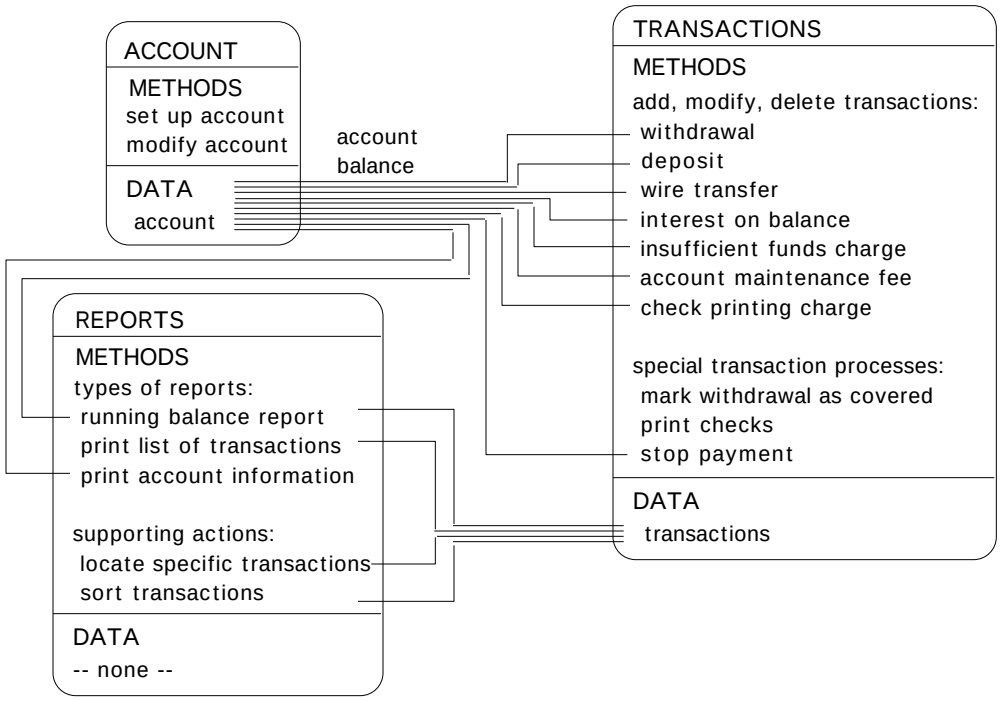

Figure 7: Cash accounting methods, data, and messages.

Example 4.2: Checkbook Accounting

I have to supply a number missing details before establishing the messaging connections betwee job costing methods and data. In this case the details are fairly obvious, but in general messagi analysis motivates design decisions about the exact information involved in each method. It also draws attention to the composition of information stored in the data areas. This analysis helps you extract details from the specifications that are either unstated, overlooked, or misunderstoo

In the Customers module the customer statement and customer income statement need to access information concerning items billed and paid on each job. Consequently they both draw on the payments and job items data. In the Jobs module the addition of a new job requires access to customer information, since the job is always assigned a customer. The job invoice method also requires customer information to print the customer's address on the invoice. Data accessed by the methods "assign payments to items," "specified by customer," and "specified by job" are se evident. Drawing the message lines results in Figure 8.

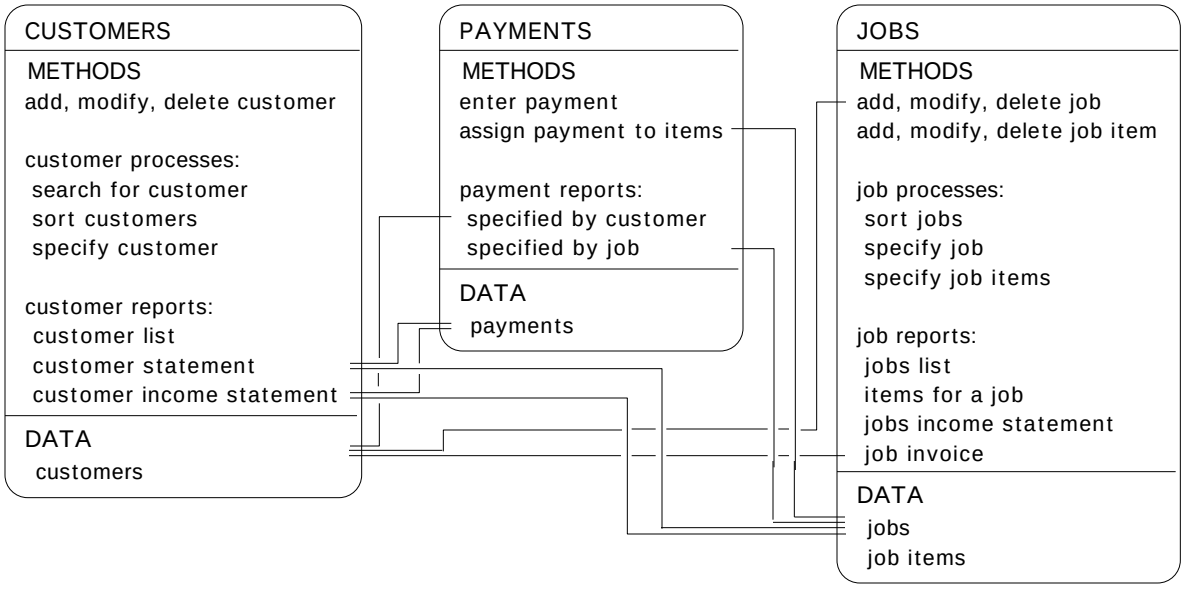

Figure 8: Job Costing methods, data, and messages.

Step Five: Rate the design

As mentioned above, the more messages passed between modules, the less independent they are. Each message means more programming overhead. And since a message represents data being use in a somewhat foreign context, each message is another point where data may be incorrectly written or interpreted. Consequently you can use a simple rating system based on the number o inter-module messages. The object is to design software with as few messages as possible.

The number of messages is a relative measure useful for judging alternatives — it's not an absol measure of the quality of a design. This is admittedly a rough and easily misapplied system: yo could simply lump all methods in a single module and have no messages! But this wouldn't represent the modular structure of the problem domain, so you need to exercise some judgment.

Example 5.1: Checkbook Accounting

There are 14 messages passed in the checkbook system in Figure 7. Six of these involve methods in the Reports module, a module that doesn't store any data of its own. The question is, do the report methods deserve a module of their own? Consider some of the benefits and drawbacks of this design.

The existence of a Reports module says that there is something about report methods that makes them independent from other modules and similar among themselves. If these reports could be

driven by a common core of reporting procedures, then grouping them together would be a good idea. If the reports shared common layout specifications, this would make even more sense.

On the other hand, if the forms of the reports are determined by the structure of the data they draw on, then a change in data structure will change the reports. In that case, reports represent data structures and should be methods of the modules that store the data.

In most cases reports are a reflection of the data they represent and the methods used to define these data. While it is also true that reports share common formatting, this is usually of minor importance. In 4D reports are usually specific to a data file — they rarely share the same output layout. From the perspectives of both problem domain and modular independence, then, it makes more sense to incorporate the reports into the Account and Transactions modules. Doing this for Cash Accounting yields the final design shown in Figure 9, which involves nine messages.

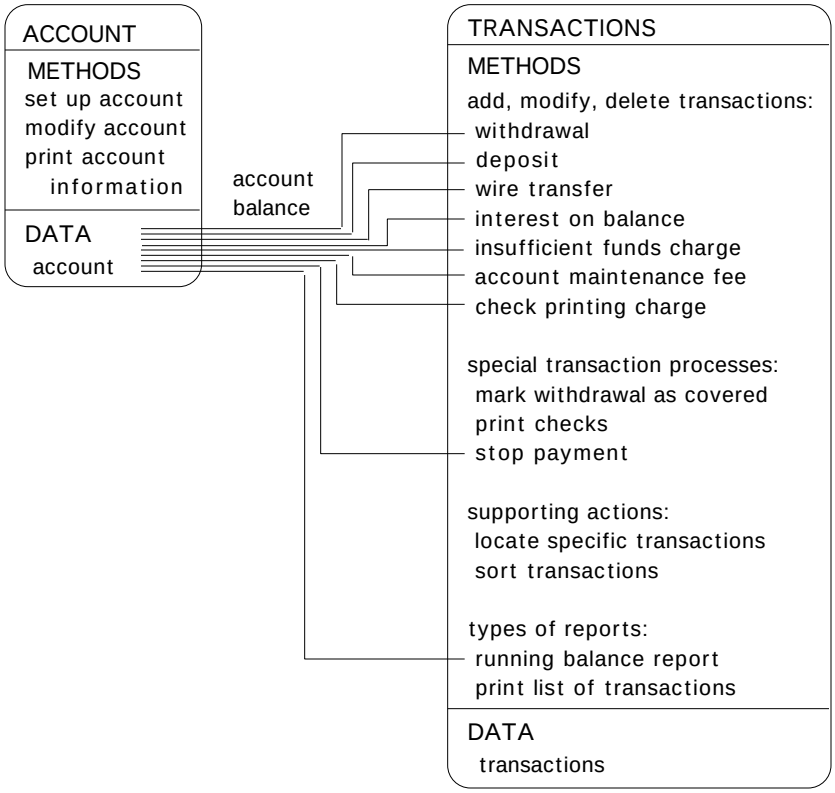

Figure 9: Final cash accounting modules.

Those reading this series of articles may notice that the modules of Figure 9 do not match those the example from the previous article. In the latter case, extra specifications differentiated the bank from the account, and the services from the transactions.

Summary

I've described a method for designing a modular application. The method has few steps, avoids technical details, and only requires a study of the application specifications, which you have to do anyway. The method is so simple, in fact, that it can often be implemented in a single step: study the specs, do a little doodling, and just write out the modules and messages. Drawing message lines helps a lot in determining whether your design is complete. And the number of messages makes the complexities of your design immediately apparent.

I have avoided discussing how to convert design into code and file structure for two reasons. Fi I already discussed this a bit in the Implementing Modules section of installment III [Stoller, 1993]. But more importantly, I believe programmers put too little effort into thinking their problem through before starting to code. This method is primarily a thinking tool — its purpose to help the programmer see how current choices affect subsequent alternatives.

Try this method before starting your next project — I guarantee you'll fine it useful. And if you want to create more valuable and maintainable applications, don't start coding until you can see the forest for the trees.

References

Graham, Ian 1990. Object Oriented Methods, Wokingham, England: Addison-Wesley.

Nerson, Jean-Marc 1992. Applying Object-Oriented Analysis and Design, Communications of the ACM, Vol. 35 N°9, p.63-74.

Stoller, Lincoln 1992. Working Toward Step One, a Method for Deriving a 4D File Structure, Dimensions, Vol. 1 N°5, p.39-46.

Stoller, Lincoln 1993. Writing Maintainable Code III: Stable Code, Dimensions, Vol. 2 N°6, p.31-35.

Wirts-Brock, Rebecca, Brian Wilkerson and Lauren Wiener 1990. Designing Object Oriented Software, Englewood Cliffs, NJ: PTR Prentice Hall