What is Software Design?

by

Jack W. Reeves

Object oriented techniques, and C++ in particular, seem to be taking the software world by storm. Numerous articles and books have appeared describing how to apply the new techniques. In general, the questions of whether O-O techniques are just hype have been replaced by questions of how to get the benefits with the least amount of pain. Object oriented techniques have been around for some time, but this exploding popularity seems a bit unusual. Why the sudden interest? All kinds of explanations have been offered. In truth, there is probably no single reason. Probably, a combination of factors has finally reached critical mass and things are taking off. Nevertheless, it seems that C++ itself is a major factor in this latest phase of the software revolution. Again, there are probably a number of reasons why, but I want to suggest an answer from a slightly different perspective: C++ has become popular because it makes it easier to design software and program at the same time.

If that comment seems a bit unusual, it is deliberate. What I want to do in this article is take a look at the relationship between programming and software design. For almost 10 years I have felt that the software industry collectively misses a subtle point about the difference between developing a software design and what a software design really is. I think there is a profound lesson in the growing popularity of C++ about what we can do to become better software engineers, if only we see it. This lesson is that programming is not about building software; programming is about designing software.

Years ago I was attending a seminar where the question came up of whether software development is an engineering discipline or not. While I do not remember the resulting discussion, I do remember how it catalyzed my own thinking that the software industry has created some false parallels with hardware engineering while missing some perfectly valid parallels. In essence, I concluded that we are not software engineers because we do not realize what a software design really is. I am even more convinced of that today.

The final goal of any engineering activity is the some type of documentation. When a design effort is complete, the design documentation is turned over to the manufacturing team. This is a completely different group with completely different skills from the design team. If the design documents truly represent a complete design, the manufacturing team can proceed to build the product. In fact, they can proceed to build lots of the product, all without any further intervention of the designers. After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings.

There are probably enough arguments both for and against this premise to fill numerous articles. This article assumes that final source code is the real software design and then examines some of the consequences of that assumption. I may not be able to prove that this point of view is correct, but I hope to shown that it does explain some of the observed facts of the software industry, including the popularity of C++.

There is one consequence of considering code as software design that completely overwhelms all others. It is so important and so obvious that it is a total blind spot for most software organizations. This is the fact that software is cheap to build. It does not qualify as inexpensive; it is so cheap it is almost free. If source code is a software design, then actually building software is done by compilers and linkers. We often refer to the process of compiling and linking a complete software system as "doing a build". The capital investment in software construction equipment is low -- all it really takes is a computer, an editor, a compiler, and a linker. Once a build environment is available, then actually doing a software build just takes a little time. Compiling a 50,000 line C++ program may seem to take forever, but how long would it take to build a hardware system that had a design of the same complexity as 50,000 lines of C++.

Another consequence of considering source code as software design is the fact that a software design is relatively easy to create, at least in the mechanical sense. Writing (i.e., designing) a typical software module of 50 to 100 lines of code is usually only a couple of day's effort (getting it fully debugged is another story, but more on that later). It is tempting to ask if there is any other engineering discipline that can produce designs of such complexity as software in such a short time, but first we have to figure out how to measure and compare complexity. Nevertheless, it is obvious that software designs get very large rather quickly.

Given that software designs are relatively easy to turn out, and essentially free to build, an unsurprising revelation is that software designs tend to be incredibly large and complex. This may seem obvious but the magnitude of the problem is often ignored. School projects often end up being several thousand lines of code. There are software products with 10,000 line designs that are given away by their designers. We have long since passed the point where simple software is of much interest. Typical commercial software products have designs that consist of hundreds of thousands of lines. Many software designs run into the millions. Additionally, software designs are almost always constantly evolving. While the current design may only be a few thousand lines of code, many times that may actually have been written over the life of the product.

While there are certainly examples of hardware designs that are arguably as complex as software designs, note two facts about modern hardware. One, complex hardware engineering efforts are not always as free of bugs as software critics would have us believe. Major microprocessors have been shipped with errors in their logic, bridges collapsed, dams broken, airliners fallen out of the sky, and thousands of automobiles and other consumer products have been recalled - all within recent memory and all the result of design errors. Second, complex hardware designs have correspondingly complex and expensive build phases. As a result, the ability to manufacture such systems limits the number of companies that produce truly complex hardware designs. No such limitations exist for software. There are hundreds of software organizations, and thousands of very complex software systems in existence. Both the number and the complexity are growing daily. This means that the software industry is not likely to find solutions to its problems by trying to emulate hardware developers. If anything, as CAD and CAM systems have helped hardware designers to create more and more complex designs, hardware engineering is becoming more and more like software development.

Designing software is an exercise in managing complexity. The complexity exits within the software design itself, within the software organization of the company, and within the industry as a whole. Software design is very similar to systems design. It can span multiple technologies and often involves multiple sub-disciplines. Software specifications tend to be fluid, and change rapidly and often, usually while the design process is still going on. Software development teams also tend to be fluid, likewise often changing in the middle of the design process. In many ways, software bears more resemblance to complex social or organic systems than to hardware. All of this makes software design a difficult and error prone process. None of this is original thinking, but almost 30 years after the software engineering revolution began, software development is still seen as an undisciplined art compared to other engineering professions.

The general consensus is that when real engineers get through with a design, no matter how complex, they are pretty sure it will work. They are also pretty sure it can be built using accepted construction techniques. In order for this to happen, hardware engineers spend a considerable amount of time validating and refining their designs. Consider a bridge design, for example. Before such a design is actually built the engineers do structural analysis; they build computer models and run simulations; they build scale models and test them in wind tunnels or other ways. In short, the designers do everything they could think of to make sure the design is a good design before it is built. The design of new airliner is even worse; for those, full scale prototypes must be built and test flown to validate the design predictions.

It seems obvious to most people that software designs do not go through the same rigorous engineering as hardware designs. However, if we consider source code as design, we see that software designers actually do a considerable amount of validating and refining their designs. Software designers do not call it engineering, however, we call it testing and debugging. Most people do not consider testing and debugging as real "engineering"; certainly not in the software business. The reason has more to do with the refusal of the software industry to accept code as design than with any real engineering difference. Mock-ups, prototypes, and bread-boards are actually an accepted part of other engineering disciplines. Software designers do not have or use more formal methods of validating their designs because of the simple economics of the software build cycle.

Revelation number two: it is cheaper and simpler to just build the design and test it than to do anything else. We do not care how many builds we do -- they cost next to nothing in terms of time and the resources used can be completely reclaimed later if we discard the build. Note that testing is not just concerned with getting the current design correct, it is part of the process of refining the design. Hardware engineers of complex systems often build models (or at least they visually render their designs using computer graphics). This allows them to get a "feel" for the design that is not possible by just reviewing the design itself. Building such a model is both impossible and unnecessary with a software design. We just build the product itself. Even if formal software proofs were as automatic as a compiler, we would still do build/test cycles. Ergo, formal proofs have never been of much practical interest to the software industry.

This is the reality of the software development process today. Ever more complex software designs are being created by an ever increasing number of people and organizations. These designs will be coded in some programming language and then validated and refined via the build/test cycle. The process is error prone and not particularly rigorous to begin with. The fact that a great many software developers do not want to believe that this is the way it works compounds the problem enormously.

Most current software development processes try to segregate the different phases of software design into separate pigeon-holes. The top level design must be completed and frozen before any code is written. Testing and debugging are necessary just to weed out the construction mistakes. In between are the programmers, the construction workers of the software industry. Many believe that if we could just get programmers to quit "hacking" and "build" the designs as given to them (and in the process, make fewer errors) then software development might mature into a true engineering discipline. Not likely to happen as long as the process ignores the engineering and economic realities.

For example, no other modern industry would tolerate a rework rate of over 100% in its manufacturing process. A construction worker who can not build it right the first time, most of the time, is soon out of a job. In software, even the smallest piece of code is likely to be revised or completely rewritten during testing and debugging. We accept this sort of refinement during a creative process like design, not as part of a manufacturing process. No one expects an engineer to create a perfect design the first time. Even if she does, it must still be put through the refinement process just to prove that it was perfect.

If we learn nothing else from Japanese management techniques, we should learn that it is counter-productive to blame the workers for errors in the process. Instead of continuing to force software development to conform to an incorrect process model, we need to revise the process so that it helps rather than hinders efforts to produce better software. This is the litmus test of "software engineering." Engineering is about how you do the process, not about whether the final design document needs a CAD system to produce it.

The overwhelming problem with software development is that everything is part of the design process. Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design. Software may be cheap to build, but it is incredibly expensive to design. Software is so complex that there are plenty of different design aspects and their resulting design views. The problem is that all the different aspects interrelate (just like they do in hardware engineering). It would be nice if top level designers could ignore the details of module algorithm design. Likewise, it would be nice if programmers did not have to worry about top level design issues when designing the internal algorithms of a module. Unfortunately, the aspects of one design layer intrude into the others. The choice of algorithms for a given module can be as important to the overall success of the software system as any of the higher level design aspects. There is no hierarchy of importance among the different aspects of a software design. An incorrect design at the lowest module level can be as fatal as a mistake at the highest level. A software design must be complete and correct in all its aspects, or all software builds based on the design will be erroneous.

In order to deal with the complexity, software is designed in layers. When a programmer is worrying about the detailed design of one module, there are probably hundreds of other modules and thousands of other details that he can not possibly worry about at the same time. For example, there are important aspects of software design that do not fall cleanly into the categories of data structures and algorithms. Ideally, programmers should not have to worry about these other aspects of a design when designing code.

This is not how it works, however, and the reasons start to make sense. The software design is not complete until it has been coded and tested. Testing is a fundamental part of the design validation and refinement process. The high level structural design is not a complete software design; it is just a structural framework for the detailed design. We have very limited capabilities for rigorously validating a high level design. The detailed design will ultimately influence (or should be allowed to influence) the high level design at least as much as other factors. Refining all the aspects of a design is a process that should be happening throughout the design cycle. If any aspect of the design is frozen out of the refinement process, it is hardly surprising that the final design will be poor or even unworkable.

It would be nice if high level software design could be a more rigorous engineering process, but the real world of software systems is not rigorous. Software is too complex and it depends on too many other things. Maybe some hardware does not work quite the way the designers thought it did, or a library routine has an undocumented restriction. These are the kinds of problems that every software project encounters sooner or later. These are the kinds of problems discovered during testing (if we do a good job of testing), for the simple reason that there was no way to discover them earlier. When they are discovered, they force a change in the design. If we are lucky, the design changes are local. More often than not, the changes will ripple through some significant portion of the entire software design (Murphy's Law). When part of the effected design can not change for some reason, then the other parts of the design will have to be weakened to accommodate. This often results is what managers perceive as "hacking", but it is the reality of software development.

For example, I recently worked on a project where a timing dependency was discovered between the internals of module A and another module B. Unfortunately, the internals of module A were hidden behind an abstraction that did not permit any way to incorporate the invocation of module B in its proper sequence. Naturally, by the time the problem was discovered, it was much too late to try to change the abstraction of A. As expected, what happened was an increasingly complex set of "fixes" applied to the internal design of A. Before we finished installing version 1, there was the general feeling that the design was breaking down. Every new fix was likely to break some older fix. This is a normal software development project. Eventually, my colleagues and I argued for a change in the design, but we had to volunteer free overtime in order to get management to agree.

On any software project of typical size, problems like these are guaranteed to come up. Despite all attempts to prevent it, important details will be overlooked. This is the difference between craft and engineering. Experience can lead us in the right direction. This is craft. Experience will only take us so far into uncharted territory. Then we must take what we started with and make it better through a controlled process of refinement. This is engineering.

As just a small point, all programmers know that writing the software design documents after the code instead of before, produces much more accurate documents. The reason is now obvious. Only the final design, as reflected in code, is the only one refined during the build/test cycle. The probability of the initial design being unchanged during this cycle is inversely related to the number of modules and number of programmers on a project. It rapidly becomes indistinguishable from zero.

In software engineering, we desperately need good design at all levels. In particular, we need good top level design. The better the early design, the easier detailed design will be. Designers should use anything that helps. Structure charts, Booch diagrams, state tables, PDL, etc. -- if it helps, then use it. We must keep in mind, however, that these tools and notations are not a software design. Eventually, we have to create the real software design, and it will be in some programming language. Therefore, we should not be afraid to code our designs as we derive them. We simply must be willing to refine them as necessary.

There is as yet no design notation equally suited for use in both top level design and detailed design. Ultimately, the design will end up coded in some programming language. This means that top level design notations have to be translated into the target programming language before detailed design can begin. This translation step takes time and introduces errors. Rather than translate from a notation that may not map cleanly into the programming language of choice, programmers often go back to the requirements and redo the top level design, coding it as they go. This, too, is part of the reality of software development.

It is probably better to let the original designers write the original code, rather than have someone else translate a language independent design later. What we need is a unified design notation suitable for all levels of design. In other words, we need a programming language that is also suitable for capturing high level design concepts. This is where C++ comes in. C++ is a programming language suitable for real world projects that is also a more expressive software design language. C++ allows us to directly express high level information about design components. This makes it easier to produce the design, and easier to refine it later. With its stronger type checking, it also helps the process of detecting design errors. This results in a more robust design, in essence a better engineered design.

Ultimately, a software design must be represented in some programming language, and then validated and refined via a build/test cycle. Any pretense otherwise is just silliness. Consider what software development tools and techniques have gained popularity. Structured programming was considered a breakthrough in its time. Pascal popularized it and in turn became popular. Object oriented design is the new rage and C++ is at the heart of it. Now think about what has not worked. CASE tools? Popular, yes; universal, no. Structure charts? Same thing. Likewise, Warner-Orr diagrams, Booch diagrams, object diagrams, you name it. Each has its strengths, and a single fundamental weakness -- it really isn't a software design. In fact the only software design notation that can be called widespread is PDL, and what does that look like.

This says that the collective subconscious of the software industry instinctively knows that improvements in programming techniques and real world programming languages in particular are overwhelmingly more important than anything else in the software business. It also says that programmers are interested in design. When more expressive programming languages become available, software developers will adopt them.

Also consider how the process of software development is changing. Once upon a time we had the waterfall process. Now we talk of spiral development and rapid prototyping. While such techniques are often justified with terms like "risk abatement" and "shortened product delivery times", they are really just excuses to start coding earlier in the life cycle. This is good. This allows the build/test cycle to start validating and refining the design earlier. It also means that it is more likely that the software designers that developed the top level design are still around to do the detailed design.

As noted above -- engineering is more about how you do the process than it is about what the final product looks like. We in the software business are close to being engineers, but we need a couple of perceptual changes. Programming and the build/test cycle are central to the process of engineering software. We need to manage them as such. The economics of the build/test cycle, plus the fact that a software system can represent practically anything, makes it very unlikely that we will find any general purpose methods for validating a software design. We can improve this process, but we can not escape it.

One final point: the goal of any engineering design project is the production of some documentation. Obviously, the actual design documents are the most important, but they are not the only ones that must be produced. Someone is eventually expected to use the software. It is also likely that the system will have to be modified and enhanced at a later time. This means that auxiliary documentation is as important for a software project as it is for a hardware project. Ignoring for now users manuals, installation guides, and other documents not directly associated with the design process, there are still two important needs that must be solved with auxiliary design documents.

The first use of auxiliary documentation is to capture important information from the problem space that did not make it directly into the design. Software design involves inventing software concepts to model concepts in a problem space. This process requires developing an understanding of the problem space concepts. Usually this understanding will include information that does not directly end up being modeled in the software space, but which nevertheless helped the designer determine what the essential concepts were, and how best to model them. This information should be captured somewhere in case the model needs to be changed at a later time.

The second important need for auxiliary documentation is to document those aspects of the design that are difficult to extract directly from the design itself. These can include both high level and low level aspects. Many of these aspects are best depicted graphically. This makes them hard to include as comments in the source code. This is not an argument for a graphical software design notation instead of a programming language. This is no different from the need for textual descriptions to accompany the graphical design documents of hardware disciplines. Never forget that the source code determines what the actual design really is, not the auxiliary documentation. Ideally, software tools would be available that post processed a source code design and generated the auxiliary documentation. That may be too much to expect. The next best thing might be some tools that let programmers (or technical writers) extract specific information from the source code that can then be documented in some other way. Undoubtedly, keeping such documentation up to date manually is difficult. This is another argument for the need for more expressive programming languages. It is also an argument for keeping such auxiliary documentation to a minimum and keeping it as informal as possible until as late in the project as possible. Again, we could use some better tools, otherwise we end up falling back on pencil, paper, and chalk boards.

To summarize:

o Real software runs on computers. It is a sequence of ones and zeros that is stored on some magnetic media. It is not a program listing in C++ (or any other programming language).

o A program listing is a document that represents a software design. Compilers and linkers actually build software designs.

o Real software is incredibly cheap to build, and getting cheaper all the time as computers get faster.

o Real software is incredibly expensive to design. This is true because software is incredibly complex and because practically all the steps of a software project are part of the design process.

o Programming is a design activity -- a good software design process recognizes this and does not hesitate to code when coding makes sense.

o Coding actually makes sense more often than believed. Often the process of rendering the design in code will reveal oversights and the need for additional design effort. The earlier this occurs, the better the design will be.

o Since software is so cheap to build, formal engineering validation methods are not of much use in real world software development. It is easier and cheaper to just build the design and test it than to try to prove it.

o Testing and debugging are design activities -- they are the software equivalent of the design validation and refinement processes of other engineering disciplines. A good software design process recognizes this and does not try to short change the steps.

o There are other design activities -- call them top level design, module design, structural design, architectural design, or whatever. A good software design process recognizes this and deliberately includes the steps.

o All design activities interact. A good software design process recognizes this and allows the design to change, sometimes radically, as various design steps reveal the need.

o Many different software design notations are potentially useful -- as auxiliary documentation and as tools to help facilitate the design process. They are not a software design.

o Software development is still more a craft than an engineering discipline. This is primarily because of a lack of rigor in the critical processes of validating and improving a design.

o Ultimately, real advances in software development depend upon advances in programming techniques, which in turn mean advances in programming languages. C++ is such an advance. It has exploded in popularity because it is a mainstream programming language that directly supports better software design.

o C++ is a step in the right direction, but still more advances are needed.