Category Overview

8/27/07

Design

Reliable and dependable form and function

In 1968, the NATO workshop on software systems declared the entire software industry to be in a state of perpetual crisis because the size and complexity of needed systems always seemed to exceed our tools and skills for building them. They called for a new field of “software engineering” to bring rigorous engineering methods to software development.

In 1986, software pioneer Fred Brooks wrote “No Silver Bullet”, an assessment of the progress of software engineering. He said that despite tremendous advances in tools, our ability to create dependable, reliable, usable, safe, and secure (DRUSS) software systems had not materially improved. He said that the hard part of software design was getting an intellectual grasp of the problem to be solved by software. That will never be easy. Success depends largely on the cultivation of people who have the requisite skills.

Design, the verb, means to plan an arrangement of components that enables its users to meet certain objectives effectively. An engineer designs systems that meet stated requirements. An architect designs buildings that facilitate people doing their work. A software developer designs software systems that enable users to calculate or communicate.

It is often said that designers create not only artifacts or architectures, but also practices. The designer of the video tape recorder assumed the job was to record in a time slot rather than to capture a show, and produced machines that few people could program successfully. In contrast, to avoid confusing users, the designer of the bank ATM aligned the ATM functions with the existing practices of depositing and withdrawing. Designers are always involved with anticipating how human users will react to and behave with their designs.

A big challenge for software developers is that the components of their designs are abstract objects. The lack of concreteness makes it difficult for users to test the fit between the design and their existing practices and environments. Software developers therefore need to prepare many prototypes and demonstrations.

Design

Anyone who has studied computing even for a short time will be struck by the fineness of detail that must be mastered to get computing devices to function as intended. Even at the abstract level, Turing machine programs for the simplest tasks, such as adding or multiplying numbers, are quite intricate. A single bit error in a file containing the machine code of a large program can render the entire program inoperative. Or worse: its malfunction may be obscured -- by the time humans notice the accumulation of bad results, they'll be in a real pickle to correct them.

In software development, the word design means two things: architecture and process. Architecture is a division of a system into components, their interactions, and their layout. Process means the steps followed to produce an architecture.

The four main criteria for good design are:

(1) Correctness: Does it work properly?

(2) Speed: Is it fast enough?

(3) Fault tolerance: Does it keep working?

(4) Fitness: Does it align well with the environment?

Correctness means that the software provably meets precise specifications. Correctness is challenging because precise specifications for complex systems are difficult to come by, and the proofs themselves are often computationally intractable.

Speed means that we are able to predict that the system completes tasks will within the expected deadlines.

Fault tolerance means that the software and its host systems can continue to function despite small errors, and will refuse to function in case of a large error.

Fitness means that the dynamic behavior of a system aligns with its environment of use. Fitness is challenging because the DRUSS assessments are context sensitive and much of the context is not obvious even to the experienced observer.

Does it Work Correctly?

Since the time of Issac Newton in the 1700s, astronomers, navigators, and physicists relied on hand-calculated numerical tables of logarithms, exponents, and trigonometric functions to compute the orbits of planets and positions of the stars. Imagine yourself as the author of one of those thick books of numbers. You have been working out the numbers for twenty years, using "difference methods" that calculate each new line of number by adding a small "difference" to the previous line. You have done this meticulously and you have always double-checked your arithmetic. One day you discover to your horror that a line you wrote five years ago is wrong. Every number in your book since that time is in error. You imagine the shipwrecks that were caused when navigators used your erroneous numbers.

Back in the 1830s, Charles Babbage was deeply troubled by exactly this issue. He thought that these sorts of errors could be completely eliminated by a machine that could compute the differences and print out the lines of numbers. He built such a machine and called it the Difference Engine. The machine was not subject to boredom, daydreaming, distraction, or fatigue. Its results were trustworthy and reliable.

Babbage went on to design the analytical engine, which by today's standards would be a general-purpose computer with a microprogrammed control unit. Lady Augusta Ada Lovelace assisted him by writing out algorithms for the machine; history has accorded her the honor of the first programmer. She and Babbage concluded that a perfect machine could be rendered useless by an erroneous or ill-considered algorithm.

For various reasons, Babbage did not succeed in constructing a working model of his analytical engine. Programmable computing technology died with him, laying dormant for nearly a century.

During World War II, two wartime needs resurrected the idea of a programmable computer: calculation of ballistic trajectories for ordnance, and code breaking. The people who worked on these projects quickly learned that they would be spending a large part of their lives finding mistakes in their own programs. They were keenly interested in design principles that would improve the odds they could design error-free programs that met their specifications exactly.

A mistake in a program is a logical or syntactic error that causes the program to compute incorrect values, or to enter infinite loops, for some inputs. Correctness is defined by the program's specifications. In an effort to eliminate such mistakes, software designers have

developed specification languages, methods to trace logical truth predicates throughout programs, and theorem provers that demonstrate when programs meet their specifications.

These efforts can be thwarted by large complex programs (the theorem provers take too long) and by ambiguities and oversights in the specifications. Thus designers have pursued methods that decompose complex programs into small components, enabling the theorem provers and specifications to be simpler and more effective. These modularization efforts trace back to the first digital computer projects, such as the Atlas project at University of Manchester in the 1950s. Among their many innovations, the Atlas designers created call and return instructions for subroutines, which allowed programmers to encapsulate a complex routine behind a simple interface. Through their file system, they provided libraries of standard subroutines for all users to share.

Is it Fast Enough?

In the Evaluation section we discussed why the speed question for software systems is not the same as the complexity class question for the software. The complexity measures assume that the program runs in a flat memory and gets uninterrupted use of the CPU. Real systems have limited resources; multiple jobs, running as threads in the same software, contend for them. The delays experienced by a job therefore depend on queueing, bottlenecks, service times, visit ratios, and total load on the system. Sophisticated queueing models have been used to answer performance questions in these case.

Does it Keep Working?

Most physical systems obey continuum laws that guarantee that a small change in one variable produces a small corresponding change in other (dependent) variables. Thus the system can naturally tolerate a small error. Many biological systems, including human and animal immune systems, contain self-repair mechanisms that respond to errors through feedback and correct them.

In contrast, the virtual worlds created by software tend to be highly sensitive to errors. A single bit changed in a program can drastically change the algorithm represented by the program. Moreover, it is easy to create software whose actions conflict with physical laws, leading to errors when the software interacts with the world.

For this reason, error confinement and recovery are important aspects of software design. Error confinement means to limit the number of

objects that an error can influence before it is detected. Error recovery means to remove the error and restore an error-free condition. Error recovery is easier when errors are confined.

The *principle of least privilege* means that the designer sets the default access of each process to the smallest possible set of objects needed for that process to do its job. This supports error confinement by limiting the number of objects that an error can influence.

Mechanisms for error confinement are of three kinds:

(1) Static checks in the software. Primary examples are type checks performed by the compiler: floating point operations apply to floating point data only, character strings only can be passed to subroutines expecting string arguments, file handles are passed only to file system functions.

(2) Dynamic checks in the software. Primary examples are array bounds checking: the subscript of an array is not allowed outside the range of the array, or the size of an argument to a procedure may not exceed the storage allocated for the argument.

(3) Dynamic checks in the host environment in which the software runs. Primary examples are general checks on data tags and storage bounds: a handle passed to any file system procedure must be tagged "file", an instruction that changes the memory boundaries of another process must be blocked, a memory reference to a page of another process must be prevented.

Even the best designs will need help from the host environment because software usually contains faults that the designer did not know were present.

Does it Fit?

The electronic automatic teller machine (ATM) was introduced into banking by Barclay's in 1967. It performs the basic transactions of deposit, withdrawal, and transfer through a computer interface instead of a direct interaction with a teller. It precisely imitates the steps that someone would use with a teller. Because the ATM automated the teller without changing any banking practice, it was easy for customers to use. They simply did what they always did and it worked.

The ATM is an example of an excellent fit between a machine and the standard practices of its user community. Experienced software designers understand that they are not designing a mechanism, but instead a practice for users to engage in. The closer the practice is to

other practices the users are familiar with, the easier it will be for them to use the software. The principle of fitness says that the designer should align the practices of using the software with the preexisting standard practices of the user community.

Other examples of good fit: games, Amazon.com, Bayesian spam filters, semantic web, Google, linkers and loaders, thrashing controllers, and forensics tools.

An example of poor fit is the classical video cassette recorder (VCR). These machines allowed users to create list of time slots in which the recorder would come on and record a show. They were notoriously difficult to program and there were no standard practices for them to align with. The newer generation of digital video recorders (DVR) with features like TiVo are much better aligned and have won a higher level of user acceptance. The VCR had so much difficulty because there were no standard practices for them to align with and because the thought processes they demanded did not match the thought processes of people interested in recording TV shows. For example, you might be interested in recording all episodes of a show, no matter what time they are shown; the modern DVR lets you get a "season pass" and finds the show time automatically from a schedule. With the old VCR you could only request that the recorder come on at specified time slots; the recorder knew nothing about "shows".

Other examples of bad fit: blue screen of death, voice recognition units, on-line help systems, and thrashing.

The principles of fitness are: (1) Every technology enables a practice; the software designer is designing a new practice or aligning with an existing practice. (2) Fit is most easily achieved when the practice aligns with preexisting practice or is intuitive and obvious. Sometimes software that has good fit is called "context aware" because the software and designer both understand deeper, contextual issues know to users but not spoken about.

Getting to Reliable Software

The quest for methods to improve the odds of design correct, fit software produced a rich trove of technologies to help programmers. These include:

- **High-level languages,** which allow complex operations to be expressed with simple statements.
- **Time Sharing,** which allow interaction with the program during testing and facilitated sharing of program modules.

- **Unified programming environments,** which allow sharing and reuse through libraries, common file formats, pipes, and filters.
- **Artificial Intelligence,** which provides heuristic and inference methods to develop programs.
- **Automatic programming,** which generates programs from specifications.
- **Graphical programming,** which generates programs from pictures and diagrams of control and data flow.
- **Program verification,** which proves that a program performs its function correctly.
- **Programming environments and development tools,** which keep track of versions, linkages, formats, design choices, and interfaces.
- **Object orientation,** which constructs software systems from "objects", which are abstract computational entities that perform selected operations on internal data. Objects are the most advanced form of abstraction now used in software.
- **Development process management.** Structured ways to manage the development process such as spiral, rapid prototyping and testing, user-centered, agile programming, and extreme programming.

In assessing the ability of these technologies to help achieve reliable software, Fred Brooks concluded that these tools were incapable of eliminating all errors or improving fitness. He said: "I believe the hard part of building software to be the specification, design, and testing of the conceptual construct, not the labor of representing it and testing the fidelity of the representation." He noted four approaches to dealing with the intellectual challenges of software:

- **Buy versus build:** if you can find it in a library, reuse it, it's cheaper than building it again yourself.
- **Rapid prototyping:** build models and prototypes and test them, using the results to refine your conceptual understanding of the design.
- **Incremental development:** grow and extend software rather than starting from scratch.
- **Cultivate great designers:** find the people who have the greatest talent at programming and design, and reward them.

Brooks thought that cultivating great designers is the approach most likely to yield reliable software, but it is also the greatest challenge.

The talent and skill of software developers is not a trivial point. It has been well documented that the very best programmers can be as much as 50 times more productive than entry-level programmers. They can "see" large systems at all levels of detail in their heads, and transform their vision into working code very quickly. Although these "50x" programmers are rare, those who are "10x" are not. It is well worthwhile for a software company to find a 10x programmer and pay twice the normal salary. This is far superior to hiring 10 entry-level programmers and trying to manage them well.

Driving Concerns

For the reasons outlined above, many design principles are formulated as guidelines for software engineering practice, rather than as natural laws. These principles support five driving concerns of everyone who users and designs software:

- **Simplicity:** We exploit various forms of abstraction and structure that overcome the apparent complexity of applications.
- **Performance:** We try to predict how long computations will take, how many of them a system can support, and how much capacity our systems need to meet targets of throughput and response time.
- **Resilience:** We want our systems to "bounce back" from errors, and so we build in redundancy, error detection and recovery, and end-to-end error checking.
- **Evolvability:** We want our systems to adapt to changes in function and scale.
- **Security:** We want our systems to control access, ensure secrecy and privacy, authenticate users and transactions, maintain integrity, and operate safety.

Now you can see why the challenge facing the great designer is so considerable. That person must:

(1) Be able to visualize software modules operating at their different levels of abstraction and move the mind to any level easily.

(2) Be able to conform the software to a good specification.

(3) Be able to fit the software to the standard practices of the user community.

(4) Be sensitive to the user context.

(5) Be aware of how users will assess simplicity, performance, resiliency, evolvability, and security.

Much of the job of a designer is to do these things with great skill, using available software development tools to assist.

Design Principles

There are clear principles that help designers meet the challenges. If you look through software engineering literature, you will find long lists like:

abstraction	version control
separation of concerns	divide-and-conquer
modularity	functional levels and hierarchies
interfaces	layering
information hiding	virtual machines
encapsulation	prototype and test
separate compilation	process-centered design
reuse	human-centered design
packages	

Many of these principles are derivatives of others. The most fundamental principles are described below.

Hierarchical Aggregation

Hierarchical aggregation means that objects (components) consist of interconnected groups of smaller objects, and are themselves components of larger objects. You interact with an object as a unity and are not concerned about the individual parts constituting it. When you do look inside, you focus on the interactions among components without concern about what is going on in the external environment. Thus there is hierarchy with smaller aggregates making up large aggregates. Aggregates at every level of the hierarchy are also insulated from lower and higher level details.

Hierarchical aggregation is common in nature. In physics and astronomy, objects are formed into hierarchies with levels including quarks, electrons and protons, atoms, molecules, materials, planets, solar systems, galaxies, clusters, and quasars. The scale at the high end is about 10^{27} meters and at the low end 10^{-14} meters.

In biology, living organisms have hierarchies with levels including DNA, genes, cells, organs, nervous systems, plants, animals, and social systems.

In mathematics, fractals are sets whose components follow the same structure rules as the set. The same structure is observed at all levels of aggregation.

In computing, the principle of locality, discussed under Principles of Recollection, is a consequence of hierarchical aggregation. A single interaction with an object can trigger a long sequence of internal actions among the object's components. All those interactions are confined to the object's components over an extended period.

The principle of hierarchical aggregation underlies software principles including (1) functional decomposition, (2) modules and interfaces, (3) information hiding, (4) encapsulation, and (5) abstraction.

Levels

Levels is a form of hierarchical aggregation that stratifies the levels of abstraction. All objects in the same level are treated as peers with respect to how they contribute to the higher levels. The levels structure of the universe, mentioned earlier, illustrates this. At the level of atoms, we are concerned with issues like chemical bonding, electron deficiencies, and molecular shapes; but not with what materials the atoms belong to, or with what holds the protons and neutrons together inside atoms.

The levels principle has been used in computing to structure very complex software systems and enable them to be proved correct. One example is the operating system. In 1968, Edsger Dijkstra designed an operating system at Technische Hogeschool Eindhoven, known to posterity as "THE operating system". He organized it into 7 levels, each containing software components that realized a particular abstraction. The "processes" level, for example, abstracted away from the processor by creating an abstraction of a computation that always moved forward except when it was waiting for a signal. All software components above that could be programmed with processes instead of subroutine calls and "CPU switches". The problem of multiplexing the CPU among processes to give the illusion of joint progress was solved once and for all at the processes level. A decade later, a group at SRI, Inc., built a "provably secure operating system" of 14 levels, in which they were able to prove that a level is secure given that all levels below it are secure.

Another venue for level structure in software has been the network protocols such as in the Internet. The file transfer protocol TCP, for example, is built on several lower level layers including IP protocol, routing protocol, data link protocol, and physical signal protocols.

The levels principle applies nature's learning to complex networked software systems.

Virtual Machines

A virtual machine is a simulation of one computer by another. The idea traces back to Alan Turing's Universal Machine. The simulation principle behind it says that a more complex system can be simulated from less complex components.

Today there are three uses of the term virtual machine.

The first use refers to the simulation of one machine by another. The simulating machine has subroutines that carry out the effect of the machine instructions on the other. This idea came into practice with the second generation of computers (late 1950s). The new computer had to run all the software written for previous versions of the computer. So the new computer's instruction set contained software or firmware that simulated the old computer's instruction set. This type of simulation was called "emulation". As the old programs were brought up to date and recompiled, they would use the new, more efficient instruction set and would run faster. Emulation is still used today in the programs on Apple Computers that host the Windows environment. Its most widespread use today is the Java Virtual Machine, which emulates the Java "byte code" instruction set on a host computer. This allows great portability of Java programs.

IBM pioneered a second use of the virtual machine principle beginning in the mid 1970s. The IBM virtual machine was a complete simulation of an IBM mainframe identical in every way to the original except that it had a reduced main memory. The simulator partitioned the real main memory and placed a copy of the operating system in each block of the partition. The program running on the operating system in a block executed all instructions normally, except for a few that were capable of affecting other users. A similar idea is used today in the multitasking features of operating systems like Mac OS Tiger and Windows Vista. This approach allows the virtual machine to run at nearly the same speed as the real machine; there is no significant performance loss.

A third use of the virtual machine principle was pioneered in the Multics system at MIT in 1968 and the Unix system at Bell Labs in 1972. A " process" was defined as a program in execution on a virtual machine. The virtual machine was simply a standard template for providing input and output to a running program and connecting with any sub-machines it may have spawned. Any user program would be

embedded into the standard process. Because all the virtual machines have an identical form, they enable arbitrary programs to be interconnected in pipelines of any length.

Objects

Because software programs can transform a computer into any other virtual machine, it has become standard to say that software creates abstractions that execute and perform actions. The designer's job is to find a decomposition of the original problem into a hierarchy of abstractions that solve the problem. To make this process more reliable, over time we

- developed standard programming practices of hiding data structures behind an interface of functions defined by subroutines that manage the data structure without the user having to see it;
- developed notations for expressing abstraction and included them in our programming languages;
- developed ways to define classes of abstractions that all use the same operations, for example files; and
- developed ways to share abstractions without causing race conditions.

We packaged all this up into object oriented languages and we teach it to computing students almost as soon as they enter the major.

Objects are often presented as fundamental because they manifest the fundamental principle of abstraction. However, they are actually an advanced concept because they provide a unified way to deal with a host of structural and synchronization problems.

End-to-End Principle

Consider any network in which data are moved from one machine to another. If we check the integrity of the bits at any point on the path, but not the absolute end, there is a possibility that an error can occur in the final segment. Therefore, we must check that the bits at the end are the same as those at the beginning. This is called end-to-end error checking.

The TCP (transport control protocol) of the Internet uses end-to-end checking to assure that files are transferred with 100% reliability. The source file is broken into packets, which are sequence-numbered and sent to the receiver. The receiver checks parity bits to detect corrupted packets, and it notes gaps in its list of received packets to

detect missing packets. It sends acknowledgements to the sender that confirm the number of packets successfully received. Both sender and receiver have time-out alarms after which they resend packets for which there has been no acknowledgement. There is no checking in the Internet itself to be sure that packets have not been corrupted or lost. The Internet is simply a best-effort medium that tries to get packets delivered but may not succeed, and it does not retry. The end-to-end checking of the TCP makes file transfer reliable even though packet transfer is unreliable.

Another approach to file transfer is bulk: simply to move the entire file along with a checksum in a single transfer. The receiver computes the checksum of the received bits and compares with the received checksum. If they are the same, it accepts the transfer. If different, it requests the sender to send the whole file again. This scheme will work well if the transfer time of the largest files is smaller than the mean time between errors. Otherwise, it will generate a lot of resend-the-whole-file requests, which can become quite expensive. In such cases, the TCP protocol is more efficient because it requires transmission only of missing packets, not the whole file.

Notice that the bulk-file transfer protocol is likely to transmit the least number of bits when the error rate is low, but likely to transmit huge numbers of bits when the error rate is high. This is a problem with end-to-end protocols: they may overload. Designers need to check that the assumptions that make the end-to-end tests simple actually hold.

The only reason to include checks on the path would be to improve performance. For example, routers could cut retransmission times in TCP by caching packets. Such checks do nothing to improve reliability.

Design Hints

In 1983 Butler Lampson, a superb and accomplished designer, summarized a number of guidelines that help designers, but cannot be considered as principles that work in all cases. (See his "Hints for Computer System Design," in Proc. SOSP-9, 1983.) He called them design hints. The table below summarizes his hints as slogans. We won't explain them. The point is that there is considerate art in designing. Lampson has outlined the best practices of that art.

	Correctness & Fit	**Speed**	**Fault Tolerance**

Use cases	Separate normal and worst cases	Safety first Shed load End-to-end	End-to-end
Interface	Keep it simple Do one thing well Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep interface stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Rerse a good idea Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints