

# Software Design Principles and Practices

Kirk Kandt  
Jet Propulsion Laboratory  
ronald.k.kandt@jpl.nasa.gov

## Abstract

*Jet Propulsion Laboratory (JPL) has an on-going process improvement program that is assessed against ISO and CMMI criteria. To help JPL meet these criteria, a Software Quality Improvement project was recently established to define software requirements, guidelines, and processes consistent with these two standards and infuse this technology into the organization. In support of this task, the author identified hundreds of software development practices. This paper discusses those practices, and underlying principles, that address common design problems. In addition, some brief ideas are presented about the influence of good design practices on the design of programming languages and integrated development environments. Lastly, some hints are provided for creating improved design methodologies.*

## 1. Introduction

In 1999, JPL lost two spacecraft. As a result, NASA investigated these losses and reported on the possible causes of them. These reports and an on-going process improvement effort caused JPL to reexamine its management and engineering processes. For software, JPL levied new requirements and guidelines on projects and made it much more difficult to obtain waivers on requirements. Some of these new requirements and guidelines affected software development processes.

To help define these new software requirements and guidelines, the author identified hundreds of best practices. These practices span the entire software lifecycle, cover both management and technical processes, and have process- and product-orientation. This paper reports on general design practices of use across most types of software development activities. Another paper will deal with those practices specific to the development of flight software systems, which are autonomous, mission-critical, and real-time.

The identification of these software design practices was accomplished using several methods. First, the author identified those practices that he could draw upon by introspection. Many of these practices were not necessarily original (e.g., coupling and cohesion). Second, he reviewed some of the published literature (e.g., [12],

[14]). Third, he interviewed approximately 30 senior software personnel, of an estimated 1,100 software practitioners at JPL, to learn what they had to say about the subject. All of this information was synthesized and condensed to about 60 software design practices, of which 42 are reported herein.

These practices can be used in many ways. First, organizations can use these practices to train software engineers and evaluate their software development practices. Second, organizations can use them to define a core set of software design practices. Third, the identification of these practices can motivate others to explore opportunities for automating their use, enforcement, and verification. The remainder of this paper will describe these practices, as well as their underlying principles.

## 2. Principles

A software design principle is a comprehensive and fundamental doctrine or rule that governs the creation of quality software designs. The producer-based meta-principle is to maximize profit by reducing overall development and maintenance costs and producing artifacts having the features yielding the greatest number of sales. The customer-based meta-principle is to satisfy and delight the customer. To satisfy customers an organization must design software that meets their needs. To delight customers requires that an organization do something unexpected and beneficial for them. For example, integrated development environments that generate exhaustive statement or path coverage test suites for components most likely to have defects would delight their users. Several software design principles that elaborate these meta-principles follow. These principles are not exhaustive, but provide the underlying motivation for the software design practices that a later section identifies.

*Principle 1: Design quality software and measure the quality of software designs.* Quality reflects the essential character and inherent features and properties of artifacts, as well as their degree of excellence. Thus, the quality of a software design is a reflection of how well its features and properties support the users and developers needs. Quality is often associated with the number of defects that a software artifact has. However, this is just one measure

of quality. Quality is also related to many other design properties such as its reusability, understandability, flexibility, reliability, efficiency, and so on. The better a design, the more effective it will be to the developing organization and its customers.

*Principle 1.1: Design reliable software systems.* Customers should be able to rely on software to always work and yield the same results when given the same inputs. In many cases, human life is dependent on such software. For example, many devices that monitor the health and well being of people are dependent on software. In other cases, the cost of lost or damage caused by unreliable software is significant. For instance, the cost of deep space missions is in the hundreds of millions of dollars. An unreliable piece of code can cause a loss of an entire mission [4].

*Principle 1.2: Design flexible software systems.* Software systems should be capable of easily adapting to new, different, or changing requirements. In an idealized world, the problem that an organization is to solve and its problem domain would be well understood, leading to requirements that it or another organization completely and unambiguously defines once, and only once, before software design begins. Unfortunately, this fantasy rarely, if ever occurs. In the real world, user needs change, technology changes, personnel change, funding changes, and so on. Consequently, to develop quality systems on time and within budget that satisfies and delight customers requires that software designers preplan their designs for software product evolution. Such preplanning requires the development of flexible designs.

*Principle 1.3: Design understandable software systems.* Software engineers should be able to mentally grasp the meaning, purpose, character, and reasonableness of software systems. Thus, it is important to design software systems so that they are understandable. There are various ways of making software systems more understandable, such as using improved design and documentation methodologies.

*Principle 1.4: Design simple software systems.* One of the ways an organization produces software of high quality is by minimizing the complexity of its designs. Thus, software architects should only design and programmers code those features that a system needs to achieve the objectives of a project. Adding non-essential features simply creates greater risk. Since the enormous complexity associated with developing most modern software systems quickly stresses intellectual limitations of people, the best solutions are those created by people who tailor their solutions accordingly [6]. Regardless of the reason for overly complex designs and implementations, the cost per line of code and defect rate per line of code increase as software size increases [17].

*Principle 2: Plan software development.* Planning a software development activity helps an organization

efficiently and effectively use its human and technological resources.

*Principle 2.1: Efficiently utilize software development personnel.* To efficiently use software development personnel, an organization must assign personnel to tasks that have the appropriate skills and experience, provide software tools and a work environment that enhances their productivity, and schedule personnel to maximize their utilization. This requires planning.

*Principle 2.2: Reduce or eliminate the number of new, changed, and deleted requirements.* To reduce requirements creep and churn, an organization must manage the definition of requirements, assign work tasks based on the requirements and related requirement groupings, evaluate the effects of the addition or change of a requirement, and control the realization of such requirements. The reduction of requirements churn and creep helps to prevent or eliminate design changes, which has a positive effect on both product quality and development effectiveness.

*Principle 3: Validate critical design characteristics.* Critical aspects of a design can determine the success or failure of a software system. Therefore, an organization should validate vital design characteristics. These characteristics may involve the performance of an overall system or component, the suitability of a purchased product or a reused component, the technical feasibility of a framework, the usability of a user interface, etc. Common methods of validating a design include reviewing, simulating, and prototyping a design.

### 3. Practices

A software design practice is a customary action that a software designer repeatedly performs to proficiently derive quality software designs. Several software design practices follow and are grouped into several categories.

#### 3.1. Software Process Practices

*Practice 1: Define and use criteria and weightings for evaluating software design decisions.* Such criteria should minimally include the following qualities: simplicity, reliability, generality, efficiency, testability, modularity, portability, and understandability. Using the relative merit of these criteria, software managers can objectively estimate the quality of the final product, as well as intermediate artifacts and alternatives to them. One effective method of defining criteria and their weightings is Quality Function Deployment (QFD) [20]. Software tools are available that implement the QFD methodology.

*Practice 2: Create architectural specifications for software systems.* An architectural specification should identify many things.

- Security, reliability, availability, performance, safety, and interoperability requirements.
- All external interfaces, including descriptions of their source, format, structure, content, and method of support.
- All internal software interfaces, which should comply with applicable public, open, and interoperability standards.
- Static views of structures that include threads of collaboration.
- Dynamic views of structures that show temporal, concurrency, and synchronization behavior and include interactions in time sequences and sequences of states.
- Physical views of structures that describe the allocation of software to hardware.
- A reuse strategy that minimizes modifications and additions to the reused software.
- Standard components and component frameworks that the architecture uses.

Adherence to this practice will result in improved design realizations.

*Practice 3: Create appropriately sized specifications.* A specification provides a common point of reference for a project team. Providing a very brief specification will not enhance a shared understanding by the development team because it will leave too many issues unspecified or ill specified. An extremely detailed specification, on the other hand, indicates that the project team has forgotten that the application is the final product, not its specification. An overly detailed specification also breeds overconfidence, causing a team to poorly prepare for required changes and other unexpected events. This causes teams to ignore customer needs, resist changing specifications, and schedule projects without considering schedule slips. By creating right-sized specifications, an organization will bypass these problems and derive greater benefit.

*Practice 4: Identify fallback options during design conceptualization.* Since some decisions do not yield the expected results, it makes sense to consider and document alternatives. This is especially true when accepting high risks (e.g. using new technology). By having fallback positions, a project can assume greater risk because it has made contingency plans that permit quick recovery.

*Practice 5: Create working prototypes of components that implement key system features.* Since the sole purpose of a prototype is to validate a prior decision or design, the software engineering team generally ignores its defined software processes to reduce development time of the prototype. Because of this, software engineers rarely can cost effectively scale it into a workable solution. Consequently, a project team must constantly reinforce the purpose of a prototyping effort to its customers, as well as the limitations of the resulting prototype. In addition, an organization must follow an

effective quality assurance process that imposes various quality standards to prevent an organization from delivering prototypes to customers.

*Practice 6: Use tools to design software that enforce notation and semantic rules, especially ones that automate vital functions.* A key purpose of a design effort is to communicate a design to others – developers, testers, and customers – so that they will better understand how it is suppose to function. By using software tools, an organization can capture, syntactically check, and further manipulate its designs. It can also enforce syntactic and semantic rules that help to make the interpretation of a specification very precise, which avoids confusion regarding what a system is suppose to do among the entire team and how its modules will communicate.

*Practice 7: Use standard software components and component frameworks.* When software engineers reuse components and component frameworks, they improve their own productivity and enhance product reliability because they analyze, design, and implement less software. Another benefit of using components and component frameworks is that they create a language for discussing future needs and extensions of application domains. From this perspective, reusable components and component frameworks represent reusable design and implementation patterns.

### 3.2. Design Heuristics

*Practice 8: Defer decisions as long as possible.* Software architects should postpone decisions as long as possible because they will continuously increase their understanding of the problem domain and the design activity, which will enable them to make better decisions and create better designs later on [22].

*Practice 9: Produce models of problem domain concepts and interactions as a basis for software system architectures.* A software system should manipulate representations that model real-world objects, but generally at a reduced level of fidelity. The benefits of modeling a domain are several. First, software engineering personnel can use a domain model to check the specifications and requirements of a new system. Second, software organizations can use the domain model to educate people and provide them with a general understanding of the structure and operation of developed systems of the problem domain. Third, programmers can derive working systems directly from a specification described using the domain model. Fourth, domain modeling makes it easier to inject faults into software simulations, which eases debugging and test activities.

*Practice 10: Reduce large systems into modules whose size are about 5,000 lines of source code.* For most programmers, productivity rates drop off sharply when programs grow much larger than about 5,000 source lines

of code. More specifically, for each ten-fold increase in system size, programmer productivity decreases by roughly forty percent [12]. Thus, if software architects successfully decompose an application that is 500,000 source lines of code into one hundred components of 5,000 source lines of code, assuming no additional cost for partitioning, they will increase productivity of the programming staff by about seven hundred percent. Additionally, software engineering teams would increase their ability to adhere to schedules from about thirty-seven to eighty-six percent [12]. Thus, developing right-sized components, with well-defined interfaces is vital to the success of an organization.

*Practice 11: Assign responsibilities to software components with the goal of minimizing coupling.* More specifically, artifacts should be dependent on or interact with nine or less artifacts. By controlling the interactions among system components and global resources, system architects enhance their ability to reuse components and create adaptable systems. In addition, the more loosely coupled two routines or classes are, the less complex their relationship. This permits software to be more understandable.

*Practice 12: Assign responsibilities to software artifacts with the goal of maximizing cohesion.* Maximizing cohesion allows software architects to better manage complexity and others to better understand software systems. Artifacts having related responsibilities will form larger-grained modules with well-defined interfaces. In object-oriented programming languages, such schemes cause designers to assign highly cohesive sets of responsibilities to classes.

*Practice 13: Create components that use well-defined interfaces that encapsulate internal behavior.* Software architects should define interfaces using a module interconnection language or a similar mechanism [5]. Well-defined interfaces increase the ability of software personnel to understand and maintain the basic function of components and may be the most powerful design heuristic for reducing a program's complexity. Furthermore, software architects should use the principal of maximum information hiding to determine whether they should use stepwise refinement or layers of virtual machines to define software architectures [21].

*Practice 14: Use hierarchies and abstractions.* Two of the most effective general means of managing complexity are the use of hierarchies and abstractions. A hierarchy is a tiered, structured organization that divides a problem space into ordered and ranked levels, where a system handles different aspects of a problem at different levels. Abstraction is a more general concept than hierarchical organization because it can reduce complexity by spreading details across a loose network of components. The use of this practice increases programmer productivity, as long as the depth of hierarchies is 5 or

less. Consider, for example, that the largest single productivity gain made in software development resulted when programmers switched from machine language to higher-level languages [3]. In addition, hierarchies and abstractions reduce the total number of details that programmers must remember and make software systems easier to understand and test, leading to greater reuse achieved by separating common functionality.

*Practice 15: Define and use common protocols for common operations.* Using common protocols makes it easier for developers to understand and enhance a software system. Several common object-oriented design heuristics follow.

- When one class contains or aggregates another, it should create, initialize, and destroy the contained or aggregated class.
- When one class contains the data used to initialize another class, it should create, initialize, and destroy the initialized class.
- When one class closely uses another one, it should create, initialize, and destroy the used class.
- A controller class should handle system events. The class may represent an overall system, business, or organization or an animate domain object that performs the work.
- Distribute the responsibility of a behavior that varies by class – using polymorphic operations – to each class. Such *divide and conquer* strategies reduce larger problems into smaller, more manageable ones.
- When member functions manipulate data, the data should belong to the class.

The benefits of using these specific heuristics, and other common protocols, is that they reduce complexity by repeatedly using a core set of techniques for dealing with common kinds of problems, making it easier for software personnel to understand and develop systems.

*Practice 16: Design software to support the measurement of constrained resources, such as throughput and memory.* Since software should always operate in a timely manner, it should monitor its resource consumption to avoid exhausting a resource and not fulfilling a system need. Currently, programmers must write software to monitor resource consumption and others must verify that such software functions correctly.

*Practice 17: Use metrics that measure design quality.* Several people have defined various metrics over the years that measure functional (e.g., [10], [16]) and object (e.g., [11]) complexity. Numerous others have attempted to determine those metrics that are superior to others and how individual metrics correlate (e.g., [19]). These metrics, as well as several others, provide a good basis for distinguishing those components that have high quality from those that do not. Since low quality components have more defects and are harder to understand and adapt

to meet new demands than high quality ones, this practice should be of particular interest to an organization.

### 3.3. Data Storage Practices

*Practice 18: Categorize data that represent measurements with specific units of measurement.* It is not sufficient to represent 5.3 inches as a floating-point number. Instead, a non-primitive data type should represent a measurement so that a compiler can discover incompatible component interfaces during compilation instead of during execution or test. This is a specific analysis pattern [7], whose lack of use caused the Mars Climate Orbiter Mission failure [4].

*Practice 19: Specify a nominal value, precision, accuracy, and an allowable range for data elements.* This ensures that applications do not violate interface requirements. It also prevents invalid data that is out of range or beyond the representational capability of the hardware.

*Practice 20: Design applications to initialize all variables upon startup, including clocks, consistent with a global state.* When someone halts an application or it aborts itself, other components of the system may continue to run. Thus, when an operator restarts an application, it must synchronize itself with a shared global state, if it exists, and initialize local variables to accurately reflect the global state. This practice helps to produce reliable software since processes will share a common, global state.

*Practice 21: Avoid creating and using redundant data.* The problem with using redundant data is that a program must consistently maintain it. Unfortunately, it is easy for programmers to overlook this fact. However, the use of redundant data is often necessary to meet the performance requirements of a system. In these cases, programmers should use well-defined protocols (see Practice 15) for maintaining the consistency among the redundant data.

### 3.4. Data Input and Output Practices

*Practice 22: Design software systems to use all user and sensor inputs.* If a software system does not use all sensor and user input, it is likely that there is an omission in the requirements, design, or implementation. Therefore, an organization should never allow exceptions to this practice. If a system does not use an input, it should at least log it in some way so that others can monitor when an application is not using input data. Adherence to this practice helps to produce correct software that processes information in a complete and consistent manner.

*Practice 23: Design software systems to check critical inputs for correctness and proper timing.* Software architects must specify the minimum and maximum bounds of each variable and the expected time between

inputs. When an application receives an out-of-range or unexpected value or the time between arrivals is not as expected then the application should, at a minimum, record such defects. Adherence to this practice will increase the reliability of software systems and provide a basis for product improvement. It is possible to automate part of this practice. Compilers for strongly typed languages, for instance, fulfill some aspects of this practice.

*Practice 24: Ensure that the output absorption rate of a component equals or exceeds that of its input arrival rate for the longest interval for which input occurs.* To prevent long-term overload conditions, the output must be consumed at a faster rate than input is received. Even so, the ability to buffer both input and output should be considered as a precaution for preventing data loss. By ensuring that output absorption is at least as great as input arrival, a development team will enhance system reliability.

*Practice 25: Specify contingency actions to mitigate when components exceed their output absorption rate limits.* For the short-term, a program can buffer inputs and prevent sending excessive outputs. This, however, could result in a buffer overrun. Consequently, the system must perform contingency actions for this situation, as well as when the long-term output capacity is less than the long-term input capacity. If a system does not properly handle this type of failure, a catastrophic error can occur.

*Practice 26: Specify software behaviors regarding automatic update and deletion of information in data queues.* Establish and convey a policy regarding the modification of data queue information, which will enhance software predictability because programmers will consistently handle anomalies.

*Practice 27: Specify timeout conditions and recovery actions for all input and output streams.* By defining timeout conditions and recovery actions, systems can continue to function in the absence of input and output data. Adhering to this practice improves software safety and reliability. Encapsulating this practice as a software component greatly eases the burden placed on programmers since they would no longer need to create event loops that monitor and recover from these conditions.

### 3.5. Process Practices

*Practice 28: Design applications to reflect the actual process state at initial startup and after temporary suspension.* When a process is suspended, the real world continues to change. Hence, the internal model of the suspended process may be inconsistent with the real world. Therefore, it must synchronize its internal model with the external world when it is continued, or when it is started. It is possible to encapsulate this practice as a class

within an object-oriented programming language, which would help to improve software reliability.

*Practice 29: Specify the minimum and maximum time that a process or thread should wait before the first input.* If a software system has not received input within a reasonable amount of time after startup, it most likely has not properly initialized itself and it should take corrective action to repair itself or notify an operator of the problem [14]. Similarly, this is also true if the time of receipt occurs too soon after start up. Adherence to this practice helps to produce reliable software.

*Practice 30: Design software to be free of deadlocks.* When a deadlock occurs, a system ceases to process data. Software professionals can verify that programs are free of deadlocks using formal and informal methods. Alternatively, software engineers could design systems to recover from such situations even though some data or processing will be lost.

### 3.6. Failure Monitoring and Recovery Practices

*Practice 31: Identify and address expected faults as early as practical.* Anticipating the types of faults that can occur within a software system permits its designers to propose solutions to them, which helps to produce reliable and safe software.

*Practice 32: Design software systems to continuously monitor their health and perform corrective measures, when possible.* Software systems should perform several actions to support this practice.

- Software systems should monitor assumptions during runtime and respond appropriately to deviations from them because violations of critical assumptions may cause catastrophic errors. At a minimum, systems should log each violation of an assumption. This implies that system analysts and software architects should identify the assumptions they make during analysis and design tasks.
- Software systems should respond to situations where resource usage is oversubscribed (e.g., violations of load assumptions). Common methods for dealing with this kind of overload include logging the problem, instructing external systems to reduce their load, locking out interrupts for overloaded channels, and reducing the functionality of the software system, or even halting or suspending a process or a computer. In addition, software can change operational behavior to handle the load. For example, the system may use faster algorithms to keep up with the workload, but produce outputs with reduced accuracy.
- Software systems should define the desired response to an overload condition. Typically, the performance of a software system will degrade, which implies that the system may disable various capabilities. In these

situations, software systems should inform operators of the degradation.

Adhering to this practice allows software systems to continue to operate and catch up to the input data or processing rates, when possible and appropriate.

*Practice 33: Design software systems to query their environments.* In some instances, system problems may arise because of defects in external systems. Therefore, some software systems may need to query external components to determine if they have failed [14]. For example, if a software system continues to timeout because it has not received input from an external system then it may make sense to shutdown the system or change its operation until someone repairs the external system. This practice may help to reduce the number of falsely reported defects.

*Practice 34: Use feedback loops to detect internal and external failures.* A feedback loop should execute a diagnostic procedure that determines if a failure has occurred, the source of the failure, and if corrective action is required. Unused inputs from a feedback loop probably indicate a deficiency in the design or implementation of the software. Similarly, a missing feedback loop may indicate an incomplete design or implementation. This practice benefits software systems by making them more reliable.

*Practice 35: Specify the minimum and maximum expected execution times for a computation.* If a computation requires less time than expected then a failure may have occurred preventing the system from performing the desired computation. If a computation has taken longer than expected then the system may be in a deadlock situation or the system may be waiting for input from a defective component, module, or external system. In either case, something may have gone wrong and adherence to this practice tends to improve system reliability and safety.

A particularly important type of computation is an atomic transaction. Since systems must revoke transactions of this nature, revocation of partially executed transactions may require that system analysts specify the times and conditions when cancellation can automatically occur, as well as the warnings that the system may issue.

*Practice 36: Specify the conditions where software systems can return to normal processing load after encountering an anomaly.* Adherence to this practice prevents a system from reentering an anomalous situation. After detecting a capacity violation, for example, the system should not begin normal processing too quickly because the circumstances that caused the capacity violation may still exist and cause the system to violate the capacity limitation once again.

*Practice 37: Design software to handle all credible anomalies.* This practice enables software to detect,

isolate, and correct single anomalies, which improves the reliability of software.

*Practice 38: Design software to handle successive, interrelated anomalies critical to its well being.* Analyzing multiple anomalies and determining what to do about them is a difficult task; therefore, it is prudent to address only interrelated anomalies that can do significant harm.

### 3.7. Finite State Machine Modeling Practices

*Practice 39: Design system-wide control logic as table-driven deterministic finite state machines.* Using finite state machines as a basis for system design, all components of a system, with the exception of the controller that controls those components, are passive and simply provide services to the active controller. Since the controller is the only active component in a system, software professionals can easily understand, validate, and modify the system behavior. In addition, software systems using a table-driven approach operate very efficiently. Further, data stores can represent finite state machines, which permits programmers to define multiple instances of finite state machines and their associated behaviors without actually changing code. In sum, this approach provides a very flexible and reliable means for controlling a software system.

*Practice 40: Design finite state machine models of software systems to satisfy the following constraints: (1) every state must have a defined transition for every possible input and (2) every state must have a defined transition to handle cases where a timeout occurs.* This scheme guarantees that the system will process all valid and invalid data, as well as handling situations where the system expects data, but the data does not arrive when expected. The beneficial aspect of this practice is that it ensures that all data is processed or that a failure is recognized. A program could validate this practice if it had access to machine-readable representations of finite state machines.

*Practice 41: When modeling a system using finite state machines, ensure that every state is reachable from its start state.* An unreachable state indicates a flaw in the requirements or design. Given machine-readable representations of finite state machines, a program could verify adherence to this practice.

*Practice 42: Eliminate soft and hard failure modes for all risk-reducing states, and define both soft and hard failure modes for risk-increasing outputs.* Whenever a system is attempting to transition to a more functional state, it should never permit a failure to occur that would put its health in worse shape. Contrarily, when a system enters a less capable state it should have both graceful and hard fallback options [14].

## 4. Summary and Conclusions

As part of an ongoing software process improvement activity, JPL has defined new requirements that software development personnel must follow and guidelines that they should follow. Hundreds of best practices were identified to support the effort, of which 42 were reported herein as general software design practices. These practices can be used to train software engineers, evaluate and define software development practices, and as a source of ideas for automating software design.

Unfortunately, few specification, modeling, or programming languages or integrated development environments are available to practicing software engineers that provide or encourage the use of many of these practices. Instead, current software engineering practice still places most of the burden of good analysis, design, and programming on the software professional, although software tools, such as compilers and integrated development environments could handle more of it.

Conventional programming languages, for example, suffer from several limitations. The software design practices already discussed identify five of them. First, conventional programming languages do not treat numbers as importantly as they should. More specifically, they do not allow programmers to specify the required precision or accuracy of numbers, nor do they compute the accuracy of numeric computations. Furthermore, they do not generally permit the specification of the allowable range of values that a numeric variable can assume. Second, programming languages do not have built-in data types corresponding to common types of measurements. Third, programming languages do not provide mechanisms for bounding the length of a computation or raising errors when such bounds are violated. Fourth, programming languages do not have built-in representations for state machines. Fifth, modern programming languages do not provide higher-level abstractions as exemplified by QA4 [23] and SETL [24].

Practitioners need these capabilities in their programming languages because they are overwhelmed by the problems that they are asked to solve. For example, the navigation software that JPL uses consists of 8 million source lines of code, and it is growing. The complexity of this one subsystem, of a much larger system is beyond human comprehension and is not verifiable. To make such a subsystem understandable requires an order of magnitude reduction in specification complexity. It has also been stated that such an order of magnitude improvement is required in language design to have people migrate from one language to another [13]. Although Java is an improvement over C and C++, it does not provide an order of magnitude improvement in personnel productivity or software quality. Instead, it has succeeded by providing modest improvements compatible

with existing, widely used languages. Hence, it does not help improve society's need to build systems that are more complicated in a time and cost effective manner.

Visual modeling languages, such as the Unified Modeling Language (UML) [2], are great for enforcing a common interpretation of a design. That is, UML defines rules governing the interpretation of diagrams. However, graphical representations have several limitations. First, deployment and class diagrams have marginal value. That is, most editors and integrated development environments provide utilities to rapidly define and acquire this same information. Second, sequence diagrams are difficult to specify repetitive operations. For example, try expressing  $\forall i \in S \exists p(i) \text{ do } s_1, s_2, \dots, s_n$  where  $S$  is a set of elements,  $p$  is a Boolean predicate, and  $s_i$  is a language statement. Third, UML modeling tools do not provide capabilities to validate state diagrams, although they could ensure that (1) every state has a transition for every possible input, (2) time-dependent states have a state transition for handling timeout conditions, and (3) every state is reachable.

Most integrated development environments also are severely flawed. First, it is extremely rare to find an environment that has the capability to measure design (or code) quality, although several studies have shown the benefits of metrics that measure coupling, cohesion, encapsulation, class inheritance depth, and function fan-out [1]. Second, no integrated development environment provides a capability to capture design options and decisions or the rationale for choosing one option over another, although there are numerous formal decision making techniques that have been applied in other industries and significant research has been conducted in design rationalization (e.g., [8], [18]).

Markowitz created modern portfolio theory in the early 1950s [15] and similar ideas can be applied to software. His contributions were two-fold. First, he determined how one could compute an efficient frontier of optimal solutions measured by return on investment for any level of risk. Second, he developed utility theory, which are methods for selecting an acceptable level of risk. The assets that he could invest in at the time were various stocks and bonds; he defined risk as the possible deviation of the expected rate of return of an investment.

Unfortunately, after thirty or so years of software experiments this profession has little data that indicate the relative merits between various software development schemes that improve personnel productivity and product quality. The field should be able to claim that one technique, say, software inspections, is 1.57 times more cost effective than software reviews and that the variance between the two methods for discovering defects is 0.63. If such information was available, practitioners could optimally identify the level of effort they should expend

applying each technique given the level of risk they are willing to accept. In sum, more work needs to address this issue; folklore is no longer sufficient.

Several design practices that this paper identifies can provide a foundation for an improved design methodology. First, architects should decompose software systems into smaller modules. Second, these smaller modules, and their underlying components, should model the problem domain. Such modeling helps to create natural system boundaries, which reduces module coupling and increases module cohesion. Third, architects should define strict behavioral interfaces for every module of the system. Fourth, architects should stop designing a system when they reduce the size of each module to the point where productivity is high, where introduced defects are low, and where the rate of change on each of these two dimensions begins to change rapidly. Such a point is generally around 5,000 source lines of code, or 100 function points [9]. In fact, function point computation is an appropriate way to estimate software size. This is because function point methodologies attempt to compute software complexity in terms of data and control complexity. Function point methodologies also estimate the algorithmic complexity of various problem domains (e.g., mathematical programming, real-time computing) and attempt to normalize for such problem domains. Fifth, architects should create appropriately sized specifications. Such specifications should include a description of the entire decomposition of the system into its various components and the interactions among the modules. In addition, these specifications should provide detailed descriptions of the interfaces of each module, including descriptions of the formal parameters, return values, pre- and post-conditions, and possible error conditions of each function provided by each interface, and the underlying assumptions of each interface.

Once the architects have designed the architecture, they should conduct a realistic simulation based on functional test cases by defining stubs for each module that respond appropriately to each input. Thus, after the successful execution of the simulation for every functional test, the architects would have validated the entire architecture. Afterwards, an organization could implement individual modules in parallel using as many people as there are modules, if needed. As long as each component adhered to its specification, the system would work as validated at the architectural level. If a component did not satisfy its specification, its small size would permit an organization to rapidly correct or replace it and later validate it. Most importantly, using this methodology a development team could validate a system design before it completely implements the design, which is contrary to current practice.

In conclusion, this paper has identified several design practices that help software engineers more effectively develop better products. It has also identified several weaknesses in the way the profession develops software and proposes some alternative solutions. The value of these alternatives has not yet been determined.

## 5. Acknowledgements

The author would like to thank Bill Pardee for his constructive criticism of an early draft of the design principles, which helped to improve the content of this paper. Similarly, Bruce Bullock made several useful comments on an earlier draft of this paper.

## 6. References

- [1] Boris Beizer, *Software Testing Techniques*, International Thomson Computer Press, 1990.
- [2] Grady Booch, James Rumbaugh, and Ivar Jacobson, *Unified Modeling Language User Guide*, Addison-Wesley, 1998.
- [3] Frederick Brooks, "No silver bullet: essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pg. 10-19, April 1987.
- [4] John Casini et al., *Report on the Loss of the Mars Climate Orbiter Mission*, Jet Propulsion Laboratory, Internal Document D-18441, November 1999.
- [5] Lee Coopriker, *The Representation of Families of Software Systems*, PhD Dissertation, Carnegie-Mellon University, Computer Science Department, 1979.
- [6] E. Dijkstra, "The Humble Programmer," *Communications of the ACM*, vol. 15, no. 10, pg. 859-866, October 1972.
- [7] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
- [8] P. Freeman and A. Newell, "A Model for Functional Reasoning in Design," *Proceedings of the Second International Joint Conference on Artificial Intelligence*, London, England, 1971.
- [9] David Garmus and David Herron, *Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000.
- [10] M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland, New York, 1977.
- [11] Brian Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, 1995.
- [12] Capers Jones, *Software Assessments, Benchmarks, and Best Practices*, Addison Wesley, 2000.
- [13] Donald E. Knuth, "Structured Programming with go to Statements," *ACM Computing Surveys*, vol. 6, no. 4, pg. 261-301, 1974.
- [14] Nancy G. Leveson, *Safeware: System Safety and Computers*, Addison Wesley, 1995.
- [15] Harry M. Markowitz, *Portfolio Selection: Efficient Diversification of Investments*, Blackwell Publishers, 2<sup>nd</sup> ed., 1991.
- [16] Thomas J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pg. 308-320, 1976.
- [17] Steve McConnell, "Less is More," *Software Development*, October 1997.
- [18] Thomas P. Moran and John M. Carroll, *Design Rationale: Concepts, Techniques, and Use*, Lawrence Erlbaum, 1996.
- [19] A. Nikora and J. Munson, "Determining Fault Insertion Rates for Evolving Software Systems", *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 4-7, 1998.
- [20] William J. Pardee, *To Satisfy and Delight Your Customer: How to Manage for Customer Value*, Dorset House, 1996.
- [21] David Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1971.
- [22] Mark C. Paulk et al, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 1994.
- [23] J. F. Rulifson, J. A. Derksen, and R. J. Waldinger, *QA4: A Procedural Calculus for Intuitive Reasoning*, Technical Report 73, AI Center, SRI International, November 1972.
- [24] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: An Introduction to SETL*, Springer-Verlag, 1986.