Writing Maintainable Code III: Stable Code

Lincoln Stoller, Ph.D

4/9/93

Overview

This article discusses basic principles of modular software design and explains why modular software is easier to maintain. This is the third in a series of four articles.

In the first two articles in this series I considered style-related aspects of 4D code covering th
topics of procedure breakdown, documentation, naming conventions and global variables. There
the objective was clarity and communication. In this article I describe what it means to view th
system specifications from the problem domain, and explain why code should be modularized.
Here the objective is stability, as described below. In the next article I'll present some tools fo
automating the modularization process.

Stable code is code that can be easily modified to accommodate evolving specifications. This can
be a major issue —— Horowitz and Munson have estimated that three times the original developme
time is spent in maintaining large software systems [Horowitz and Munson, 1989].

When stability is our objective we want to design programs so that the amount of reprogramming
needed to satisfy a change in specifications is comparable to the change in the system that is
being modeled. To put it another way, we want to avoid designing programs with internal barrie
that inhibit their evolution. This is a difficult goal because programs are more than a restateme
of the problem. There is extra overhead in translating system specifications to computer
algorithms, especially when high performance and ease of use are implicit. Nevertheless, we can
make progress using concepts of modular design that should become part of every programmer's
toolbox.

Common Problems in Maintaining Code

The problem of maintaining code arises any time you modify an established, working program.
This modification can be due to a change in what the program should do or an attempt to reuse o
work in a new project. It can also be part of the evolution of a protyped application that was ne

completely designed in the first place. Here are some of the problems that typically arise in th cases:

• Changes are requested requiring you to centralize functions that are currently located in different places.

• New specifications require modifying code that is logically complex and difficult to follow.

• Additional functions are added to a specific and inflexible algorithm.

• The user requests new functions that are similar to existing functions, but there is no documentation and you don't know where the existing code is located.

• The user wants to add new types of data that upset the existing structure and violate assumptions made in many algorithms.

The Problem Domain and the Solution Domain

The root of the problem is change. Not surprisingly, the key to designing maintainable software predicting which components of the physical system will change and which will stay the same. Once you know which elements of the problem are likely to change, they should be modeled in a way that will accommodate change. The effects of these changes should be isolated as much as possible from other parts of the software so that changes are contained. The problem boils down factoring the physical system into segments concerned with different aspects of the problem. Various likely and independent changes need to be isolated from each other and limited in thei effect on software. This gives you a software design that is both flexible and stable.

"Focus on building stable interfaces and encapsulate design decisions that are likely to change
- Grady Booch, Object Oriented Design with Applications

The first step in designing maintainable software is to view the system from the point of view o those who define and implement it. This requires a picture of the problem in its larger environment —— that is, an analysis of the problem that begins with the question "what are the underlying forces driving this system?" This is the view of the system the **problem domain**.
The purpose of examining the problem domain is to distinguish the system's driving elements from its supporting elements, as this will tell you which parts of the system will remain stable when other parts change.

This approach is distinguished from looking at the system from the perspective of how it's goin to be implemented. System specifications, as they're usually drafted, tell you what features nee to be supplied in software by defining the problem in terms of inputs and outputs, and by listi all the necessary functions. All this is part of a view in the solution domain. The solution, in this case, is a description of how the physical system handles and resolves the physical problem.

The problem domain view is generally not found by following traditional top-down or structured design methods, as these methods take the functions listed in the system specifications as a blueprint for the software. This is why modular software design, as espoused here by the advocates of object-oriented programming, is unfamiliar to the larger population of programmer schooled in structured programming.

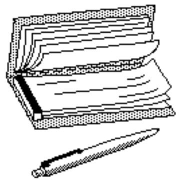
Despite its relative novelty, the idea of the problem domain is neither new, nor particularly complicated. In fact, the more you look at things from the problem domain, the more obvious an powerful a tool this approach becomes. This can be illustrated by the following example:

Checkbook Accounting Example

Let's analyze a simple personal checking system that manages deposits and withdrawals from a checking account, viewing the system from the problem- and solution domains in turn.

We'll begin with a brief specification: money is deposited and withdrawn by the depositor whil the bank provides various services, sets policy, charges fees, pays interest and supplies statements and confirmation documents. These functions are summarized in five categories give below.

The Specification



CHECKING

The depositor can write and deposit checks. Checks are "covered" when the funds are actually withdrawn and deposits are available after a certain processing time elapses.

CASH

The depositor will need to deposit and withdraw cash for general purposes. Cash transactions affect the account balance as soon as they are recorded.

SERVICES

The bank provides a variety of special services including wire transfer, stop payment, overdraft protection, check printing and interest on balance.

BANK

There are various codes, addresses and personnel associated with the bank itself. The bank sets policy on liability and access to funds.

STATEMENTS

The bank produces regular statements reconciling charges to the account with transactions, as well as deposit and withdrawal confirmation slips. The user supplies endorsed checks and deposit slips.

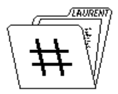
These 5 categories describe what the system does and how it works. A problem domain view focuses on the elements that underlie these specs and these elements can be identified with tho features that underlie the 5 categories. I'll take up the mechanics of arriving at the problem domain view in the final article of this series. Here I'll simply state one possible, not necessa unique, breakdown of the problem domain.

The Problem Domain

The elements underlying the specifications are transactions, services, the checking account and the bank. The interaction of these elements generates the functions and information described i the specifications.

TRANSACTIONS

There is a variety of cash, check and internal transactions that have the same effect on the account and differ only in documentation and processing.

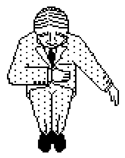


ACCOUNT

The account consists of identifying numbers, contact information, current balances, instructions for handling situations and other historical information.

BANK

The bank is a basic element because there are some data and types of processing information that do not depend on accounts or transactions.



SERVICES

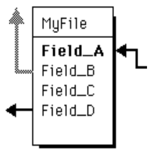
Banking services include daily account maintenance, transaction processing and the generation of reports. The history, status and instructions for performing these services are also included in this element.

The objective in this breakdown is to find the independent abstract elements from which all aspects of the problem derive. Elements in the problem domain don’t include items that are special types of other elements or derivatives thereof. In particular, neither reports nor specif processes like check or cash processing are considered elemental.

Those of us steeped in database design should note that this factoring is not giving us a set of files for a relational database structure. The elements include both data and methods for handli the data. Some of the elements consist of multiple instances, such as the transactions, but other could just as well be unique. The separation of data into files, and function into layouts, menus and procedures, does not happen until after the problem domain is factored into its basic elements.

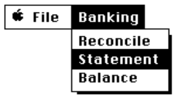
The Solution Domain

In contrast with the abstract fundamentals of the problem domain, the solution domain represen the mechanics of implementing the system. The specification itself is a solution domain view in that it describes how the bank provides checking services. At a computer programming-oriented level the solution domain view consists of the basic categories of software tools. Whatever application we ultimately create to support the checking account specifications, it will consist a combination of elements from the following tool set:



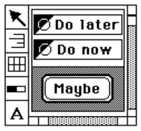
FILES

A file for system information, account information, bank services and transactions.



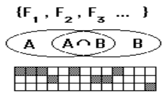
MENUS

Menus for reports, searching and sorting, entering account information and managing transactions.



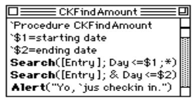
LAYOUTS

Input and output layouts for services and transactions, print selection layouts for reports, checks and deposit slips, as well as input layouts used as dialogs for displaying and printing account information.



ARRAYS, SETS, SELECTIONS

Arrays used to display multiple choices through scrolling lists and pop-up menus in dialogs. Sets to keep track of selections of transactions or services of different types. Selections to handle data for display, report printing and processin



```
`Procedure CKFindAmount
`$1=starting date
`$2=ending date
Search([Entry]; Day <=$1 ;*)
Search([Entry]; & Day <=$2)
Alert("Yo, `jus checkin in.")
```

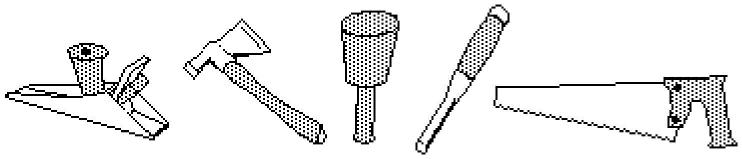
PROCEDURES

Procedures that handle all banking tasks and initialize the program, validate entered data, and manage internal tasks like assigning record IDs.

Compare the views in the problem domain with those in the solution domain. As long as these fo elements of the problem domain describe the functions of the checking account, any change to o element will have little effect on the others. For instance a new type of service could be added with little effect on the other elements. This stands in contrast to the solution domain, where a new service would likely require change in every category. As another example, consider addin multiple checking accounts to the problem. This could be done with limited alteration in the problem domain, but would require changes throughout the elements of the solution domain.

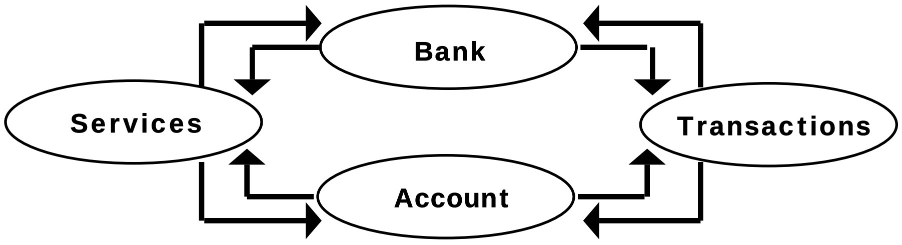
Software Modules

A software module is any segment of code that explicitly manages the information that affects i accomplishes a well-defined task, and has a well-defined effect. Modular software is more maintainable than other software designs if it is designed and implemented effectively. You can think of a module as a software tool.



Successfully designed modular software, like a successful tool-set, can be applied to a variety of problems.

Modular software techniques and an analysis of the problem domain are a perfect match: after t analysis identifies the independent concepts, the modular techniques give us a way of mapping them to independent software elements. The natural thing to do, then, is to define a software module for each element in the problem domain. This is shown in the following diagram for the checking account example. I've omitted some of the arrows for simplicity. If everything goes according to plan, the stability of the problem domain should carry over as stability of modula code.



The elements that result from factoring the problem domain form the basis for the modular software design.

Take the examples of possible changes mentioned above. In a module that handles bank services (in which all the procedures, menus, layouts and codes deal with managing services) you can be confident that the effects of any changes in services will be limited to this module.

This is not to imply that big changes in specs will only require small changes in software. This not the case. But what one expects is that all changes will have limited impact on the software a that redesign will be relatively easy. The amount of time needed to implement change will still proportional to the extent of changes to be made. However, the design, implementation, installation, documentation and debugging are all made easier than before.

Implementing Modules

Modules should share as little information as possible to preserve their independence. Except f
inputs and outputs, variables should not be shared between modules. Each module's code
structure should be hidden from other modules. This is just another way of saying that modules
should not need to know anything about how other modules operate, beyond knowing how to call
one another and what information each will return. The code structure of each module should
operate as much as possible like a "black box." One thing this implies is that modules should
avoid calling each other to obtain intermediate results. This concept of information hiding is
central to the design of modular software.

Modules are categories that describe the use of variables, procedure, functions and files. Since
does not have a mechanism to separate code into modules, we must use naming conventions to
distinguish the components of one module from another. A module consists of a collection of
interrelated variables, procedures, functions and files that are more or less isolated from othe
modules. If you define a two- or three-character code for each module and use this prefix in ea
of the module's procedures, then procedure names will be grouped together when sorted. Global
variables should be similarly prefixed to distinguish them from similarly-named variables use
in other modules. In this manner you can be sure that the mechanics in one module will never
invade the global variable space of another. Happily for us, 4D does organize layouts according
the file structure. Consequently, if we've assigned our files to the control of distinct modules,
layouts are automatically segregated as well.

Designing modules and designing your procedure structure are separate problems. We could
borrow the logic that guided the development of modular structure to infer that procedures
mapped directly from the problem domain will be the most maintainable. However, the closer yo
come to writing code, the more the language, operating system and hardware will impinge on
programming decisions. After a certain point modular design will lead to diminishing returns.
Whether it's worth the effort to stick to a modular design or just to get the job done in the most
expedient manner will depend on how much maintenance your program will need and how much
preemptive effort you can afford.

Problems with Modules

Design Overhead

Modular design adds an additional level of abstraction to software design that isn't always
appropriate. It takes a lot more thought and consultation to design a modularized system, and
there is no early payoff. The more widely you hope to reuse your design, the more factors you w

have to take into account, the longer it will take and the harder it will be to succeed. For small problems, where not enough is known about future conditions or where you've decided to use the cart-before-the-horse approach of design by iterative prototyping (which definitely has its place), this extra work may not be cost effective.

Nonfactorizable problems

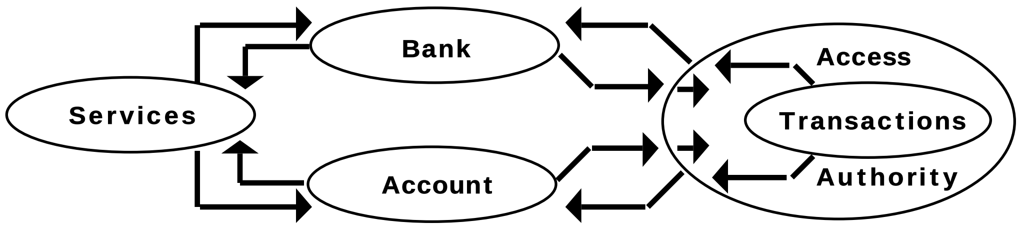
Not everything factors. Factoring and information-hiding are closely related, and where information can't be hidden, factoring doesn't work. A familiar example of information that can not be hidden is system-wide information. All of 4D's built-in procedures, as well as many utility procedures you may write yourself, also fall into this category. This is not a problem as long as these functions don't affect information managed by one module when it is called by another module.

Generic procedures and global variables always exist and usually provide information or functions needed in all modules. They either may be an encompassing feature of the problem domain, or their role may be totally internal to the workings of the code. In the first case there should be a module for information shared throughout the problem domain; in the second case there should be a module for systems and functions internal to the solution domain with no explicit role in the problem domain. As an example, the current group membership, if implemented, would be global to the problem domain, while an error code would be internal to t implementation.

Example

There are nonfactorizable structures that really do disrupt a modular structure. To illustrate such a case I'll go back to my checking account example and add to the previous specifications t condition that there are two kinds of users: limited-access and full-access users. Full-access users can print checks, view all transactions and mark certain transactions as "restricted." Limited-access users can't print checks and can't view, modify, create or delete restricted transactions. To support this example I must also say that access authority is more than just a property of transactions and needs to be handled separately.

I've revised the previous module diagram to reflect these new conditions. The modules labeled "access authority" and "transactions" are not independent. No matter how you divide up the problem domain, there will still be an overlapping situation of this sort. At the root there are n two independent specifications that apply to the same problem: the banking functions and the access conditions. Whenever something like this happens, you can factor the problem according one specification or the other, but you cannot factor the problem according to both at the same time. To put it simply, if the problem doesn't factor, then some of the modules will be interdependent.



If you can't factor the problem domain then some modules will be interdependent.

Relational Databases

4D's relational database structure is built on information sharing, not information hiding. The whole database structure —— all the files and records —— act as one huge disk-based global varia
space that is available to any procedure or module. This is an obvious clash of design methodology. Should access to the data file be controlled or modularized, and if so how? I don'
know.

Fragile Interconnections

Once you've designed your modules so that information is shared through a controlled interface
then what happens when you need to change the structure of the shared information? If the information is needed in many other modules you will have to change many interfaces. Changing
the module interfaces is a time-consuming task that can introduce many errors. This is not just
problem specific to modules —— it affects any implementation where the identity and flow of dat
is hard coded into the program. However, this drawback is especially difficult for modular architectures because they place a high priority encapsulation and involve substantial passing
information.

Over the Horizon

Object-oriented programming, or OOP, is widely discussed throughout the computer industry (e
Fichman & Kemerer, 1993), and is directly relevant to this topic. The object-oriented approach
divides into analysis and implementation, where the analysis is based on the modular design concepts that I've described. These ideas, I think, are basic problem-solving strategies —— as ol
as problem solving itself —— that are finally being applied to software design. The implementati
of OOP consists of specific tools that help to implement modular designs and overcome some of t

limitations mentioned above. These tools include support for classes, methods, encapsulation an multiple inheritance. While 4th Dimension is not an object-oriented programming language, we can still apply object-oriented ideas in our design phase. As 4D evolves it will more than likel give us tools to solve problems particular to modular design.

Summary

Designing maintainable software requires incorporation of future design possibilities in the present architecture. One method to accomplish this is to view the system from the problem domain and to factor it into a set of distinct elements. These elements represent persisting structures that, when mapped into software modules, allow for more stable and reusable softwar When there are many similar and interdependent systems exchanging information, the factoring process is complex. In the next article I'll give some techniques for automating the factoring process.

Acknowledgments

Thanks to Scott Ribe, Tracy Harms and Michael Billesbach for their helpful comments and to “th crew” on ACIUS compuserve forum for reading the manuscript and providing suggestions. Thank to Foofer D. for editorial assistance.

References

Booch, Grady 1991. Object Oriented Design with Applications, Redwood City, CA.:Benjamin/ Cummings, page 205.

Fichman, Robert and Chris Kemerer 1993. Adoption of Software Engineering Process Innovations the Case of Object Orientation, Sloan Management Review, Vol. 34 N°2.

Horowitz, Ellis and John Munson 1989. An Expansive View of Reusable Software, in Software Reusability vol. I: Concepts and Models, Reading, MA: Addison-Wesley, ACM Press, p.19-43.