Design Principles

8/27/07

These principles concern how to design software and computing systems that are dependable, reliable, usable, safe, and secure (DRUSS).

Design principles are conventions for planning and building correct, fast, fault tolerant, and fit software systems.

A. Design means two things: architecture and process. Architecture is a division of a system into components, their interactions, and their layout. Process is the steps producing an architecture.

B. Design process is adapted from engineering: (1) requirements, (2) specifications, (3) prototype, and (4) testing. The outcome of testing is either an acceptable system or feedback to repeat a previous step. The process can be repeated incrementally and iteratively until testing is satisfactory. The process produces a series of increasingly concrete representations of the system, each in its own language.

C. Four main criteria for good design are correctness, speed, fault tolerance, and fit.

D. Correctness means that the software system provably meets precise specifications. Correctness is challenging because of the difficulty of getting precise specifications for complex systems and the computational intractability of formal proofs for large systems.

E. Speed means that the system completes tasks within acceptable time limits. This can be challenging because system speed depends not only on the running times of tasks, but on the number of other tasks, bottlenecks, and uncertainty of the demands of other tasks for resources.

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

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

H. Design principles are not laws of nature, but conventions that have been found to lead software developers to the most success when pursuing correctness, speed, fault tolerance, and fitness objectives.

I. A software designer's success depends ultimately on two things: tools and intellectual grasp. Tools are various software that assist the design process: languages, editors, debuggers, testers, graphical interfaces, version control systems, and the like. Over time, tools improve. The real challenge is intellectual grasp of the problem and how a computational system might solve it. Our design principles help organize our thoughts and actions to aid as much as possible with the intellectual side of every design problem.

Error confinement and recovery are much harder in the virtual worlds of software than in the real world of physical objects.

A. An error is an item of data that, when processed by the normal algorithms of a system, produces behavior not authorized by the specifications. Errors can be input or output of a program.

B. A fault is a defect or mistake in an algorithm, which when executed produces an error. Faults can produce errors that are not detected for some time after the fault is exercised.

C. Error tolerance means that the actual behavior under an error does not deviate far from normal behavior. Many physical systems follow continuum laws that guarantee that a small change in one variable produces small corresponding changes in other (dependent) variables; thus the system can naturally tolerate a small error. Many biological systems are highly error tolerant: they contain self-repair mechanisms that respond to errors through feedback and correct them. Many animals contain immune systems, an error recovery system that detects and destroys intruding organisms.

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

E. Error confinement means to limit the number of objects that an error can influence before it is detected. The best possible error confinement limits the error to the object in which it was created. Error recovery means to take an action that removes the error and restores the data to an error-free condition. Error recovery is easier when errors are confined.

F. Mechanisms for error confinement are of two kinds: (a) dynamic checks in the software itself, and (b) dynamic checks in the host environment in which the software runs. In-software checks are implemented as various tests whether data are staying within allowed ranges; these checks can generate substantial overhead and can be subverted by other errors in the software. External checks are performed independently and in parallel by the environment, do not generate overhead for the software, and cannot be subverted by the software. Operating systems, for example, place individual user programs into memory partitions defined by hardware that is inaccessible to the user's software; errors generated by the software are confined to the partition and cannot affect any other program.

G. The *Principle of Least Privilege* supports error confinement by making the default access for any process to be the smallest set of objects possible.

H. Mechanisms for error recovery are partly in the host environment and partly in the software itself. Operating systems, for example, contain programs that scan and fix errors in directory entries; these programs are invoked when a user program encounters a directory lookup error.

I. Software designers cannot rely solely on error confinement and recovery built in to the host environment. They must also design to minimize the chances of error and maximize the tolerance to error, and they must provide means for error recovery. Even the best designs will need help from the host environment because software usually contains faults that the designer did not know were present.

The four base principles of software design are hierarchical aggregation, levels, virtual machines, and objects.

A. Software design principles organize the intellectual approach to expedite a correct, fast, fault tolerant, and fit software system.

B. Hierarchical aggregation means that objects (components) consist of interconnected groups of smaller objects and are themselves components of larger objects. The principles of abstraction, information hiding, and decomposition all flow from this one.

C. Levels means to organize the functions of a system into a hierarchy with the constraint that the higher-up functions can only call lower-down functions.

D. Virtual machines means to create simulations of logical computing machines and use them to solve the problem. Virtual machines can be hierarchically nested.

E. Objects means to encapsulate a data structure and its applicable functions in a single package.

Abstraction, information hiding, and decomposition are complementary aspects of modularity.

A. Modularity is a process of dividing a large system into a hierarchy of aggregates (modules) that interact across precisely defined interfaces.

B. Abstraction means to define a simplified version of something and to state the operations (functions) that apply to it. By bringing out the essence and suppressing detail, an abstraction offers a simple set of operations that apply to all the cases. In a hierarchy, an abstraction corresponds to an aggregate; forming a hierarchy is a process of abstraction. Abstraction is one of the most fundamental powers of the human brain. Abstractions in classical science are mostly explanatory -- they define fundamental laws and describe how things work. Computer science abstractions do more: they define computational objects, and they perform actions. For example, the bit (0 or 1) is an abstraction of all sorts of media that rely on two states to store or transmit information -- pits and peaks on a CD, magnetized patches on a disk, voltages on a wire.

C. A file (a named sequence of bits) is a common abstraction representing text documents, graphs, spreadsheets, images, movies, sounds, directories, and more. A file system provides create, delete, open, close, read, and write operations that work on any file. Any program's output (already represented as bits) can be stored in a file. The file system does not have to understand the differences between file formats assigned by applications -- it just stores and retrieves the bits.

D. Information hiding means to hide the details of an implementation so that users do not see them. It protects against errors caused by changes in the details that do not concern users. It is a policy that supports abstraction by preventing users of the abstraction from gaining access to the suppressed details behind the abstraction. In a hierarchy, it is a decision to hide the component structure of an aggregate, allowing that structure to be rearranged without changing the behavior of the aggregate. A software module implements a software function by hiding internal details behind a simple interface.

E. Information hiding prevents users of a file system from seeing disks, disk drivers, records, index tables, disk addresses, buffers, caches, open file control blocks, and RAM copies of files. This benefits the user in two ways: (1) the user never makes assumptions about the details,

simplifying the user's task, and (2) the maintainer of the abstraction can change or improve any of the details without forcing any user to change anything.

F. It's possible to have abstraction without information hiding. An organizational hierarchy, for example, is a set of abstractions that group people by functions. Executives can micromanage a function by looking inside it.

G. Decomposition means to subdivide a large problem into components that can be designed separately and then assembled into the full system. In a hierarchy, identifying the components of an aggregate is an act of decomposition. A module is an abstraction of the components that compose it.

H. Designing by decomposing a system into modules and interfaces is often insufficient in large systems. When the independently designed modules are plugged together, the system often does not work even though the module designers insist that the individual modules meet specifications. The problem is that the interactions among the modules are as important as the internal computations of the modules; independent module designers do not see or test the interactions. Considerable testing of the system as a whole is needed, leading to redesign of modules until the entire system works.

Levels organize the functions of a system into hierarchies that allow downward invocations and upward replies.

A. The levels principle is derived from a traditional science principle to explain complex systems by dividing into levels that can be understood in terms of their own abstractions. For example, physical materials can be seen as assemblies of molecules, molecules as assemblies of atoms, atoms as assemblies of electrons and protons, and so on.

B. In computing, levels means to organize the functions of a large system into a hierarchy. Functions in a given level are components (decompositions) of a function at a higher level. Functions in a given level may invoke functions in a lower level, but never a function in a higher level. Therefore the designer of a function can use lower level functions as part of the implementation, but can make no assumptions about functions at higher levels.

C. This structure simplifies the proof of correctness, which can be considered one level at a time. The lowest level, which does not call lower-level functions, is proved first. The next level is proved by demonstrating its functions are correct given that the lower level functions are.

D. The levels structure also facilitates testing. The lowest level is tested first. Then the next level, which depends on the correct function only of the lowest level, is tested. This continues until all levels are tested.

E. The earliest systematic use of levels was in the THE operating system (Dijkstra, 1968). The principle has been used in many other operating systems and has led to significant reductions in kernel size. The principle was adopted into the Internet model (OSI model, protocol stack), where it allows the designer of a level to pretend that he is interacting with the software of the same level on another machine.

F. The levels structure can improve the stability of a modules-interfaces design by imposing system-wide constraints on information flows.

Virtual machines organize software as simulations of computing machines.

A. A virtual machine is a simulated computer. It has its own memory, processing, instruction set, and input-output mechanisms. Virtual machines are used to organize sets of modules into working subsystems. An example is a virtual disk, which consists of a disk unit, an interface for submitting read and write disk requests, and an internal process that monitors its input queue and the disk.

B. The term virtual machines is used in three main ways. One is as a complete simulation of the host computer, but with less memory. Examples of virtual machines are: (1) The IBM Virtual Machine operating system; each user process is assigned its own virtual machine. Because the simulation completely isolates the virtual machines, a crash or error in any one will not affect any others, leading to very stable operating systems. (2) The Intel x86 series gives each process its own virtual machine. (3) The Java compiler translates Java programs into “bytecode”, the instruction set of the Java Virtual Machine (JVM). A JVM is implemented for each computing platform, using the instruction sets of that platform. This allows portability of Java programs to many machines and platforms.

C. The second meaning of virtual machine is the simulation of one computer’s instruction set on another. Known also as emulation, it was used in many early mainframe computers so that the next generation computer could simulate the instruction set of the previous generation and allow old programs to run without recompilation. Emulation is used in some modern computers to simulate one machine on another, for example, Microsoft’s Virtual PC for simulating Windows on Apple computers. In the simulation, each instruction of the

simulated machine is implemented as a subroutine on the simulating machine; thus the simulated version runs slower than the real version.

D. The third meaning of virtual machine is any program in execution in a standard environment. An example is the Unix system, which encapsulates any user program into a standard environment (the Unix Process), thereby providing a standard interface to interact with any executing program. The standard interface includes a standard input (IN) and standard output (OUT). All programs are written to read from IN and write into OUT. Then any programs can be piped together by connecting the OUT of the first to the IN of the second. Pipelining is a powerful way to create a complex function from existing parts.

Objects organize software into networks of shared entities that activate operations in each other by exchanging signals.

A. A software object is an abstraction of a data entity packaged together with the set of functions that operate on it. The user of an object signals the desired function; the object performs that function, updates its internal state, and returns an answer to the user.

B. Objects are defined as members of a class of similar objects, usually called a type. All objects of the same type have the same operations. Types are defined in terms of higher-level types (hierarchies) so that an object can inherit properties from its immediately defining class and from all higher classes.

C. Objects have locks so that parallel processes can access them only one at a time (mutual exclusion).

D. Objects have global handles so that they can be shared among many users.

In a distributed system, it is more efficient to implement a function in the communicating applications than in the network itself (end-to-end principle).

A. This claim, known as the end-to-end principle, is simply that the network itself does not know all the requirements of the applications that connect to it; therefore any attempt to implement in the network a function used solely by the applications will be less efficient than implementing the function in the applications but not the network.

B. An example is in the TCP protocol for the Internet. This protocol ensures 100% reliable delivery of a file even though the network may lose some of the packets during any transmission. TCP is designed with acknowledgements and time-outs so that the sender can resend

missing packets. Only the sender knows the full file to be sent, and only the receiver knows which parts of the file have been received. Any attempts by the network to cut down on packet loss at a router or on a link do not absolve TCP from sending acknowledgments, and are thus redundant. The network will be more efficient by omitting them.