Security Design Principles: Design for Iteration Least Astonishment

CS-3113: PRINCIPLES OF CYBER SECURITY

BENJAMIN R. ANDERSON

Review: 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***
4. ***Least Astonishment***

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

Review: Saltzer and Schroeder's Security Design Principles

Many security issues are a result of poor coding techniques which lead to flaws in the program which can result in a security hole that can be exploited

Saltzer and Schroeder came up with a list of 8 security design principles that, if followed, would help programmers reduce the number of errors and design more secure software

1. Economy of mechanism – A simple design is easier to test and validate
2. Fail-safe defaults –In computing systems, the safe default is generally "no access" so that the system must specifically grant access to resources
3. Complete mediation – Access rights are completely validated every time an access occurs
4. Open design –secure systems, including cryptographic systems, should have unclassified designs
5. Separation of privilege – A protection mechanism is more flexible if it requires two separate keys to unlock it, allowing for two-person control and similar techniques to prevent unilateral action by a subverted individual
6. Least privilege – Every program and user should operate while invoking as few privileges as possible
7. Least common mechanism – Users should not share system mechanisms except when absolutely necessary
8. ***Psychological acceptability – users won't specify protections correctly if the specification style doesn't make sense to them***
    - ***Related to the principle of Least Astonishment***

Review: Methods for Reducing Complexity

Goes along with the Simplicity Design Principle

They are:

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

Controlling Complexity

We previously mentioned the importance of ***simplicity*** and avoiding complexity where we can

- Modularity, abstraction, layers, and hierarchy assist in this goal
- By themselves, they are not enough

There is an underlying assumption to these four methods:

- ***The designer understands the system being designed***

This is an assumption – and recall what Einstein said about assumptions

It is hard to choose the right modularity/abstraction/layering/hierarchy from all the potential options

- Consider how many NPM packages exist
- How does a designer or developer select the right modules to design or code into an application?
- This is especially difficult if the developer doesn't have a good understanding of the design
- Unclear requirements can make designing a system difficult – even for an experienced designer

Designers of computer systems have developed and refined at least one additional technique to cope with complexity: ***Iteration***

- Iteration addresses the issue that you may not fully understand what it is that you are building when you begin the process

Design for Iteration

You won't get it right the first time, so design it to be easy to change!

The basics of iteration:

- Start by building a simple, working system with only a small subset of the requirements or use cases
- Once you have a simple, working system – implement other requirements or use cases to evolve the system to be more complete
- Small steps can reduce the risk that complexity will overwhelm the system design
- Step-by-step Development not "Big Bang Development"

Keep in mind that you also iterate when implementing the original, simple system

- *This is not just about future versions – or the final product*
- There is always the chance priorities or timelines will change and the system will be used "As Is"

Advantages:

- Having a working system available at all times helps provide assurance that something can actually be built
- It provides on-going experience with the current technology, ground rules, and an opportunity to discover and fix bugs
  - Lessons learned can be incorporated into future iterations
- It is easier to incorporate technology changes that arrive during the system development

How many versions of the iOS or Android operating system have you seen?

- The release numbers capture their iterations

Considerations

Considerations

- Take small steps
  - Allows discovery of design mistakes and bad ideas quickly
  - This allows them to be changed or removed before they become too embedded in the system to change
    - Watch out for the Sunk Cost Effect! https://en.wikipedia.org/wiki/Sunk_cost
  - If code reviews and automated testing are used, implementation mistakes can be found quickly
    - This allows changes while the developers are still familiar with the code (and not the next module, or the one after that)
- Don't rush
  - Each individual step must still be well planned
  - Have to avoid the temptation to rush the current iteration so you can begin the next one
  - Development is a marathon, not a sprint (Not to be confused with a sprint in the Scrum Framework)
- Plan for feedback
  - Feedback needs to be at all levels and encouraged by management
  - Feedback can also include users and customers
    - “Send usage information” and “Report a bug” features
- Study failures with the goal of learning from them, rather than assigning blame for them
  - Need a culture of improvement – not blame
  - If problems are penalized, there is an incentive to hide them, causing future (and probably bigger) problems
  - This information can then be used in improving the product, future iterations, and other future products

Quotes and Comics

Plan to throw one away; you will, anyhow.

- Frederick P. Brooks, *The Mythical Man Month* (1974)

If you design it so that it can be assembled wrong, someone will assemble it wrong.

- Edward A. Murphy, Jr. (The original Murphy's law, 1949)
- This also applies to system configuration – especially system defaults

I was to learn later in life that we tend to meet any new situation by reorganization; and what a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency and demoralization.

- Shortened version of an observation by Charlton Ogburn, *Merrill's Marauders: The truth about an incredible adventure*, Harper's Magazine (January 1957)
- ***Note***: Don't confuse reorganization with design for iteration

A system is never finished being developed until it ceases to be used.

- Attributed to Gerald M. Weinberg

https://jobmob.co.il/blog/funny-ikea-job-interview-cartoon/

Software Development Processes

At this point in your academic career, you have all developed multiple programs

There are many ways to approach coding a specific problem
- When starting out, this is often the “Jump in and use the Big Bang approach”
- Little time is spent on designing the program
- This method is often… contraindicated

A structured development process is required to develop any non-trivial system correctly

The generic steps can be defined as:
- Specification
- Design
- Validation
- Evolution

A software process model is an abstract representation of the steps in the development process
- It presents a description of a process from some particular perspective

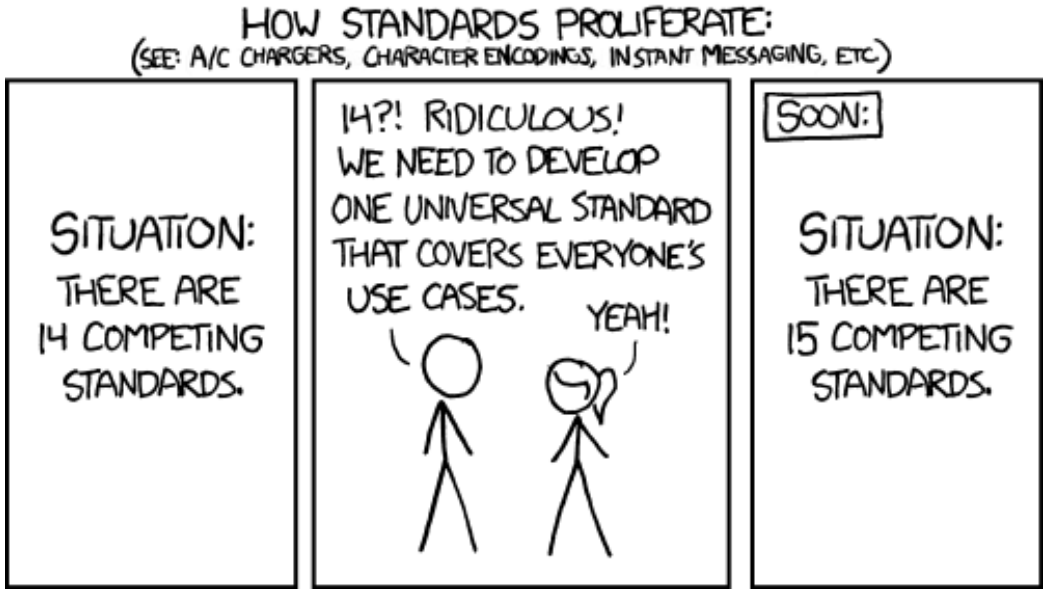
There are many different software development processes
- These different processes also have different security implications
- For example, the process may dictate how often a security analysis must be performed

Generic Software Development Models

We will be looking specifically at the following software development lifecycle models:

- Waterfall
- Evolutionary Development
- Component-based Software Engineering
- Spiral Modal
- Agile

There are a lot of models – and variants – to choose from!



Comic from XKCD: https://xkcd.com/927/

Waterfall Model

This model is characterized by separate and distinct phases of specification and development

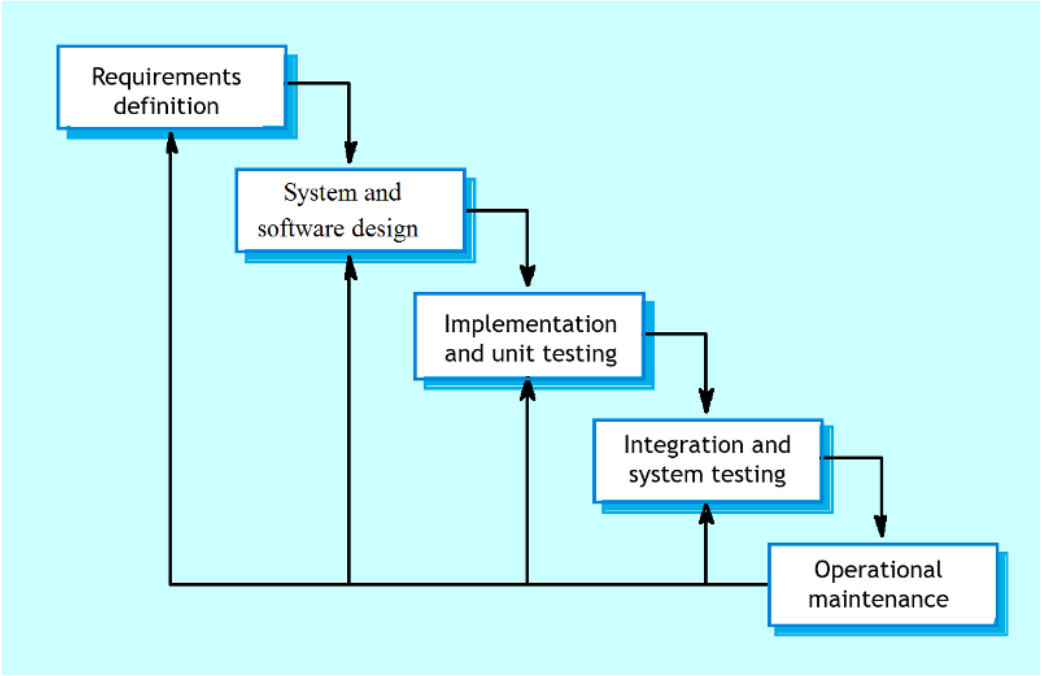
Main drawbacks:

- Difficulty in accommodating change
- Must complete the previous stage before moving on – no parallel efforts
- Can only go back one step if there is a problem
  - The previous step is done again and not just "jumped into" to make a quick change
  - If a problem with the requirements is discovered in implementation, the model does not allow modification of requirements

Other issues:

While an organized approach, it does have several problems:

- Can't easily respond to changing customer requirements or feature requests
- Only appropriate when the requirements are well-understood and minimal changes are expected in the design phase
- For most systems, requirements are not stable enough
- Mostly used for large systems engineering projects – like the James Webb Space Telescope



Evolutionary Development

An improvement on the Waterfall Model

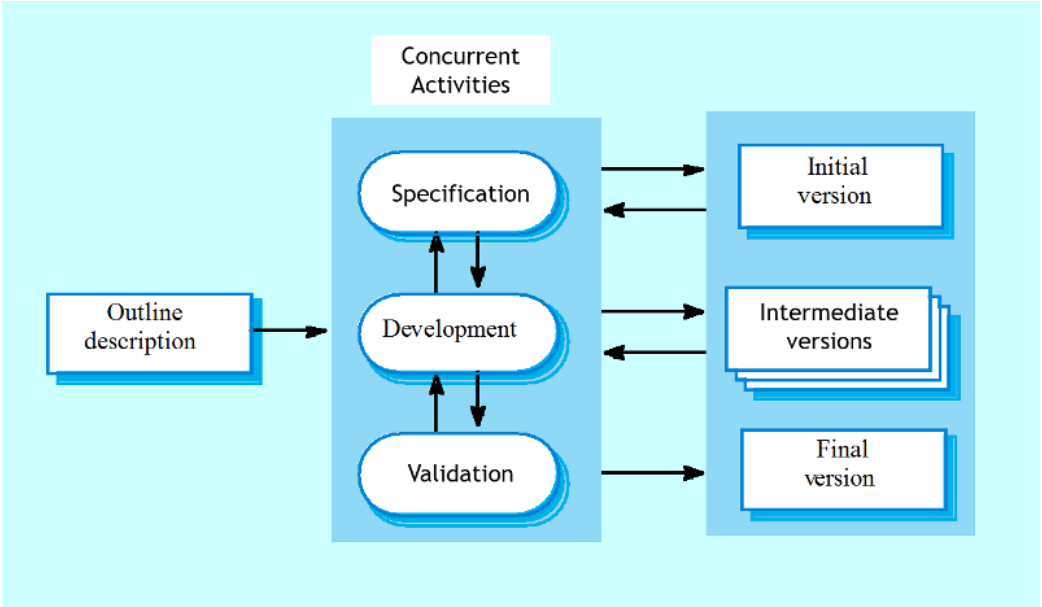
Allows exploratory development

- Can work with the customer to evolve a final system design from an initial specification
- Need to start with well-understood requirements and add new features as proposed by the customer

Can use throw-away prototyping

- Prototypes can be used if the customer isn't sure of what they want (which happens quite often)
- Objective is to understand the system requirements
- In this case you start with poorly understood requirements and iterate on prototypes to clarify what is really needed

Note that the activities in the diagram can be done in parallel



Evolutionary Development

Problems:

- Lack of process visibility
- Systems are often poorly structured
- Special skills (e.g. in languages for rapid prototyping) may be required

Applicability:

- For small or medium-size interactive systems
- For parts of large systems (e.g. the user interface, or database API)
- For short-lifetime systems

Comment on Throw-Away Prototypes

- Far too often the “throw-away” part is forgotten
- Once the prototype is working, budget and business needs may say:
  - “Just use that!”
  - No matter how bad the hastily thrown-together code is

Quote regarding software engineering:

- “Design it from the top down; then build it from the bottom up. Once that is done – throw it all out and do it again. This time correctly.”
  - Former coworker at Motorola, Inc.

Component-Based Software Engineering

Based on systematic reuse of existing components

- Open source or commercially available

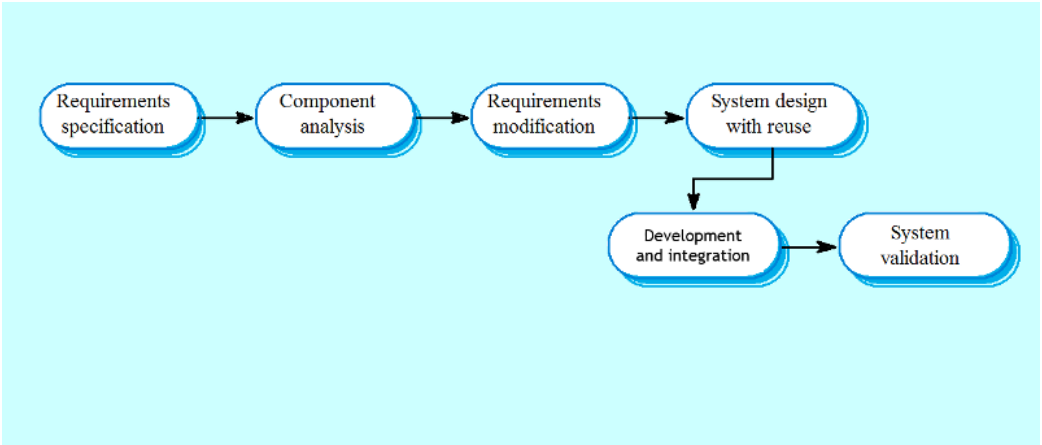
Process Stages:

- Component analysis
- Requirements modification
- System design with reuse
- Development and integration

This approach is becoming increasingly used as component standards have emerged

- Web applications can use dozens of modules or plug-ins to provide functionality

If requirements change, components can be added or changed



Incremental Delivery

With this model, development and delivery is broken into ***increments***

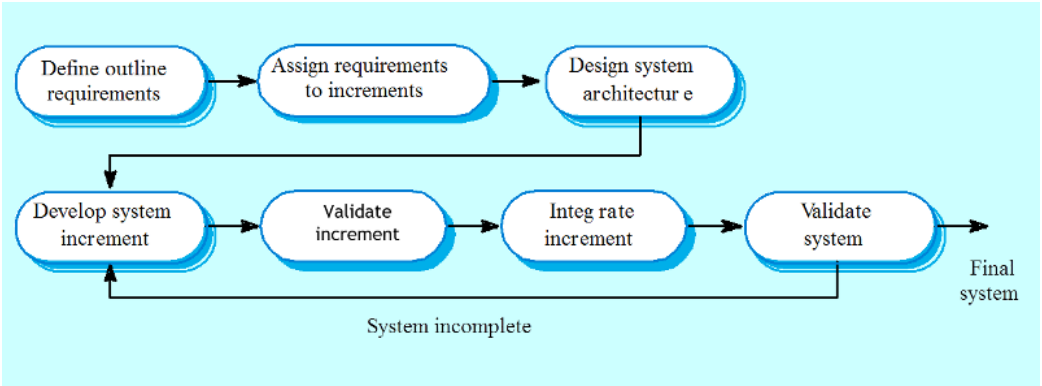
- Each increment delivers part of the required functionality
- Requirements are prioritized – and the highest priority ones are delivered first
- Requirements are frozen for each increment, but can then be changed

This is different than the previous methods

- Notice there is a loop for each increment
- This continues until the system fully developed

There are multiple advantages to this approach:

- Value can be delivered with each increment – producing benefits faster
- Early increments act as a prototype to inform requirements, design, and implementation of later increments
- Lower risk of overall project failure
- The highest priority system services tend to receive the most testing since they were in the early increments



Spiral Model

First described by Barry W. Boehm – primary paper:

- B. W. Boehm, "A spiral model of software development and enhancement," in *Computer*, vol. 21, no. 5, pp. 61-72, May 1988, doi: 10.1109/2.59.
- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=59

Read: Wikipedia article on the Spiral Model

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

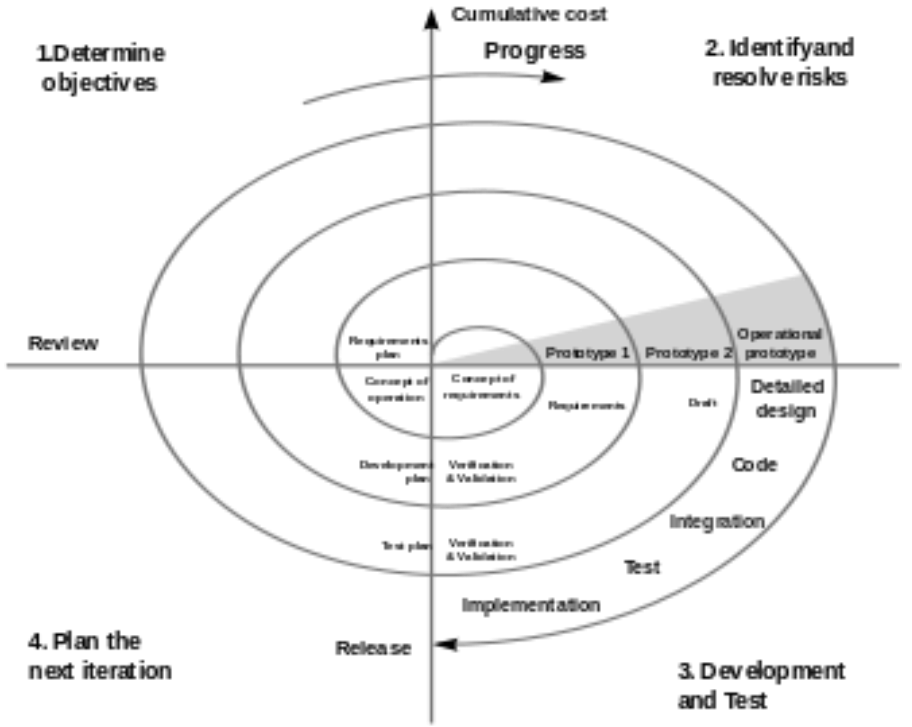


Image from Wikipedia:
https://en.wikipedia.org/wiki/Spiral_model#/media/File:Spiral_model_(Boehm,_1988).svg

Spiral Model

The four phases are sometimes labelled:

- Planning (Determine Objectives)
- Risk Analysis (Identify and Resolve Risk)
- Engineering (Development and Test)
- Evaluation (Plan the Next Iteration)

Each spiral is made up of these four phases

Other points about the Spiral Model:

- The process is represented as a spiral rather than as a sequence of activities with backtracking
- Each loop in the spiral represents an iteration of the process
- There are no fixed phases such as specification or design - loops in the spiral are chosen depending on what is required
- Risks are explicitly assessed and resolved throughout the process

Advantages of the Spiral model:

- High amount of risk analysis hence, avoidance of risk is enhanced
- Good for large and mission-critical projects
- Strong approval and documentation control
- Additional Functionality can be added at a later date
- Software is produced early in the software life cycle

Disadvantages of the Spiral model:

- Can be a costly model to use
- Risk analysis requires highly specific expertise
- Project's success is highly dependent on the risk analysis phase
- Doesn't work well for smaller projects

Agile Software Development

Published in 2001, the Agile Manifesto that gave us a different paradigm in software development

It states that it values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan
- That is, while there is value in the items on the right, we value the items on the left more.

This manifesto, and the 17 signatories, can be found at:

- http://agilemanifesto.org/

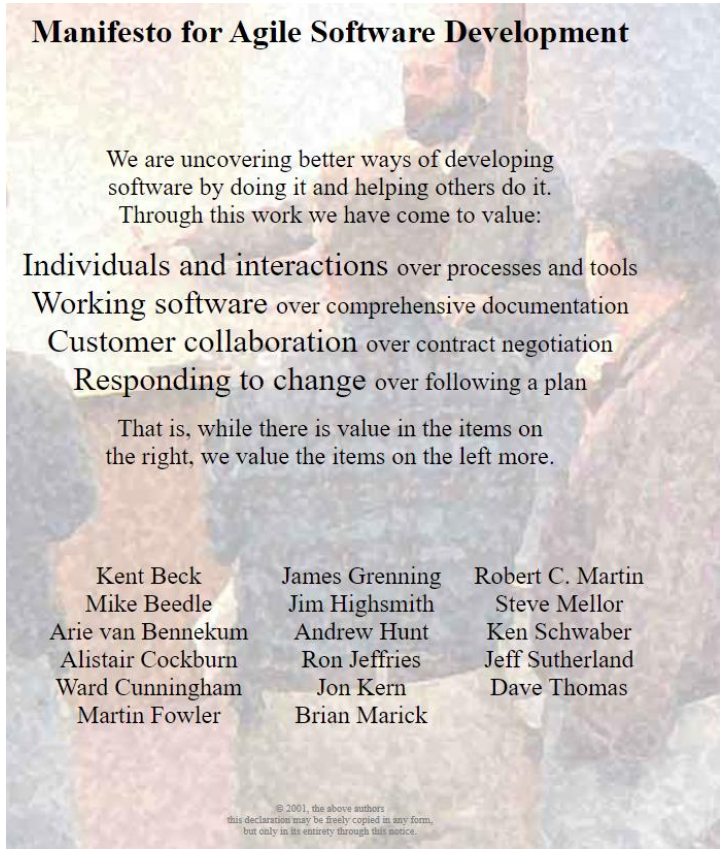


Image taken from: http://agilemanifesto.org/

Agile Software Development

The Agile Manifesto defines 12 principles that underly the Agile philosophy

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
7. Working software is the primary measure of progress
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility
10. Simplicity--the art of maximizing the amount of work not done--is essential
11. The best architectures, requirements, and designs emerge from self-organizing teams
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

Available at: http://agilemanifesto.org/principles.html

Further reading: https://www.agilealliance.org/agile101/

Video: *How the agile methodology really works*: https://www.youtube.com/watch?v=1iccpf2eN1Q

Requirements Engineering

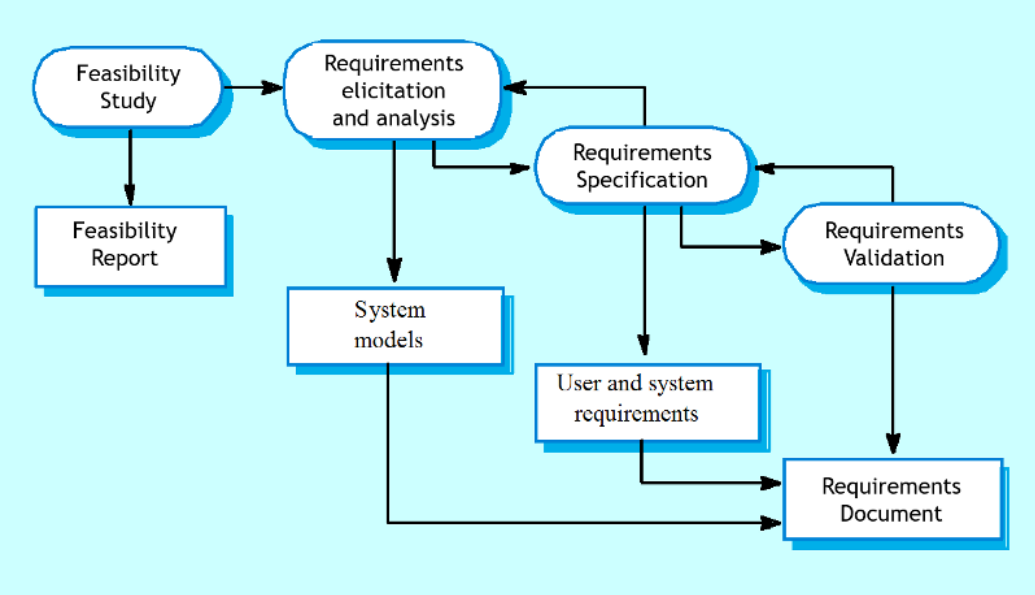
Another approach is to spend time engineering the requirements before design or development. This establishes:

- What services are required
- Constraints on the system's operation and development

Steps of the requirements engineering process:

- Feasibility study
- Requirements elicitation and analysis
- Requirements specification
- Requirements validation

Note: The output is a requirements document and not the code (or system) itself



Software Design and Implementation

Once a (correct) specification and requirements list has been developed, the process of creating a running system consists of two phases:

- ***Software design***: Design a software structure that realizes the specification
- ***Implementation***: Translate the designed structure into an executable program

These two activities are closely related and may be interleaved

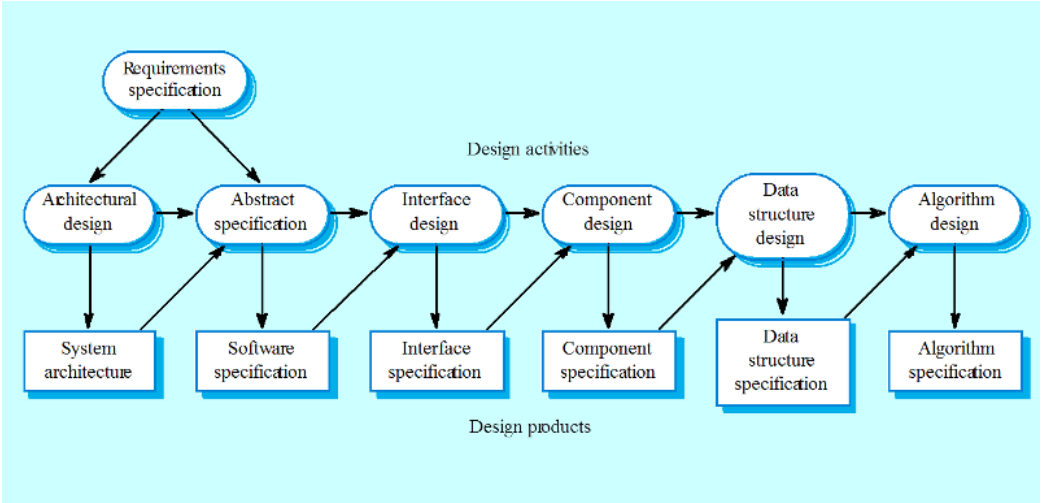
Design Process

There are many different approaches (and associated activities) for the design process

We will define the process as consisting of the following phases:

- Architectural design
- Abstract specification
- Interface design
- Component design
- Data structure design
- Algorithm design

Notice the outputs of each phase are used as inputs to the next phase



Programming and Debugging

Once the design is complete, the next phase is programming (implementation)

- This consists of translating a design into (working) code – and removing errors from that program (debugging)
- Generally, the programming is done as a personal activity without strict processes
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process

The steps to debugging are generally:

- Locate Error
- Design Error Repair
- Repair Error
- Re-test Program

Debugging also assumes that you are able to identify an error and locate what is causing it

- This is not always true – particularly with intermittent errors
- “Closed – Cannot Reproduce” is used often with these types of bugs

Software Validation

Verification and validation (V&V) is another critical step in developing a system

V&V is intended to show that a system conforms to its specification and meets the requirements of the system customer

Involves checking and review processes and system testing

System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system

The Testing Process

Testing Stages:

Component or unit testing

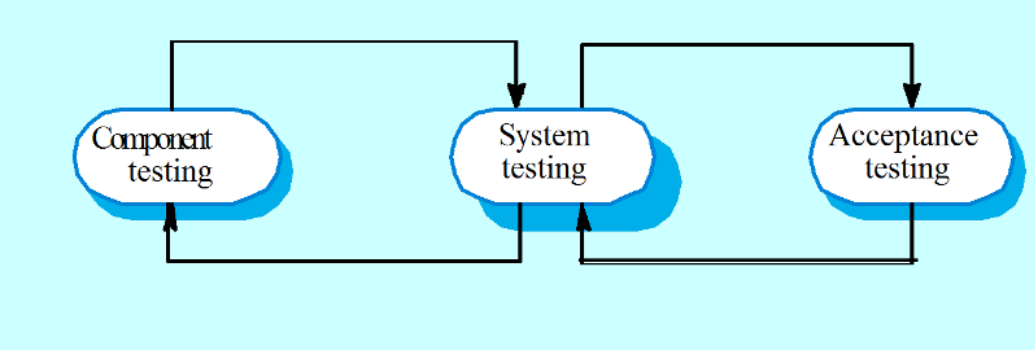
- Individual components are tested independently
- Components may be functions or objects or coherent groupings of these entities

System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

Acceptance testing

- Testing with customer data to check that the system meets the customer's needs.



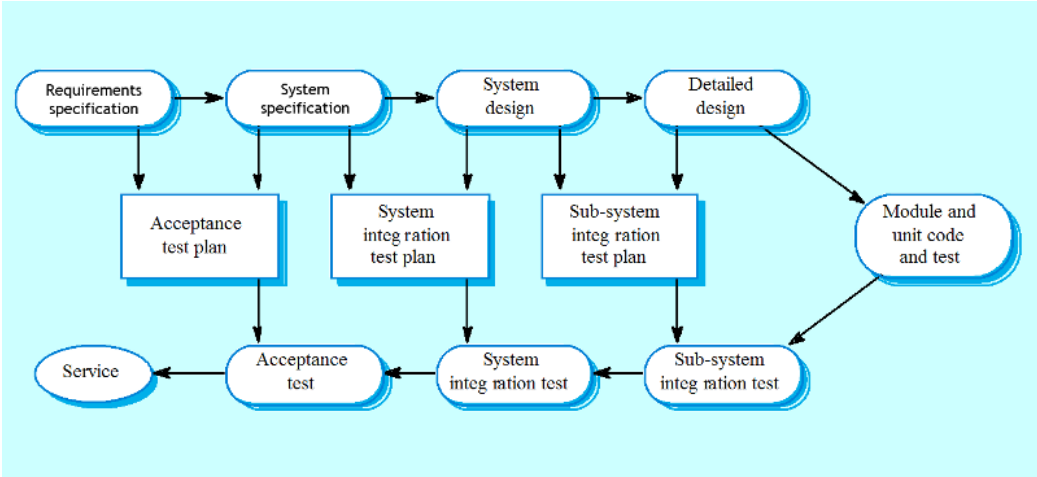
The Testing Process

The three different testing phases may be iterative

- Individual components will be unit tested and then assembled into a full system
- If there is a failure in the system testing, fixes will be applied and unit tests will be re-run

Once System Tests are passed, the system is given to the customer for acceptance testing

- This testing uses customer conditions and data to ensure proper performance
- It will also verify that it meets the customer's needs
- This is why requirements are so important
  - If it doesn't meet customer needs, but meets the documented requirements, the customer has a problem
  - If it doesn't meet the documented requirements, the developer has a problem



Testing Not Always the Prime Concern

Testing is a very important part of software development

It is often treated as an “add on” and not treated as an “equal partner” in the overall development process

- Meta’s Old Motto: “Move fast and break things”
- https://www.businessinsider.com/meta-mark-zuckerberg-new-values-move-fast-and-break-things-2022-2

If you rush to get a product out the door, it is unlikely to result in secure code

- Facebook has had a number of privacy breaches over the years
- Details on a leak that revealed details about 533 million users:
  - https://www.bbc.com/news/technology-56815478

Software Testing

Testing software consists of running a battery of test cases using multiple techniques against a specific use case and evaluating the results for pass or fail marks. The testing phase of any application is a major phase of the life cycle and should be treated as important as the development itself by the project management team. As a developer, you will play a key role in helping the management team create, execute, and evaluate test cases. Therefore, getting a general idea of what secure testing is and how to administer it will help you contribute more to the success of the project.

Testing vs. Assurance

Software testing is the discipline of examining the functionality of application software to determine whether it meets the requirements (verification) and satisfies the need (validation). Software Assurance (SwA) is how effective security is tested and implemented within the software.

During the software assurance process, the development team members must demonstrate their competency with software security testing through the use of artifacts, reports, and tools. It is in this phase that the design and code for each of the misuse cases created in requirements will be executed and thoroughly tested. Testing for both functionality and security requires the execution of the code and validation/verification of the results.

Reference: Jason Grembi, *Secure Software Devleopment: A Security Programmer's Guide*, Cengage Learning, May 2008.

Software Testing

Security testing is different

- Usually testing is accomplished in order to verify that some feature works properly
- If the feature doesn't perform as its specifications outline it is considered a 'bug' and needs to be addressed
- Security testing is different in that it generally is about trying to prove that the program's defensive mechanisms work correctly

"Good security testers are also good testers who understand and implement important testing principles. Security testing, like all other testing, is by its nature subject to the tester's experience, expertise, and creativity. Good security testers exhibit all three traits in abundance."

- Michael Howard in *Writing Secure Code*, 2ed

"A study by Gartner, IBM, and The National Institute of standards and Technology (NIST) revealed that 'the cost of removing an application security vulnerability during the design/development phase ranges from 30-60 times less than if removed during production.'"

- From: *Secure and Resilient Software Development* by Merkow and Raghavan
- **It is important to catch problems early!**

Testing Interfaces

For security testing, a major part of any program to test are the interfaces – especially those that involve user inputs

There is a 5 step process for testing of interfaces:

- Decompose the application into its fundamental components
- Identify the component interfaces
- Rank the interfaces by potential vulnerability
- Ascertain the data structures used by each interface
- Find security problems by injecting mutated data (fuzzing)

Testing Interfaces

The last step is one of the most interesting types of testing that takes place

Data Mutation – often referred to as “Fuzzing” – involves sending unexpected data to get the code to act in an insecure manner

- What happens when the code assumes something about the data that isn’t true?
- Sometimes this behavior isn’t defined at all – and we end up with vulnerabilities like those that allow buffer overflow attacks

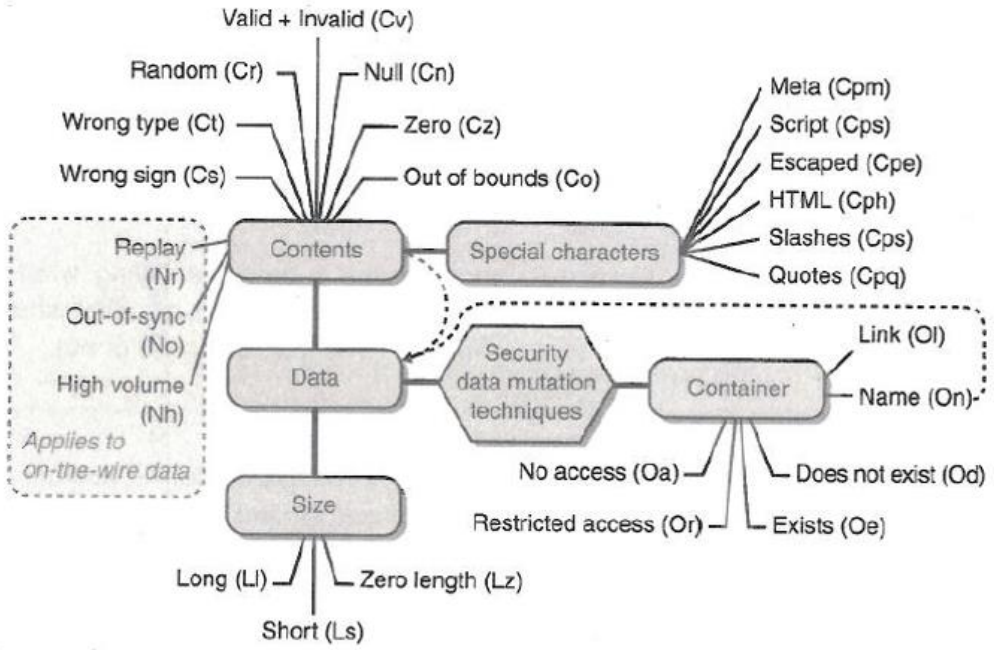


Figure 19-1 Techniques to perturb applications to reveal security vulnerabilities and reliability bugs.

From: "Writing Secure Code, 2ed" by Howard and Leblanc

When A Bug Is Found

When a bug is found, the tester needs to test variations to see if there are deeper problems

These are the general steps to use:

- *Reduce the Attack*: Determine what the 'base' exploit is and what is the simplest form of the problem
- *Identify fundamental exploit variables*: See what parts of the exploit can be modified to create other possible variations
- *Identify possible meaningfully distinct variable values*: Important values in an exploit are often little-used or under-documented
- *Test the full matrix of variables and variable values*: This may not always be possible, but the ones most likely to have issues should be tested

Fuzzing

OWASP defines Fuzzing as:

- *…a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.*

Read: OWASP page on Fuzzing here:

- https://owasp.org/www-community/Fuzzing

Three Parts of Fuzz Testing

- The input is random. We do not use any model of program behavior, application type, or system description. This is sometimes called black box testing. In the command-line studies (1990, 1995, and 2006), the random input was simply random ASCII character streams. For our X-Window study (1995), Windows NT study (2000), and Mac OS X study (2006), the random input included cases that had only valid keyboard and mouse events.
- Our reliability criteria is simple: if the application crashes or hangs, it is considered to fail the test, otherwise it passes. Note that the application does not have to respond in a sensible manner to the input, and it can even quietly exit.
- As a result of the first two characteristics, fuzz testing can be automated to a high degree and results can be compared across applications, operating systems, and vendors.
- Barton Miller at UW Madison: https://pages.cs.wisc.edu/~bart/fuzz/fuzz.html

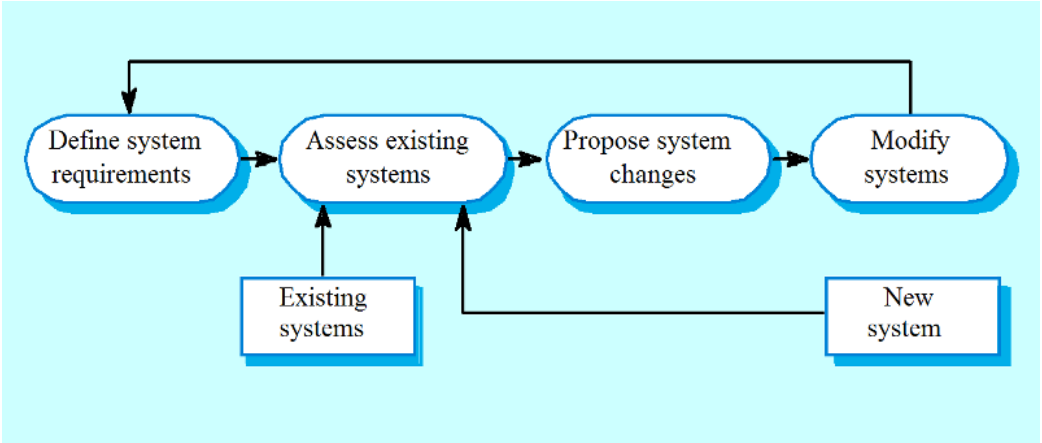
Software Evolution

Software is inherently flexible and can change – sometimes rapidly

As requirements change through changing business circumstances, the software that supports the business must also evolve and change

Traditionally, there has been a demarcation between development and evolution (maintenance)

- This becomes less true with the constant integration of new features
- Also blurred by porting software to different platforms and frameworks
- Is it evolution, or something new?



Principle of Least Astonishment

The issue:

- Users expect their software to be easy-to-use, and similar to other applications that they use
- Interfaces should have familiar buttons, controls, etc.
- Touchscreens should have standard functionality like scrolling, zooming, and interacting
- Web browsers should have an address bar to enter desired websites
- These interfaces abstract incredibly complex underpinnings of memory, processor, communication, and display management
- Users ***want*** abstraction – as long as it is consistent

Definition of Least Astonishment (Wikipedia):

- It [Least Astonishment] proposes that a component of a system should behave in a way that most users will expect it to behave. The behavior should not astonish or surprise users.
- Also: People are part of the system. The design should match the user's experience, expectations, and mental models.
- https://en.wikipedia.org/wiki/Principle_of_least_astonishment

Principle of Least Astonishment

In general engineering design contexts, it means components should behave in a consistent manner

- A car horn is located in the middle of the steering wheel, and is activated by pressing
- A turn signal is on the left side of the steering column and up is “right” and down is “left”
- The hands of a clock turn clockwise – and the numbers increase in a clockwise direction

This same principle applies equally to security features of an application or system

Principle of Least Astonishment

This principle also aims to minimize the learning curve for users

- New systems and applications can leverage pre-existing knowledge
- For example, on Windows:
  - Ctrl-S is Save
  - Ctrl-C is Copy
  - Ctrl-V is Paste

In *The Art of Computer Programming*, Eric Steven Raymond says:

> *The Rule of Least Surprise is a general principle in the design of all kinds of interfaces, not just software: "Do the least surprising thing". It's a consequence of the fact that human beings can only pay attention to one thing at one time (see The Humane Interface [Raskin]). Surprises in the interface focus that single locus of attention on the interface, rather than on the task where it belongs.*
>
> *Thus, to design usable interfaces, it's best when possible not to design an entire new interface model. Novelty is a barrier to entry; it puts a learning burden on the user, so minimize it. Instead, think carefully about the experience and knowledge of your user base. Try to find functional similarities between your program and programs they are likely to already know about. Then mimic the relevant parts of the existing interfaces.*

- http://www.catb.org/esr/writings/taoup/html/ch11s01.html
- ***The above paragraphs are important!***

Principle of Least Astonishment

Saltzer and Schroeder:

- Psychological acceptability – users won't specify protections correctly if the specification style doesn't make sense to them

Carnegie Mellon University via CISA:

- Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.
- Slide #33
- https://niccs.cisa.gov/sites/default/files/documents/pdf/intro%20to%20assured%20sw%20eng_classpresentations.pdf

Principle of Least Astonishment

Unfortunately, people can be trained to respond to certain things

Windows Vista had the User Account Protection (UAP) feature

- Constantly alerted the user to operations – even standard ones
- There were so many of them, users developed the reflex to click “OK” and not even read the warnings any more

More information: https://blog.codinghorror.com/windows-vista-security-through-endless-warning-dialogs/

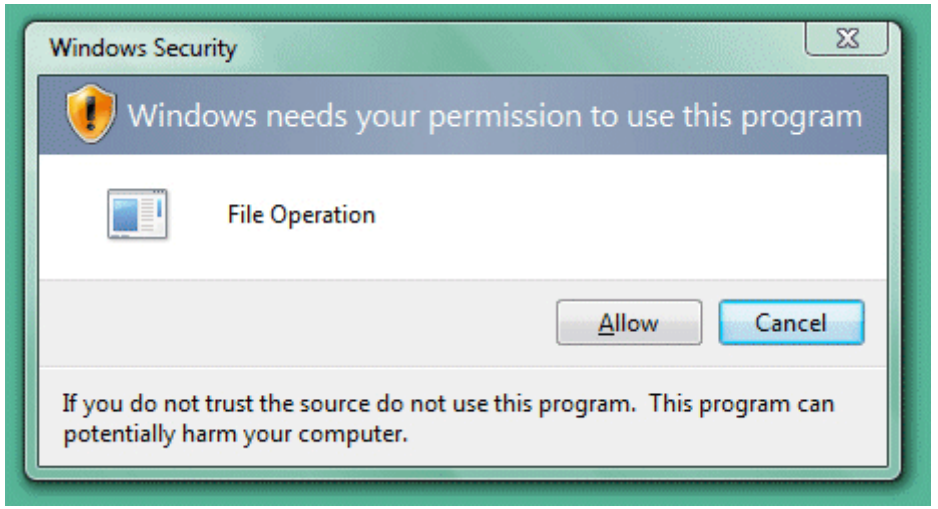


Image from Coding Horror, https://blog.codinghorror.com/windows-vista-security-through-endless-warning-dialogs/

Principle of Least Astonishment

The Windows Vista example illustrates an old adage:

- If a protection mechanism is easy to use, it is unlikely to be avoided
- If it is annoying, the opposite is true

Remember: Change (or a security feature) isn't always accepted

Image from XKCD, https://xkcd.com/1172/

Principle of Least Astonishment

Advice From Microsoft:

Effective error messages inform users that a problem occurred, explain why it happened, and provide a solution so users can fix the problem. Users should either perform an action or change their behavior as the result of an error message.

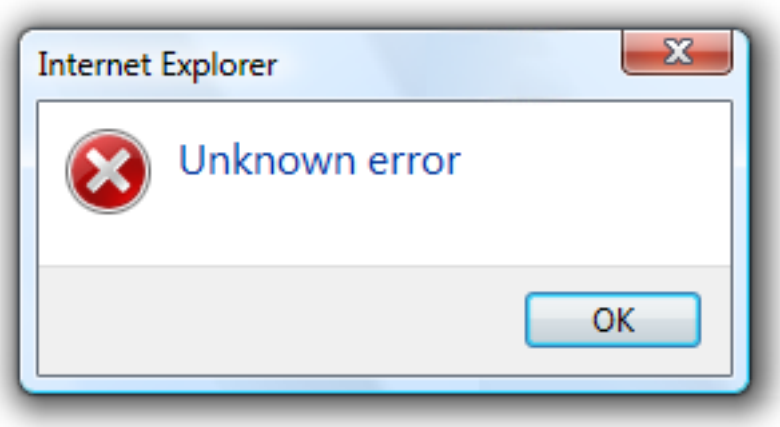
Well-written, helpful error messages are crucial to a quality user experience. Poorly written error messages result in low product satisfaction, and are a leading cause of avoidable technical support costs. Unnecessary error messages break users' flow.

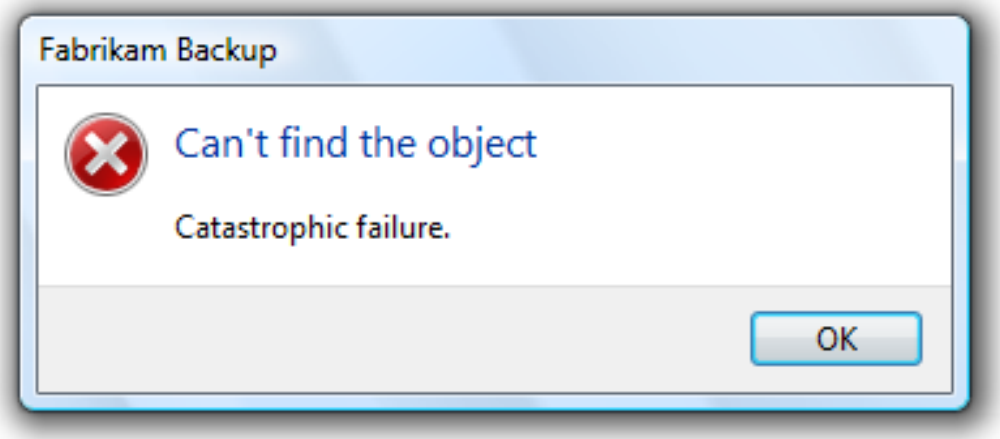
- https://docs.microsoft.com/en-us/windows/win32/uxguide/mess-error?redirectedfrom=MSDN

Unfortunately, even with this advice, there are many annoying, unhelpful, and poorly written error messages.

- When presented as modal dialogs, they also interrupt the user's current activity
- Part of the problem is that there are so many ways to do it wrong.

Principle of Least Astonishment





Images from Microsoft, https://docs.microsoft.com/en-us/windows/win32/uxguide/mess-error?redirectedfrom=MSDN

Functionality vs. Assurance

One of the most challenging issues in information security:

- Functionality vs. Assurance
- To illustrate this, refer to your own first-hand experience with computers
  - How many times has a computer failed to do something that you expected of it? (Load a Social Media page, connect to the Wi-Fi)
  - How many times did it do something you didn't want it to do? (Follow a link unexpectedly, apply an update and restart)
  - This difference between our expectations (as well as the vendors' advertising of product features) and what happens in fact that is referred to as functionality versus assurance.

A system may *claim* to implement a dozen security features

- However this has to be verified
- During an assessment, it was found that a network equipment vendor had a setting to “Prevent Data Exfiltration via DNS Requests”
- Enabling this feature did nothing

Another way of looking at the functionality versus assurance issue is that functionality is about what a system can do and assurance is about what a system will not do

Closing

Design for Iteration: You won't get the ideal system the first time

Everyone must be on guard to make sure that the overall design rationale remains clear despite changes made during iterations

Least Astonishment: A component of a system should behave in a manner consistent with how users of that component are likely to expect it to behave

Least Surprise: Behavior should be that which will least surprise the user

Psychological Acceptability: Users won't specify protections correctly if the specification style doesn't make sense to them

Functionality versus Assurance: The difference between our expectations (as well as the vendors' advertising of product features) and what happens