Fifteen principles of Davis :

1. Make quality Number 1: It is shortsightedness to deliver a product that is of poor quality. Quality has many definitions and means different things to different constituents. For example, to the developer, it may mean “elegant design,” and to the customers it may mean ”good response time." In any case, quality must be viewed as the first requirement.

2. High-quality software is possible: Large software systems can be built with good quality, although it may carry a high price.

3. Give products to customers early: It is very difficult to completely understand and capture the users' needs during the requirements phase; thus it is more effective to give the users a prototype of the product and let them play with it. Gather the feedback and then go into full-scale development of the product.

4. Determine the problem before writing the requirements: Before the software engineers rush to offer the solution, ensure that the problem is well understood. Then explore the potential solution and various alternatives.

5. Evaluate design alternatives: After the requirements are understood and agreed upon, explore a variety of design architecture and related algorithms. Ensure that the selected design and algorithms are the best match to satisfy the goals of the requirement.

6. Use an appropriate process model: Since there is no universal process model that applies to all projects, each project must select a process that best fits the project based on parameters such as corporate culture, project circumstances, user expectations and requirements volatility, and resource experiences.

7. Use differentlanguages for differentphases: No one single language is optimal for all phases of software development. Therefore select the best methodology and language for the different phases of software development. The d ifflculty in transition from one phase to another is not necessarily solved by using a single language across all phases.

8. Minimize intellectual distance: It is easier to create the solution if the distance between the real-world problem and the computerized solution to that problem is minimized. That is, the software solution structure needs to be as close as possible to the real-world problem structure.

9. Put technique before tools: Before using the tool, the technique needs to be well understood. Otherwise, the tool just rushes us into performing the wrong thing faster.

10. Get it right before you make it faster: It is essential to make the software execute correctly first and then work on improving it. Do not worry about optimization for either execution speed or code during initial coding.

11. Inspectcode: Inspection as first proposed by IBM’s Mike Fagan is a much better way to find errors than testing. Some early data in inspections showed a reduction of 50% to 90% of the time-to-test.

12. Good management is more important than good technology: A good manager can produce extraordinary results even with limited resources. Even the best technology cannot compensate for terrible management because it cannot motivate the people as good mangers can.

13. People are the key to success: Software is a labor-intensive profession. and people with experience, talent, and appropriate drive are the key. The right people can overcome many of the shortcomings in process, methodology, or tools. There is no substitute for quality people.

14. Follow with care: Be careful in adopting tools, process, methodology, and so on. Do not follow just because someone else is doing it or using it. Run some experiments before making a major commitment.

15. Take responsibility. If you developed the system, then you should take responsibility to do it right. Blaming the failure or the problem on others, on the schedule, or on the process is irresponsible.

1. Make quality #1. Quality must be quantified and mechanisms put into place to motivate its achievement.

a Defining quality commensurate with the project at hand is important but is not easily done at the outset of a project. Consequently, a modern process framework strives to understand the trade-offs among features, quality, cost, and schedule as early in the life cycle as possible. Until this understanding is achieved, it is not possible to specify or manage the achievement of quality.

2. High-quality software is possible. Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people.

a This principle is mostly redundant with the others.

3. Give products to customers early. No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it.

a This is a key tenet of a modern process framework, and there must be several mechanisms to involve the customer throughout the life cycle. Depending on the domain, these mechanisms may include demonstrable prototypes, demonstration-based milestones, and alpha/beta releases.

4. Determine the problem before writing the requirements. When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.

a This principle is a clear indication of the issues involved with the conventional requirements specification process. The parameters of the problem become more tangible as a solution evolves. A modern process framework evolves the problem and the solution together until the problem is well enough understood to commit to full production.

5. Evaluate design alternatives. After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an "architecture" simply because it was used in the requirements specification.

▲ This principle seems anchored in the waterfall mentality in two ways: (1) The requirements precede the architecture rather than evolving together. (2) The architecture is incorporated in the requirements specification. While a modern process clearly promotes the analysis of design alternatives, these activities are done concurrently with requirements specification, and the notations and artifacts for requirements and architecture are explicitly decoupled.

6. Use an appropriate process model. Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.

a It's true that no individual process is universal. I use the term process framework to represent a flexible class of processes rather than a single rigid instance. Chapter 14 discusses configuration and tailoring of the process to the various needs of a project.

7. Use different languages for different phases. Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle. Why should software engineers use Ada for requirements, design, and code unless Ada were optimal for all these phases?

a This is an important principle. Chapter 6 describes an appropriate organization and recommended languages/notations for the primitive artifacts of the process.

8. Minimize intellectual distance. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure.

a This principle has been the primary motivation for the development of object-oriented techniques, component-based development, and visual modeling.

9. Put techniques before tools. An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.

a Although this principle is valid, it misses two important points: (1) A disciplined software engineer with good tools will outproduce disciplined software experts with no tools. (2) One of the best ways to promote, standardize, and deliver good techniques is through automation.

10. Get it right before you make it faster. It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding.

▲ This is an insightful statement. It has been misstated by several software experts more or less as follows: "Early performance problems in a software system are a sure sign of downstream risk." Every successful, nontrivial software project I know of had performance issues arise early in the life cycle. I would argue that almost all immature architectures (especially large-scale ones) have performance issues in their first executable iterations. Having something executing (working) early is a prerequisite to understanding the complex performance trade-offs. It is just too difficult to get this insight through analysis.

11. Inspect code. Inspecting the detailed design and code is a much better way to find errors than testing.

▲ The value of this principle is overhyped for all but the simplest software systems. Today's hardware resources, programming languages, and automated environments enable automated analyses and testing to be done efficiently throughout the life cycle. Continuous and automated life-cycle testing is a necessity in any modem iterative development. General, undirected inspections (as opposed to inspections focused on known issues) rarely uncover architectural issues or global design tradeoffs. This is not to say that all inspections are ineffective. When used judiciously and focused on a known issue, inspections are extremely effective at resolving problems. But this principle should not be in the top 15, especially considering that the industry's default practice is to overinspect.

12. Good management is more important than good technology. The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Good management motivates people to do their best, but there are no universal "right" styles of management.

▲ My belief in this principle caused me to write this book. My only argument here is that the term meager resources is ambiguous. A great, well-managed team can do great things with a meager budget and schedule. Good management and a team meager in quality, on the other hand, are mutually exclusive, because a good manager will attract, configure, and retain a quality team.

13. People are the key to success. Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed. The wrong people with appropriate tools, languages, and process will probably fail.

▲ This principle is too low on the list.

14. Follow with care. Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping—all these might increase quality, decrease cost, and increase user satisfaction. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal.

▲ This is sage advice, especially in a rapidly growing industry in which technology fads are difficult to distinguish from technology improvements. Trading off features, costs, and schedules does not always favor the most modern technologies.

15. Take responsibility. When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.

▲ This is a great corollary to item 14. It takes more than good methods, tools, and components to succeed. It also takes good people, good management, and a learning culture that is focused on forward progress even when confronted with numerous and inevitable intermediate setbacks.