Abstract

This paper reviews David L. Parnas’ paper titled “Designing Software for Ease of Extension and Contraction”, discusses some of the key ideas, and relates them to current times.

1 Introduction

In the paper titled “Designing Software for Ease of Extension and Contraction” [1], published at ICSE 3 in Atlanta in 1978, David Parnas discusses, what essentially amounts to: software requirements and how to deal with them.

Basically, we (software developers) are taught to think in terms of programs. We are given a task, and we design a program to solve that task. Often, we do not concern ourselves with the fact that requirements can change, etc.

What David Parnas is suggesting is that we should build software not as an individual ‘system’ (as a single program) but as a family of programs. Each of those ‘programs’¹ would have distinct functionalities, and be precisely specified. These programs are allowed to use other programs to facilitate reuse, etc.

¹We have to remember that this paper was written in 1970s, and it is likely that what Parnas is implying is that software should be designed using modules, or by using object oriented techniques, etc., and not the actual separate executable programs he’s implying in the article.
Parnas’ paper also spends a great deal of time talking about requirements, and how to go about breaking up the functionality into these distinct programs, and how to design programs that would use other programs.

2 Software as a Family of Programs

The key idea of the paper is: when faced with a task of building a complex system, don’t build the whole thing as a single chunk of code. Rather, build it as a collection of many programs.

One can see that even with relatively unstable requirements, if these subset programs are designed to be independent of other unrelated programs, then it is relatively easy to go about changing pieces of software. Either these programs have to be modified, or how they interoperate and what they do collectively to solve a problem.

2.1 Identifying the Subset

The way to go about designing these types of system is to first identify the important subsets of programs that are shared by most likely variations of requirements.

For example, if we’re designing a database to hold contacts data, we may identify the file I/O operations as the subset, since no matter what data we’ll be storing, we’ll still need to save it.

Basically we have to look at requirements, and predict what may be the most likely (possible) changes, and then place more emphasis on designing ‘programs’ (or modules) that are shared by those most likely changes to requirements. That way, even if the requirements change, majority of the work should should still be usable.

---

2 Possibly modules.

3 Programs that handle some subset of a problem.
2.2 Modules

The motivation for this ‘family of programs’ to solve some problem is that each program behaves as an independent unit; a modern day module or component. Data hiding and separation of concerns is the key. If a change in requirements occurs (as it often does), hopefully only a few of these programs will be affected, and the effect won’t propagate throughout the system and will be confined to within a few updated ‘programs’.

That being said, Object Oriented Programming (OOP) was supposed to alleviate these issues, and it hasn’t. Writing software for ease of extension and contraction is still a major effort, and to a large extent, still generally unsuccessful.

3 Chain of Programs

Often, the job of the computer system is to do some processing. By breaking up the monolithic system into a set of individual programs, we introduce the idea of ‘use’. Basically programs have to use each other to perform tasks. Programs must pass data around themselves to accomplish things.

One idea of how to arrange these is to pipe the input from one program to the next, until the last program produces the desired output. This is often used in modern times on UNIX; many UNIX commands allow you to pipe data from one program to the next in a chain.

The paper mentions some issues with this approach: programs that don’t do any processing. The situation may happen like this. At first, you have several programs, each one passing the data from one to the next, each one doing some calculations, etc. Then requirements change, and you realize you don’t need a certain middle step. So you just erase that program. The problem is that the next step expects data to be in the format that’s produced by the now missing step, so you even through you removed the extra processing step, you still need a data transformation step so the following steps in the chain don’t fail.
4 Uses Relation

Parnas’ paper then goes on to describe how to separate the functionality of these programs so that each one is relatively independent from the other. He introduces the idea of a ‘uses’ relation, whereby programs can ‘use’ other programs.

The paper details some rules as to which program should use which other program. Basically if program $A$ has functionality that’s needed by program $B$, then program $B$ should use program $A$. According to the paper, the used program should $\textit{not}$ be more complex than the using program, and the used program should not itself use the program that it is being used by, whereby creating a circular ‘uses’ relation.

In modern times, the focus shifted a bit. Now it is likely that you’d want to use a ‘program’ (or library) that’s more complicated because you don’t want your program to be $\textit{that}$ complicated. In fact, many times, the libraries used are far more complicated than the software that uses them.

There is also the general shift towards creating higher and higher level programming environments, whereby the details of input/output, and other low level functions is hidden away from the real business logic code that solves the problem. Basically as described in my paper on code generation, the key is to make it easier to express semantic portions of problems, by simplifying (or generating—as it applies to my paper) the syntactic portions$^4$.

5 Conclusion

In this conclusion, I’d like to point out the ‘Summary’ portion of the paper, which is surprisingly detailed and explanatory (more so than the paper itself in my opinion). Parnas outlines the key ideas, and then makes a few comments on each one. So let’s start.

$^4$These include repetitive or boilerplate code.
5.1 The Requirements Include Subsets and Extensions

Basically, we need to know what parts of requirements won’t change over time, and what *may* change. We should be able to spot subsets (and utilize them in our design—working on subsets first), and ‘guess’ extensions (basically what can possibly change that would effect our deadlines).

5.2 Virtual Machine Bit

There is a bit about implementing all these ‘programs’ as virtual machines. The definition of the virtual machine is quite different than from what we know of Java, etc., Basically the paper’s view is that these programs would be implemented as commands (or instructions) of this virtual machine. The idea is so unconventional and outdated, I failed to see the benefits of it.

5.3 Generality vs. Flexibility

There is a mention of generality vs. flexibility. Basically generality means that we can use that same program in many circumstances. While flexibility means that we can extend (modify) that program easily to handle many situations.

5.4 Subprograms, Modules, Levels

While I took the idea that the ‘subprograms’ basically mean modern day ‘modules’ this section in the summary turns that view up side down. Basically, sub-programs are sub-programs, and modules are modules.

I think the paper still mostly fits in the concept of modules. Maybe at the time, modules were very much software dependent, so for example having a database management module as a ‘module’ itself would be uncommon. Much of the discussion of data hiding and separation of concerns that is discussed, can, in my view, be accomplished via modules, components,
or, as the paper says, sub-programs.

There is also the concept of ‘levels’. Basically we should also level our ‘sub-programs’ and modules, so there is a low-level (that handles issues close to hardware, etc.) and high-level modules (that handle business layer logic, etc.)

5.5 Avoiding Duplication

There is mention of avoiding duplication. Basically by properly separating out the problem space into programs, duplication can be minimized. There are certain instances where this won’t work out well, like in hierarchical structures, where several branches may need to duplicate some functionality.

5.6 Conclusion of a Conclusion

Basically the paper ends with a summary of these key points. While many of these ideas are still valid, some are quite outdated. Nobody would design a simple ‘address management’ software (as Parnas’ paper does) using several dozen separate programs. These types of programs may have been complex for the 1970s, but they’re not now.

Also, most of the ideas of the paper have either been incorporated into ‘patterns’ to be followed, or have been shown to be ineffective\(^5\).

Overall, it is an interesting paper, describing things that most software developers would find relatively common sense in today’s world.

References


\(^5\)Software is still hard to build, and all the problems mentioned in the paper have yet to be resolved—even with the creation of OOP languages, and virtual machines, etc.