Security Design Principles: Simplicity

CS-3113: PRINCIPLES OF CYBER SECURITY

BENJAMIN R. ANDERSON

Design Principles

The National Security Agency (NSA) has designated 11 design principles as part of the Cyber Operations Centers for Academic Excellence (CAE-CO)

Their purpose:

- Help programmers develop more secure code
- When followed, code should:
  - Contain fewer errors (often a source of security vulnerabilities)
  - Be designed with security built in (as opposed to added development is complete)

These principles are based on the previous work by Saltzer and Schroeder that we covered in the previous lesson (Differences between the two will be noted)

The Eleven Design Principles

General/Fundamental Design Principles

1. Simplicity (related to Economy of Mechanism but not exactly the same)
2. Open Design
3. Design for Iteration (not specifically identified by Saltzer and Schroeder)
4. Least Astonishment (related to Psychological Acceptability)

Security Design Principles

5. Minimize Secrets (not specifically identified by Saltzer and Schroeder)
6. Complete Mediation
7. Fail-safe Defaults
8. Least Privilege
9. Economy of Mechanism
10. Minimize Common Mechanism (related to Least Common Mechanism)
11. Isolation, Separation and Encapsulation

Methods for Reducing Complexity

Goes along with the Simplicity Design Principle

They are:

1. Abstraction
2. Modularity
3. Layering
4. Hierarchy

Defining the Problem

Quote by Viega and McGraw in *Building Secure Software*:

- "Since no one knows how to build a system without flaws, the alternative is to rely on eight design principles, which tend to reduce both the number and the seriousness of any flaws: Economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability.“

Note: They use the 8 Principles by Saltzer and Schroeder

Errors in Software

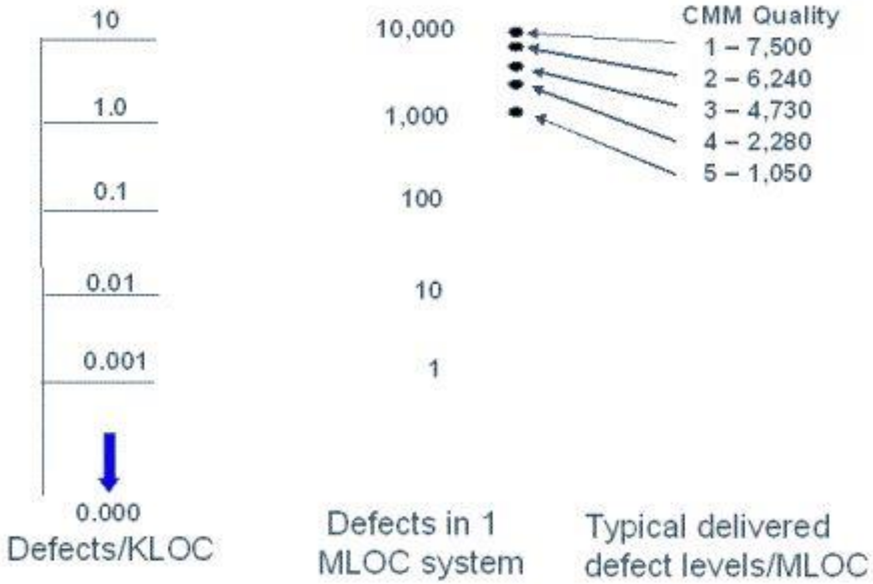
One of the greatest sources of security problems is errors in the code

It is very difficult to deliver code with no errors in it, and we have to answer some key questions:

- Just how hard is it to deliver code with no errors in it?
- What is the industry average for errors per thousands of lines of code (KLOC) or millions of lines of code (MLOC)?

In 1986 the Software Engineering Institute (SEI) developed the CMM (Capability Maturity Model) to assess the capability of software development organizations

- The CMM level is a measure of how mature the software development process is for an organization
- The higher the maturity level, the more mature the process
- This (usually) results in fewer errors in delivered code



Errors in Software

Software is more than just lines of code – and errors are not always linear with KLOC

The number and design of interfaces can add complexity and add additional problems

In addition, designed and expected behavior can end up as a vulnerability if the computing environment changes

- In 2014, the Shellshock vulnerability was discovered
- Affects the Bash shell and allows an attacker to execute arbitrary commands
- The behavior was expected, and was introduced into Bash on September 1989
- https://en.wikipedia.org/wiki/Shellshock_(software_bug)

Errors in Software

It is difficult to find accurate numbers for errors per KLOC since companies generally don’t discuss things that could reflect negatively on them

The book *Code Complete* published by Steve McDonnell in 2004 provides a range of answers on this topic:

(a) Industry Average: "about 15 - 50 errors per 1000 lines of delivered code." He further says this is usually representative of code that has some level of structured programming behind it, but probably includes a mix of coding techniques

(b) Microsoft Applications: "about 10 - 20 defects per 1000 lines of code during in-house testing, and 0.5 defect per KLOC (KLOC IS CALLED AS 1000 lines of code) in released product (Moore 1992)." He attributes this to a combination of code-reading techniques and independent testing (discussed further in another chapter of his book).

(c) "Harlan Mills pioneered 'cleanroom development', a technique that has been able to achieve rates as low as 3 defects per 1000 lines of code during in-house testing and 0.1 defect per 1000 lines of code in released product (Cobb and Mills 1990). A few projects - for example, the space-shuttle software - have achieved a level of 0 defects in 500,000 lines of code using a system of format development methods, peer reviews, and statistical testing."

Book reference: Steve McDonnell, Code Complete, 2nd Edition. Redmond, Wa.: Microsoft Press, 2004.

Source Lines of Code (SLOC)

Even at small error rates per KLOC or MLOC, the sheer size of applications, operating systems, and ecosystems (Google or Amazon) can result in large numbers of defects

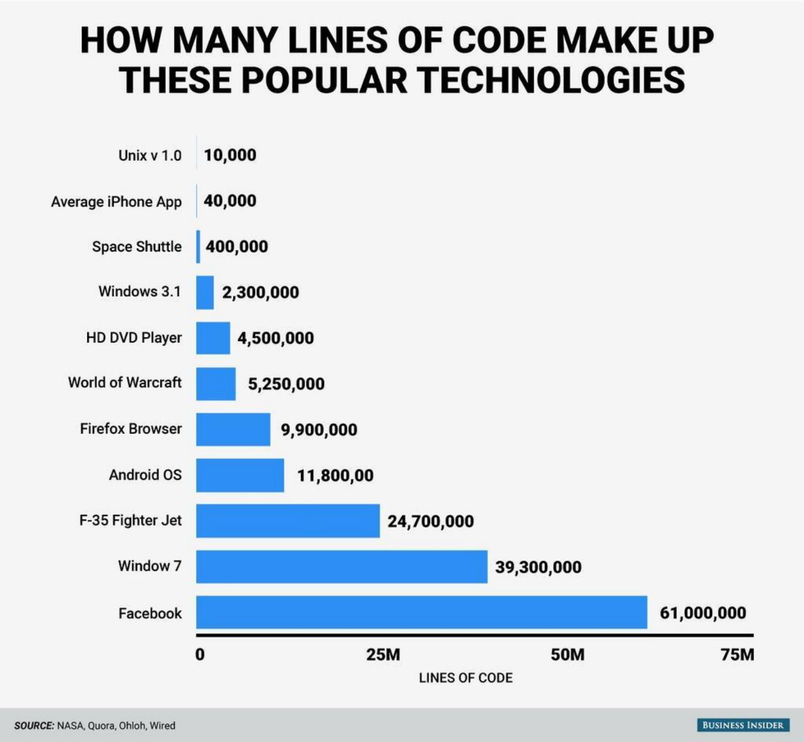
You can see from the diagram that Windows 7 is estimated at ~40 MLOC and Facebook is ~61 MLOC

In 2015, the Linux kernel (pre-4.2) was estimated at ~20 MLOC

- https://en.wikipedia.org/wiki/Source_lines_of_code

More estimates on software size can be found here:

- https://www.informationisbeautiful.net/visualizations/million-lines-of-code/



SOURCE: NASA, Quora, Ohloh, Wired

Image from: https://twitter.com/saanpaurseedi/status/819876736745631744

Source Lines of Code (SLOC)

From Wired:

- *Google's Rachel Potvin came pretty close to an answer Monday at an engineering conference in Silicon Valley. She estimates that the software needed to run all of Google's Internet services—from Google Search to Gmail to Google Maps—spans some 2 billion lines of code. By comparison, Microsoft's Windows operating system—one of the most complex software tools ever built for a single computer, a project under development since the 1980s—is likely in the realm of 50 million lines.*
- https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/

This is:

- 32x Facebook
- 50x Windows 7
- 100x Linux kernel

Do Software Errors Mean A Security Problem?

Modern applications, operating systems, and ecosystems can be enormous

If the number of errors per lines of code is still the same as the earlier studies we can expect thousands of undiscovered errors in delivered code

- Modern tools help identify errors, but the complexity of modern applications is much higher – and the development cycle much faster – so the overall rate probably hasn’t changed very much

Is every software error going to result in a security problem?

- No – but each error needs to be checked to ensure that it doesn't cause a security issue

Is every security problem the result of a software error?

- No – misconfiguration and user errors are also major sources of security problems

Adopt Sweeping Simplifications

Complexity is the worst enemy of security – Bruce Schneier

Complex systems are inherently more insecure because they are difficult to design, implement, test, and secure

Consider errors per lines of code

- Unless designed very carefully and implemented in a controlled process, the larger the program, the more complex it will be
- This has a multiplying effect on errors

There are also design errors

- If the design of the program contains errors, when it is coded, those errors will be implemented
- In this case, the error is not due to the coder making a mistake but the designer of the program

The more complex a system, the less assurance we may have that it will function as expected

“Complexity the Worst Enemy of Security”

From a 2012 article in Computerworld Hong Kong by Bruce Schneier:

> ***BS****: The Internet and all the systems we build today are getting more complex at a rate that is faster than we are capable of matching. So while security in reality is actually improving but the target is constantly shifting and as complexity grows, we are losing ground.*
>
> ***CWHK****: And is this the reality that we have to accept today and for the foreseeable future?*
>
> ***BS****: I'm sure that this isn't the answer that many would want to hear but yes this is the reality today. I'm sure that out there somewhere is a point where the complexity slows down and we find a way to gain back some ground. But it's hard to envisage as there is so much change and it's happening so fast that every new thing brings added complexity. And complexity is the worst enemy of security.*
>
> ***CWHK****: So how do we reconcile the irony that complexity is something we desire?*
>
> ***BS****: The thing is we absolutely love complexity. It's down to using these new apps on our smartphones, it's using Skype on our work device while using the airport WiFi. We all like these things and having access to our data at all times, but this creates more complexity and it makes security harder.*
>
> *There's no way I would advise anyone to stop doing these things so we just have to find ways to live with this.*
>
> *If you look back to five years ago, we were all discussing how to lock down all our access points to the enterprise. Today all the data resides outside the network, so who cares about where the access points are today? That's the ongoing evolution we have to accept and deal with.*
>
> ***CWHK****: So do we have to constantly redefine the meaning of security?*
>
> ***BS****: We do that almost on a daily basis anyway. In the real world we do this, as security is a very much a local construct. What it means to be secure in Hong Kong is very different to say Manila or downtown Kabul. We as humans are very good at adapting to scenarios to create a new sense of normal.*

Source: https://www.schneier.com/news/archives/2012/12/complexity_the_worst.html

Adopt Sweeping Simplifications

To combat complexity, we must “simplify” our code

This will allow developers to:

- Easily see what they are doing (and how it fits)
- Keep control of the complexity

The provides many advantages to the developers

- Allows the designer to make compelling arguments for correctness
  - It's easier to validate a simple design rather than a complex one
- Makes detail irrelevant
  - Simplified designs will lead to an easier coding process thus the details on how it will be implemented become less important
- Makes clear to all participants exactly what is going on
  - Simple designs are easier for folks to understand and are less likely to be misinterpreted

Complexity of information systems and processes are likely to increase with our increasing expectations of functionality

- Need to carefully draw the line between avoidable and unavoidable complexity
- We shouldn’t sacrifice security for additional features, only to regret it later

When you have to choose between a complex system that does much and a simple system that does a bit less but still enough, choose the simple one

Spectacular Software Failures

Daniel Martin of Raygun has a blog post: *11 of the most costly software errors in history*

- *https://raygun.com/blog/costly-software-errors-history/*
- Read through these and see how errors can have a great impact

Software errors can also cost lives – like the Therac-25 Medical Accelerator

- The Therac-25 emitted beams of electrons to kill cancer cells
- It had three modes:
  - Field light mode (visible light)
  - Low-powered direct electron beam
  - Megavolt X-ray mode
- The Megavolt mode required shielding and filters and an ion chamber to keep the dangerous beams safely on target
- The software that powered the unit was repurposed from the previous model, and wasn't adequately tested
- If the operators changed the mode of the device too quickly, a race condition occurred
  - Two sets of instructions were sent, and the first one to arrive set the mode
- In six documented cases, this meant that megavolt X-rays were sent, unfiltered and unshielded, toward patients requiring direct electron therapy
  - At least two of them screamed in pain and tried to run from the room
  - All of them suffered radiation poisoning and 4 people were killed

Complexity

As the number of requirements grows, so can the number of exceptions, corner cases, and the overall complexity

- All of the special cases in the United States tax code makes filling out an income tax return a complex task
- The impact of any one exception may be minor, but the cumulative impact of many interacting exceptions can make a system so complex that no one can easily understand it

Complications also can arise from outside requirements

- Requiring software to work with a specific type of hardware or environment
- Industrial control systems may require trade-offs given the extreme environments they are deployed in
  - A temperature sensor that can handle 1000 degrees may not be able to include digital components
  - Therefore, other parts of the system may have to compensate by handling the conversion of analog signals to digital

Unfortunately, generality also contributes to complexity

- A designer has to use good judgment in the trade-off between simplicity and generality

Complexity

Systems grow in complexity with the passage of time

Even the simplest change to repair a bug, has an increasing risk of introducing another bug

A common phenomenon in older systems is that the number of bugs introduced by a bug fix release may exceed the number of bugs fixed by that release

The lifetime of a system is usually limited by the complexity that accumulates as it evolves farther from its original design

Avoid excessive generality:

- ***If it is good for everything, it is good for nothing.***

Methods to Handle Complexity

Many systems have common problems and common sources of complexity

Techniques for coping with these sources of complexity have been developed

These techniques can be loosely divided into four general categories:

1. Modularity
2. Abstraction
3. Hierarchy
4. Layering

Modularity

The simplest, most important tool for reducing complexity is the divide-and-conquer technique

- Analyze or design the system as a collection of interacting subsystems
- These are called modules

The power of this technique lies primarily in being able to consider interactions among the components within a module without simultaneously thinking about the components that are inside other modules

The feature of modularity that we are taking advantage of here is that it is easy to replace an inferior module with an improved one, allowing incremental improvement of a system without completely rebuilding it

Modularity

In the 1960s, computer systems were a vertically integrated industry

- Companies like IBM, Burroughs, Honeywell, and others each provided top-to-bottom systems and support
  - They did everything: processors, memory, storage, operating systems, applications, sales and maintenance
  - IBM even manufactured its own chips

By the 1990s the industry had transformed into a horizontally organized one

- Hardware: Intel sells processors, Seagate sells hard drives, Broadcom sells network components
- Software: Microsoft sells operating systems, Adobe sells text and image applications, Oracle sells database systems
- Systems: Dell sells complete computer systems
- This was a dramatic change from only 20 years earlier

This trend is reversing in some places today

- Apple and Microsoft both supply systems (Mac and Surface)
- Apple even designed the M1 chip for use in its systems (but they are produced by TSMC)

However, for the majority of the industry, once this modularity was defined and proven to be effective, other vendors were able to leap in and turn each module into a distinct business

Open Problem: How do we decide where and how to modularize?

Abstraction

Ideally, divisions usually follow natural or effective boundaries

An additional requirement on modularity is called abstraction

- Abstraction is a separation of interface from internals, of specification from implementation
- Every module should be able to treat all the others entirely on the basis of their external specifications, without need for knowledge about what goes on inside

Enforcing modularity:

- The goal of minimizing interconnections among modules may be defeated if there are **unintentional or accidental interconnections**. These can arise from:
  - Implementation erros
  - Design attempts (or “sneaky coding”) that bypass modular boundaries in order to improve performance or meet some other requirement

Software is particularly subject to this problem, because the modular boundaries that separately compiled subprograms provide are actually somewhat soft and easily penetrated by errors in using pointers, filling buffers or calculating array indices

To combat this, system designers prefer techniques that enforce modularity by interposing impenetrable walls between modules. These techniques assure that there can be no unintentional or hidden interconnections.

Abstraction

Closely related to abstraction is an important design rule that makes modularity work in practice:

The Robustness principle:

- ***Be tolerant of inputs and strict on outputs***

This principle means that a module should be designed to be liberal in its interpretation of its input values

- Accepting them even if they are not within specified ranges
- As long as it is still apparent how to sensibly interpret them

Example:

- The HTML for a webpage is missing a close paragraph tag </p> before an opening <p> tag and the browser simply formats a new paragraph; or it ignores nested italics flags: <i>Text<i><i><i>Text</i>

Abstraction

Buying things on the Internet

- It is not uncommon for web sites to not adequately check inputs for valid entries
  - Programmers don't always consider as many possible inputs, valid or invalid, as possible
  - This topic will be covered in more depth later in the course
- For example, many online carts have a field for “Quantity”
  - What happens if the user inputs a negative number?
  - A very large (outlandish) number?
  - A real number? (But one that’s more than the quantity they have in stock)
  - Alphanumberic or special characters?
  - These cases are why many online shops have dropdown boxes for the quantity instead of a text field

- Do you use any shopping websites that don’t properly validate inputs?

Layering

Systems that are designed using good abstractions tend to minimize the number of interconnections among their component modules

One powerful way to reduce module interconnections is to employ a particular method of module organization known as layering

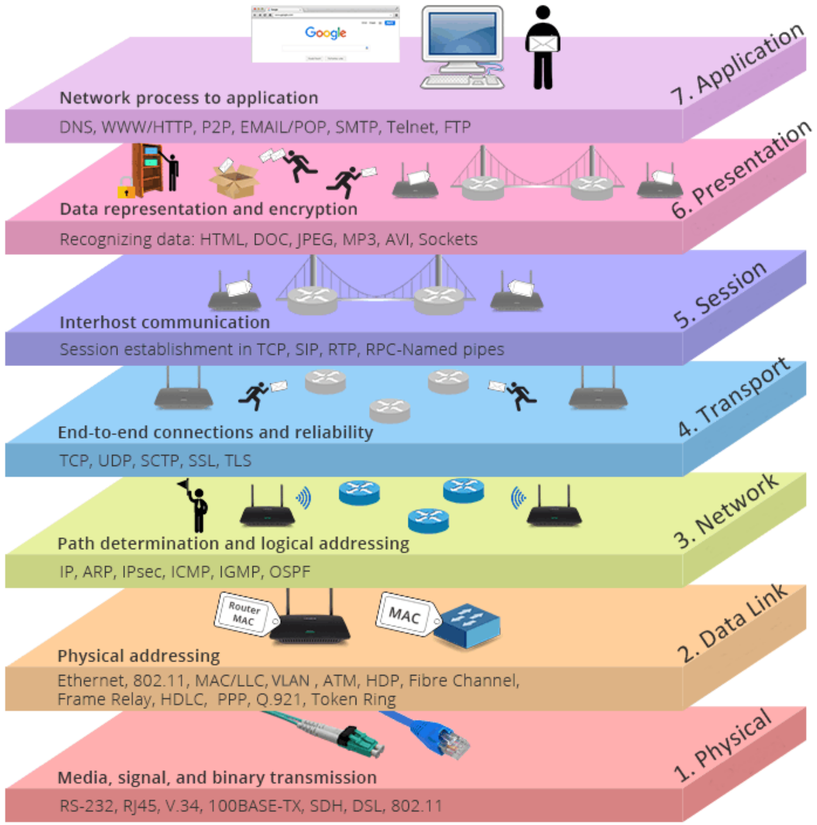
Designing with layers

- One builds on a set of mechanisms that are already complete (a lower layer), and uses them to create a different complete set of mechanisms (an upper layer)
- A layer may itself be implemented as several modules but, as a general rule, a module of a given layer interacts only with its peers in the same layer and with the modules of the next higher and next lower layers
- That restriction can significantly reduce the number of potential inter-module interactions in a large system

Layering

The OSI Model in networking is a perfect example of layering

- In this model, each layer ONLY interacts with the layer above it and below it
- Your web browser doesn't need to know what kind of network (Ethernet, wireless, fiber, etc.) is being used for the HTTPS request – it just hands the request off to the presentation layer
- The physical layer isn't concerned with what kind of session is being used, or what routing is done – it just provides the electrical, mechanical, and procedural interface to the data link layer to send or receive information



7. Application
Network process to application
DNS, WWW/HTTP, P2P, EMAIL/POP, SMTP, Telnet, FTP

6. Presentation
Data representation and encryption
Recognizing data: HTML, DOC, JPEG, MP3, AVI, Sockets

5. Session
Interhost communication
Session establishment in TCP, SIP, RTP, RPC-Named pipes

4. Transport
End-to-end connections and reliability
TCP, UDP, SCTP, SSL, TLS

3. Network
Path determination and logical addressing
IP, ARP, IPsec, ICMP, IGMP, OSPF

2. Data Link
Physical addressing
Ethernet, 802.11, MAC/LLC, VLAN , ATM, HDP, Fibre Channel, Frame Relay, HDLC, PPP, Q.921, Token Ring

1. Physical
Media, signal, and binary transmission
RS-232, RJ45, V.34, 100BASE-TX, SDH, DSL, 802.11

https://community.fs.com/blog/tcpip-vs-osi-whats-the-difference-between-the-two-models.html

Hierarchy

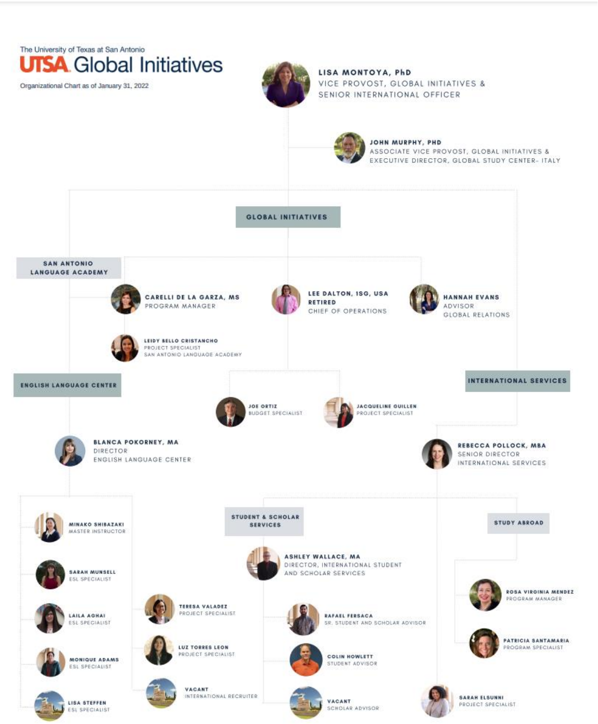
Hierarchy is the final major technique for coping with complexity

It also reduces interconnections among modules, but in a different, specialized way

- Start with a small group of modules, and assemble them into a stable, self-contained subsystem that has a well-defined interface
- Next, assemble a small group of subsystems to produce a larger subsystem
- This process continues until the final system has been constructed from a small number of relatively large subsystems

The result is a tree-like structure known as a hierarchy

- An org chart is an example of a hierarchy

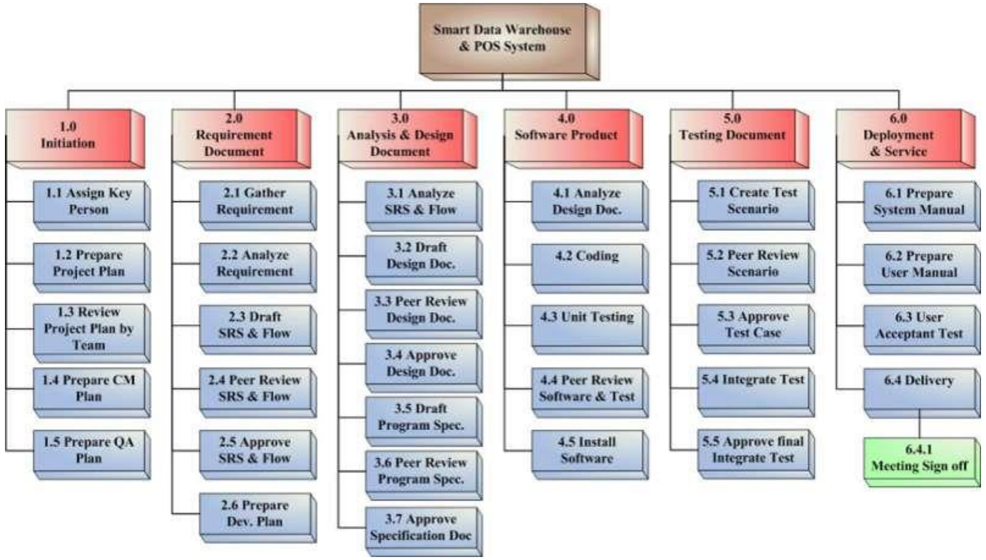


https://global.utsa.edu/home-blocks/GI-OrgChart-1.31.22.pdf

Hierarchy

This is how a point-of-sale (POS) system might be broken down

- The overall system consists of 6 subsystems
- Each subsystem has multiple components under it
- Each component might be further broken down into specific modules



Barriers to Simplicity

There are many reasons while system architects, designers, and programmers don’t adopt this principle

- There may be a push to add features to the existing system to “make it better”
- The technology has improved so much that cost and performance are no longer constraints
- The suggested new features has been successfully demonstrated in another system or platform giving the illusion of simplicity for this system
- None of the exceptions, corner cases, or other complications are hard to deal with ***individually***
- There is fear that a competitor will do it first
- Human factors like pride and overconfidence are stronger than the fear of too much complexity

The bottom line is that a computer system designer's most potent weapon against complexity is the ability to say:

- "***No. This will make it too complicated.***"

Summary

Complexity is a challenge in any complex system

Abstraction, modularity, layering, and hierarchy can help make complexity manageable

Computer systems differ from other complex systems which designers experience:

- They are not limited by physical laws
- The rate of change is unprecedented.

Hofstadter's Law:

- It always takes longer than you expect, even when you take into account Hofstadter's Law
- Douglas Hofstadter: *Gödel, Escher, Bach: An Eternal Golden Braid* (1979)