Software Quality

I’ve been doing software a long time, and it’s fair to say I’ve seen a lot more bad code than good code, which begs the question: what constitutes good software?

When asked about how to identify quality software a lot people today will start their answer with something like: “it should be object oriented …”. No. Sorry. Object oriented design is a means to an end, not an end in itself. I’ve seen “object oriented” code that could easily compete with any piece of unstructured, goto filled, FORTRAN code for “worst software of the year” status. “Object Orientedness” is not a valid metric for evaluating software. The following table lists metrics that are valid:

Metric	Meaning	Which Way
Cost to Implement	All other things (functionality, robustness, efficiency, etc) being equal, the software which costs the least to implement is better.	Lower is Better
Time to Implement	All other things being equal, the software which takes less (calendar) time to produce is better.	Lower is Better
CPU Usage	All other things being equal, the software which uses fewer CPU cycles is better	Lower is Better
MemoryUsage	All other things being equal, the software which uses less memory is better.	Lower is Better.
Complexity	All other things being equal, the software which is less complex is better.	Lower is Better
Testability	All other things being equal, the software which is easier to test is better.	Higher is Better
Portability	All other things being equal, the software which is easier to port (to a new compiler, new CPU, new OS, etc) is better.	Higher is Better
Reusability	All other things being equal, the software which is more reusable (of interest to people outside the original customer) is better.	Higher is Better
Extensibility	All other things being equal, the software which can more easily accommodate enhancements is better.	Higher is Better

WHAT NOT TO DO

The above goals are what you're shooting for if you're looking for quality in software. "Object Oriented" is too often used simply as a meaningless industry buzzword, especially by people with no engineering talent - the answer to any software problem becomes "OO". That unfortunate mindset is illustrated in the following table, contrasting how a real engineer might solve some assorted problems, versus the "solution" proposed by a software "engineer":

Project	Real Engineering Solution	Software "Engineering" Solution
Airframe for USAF fighter jet	Machined Titanium Casting – because light weight, strength, & high temperature tolerance are paramount, and cost is no object.	Machined Titanium Casting – because machined titanium casting is the latest and greatest in metal processing.
Restaurant grade kitchen sink	Stainless Steel – because corrosion resistance is essential, but strength & temperature requirements are not very demanding.	Machined Titanium Casting – because machined titanium casting is the latest and greatest in metal processing.
Eight penny common nail	Soft Steel – because low cost is essential, and soft steel has adequate strength and can be easily cold formed.	Machined Titanium Casting – because machined titanium casting is the latest and greatest in metal processing.
One gallon disposable milk jug	Polyethylene – because of its low cost, light weight, non-toxicity, and the fact that it imparts no taste or odor.	Machined Titanium Casting – because machined titanium casting is the latest and greatest in metal processing.

TOOLS

The point being, if you're an actual engineer, you'll have a laundry list of techniques that you can bring to bear on a problem to be solved. Object Oriented Design is one technique, and in fact it's a powerful and important one for those who actually understand it – but it's not the only technique. Here's a (partial) list of software engineering techniques, *all* of which should be on the table for consideration and use when implementing software:

- Modularization
- Structured Programming
- Use Cases

- Stepwise Refinement
- Documentation
- Object Oriented Design
- Layering
- Modularity
- Standards
- Rapid Prototyping
- Top Down Design
- Bottom Up Design
- Middle Out Design
- Automated Code Generation
- Table Driven Software
- State Machines
- Recursive Descent Parsers
- Delayed Decision Making

DESIGN VS IMPLEMENTATION

One of my pet peeves is people who don't understand the difference between design and implementation. This sometime becomes very clear with programmers working in OO languages like C++ and Java. Let me illustrate.

Suppose you find yourself in this environment: You're developing a software system for a specialized hardware platform, and the only tool you have for development is an assembler. Let's suppose that getting any sort of higher level tool (EG FORTRAN or Pascal or C) is not possible. The question is, do you use object oriented design?

The answer is, of course! You do almost exactly the same thing you would do if the code was going to be C++ or Java. You design the objects, their attributes, how they interact with each other. The difference comes at implementation time – you're going to have to write the code for invoking virtual function, or inheritance (or even plain subroutine calls for that matter). It's a lot more work (that's why people write compilers), but it's certainly doable. A more realistic example would involve C rather than assembler. I cringe when I hear someone say something like "you can't do OO in C because it doesn't have classes". I once saw someone implement an OO design in a shell script – and it actually worked reasonably well!

COMPLEXITY

Let me close by saying a bit more about complexity.

Complexity is the enemy of good software. Arguably, every failed software project since the beginning of time failed because of out-of-control complexity. That's an oversimplification of course, but there's also a lot of truth in it.

Complexity is the enemy of good software. Not everyone understands this. Some people are even proud of writing complex software. The thing is, with software, complexity just happens - it takes no brains or talent to generate complex software. What does take a lot of ability, experience, and hard work is writing simple software.

Complexity is the enemy of good software. I once read a post by someone who did a lot of refactoring who described his job as "deleting lines of code". That's about right. My most extreme example of "over-coding" is a C++ object oriented guy who used several hundred lines of code to implement a counter.

Complexity: Just say no!