

# **Building Software From Data**

Paul W. Homer

Copyright © 2006 Paul W. Homer

All rights reserved. No part of this publication may be reproduced stored in a retrieval system, or transmitted in any form or by any means electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher and author.

1<sup>st</sup> Edition

October 28th, 2006

## Contents

<b>THE STARTING POINT.....</b>	<b>4</b>
MY SHIFTING PERSPECTIVE.....	6
EXPLORING A SIMPLE EXAMPLE.....	9
GOING TO THE NEXT LEVEL .....	11
THE FINAL WORDS .....	14

## The Starting Point

Exploring old undocumented applications was my first real experience with computer software. The machine was an Apple ][+ clone – I had bought the dusty old hardware second hand – it came with a big box full of poorly labeled diskettes. The floppies contained a multitude of old software programs but not a single manual. I had great fun in trying to get each program to run, figure out what it did and then try to see if I could make it do something constructive. This initial experience – which drove my desire to study Computer Science – is probably all too familiar for most professional software developers.

Broadly speaking, most programmers are drawn towards software long before they decided to officially learn the craft of coding. These early experiences set our viewpoint on the essence of software. Once we learned to see software as a fixed set of functionality implemented by a collection of programs, we continue with this perspective by decomposing all new software related problems in terms of functionality. Software as a rigid set of functionality permeates all aspects of software development.

As such, the requirements we use to build a system come from analyzing the user's needs and boiling them down into a list of functions that need to be coded. Depending on the size of the project, the functionality is split into various pieces, and handed out to many different teams of developers. There may be an overall architecture, but it tends to get lost in the final design and implementation of the code. Programmers produce their own individual pieces of the system. Each piece copies in the data from somewhere else, applies some functionality to it and then copies it back out again. As the data moves throughout the program, it goes from method to method, component to component, library to library and of course function to function. Copying and converting data are the most frequent operations. Duplicate code is everywhere. This is a very common architecture for medium to large software systems.

Because the problems are initially framed as pieces of functionality, the programmers derive their answers as a sequence of steps. The execution of all of the steps implements a function. This is a perspective that is clearly driven by that underlying functional viewpoint. Implementing this directly in the code leads to breaking down the problem into the entire explicit series of steps needed to complete the tasks. The number of steps may be long and tedious, but for the most part each step is well understood and the whole sequence is easily understood. This approach exactly solves the functional problem, but nothing more. Essentially this is an approach where "brute force" is used as the primary tactic to pound the steps into the code.

Fortunately (or not), this approach to software development does not take a lot of experience to master. A smart young programmer straight out of school has the only necessary skill set – decomposition – needed to write code in this manner. If you can understand the function, you can break it down into a series of steps, code it into a language, and then beat it with a debugger until it works well enough. Enough pressure, enough young programmers, enough time and you are off to the races for the

development. That this creates horrific long-term problems is not something to spoil the day. The future be damned, we have a system to build. And yet, we are somehow left wondering if there isn't a better way, one that would produce a higher quality product? One that might make the next set of iterations a lot easier to implement? One that might better leverage the initial effort put into designing and building the code?

## ***My Shifting Perspective***

My first few years of university were extremely frustrating. I was surrounded by lots of fellow students, most of which were way ahead of me in learning how to write software. They gained experience in programming in high school – I had only discovered computers just before I started university. While I was flailing at the keyboard, they were often done, at home and probably in bed.

Mostly, my problems were caused by my bad habit of quickly belting out the code and then pounding at it recklessly until it worked. If it didn't work, I'd add an *if* statement here, or a *while* loop there. A couple of new variables and maybe that'll fix it, I'd be thinking to myself. Fortunately, I didn't save any of the code from that time, because I'm sure it was worse than awful. My approach to development was haphazard at best. My results were dismal. Because of that initial programming experience, I started searching for a better way to write code. If I was going to continue to program for a living, at very least, I needed to be good at it.

Exposure to abstract data structures was a defining moment for me. In it, data drives the underlying representations, which inverted my functionality perspective of the code. This understanding caused a profound shift in my understanding and abilities. That was when I went from barely being able to code to being able to easily and painlessly build any possible system (ok, it did take a few years to perfect it).

Both from influences in school and in my co-op work term experiences, I came to change my underlying perspective of development. Software – which had started as complex programs that implement lots of functionality, became nothing more than simple tools to manipulate data. This may not sound like that radical of a shift, but it absolutely was. The difference is in the order of importance. Previously, functionality was the most important aspect; data was necessary only as a secondary element needed to fulfill the functionality requirements. Organization of the code was thus based on the primary element – the functionality. In this perspective, the code is decomposed into smaller and smaller bits of functionality, carved up and implemented as the central core. This leads to a straightforward understanding of how to code the functions but the data in this type of implementation suffers horribly because it is copied in and out of the functionality so often. The most common results: brute force implementations, spaghetti code, redundant data copies, poor performance, and duplicated code everywhere. This perspective even percolates up to the structure and processes of the development teams, which are usually arranged around functional lines. This approach produces fragile software that is hard to understand and easily broken.

Flipping the order of importance between data and functionality changes the way code is designed and implemented. It switches from being concerned about finding a sequence of steps to representing the data as an internal structure. This new perspective not only changes the way we code, but also the architecture, the structure of the development teams and even the way we gather requirements for the systems.

Data structures helped reorient my perspective on software development. The ideas have also exerted a considerable influence within Computer Science including supporting formal idioms such as Data Directed Design and Object Oriented Design. Buried in many of the earlier ideas was an implicit change in the perspective of code. The Art of Computer Programming, by Donald Knuth, – the seminal set of textbooks on algorithms and data structures, does not talk about how to write an editor or how to get user permissions. It focuses only on the best-known ways to represent general elements of data inside of a running computer program. It focuses on the data – and presents algorithms that can be used to manipulate it.

Understanding the key points behind data structures lead to my understanding of how to easily build complex systems. The notion starts with the simple understanding that software is just a tool to manipulate data. Nothing more. All that functionality – breaks down into either viewing, adding to or changing the data in a program. It all depends on the data. Understand the data, and you understand the program. The presentation may be complex, but the underlying functionality is not. If the data is available and consistent, then implementing known algorithms on top of it is an easier problem.

All programs can be decomposed this way. Word processing programs are just textual data in a special buffer that is updated and modified then sent to a printer. Spreadsheets deal with textual and numeric data in cells that are updated automatically based on a quasi-programming language. Database engines handle generic data stored on the disk and reliably feed back to applications. Even first person shoot-em-up video games are just object data that the user interacts with to get to some fantastical goal. Underneath all are the operating systems, which are really just collections of drivers focusing on the lots and lots of data stored about various hardware devices, drivers and software programs.

In fact fiddling with data is the only thing that software can actually do. The types of fiddles may vary, can be extremely complicated and when attached to something like the Internet the amount of data available to fiddle may be beyond massive, but when it is all said and done, software can only view, modify and create data. All of the magic comes from how we interpret that data.

Because of this, internally it should be data that drives the code. Data is the core of any system, and it should only be represented within the program an absolute minimum number of times (preferably only once). Functions get applied to data, not the other way around. Systems have access the correct data or they do not. You can't implement a set of functionality, if the data for those functions is not first present in the system. The relative importance of both elements needs to be reversed. Data is the primary element in software while functionality is only a secondary element that comes as a side effect of having the data available in a running application.

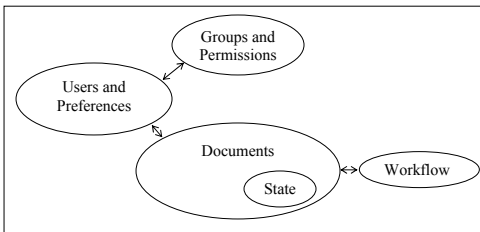
This shift is truly radical, because it asks us to switch away from our first initial perspective of software. It is also radical because it ends up driving all of the other processes that go into the development of a software system. Requirements, for instance

are usually framed as a series of functionality that must be implemented for the users to accept the system. As functions, this is a huge list of items – often containing many similarly related items, like the ability to edit the users, and the ability to edit their preferences, and the ability to edit their groups, etc. Many specifications explicitly permute all of the combinations, even when they are obvious. In the requirements are also many different repeated specifications of how to view the data. There are views for different screens, different navigations and different reports. All together this can be a huge amount of redundant information. Failing to keep all of it in sync often makes the functional specification outdated as a development guide.

## Exploring a Simple Example

From a data directed viewpoint consider a requirement specification that simply lists out the data in the system, with only simple descriptions of the obvious functionality where needed. A good example of this might come from looking at a workflow system, one with users, documents and of course, workflows. Typical requirements for a system like that include allowing the users to upload documents and track them as they move through the different states. Users, their access, the documents, and the current document state dominate the design. Another category of users, the system administrators should have edit capabilities on all administrative data, such as users and permissions. There is a function for data managers to create new workflows, push documents around and to fix any data. Ordinary users should be able to add new documents, view them in the workflows and move them from state to state. There is probably some type of review process, also a type of combined view/edit mode and maybe even an editing or annotating feature. While this isn't even close to a complete specification, this short paragraph of description is equivalent to a good ½ inch of use cases and functional requirements – but it is terse and quickly understood. There are some ambiguities, but if a developer is following standard commercial best practices, they don't amount to any more than would be commonly found in a 2-inch specification. If the screens followed common GUI standards, there might be a few questions as to how to best present some of the access to the data, but adhering to the conventions would define most of the details.

The system above, while common enough in its design patterns contains a relatively small amount of distinct internal types of data:



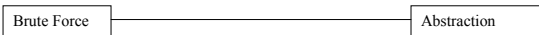
This high level perspective provides an easily abstract view of the core types of data within the system. It is crude, but effective in its simplicity. The break down of the data is important because it encapsulates the various different minor pieces of data into related data elements. Abstractly, whether the user's name is partitioned into first, middle, last sub-fields or just stored as one large full-name field, is an implementation detail. It needs to be decided, but it doesn't alter the nature or usability of the system. It matters if it is

necessary to match organizational or industry standards, and it matters as an element of an interface to be accessible by other sections of the code, but from an abstract point of view it does not affect the architecture.

A design for this system that is driven by its data starts with an understanding similar to the above of the core data and its internal relationships and then expands that into a full architecture and implementation plan for a system. From a process standpoint, the depth of the full specification varies depending on the expected size of the development, the size of the team and the organizational requirements. A very small skunk-works like development team might choose to expand out the specifications only slightly from the above by focusing on key issues like the data model / data structures used to contain the workflows. A larger organization with many separated implementation teams would probably prefer to be more precise in most of the details of the underlying data (and the critical functions) to insure that the development process is more controlled. Failure in communication between teams can be offset with more design details.

What is intriguing is that a specification of a software system based on the data within the system and some rather obvious points about how the system should work is just a fraction of the size of a similar specification based around functionality. This happens because we don't need to reiterate the underlying data used by each particular function. It appears only once. Functionality is either really generic or it is applied directly to one set of data or another. Even when it crosses multiple data sets, the definitions are almost trivial when applied against the data. The most complex functions boil down to a set of simple algorithms that are likely applied directly against one set of data. This greatly reduces the size of the specification and the amount of time to prepare the specification. It also decreases the likelihood of errors because the specification size is considerably smaller, and thus more easily corrected. Similarly, interfaces that strictly follow conventions don't need to be fully permuted if the developers are aware of and will follow the general guidelines set by the rules. Larger specifications always contain more errors.

## Going to the Next Level

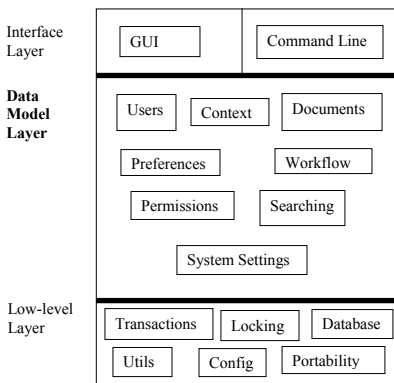


Coding style spans a spectrum between abstraction and brute force. To the far left, brute force coding is the implementation of each and every line of code with the solution. This creates simple code, but massive amounts of it. Moving to right progressively applies more and more abstraction to the code. Abstraction makes the code smaller, more general, but also on a line-by-line basis more complex. Individually, a piece of abstract code implementing something like a binary tree may be more complex than its brute force equivalent, but it needs to be understood that in general there is so much less code between the two alternatives that brute forcing a fixed tree implementation for anything larger than a trivial tree will contain considerably more complexity than the abstract implementation. The much smaller size of abstract code more than offsets the increase in complexity. Additionally, the brute force version will have a large number of fixed limitations that will need to be altered or updated as the initial tree code is redeployed for different set of problems. It will be very fragile. The fully abstract version will keep working perfectly, with no foreseeable limits (bounded by the implementation language).

The virtues of abstraction come from its ability to apply the code to a much larger domain of problems. Given that programmers have more or less a consistent and limited output for their code, it is more efficient to have programmers working on abstract implementations than on brute force ones. This way the output can be leveraged to solve a wider arrangement of problems. Getting more from less code is a considerably more effective. This principle has been shown again and again in the industry particularly in watching the progression of programming languages from assembler to C to Java. Each language offers a more abstract, higher-level working model that allows the programmers to increase their efficiency. This has clearly allowed the systems being build to massively increase in their complexity (although quality often seems unaffected). Abstraction driven by the language increases a programmer's output, so it is not hard to imagine how applying it to the design and coding of the system would also produce a significant increase.

When building systems, I prefer to architect the core with abstractions from the underlying problem domain and the technical domain. This design technique is exceptionally powerful, but the full range of the subject is beyond a simple article. Driving the development from its data lays out a natural abstraction on which to architect the rest of the system code. If a program is only a tool to manipulate the data, then the most important foundation for the code is to have a layer that contains a perfect model of the data for the running application. From an architectural point of view, the system should contain a lower layer for such things as portability, system calls, libraries, etc. Above which is a layer that provides a model for all of the data in the system. Orienting the design on the data simplifies the internal structure. Functionality, if it only applies to explicitly one type of data should be embedded close to that code. Algorithms that span

multiple sets of data can stand on their own, but only in the sense that they do not become explicitly tied to any of the different underlying data. Above the data are the various interfaces, GUI, command line, etc. The glue between the interface entry points and the data implementation should be as minimal as possible. This thinking leads to a very natural partition of the internal code:



This diagram draws very simple but clear architectural lines around the data layer. This becomes significant in development of the code. For example, a user component exists that stands alone from any of the higher level interface (and presentation) issues as well as from the lower level storage or configuration issues. As well, except for what are essentially references or soft links to other data, the user element also stands alone from the other core data elements. With this type of architectural partitioning, changing the user's first name from being a set of three strings to being one single string has a very limited impact on the rest of the system. Adding new functionality to the user model should be less likely to cause bugs in the documents model.

At this point, you might think I am misunderstanding the realities of modern software development. Doesn't it already work this way? There are so many frameworks for building systems and libraries that provide access to the different types of data. However, most of these implementations confuse their functionality with their data. They are grouped by similar functions, not by related data. As such, looking at enough libraries, you see that the underlying data is repeated over and over again in slightly different forms. Using several libraries in an implementation involves copying and converting the

data from one library to the next. By focusing on the data the difference is that the libraries would be based around their data lines. If you need access to a specific type for data, you would look for a library that provided access to that data type. Essentially if the design were normalized only one library for each set of data would be needed. There would be little need to copy and convert between the different parts of the system. Also by this type of refactoring at the base level, the divisions within the data extend all the way up through to code to the higher levels of the development project. So that for instance, the different developer teams are partitioned not by the functionality they will implement, but by the data they will make available to the implementations. There could be one team that works at a very shallow level at the top to assemble together the running applications, but all other teams would be organized by the boundaries of the data they are working with. The architecture dictates the development team structure. It minimized interaction problems with the different teams. This is a very different view of software, even though some of the results may seem similar to what already exists.

A big source of confusion in architecture often comes from the problems of making data persistent over a longer period of time than the program will run. Standard techniques include using a database, particularly a relational one to hold the data between running instances of the program. Toss in some multi-user issues, and an awful lot of technical problems (and code) arise from trying to deal with this non-core problem. The data directed approach makes this easy as well, in that the storing/loading of data in the database is a secondary problem within the system. Architecturally, the code for simplicity sake is driven from the data, so the best most accurate data model is the optimal code. Persistence then must be a secondary problem, and one that should be dealt with after the internal model of the code is decided. An approach to dealing with this for a database type like relational would be to take that internal model, cast it into the more limited relational one, and then generate a fourth normal schema, if possible. That takes advantage of the relational characteristics, but will still probably need a further denormalization to fix any mapping or performance related issues. The idea, though is to make the schema map effectively back to the best model for the software, while trying to put in the most normalized schema possible. Why limit the ad-hoc and reporting capabilities of the relational database if it is not absolutely necessary for strong technical reasons.

The data driven perspective lends itself well to iterative construction techniques as well. As the system extends, the components – based around the underlying data – are split, replaced or added with extended versions of the underlying data that have more functionality attached. Looking at change requests from the users then becomes mapping their needs back to whether or not the data is already in the system. If it is and the model is broad enough, then the functionality can be implemented. Possibly the data is there, but in a limited model. Refactoring and some extension can change the underlying data model. Adding new data, particularly if derived from a consistent set of lower-level functionality is not a major task if the data works the same as an existing data set. A type that is sufficiently different as to require new implementation techniques can be a challenge, but it is easily understood as such and so can be scheduled to fit into the overall plans. Design, extension and even scheduling all become simplified.

## ***The Final Words***

This data driven perspective leads to a very powerful partitioning of the system and the development into distinct data-related elements. From gathering the requirements as sets of data that the user wants to manipulate, to defining an architecture based around a data foundation layer, to distributing the work to various teams based on the underlying data, changing your perspective of software to place data as the primary element alters the entire process of development. This understanding, and by now lots and lots of experience in utilizing it provides the confidence to take on any type of software application development. For me, the success of a development project has become an issue of resource availability not architectural design; the system design is driven from the data, the problem domain, the technical issues and the environment; if the resources available to build it to match the constraints, it will be on time and successful.

Knowing what is really important to insure a working system is the cornerstone of being a good software developer. If I know the data that is in the system, then I can build the system quickly and efficiently. If I talk to the users and understand what they are trying to do, then I can frame that work in terms of the underlying data that they need to accomplish their goals. For more complex systems, I may need to implement some of the functionality based around very sophisticated algorithms, but generally this is rare and there is plenty of research available to get the work started. Understanding a problem domain well enough to be able to build effect tools is a complex problem, but implementing those tools as a well-rounded software package needn't be difficult. If we reorient our view to match the idea that software is nothing more than simple tools to manipulate data, then focusing on the data becomes the way in which we can utilize this knowledge to make our implementations manageable. Nothing but time should stand between us and a fully functioning software system.