Software Development Best Practices -- quotes from Steve McConnell, Code Complete 2nd Ed

On small, informal projects, a lot of design is done while the programmer sits at the keyboard. "Design" might be just writing a class interface in pseudocode before writing the details. It might be drawing diagrams of a few class relationships before coding them. It might be asking another programmer which design pattern seems like a better choice. Regardless of how it's done, small projects benefit from careful design just as larger projects do, and recognizing design as an explicit activity maximizes the benefit you will receive from it.

Design Is a Wicked Problem Horst Rittel and Melvin Webber defined a "wicked" problem as one that could be clearly defined only by solving it, or by solving part of it (1973). This paradox implies, essentially, that you have to "solve" the problem once in order to clearly define it and then solve it again to create a solution that works.

The picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will. Even the small program developments shown in textbooks and papers are unreal. They have been revised and polished until the author has shown us what he wishes he had done, not what actually did happen. â€" David Parnas Paul Clements

One of the main differences between programs you develop in school and those you develop as a professional is that the design problems solved by school programs are rarely, if ever, wicked.

Programming assignments in school are devised to move you in a beeline from beginning to end. You'd probably want to tar and feather a teacher who gave you a programming assignment, then changed the assignment as soon as you finished the design, and then changed it again just as you were about to turn in the completed program. But that very process is an everyday reality in professional programming.

Design is sloppy because a good solution is often only subtly different from a poor one.

Design is also sloppy because it's hard to know when your design is "good enough." How much detail is enough? How much design should be done with a formal design notation, and how much should be left to be done at the keyboard? When are you done? Since design is open-ended, the most common answer to that question is "When you're out of time.

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies. â€" C. A. R. Hoare

Managing complexity is the most important technical topic in software development. In my view, it's so important that Software's Primary Technical Imperative has to be managing complexity. Complexity is not a new feature of software development.

Computing pioneer Edsger Dijkstra pointed out that computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of 1 to 109, or nine orders of magnitude (Dijkstra 1989).

the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history." Of course software has become even more complex since 1989, and Dijkstra's ratio of 1 to 109could easily be more like 1 to 1015 today.

Dijkstra pointed out that no one's skull is really big enough to contain a modern computer program (Dijkstra 1972), which means that we as software developers shouldn't try to cram whole programs into our skulls at once; we should try to organize our programs in such a way that we can safely focus on one part of it at a time.

The goal is to minimize the amount of a program you have to think about at any one time. You might think of this as mental juggling the more mental balls the program requires you to keep in the air at once, the more likely you'll drop

one of the balls, leading to a design or coding error.

One symptom that you have bogged down in complexity overload is when you find yourself doggedly applying a method that is clearly irrelevant, at least to any outside observer. It is like the mechanically inept person whose car breaks down so he puts water in the battery and empties the ashtrays. â€” P. J. Plauger

At the software-architecture level, the complexity of a problem is reduced by dividing the system into subsystems. Humans have an easier time comprehending several simple pieces of information than one complicated piece.

The more independent the subsystems are, the more you make it safe to focus on one bit of complexity at a time. Carefully defined objects separate concerns so that you can focus on one thing at a time. Packages provide the same benefit at a higher level of aggregation.

…modern software is inherently complex, and no matter how hard you try, you'll eventually bump into some level of complexity that's inherent in the real-world problem itself. This suggests a two-prong approach to managing complexity: Minimize the amount of essential complexity that anyone's brain has to deal with at any one time. Keep accidental complexity from needlessly proliferating. Once you understand that all other technical goals in software are secondary to managing complexity, many design considerations become straightforward.

Levels of Design Design is needed at several different levels of detail in a software system. Some design techniques apply at all levels, and some apply at only one or two. The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are

Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are not allowed to look at any of the details of the complex concept. What you see is what you get and it's all you get!

Identify items that seem likely to change. If the requirements have been done well, they include a list of potential changes and the likelihood of each change. In such a case, identifying the likely changes is easy. If the requirements don't cover potential changes, see the discussion that follows of areas that are likely to change on any project. Separate items that are likely to change. Compartmentalize each volatile component identified in step 1 into its own class or into a class with other volatile components that are likely to change at the same time. Isolate items that seem likely to change. Design the interclass interfaces to be insensitive to the potential changes. Design the interfaces so that changes are limited to the inside of the class and the outside remains unaffected. Any other class using the changed class should be unaware that the change has occurred. The class's interface should protect its secrets. Here are a few areas that are likely to change: Business rules.Â Business rules tend to be the source of frequent software changes. Congress changes the tax structure, a union renegotiates its contract, or an insurance company changes its rate tables. If you follow the principle of information hiding, logic based on these rules won't be strewn throughout your program. The logic will stay hidden in a single dark corner of the system until it needs to be changed.

Don't use a boolean variable as a status variable. Use an enumerated type instead. It's common to add a new state to a status variable, and adding a new type to an enumerated type requires a mere recompilation rather than a major revision of every line of code that checks the variable. Use access routines rather than checking the variable directly. By checking the access routine rather than the variable, you allow for the possibility of more sophisticated state detection. For example, if you wanted to check combinations of an error-state variable and a current-function-state variable, it would be easy to do if the test were hidden in a routine and hard to do if it were a complicated test hard-coded throughout the program.

Keep Coupling Loose Coupling describes how tightly a class or routine is related to other classes or routines. The goal is to create classes and routines with small, direct, visible, and flexible relations to other classes and routines, which is known as "loose coupling." The concept of coupling applies equally to classes and routines, so for the rest of this discussion I'll use the word "module" to refer to both classes and routines. Good coupling between modules is loose enough that one module can easily be used by other modules.

Try to create modules that depend little on other modules. Make them detached, as business associates are, rather than

attached, as Siamese twins are.

Keep Your Design Modular Modularity's goal is to make each routine or class like a "black box": You know what goes in, and you know what comes out, but you don't know what happens inside.

The concept of modularity is related to information hiding, encapsulation, and other design heuristics. But sometimes thinking about how to assemble a system from a set of black boxes provides insights that information hiding and encapsulation don't, so the concept is worth having in your back pocket.

Summary of Design Heuristics Here's a summary of major design heuristics: More alarming, the same programmer is quite capable of doing the same task himself in two or three ways, sometimes unconsciously, but quite often simply for a change, or to provide elegant variation. A. R. Brown W. A. Sampson Find Real-World Objects Form Consistent Abstractions Encapsulate Implementation Details Inherit When Possible Hide Secrets (Information Hiding) Identify Areas Likely to Change Keep Coupling Loose Look for Common Design Patterns The following heuristics are sometimes useful too: Aim for Strong Cohesion Build Hierarchies Formalize Class Contracts Assign Responsibilities Design for Test Avoid Failure Choose Binding Time Consciously Make Central Points of Control Consider Using Brute Force Draw a Diagram Keep Your Design Modular

A brute-force solution that works is better than an elegant solution that doesn't work. It can take a long time to get an elegant solution to work. In describing the history of searching algorithms, for example, Donald Knuth pointed out that even though the first description of a binary search algorithm was published in 1946, it took another 16 years for someone to publish an algorithm that correctly searched lists of all sizes (Knuth 1998). A binary search is more elegant, but a brute-force, sequential search is often sufficient. When in doubt, use brute force. â€" Butler Lampson

One of the most effective guidelines is not to get stuck on a single approach. If diagramming the design in UML isn't working, write it in English. Write a short test program. Try a completely different approach. Think of a brute-force solution. Keep outlining and sketching with your pencil, and your brain will follow. If all else fails, walk away from the problem. Literally go for a walk, or think about something else before returning to the problem. If you've given it your best and are getting nowhere, putting it out of your mind for a time often produces results more quickly than sheer persistence can. You don't have to solve the whole design problem at once. If you get stuck, remember that a point needs to be decided but recognize that you don't yet have enough information to resolve that specific issue. Why fight your way through the last 20 percent of the design when it will drop into place easily the next time through? Why make bad decisions based on limited experience with the design when you can make good decisions based on more experience with it later? Some people are uncomfortable if they don't come to closure after a design cycle, but after you have created a few designs without resolving issues prematurely, it will seem natural to leave issues unresolved until you have more information (Zahniser 1992, Beck 2000).