Chapter 2
Engineering, Components, and Software

As a verb, “engineer” means to plan, manage, or construct. That’s what an engineer does. A craftsperson does the same but in a quite different way. “Craft” carries the meaning of individual creative effort, of “hand” work, of aesthetic goals as well as utilitarian ones, of less planning and more “just doing.” The engineer follows rules (learned at an accredited engineering school, of course) while a craftsperson is more likely to have served an apprenticeship, or to make it up as s/he goes along. Both have another discipline to which they may aspire though they may not attain its standard: for the craftsperson it is ‘art,’ while for the engineer it is ‘science.’

Beginning in the late 1960s, ‘software engineering’ has tried to apply engineering methods to the craft of developing software. There is an ongoing dispute about whether this is always a good idea, and much more dispute about exactly how it might be done. But even ‘software cowboys’ who despise engineering rules and management\(^1\) can enthusiastically adopt ideas that work. ‘Component-based design’ is one such idea.

2.1 Standardized Components Make Engineering Possible

When something is to be made from individual parts, there is a clear distinction between craft and engineering methods. A craftsperson makes parts in the same way as the whole object is made: each part is fitted to the others until they work

\(^1\) Much of the dispute about software engineering involves engineering methods applied to software management. The dispute is almost a replay of a similar controversy that arose at the beginning of the 20th century, when Fredrick W. (“Speedy”) Taylor introduced what he called “Scientific Management” (and wrote a book with that title). Scientific management involves finding the best (i.e., cheapest) way to do any task, and then forcing workers who do it to do it that way. Today “time and motion study” or “industrial engineering” are more common terms, but Speedy’s critics called it “Taylorism,” and said it was just another way to exploit workers. See Donald Roy, “Quota restriction and goldbricking in a machine shop,” *American Journal of Sociology*, vol. 67, 1952, no. 2, pp. 427-42.
together. If the same person makes a second similar object, it won’t be exactly the
same, and in particular its parts will differ—parts likely won’t be interchangeable
between the two. That in itself isn’t a happy situation: there could be a pile of broken
objects, each with one bad part, yet not a single working object could be salvaged
from the pile because parts can’t be substituted for each other.

The engineering approach is to design parts to specification, and include quality
parameters—usually called ‘tolerances’ for mechanical objects—that define when a
part is ‘as specified.’ Such parts can be used interchangeably, and there is the large
advantage that most of the assembly will require far less skill than for a craft object.
(Some parts may be harder to make because the necessary tolerances are difficult to
achieve.) The final step is standardization, where a limited range of parts is made in
quantity for use in designing a vast variety of objects, most not even imagined when
the parts were designed and fabricated. Mechanical fasteners are a good example:
cap screws come in only a few head designs (hex, socket, etc.), only a few diameters
(1/16th-inch gradations in the English system which now is largely confined to the
U.S.), and only a few lengths (quarter- or half-inch increments between roughly one
inch and six inches). If you want a weird cap screw to fill some special need, you
can have one made with some difficulty. But if instead you can live with one of the
standard sizes, there will be one immediately available, it will be cheap, and you
can count on it fitting.

Standardized components make engineering work because a design problem can
be broken down and many of its pieces then require no design—they are ‘off the
shelf.’ Furthermore, the limited variety of standard parts is actually an advantage
because selecting a part helps to determine what goes around it. A 1/4”-diameter
cap screw fits a 1/4” nut, it fits through a 1/4” hole (the bolt is really a tad smaller
than 1/4” in diameter), the hole can’t be too close to the edge of the material, and so
on. In today’s computer-supported world, it is no small advantage to be designing
with standardized parts. So-called computer-aided design (CAD) programs are a
big help in design, and when the designer clicks on that 1/4” cap screw the CAD
program can draw it and check if the hole is too near the edge...

Standardization plays a subtle role in designing physical systems. To illustrate,
think about two pieces of metal that must be fastened together as part of a design.
The metal parts come first, since presumably each has a purpose (to cover some-
thing, to support something, etc.). Their joint can’t be designed until they themselves
have been selected or fabricated, because its strength and location depends on their
shape, thickness, etc. Suppose the designer decides to use a cap screw, and from a
fasteners’ catalog it appears that 1/4” diameter is strong enough. Furthermore, the
screw needs to be at least 7/8” long to pass through both pieces of metal (and leave
room for the nut and a couple of washers). Both of these dimensions are properties
derived from the system design; nothing was known about them until the two metal

2 Limiting design choices is sometimes thought of as a bad thing, but every designer knows that
too much freedom wastes precious creative energy on inconsequential decisions. The hardest part
of beginning a design is reducing too many degrees of freedom so that the mental wheels can
stop spinning and start designing. The more choices that have been made for a designer, the easier
(because the more constrained) the remaining choices are to make.
parts had been designed and the choice made to bolt them together. Looking at the catalog again, the designer finds that there is no 1/4 x 7/8 cap screw (except on ‘special order’ requiring three weeks lead time), but there is 1/4 x 1, which will do. (The extra 1/8” sticking out on the nut end doesn’t interfere with anything. Perhaps it proves necessary to insert the screw from the other side for clearance.) Standardization plays a crucial role in that it allows the cap screw to be ‘off the shelf’ even though its description was not known when the shelf was stocked.

Craftspeople routinely make use of standardized components, too. A wood turner can buy ballpoint pen units or pepper-grinder mechanisms and then spend his/her creative energy on turning an elegant outer shell. The size of the hole to drill\(^3\) for the mechanism is part of its published description.

In the next two sections, superficial descriptions are given of two component-based hardware designs, with an eye to seeing what makes them successful, and how software might imitate them.

### 2.2 Mechanical Engineering of a Vacuum System

Physical scientists often use laboratory apparatus that must operate in a vacuum. A qualified lab technician is adept at designing one-of-a-kind vacuum systems with which to conduct experiments. For more than 40 years there has been a vacuum-system components industry manufacturing flanges, pipes, feed-throughs, and other parts that can be bolted together to form a custom vacuum system. These parts are made from carefully selected stainless steels welded equally carefully. Their specifications are published in catalogs (today on the Internet—search terms “vacuum flange”), and the design of each system is a textbook example of component-based design.

The lab tech starts with a set of requirements from a scientist who has an experiment to perform. The requirements are not entirely precise, but they indicate a rough shape for the system, what has to be connected to it internally and externally, and the necessary upper bound on the pressure. It is common for these requirements to be a ‘back of the envelope sketch’ such as Fig. 2.1. A requirements sketch is the beginning of the ‘top-down’ part of the system design.

The lab tech starts the ‘bottom-up’ part of the design by consulting a catalog of vacuum components. Paging through this catalog he/she notes what sizes and qualities of flanges, seals, etc., are available, and matches these with the requirements. Figure 2.2 is an example of the kind of data available for a flange. (Dimensions ‘A’ and ‘B’ in the figure, and details of the bolt holes in the flange appear in separate figures\(^4\) not shown here.) It will likely happen that available components will dictate an altered design, which will almost certainly be a wise choice compared to requesting special fabrication of custom parts. The scientist will be consulted about

\(^{3}\) With a standard-size drill bit, of course.

\(^{4}\) See [http://us.trinos.com](http://us.trinos.com), where prices are also shown.
acceptable changes; this process in turn may result in different component choices. Very quickly the top-down and bottom-up design paths will meet in a likely design.

But before the lab tech writes a purchase order, he will do some crucial calculations. Using the catalog descriptions, it will be possible to calculate important system properties. For example, the total volume is obtained as the sum of volumes of the parts, the vacuum quality will be determined by seal leak rate, the vacuum pump type and speed will depend on seals and volume, etc. These calculations show the technician whether the design will quantitatively meet the requirements, and they may be checked by the scientist. For example, it’s a bad design if there is no available pump that can achieve the required vacuum. If that’s the case, the volume or the number of seals may have to be reduced. What the tech does not do is order
the parts and assemble the system to see if it fits together and works as required\(^5\), then if it does not, start making changes. It is fully expected by everyone concerned that when the calculations check out, the required system can be routinely bolted together and it will work.

2.3 Electrical/Computer Engineering of a Laptop

A modern laptop computer is a marvel of component-based hardware design. It is literally possible to buy a CPU, memory, hard drive, CD/DVD drive, keyboard, touchpad mouse, battery, and screen, and plug them together to make most of a working computer\(^6\). Each component is itself complex, but their interfaces are simple, which is what allows the parts to be selected independently and to work together. In many cases there are several choices for interchangeable components, so a laptop builder can trade speed or quality for price.

Fixing a broken laptop may be as easy as exchanging a broken component for a replacement; in some cases the replacement slides into an existing slot without tools. How hard it is to replace a component mostly depends on packaging. For example, hard drives in a standard shape plug in; screens\(^7\) and keyboards are custom fitted to the case. A traditional way to describe component-based physical systems is an ‘exploded view,’ a projection drawing showing the whole separated into its parts. Figure 2.3 shows a typical laptop\(^8\). In Figure 2.3 about half of the 19 components shown can be obtained off the shelf, and these account for perhaps 3/4 of the laptop cost.

There is a peculiarity of the computer market that gives an insight into ‘specifications’ for laptop components. In practice, virtually all laptops are sold bundled with some version of the Microsoft Windows operating system. There is a measure of ‘fit and try’ in getting a new version of Windows to work on existing laptops, or in checking that Windows works on a new laptop. Engineers at a laptop manufacturer do the work so that when a customer buys a machine with Windows installed, it works out of the box. This effort would not be required if the published specifications for the hardware were accurate and Windows followed them. One can imagine that data on the magnitude of the adjustment required would be difficult to obtain from either side of the operation. However, there is an independent way to get data. Some laptop purchasers discard Windows and install Linux on their machines. Here there has been no fit and try—Linux, like an ideal ‘component’ developed in isolation, can use only published hardware specifications\(^9\). If Linux works ‘off

---

\(^5\) To carry this a bit farther, it would be even worse to hand a trial assembly over to the scientist untested.

\(^6\) The case that holds them and some other parts are custom designed.

\(^7\) A website advertising replacement screens lists about 5000 15.4” models. Many of them are probably interchangeable, but no specifications are published on the site.

\(^8\) Taken from the Dell information site http://support.dell.com for the Inspiron 5100.

\(^9\) And damn hard to get them from the laptop makers it is. Why is that?
the shelf, it means that the hardware specifications are solid; it may fail because they are inadequate. Judged by the traffic on ‘help’ forums and Q-A websites, these specifications have improved dramatically in the last ten years. Where it was once
necessary to pick from among a few laptop models that worked with Linux, today most models work\textsuperscript{10}.

### 2.4 Can It Be Done with Software?

“Software is different.” But is it so different that component-based designs cannot work as they do in mechanical and electrical engineering?

From the time when the first program was written for a digital computer, there has been a ‘software problem.’ This problem is created by expectations that software can be routinely made to do what its creators intend\textsuperscript{11} it to do. Many people have pointed out that the ‘problem’ is caused largely by an interplay between human aspirations and human frailty, compounded by dubious analogies to engineering. It would be as well to speak of an ‘automobile problem’ because some cars are lemons. But in the case of mechanical failures, humans share the blame with ‘Mother Nature,’ those forces that bring down the house (literally, in the case of structural engineering). Where Mother Nature is involved, engineering is well understood to be a discipline of trade-offs, in which there are no perfect solutions. A building can be designed to stand against the worst storm or the most severe earthquake in recorded history, but it becomes very expensive to build (and it is likely to be ugly). Besides, there may yet be a worse storm, or a storm coinciding with an earthquake—who knows what Mother Nature will come up with? So the structural engineer balances strength against cost and knows that there is a point beyond which any design will fail. If a building designed to withstand 70-mph wind gusts falls in a freak storm, it’s the fault of the weather and the City Fathers who limited what they would pay, not the engineer.

But for software, we humans have only ourselves to blame. Software designs do not face unpredictable weather, the software really does not cost (much) more when it covers all possibilities, and as for ugly—well, code is invisible. What seems to bring down programs is just mistakes that people make. So the ‘software problem’ is that our creations fail, we trace the causes, try to do better, yet they fail again. It isn’t flattering to explain the situation as an intrinsic limitation on human intellectual limitation. It may be true that people simply can’t accomplish some tasks that they can perfectly well set themselves, but no one likes to admit it. Instead,

\textsuperscript{10}PC models have standardized even more quickly; but here’s an anecdote: A Dell PC purchased in 2003 with a optional large display worked with Windows, but when Linux was installed the full display resolution could not be selected. When Windows was reinstalled, the high resolution was not restored. After a great deal of Internet searching, the explanation appeared: There was a mistake in the BIOS code that returned an incorrect parameter for the display. The Windows loaded in the machine initially by ‘fit and try’ fixed this by ignoring the BIOS. But Linux (and the reloaded Windows) used the erroneous BIOS value according to specification. There was probably a single Dell engineer who knew this story, which was later laboriously reconstructed by dedicated (obsessive?) Linux enthusiasts.

\textsuperscript{11}Although it may be apocryphal, the story is told that von Neumann’s first subroutine failed, and the code was examined only after checking the hardware—how could the program be wrong?
we try to solve the ‘software problem,’ and what better way than to look to older engineering disciplines12 for what has worked for them? Over the years, there have been calls for adopting this or that idea that has worked elsewhere, calls that sound good but often amount to nothing more than empty terminology. We need ‘software blueprints’ (because mechanical engineers use blueprints to get things right), or ‘software ICs’ (because electrical engineers package designs in integrated circuits), or ‘software architects’ because in construction engineering there is an important design step that comes before building starts, etc. Seeking common factors in these fads, many of them are ways of saying that for software, the ‘problem’ is in software requirements/specifications. If only we could say better what it is we mean to do, then perhaps we could get it right.

There are far too many inexcusably vague or shoddy specifications for software. But if the real difficulty is human intellectual limitations, precision in specification may not help much. More precision makes requirements harder to understand, and hence harder to implement correctly. What has been gained if software fails because a precise specification was not understood? Wouldn’t it be just as well (and of course a lot less work!) to fail to understand the real-world requirements themselves? It seems that the only real answer lies in limiting the scope of activity: precise requirements for less-ambitious software could be understood, implemented, and the software not fail. In other engineering disciplines limitations are accepted but ascribed to Mother Nature, so it isn’t necessary to admit that human rational ability, too, has its limitations13.

Apart from contested ideas about engineering management and calls for transplanting ideas for their sexy-sounding names, there are parts of conventional engineering that software developers might adopt, notably ‘components.’ There is no doubt that mechanical engineers, for example, benefit immensely from the existence of standardized parts, not least because they interact so well with CAD tools. The issue is not whether similar components would be helpful in software design—of course they would be. But is it possible? Are there software units that act as the mechanical components do? Or are ‘software components’ just another flawed analogy?

The issue of software complexity and human lack of intellectual control cuts both ways for components. On the positive side, breaking up a large system should reduce the difficulty of developing each of its parts. (The issue of putting the parts together will be taken up at the end of this section.) But on the negative size, the description of even a rudimentary software unit is more complex than the descriptions of most mechanical and electrical parts14.

12 In the 1970s, Harlan Mills and others philosophically said that software’s problems were just those of a young discipline. Give us 50 years and we would do better, as the other engineers did in that much time. We have done better, but the time is nearly up...
13 After all, whose fault is it that people aren’t clever enough to construct anything they can imagine? Mother Nature’s, that’s Who’s.
14 Many people have noticed that as computer CPU chips become more complex, they approach the intricacy of software. Then despite a long tradition of careful engineering design and good
The ability to grasp easily and accurately what a component does is one part of what makes it useful in system design. The other part is that a useful component must live up to its description. It’s no good choosing the right part if it turns out its ‘rightness’ was a lie. When it comes to certifying components—checking to see that they really are as they are described—physical objects have a huge advantage over software. To give a simple example, the strength of a threaded fastener like a cap screw is described by the torque it is safe to use in tightening it. (Too much torque and the head will shear off or the threads strip.) And to measure this parameter takes just one test: the maximum torque is applied, and if the cap screw doesn’t shear, it passes the test. Mother Nature very seldom (but not never!) allows her things to break at a small stress if they survive a larger. Mechanical engineers also cheat on Mother Nature by including safety factors in their component descriptions. If the cap screw is expected to fail at (say) 45 kg-m torque, its description might list the maximum torque as 15 kg-m, a safety factor of 3. Safety factors must cover variations in materials and fabrication and (not so obviously) mistakes in theory—the detailed explanation of how and why cap screws fail could be wrong. If despite the inclusion of a safety factor, some cap screw fails, the safety factor can be increased. However, if the reason for unexpected failure is theoretical, there may be no large enough safety factor. A minimum level of understanding is required before trial and error can succeed.

Software descriptions are not usually limited to just a few parameters, but suppose for the moment that one is this simple. A reasonable analogy to safe tightening torque for a cap screw might be a bound on response time for a program. But to establish such a bound requires knowing which input excites the longest run time. And to find that input, if it can be found at all, requires extensive trials. Even if a large number of tests execute within the bound, there is no guarantee that for some untried input the run time will not be completely out of line. A program is inherently discontinuous; what it does on one input says nothing about another input which might be different only in the least-significant bit. Another way to say this is that there can be no safety factor for programs. This subject will be extensively explored in Chapter 17.

Complexity and certification interact in an important way that makes software unlike most physical objects. Where structural engineers (say) strive to simplify and standardize component parts like I-beams so that they can be reliably described by a handful of parameters and those parameters controlled in the manufacturing process, the essence of software parts is that they are not limited in any functional way. Software can do anything, which is usually thought to be a huge positive advantage over physical artifacts. But when it comes to certification, the advantage becomes a liability: the ‘anything’ that gets done can differ arbitrarily from what is required, which may be practically impossible to discover by testing. By exerting...
ing rigid control in design and at every manufacturing step, a mechanical part is constrained to behave simply and continuity allows this to be checked; no matter how simple a software part is supposed to be, its inherent discontinuity makes it untrustworthy. In general, the response to this intrinsic software property has been to legislate against the human developers who sometimes get it wrong: “You shall do it thus (perfectly).” Perhaps it would be better to begin using software’s own power against its weakness: to exploit the ‘do anything’ to check that mistakes have not been made, or to correct them. If there is an aspect of software that has been shamefully neglected, it is self-limitation. For example, it has been known since the 1960s how to surely prevent memory-destruction problems like buffer overruns, yet the vast majority of working (so to speak!) software relies on perfect human effort to prevent them, instead of the foolproof checks that slightly slow program execution.

Much has been made of one unique software advantage: the ability to depend on all copies of a program to be exact duplicates. Manufactured physical objects, even when held to high standards and tested by good sampling methods, are not so trustworthy—Mother Nature can always throw a ringer in the batch. But continuity is worth much more than consistency when one cannot test for conformance—perfect software copies mean no more than that they are all bad. It is a common consumer-protection guarantee to promise a replacement if a product fails; but, if you got a replacement for your copy of Windows that crashes, it will be exactly the same broken Windows16.

There is nothing practical to be done about software’s complex descriptions and hard-to-test behavior, except to strive for simplicity, to be as careful as possible to understand the problem to be solved and the code that tries to solve it, and to use foolproof internal checks to detect or correct mistakes. A good argument can be made for using formal mathematical specifications, which certainly goes along with components of limited functionality—people do not seem able to use formal methods routinely except in restricted cases. The ‘component’ idea fits well with simplification and careful checking for another reason: it will be worth devoting extensive resources to requirements and certification of a unit that can be employed over and over in many developed systems. If a way can be found to have software check/correct its own actions, it is likely to be expensive; so it will only be worth incorporating in components that can be reused.

Unfortunately, complexity can be the death of standardization, which depends on being able to catalog a reasonable number of items, one that a person can skim through. Perhaps ‘standardization’ of many software components is outside the range of possibility. Even for mechanical systems there is an important role for the customized component, i.e., one not intended to be used outside of one system. Modern automobiles are full of such things, each designed for just one car, or even just one model of one car17. In structural engineering, each large building has a

16 Unless, of course, it is ‘new, improved’ Windows that is even more complicated and less well tested, i.e., probably worse. And where is the retailer offering any software guarantee?
17 Pity the car owner with a broken side mirror, who finds that its replacement costs the earth, and to get a used one requires finding a junkyard with exactly the same car. Furthermore, the parts of which the mirror is made are unique—the tiny plastic molding that is actually broken occurs in
unique design that makes crucial use of standardized components like I-beams, yet customized components may be equally important. For example, the joints between structural members can be separately designed and fabricated throughout the building, but it might be cheaper and better to design joints as examples of one custom component [3].

The ‘do anything’ nature of software certainly encourages custom components.

In summary, although there are important differences between mechanical/electrical components and software units, it should be possible to take the unique strengths and weaknesses of the latter into account and exploit component-based design. However, there remains a single significant issue that distinguishes a ‘software component’ from a mechanical/electrical one. The older engineering disciplines are able to make quantitative predictions of how their components will behave when combined into system assemblies. In the vacuum-system example, the leak rates of constituent parts can be combined (in ways that depend on the system configuration) to predict the system leakage. In the example of bolting two metal parts together, the forces that will bear on the joint can be calculated and the length is the sum of thicknesses of metal, nut, and washers. The reliability of the whole is determined by the reliability of the parts, and so on. Furthermore, these calculations carry safety factors along. It may happen that safety factors for the parts are not enough for the combination18, so an aggregate gets an additional safety factor. This ability to make safe system predictions is what makes CAD tools invaluable, and what allows a system designer to work ‘at component granularity;’ that is, to ignore details of the parts and concentrate on the combinations. Nothing remotely like this has been done for software.

It is not at all certain that properties of software systems can be calculated from properties measured for the parts, because of the problematic nature of descriptions and certifications. The best that one might hope for is that an approximate system prediction could be obtained from imperfect component descriptions, and a quantitative bound placed on the accuracy of the one (system) in terms of the accuracy of the others (components). No matter how loose such a bound might be, it would then be possible to devote sufficient effort to improving the component certifications, with the knowledge that system predictions will get better. The proof that some software-component definition is really similar to the mechanical/electrical case would be the existence of CAD tools to make system-design predictions, including the prediction of safety factors.

This monograph follows exactly that plan. It proposes an approximation to component behavior to be measured by unit testing. It displays CAD tools that use component measurements to make (approximate) system predictions, and it relates the errors that exist in approximations at both levels. The intention is not to propose that the (very restricted) components defined are adequate for practical use nor to tout the CAD tools. Rather, the book is an ‘existence proof’ for the idea that software

---

18 If so, the fault necessarily lies with inaccuracy in the theory of composition, since materials have been accounted for in the components.
components can be defined so that they act as useful components should. And it is the nature of existence proofs to illuminate the larger subject through a single concrete case. However restricted and ‘unreal’ the components and tools presented here may be, they are a fully worked out example from which insights and lessons may be drawn, insights that apply to any component scheme worthy of the name.
Composing Software Components
A Software-testing Perspective
Hamlet, D.
2010, XVII, 368 p., Hardcover
ISBN: 978-1-4419-7147-0