# Software Engineering

Early in my career I spent some time as a Quality Assurance Engineer at Colt Industries, STEMCO Truck Products Division in Longview, Texas. One of my responsibilities was to evaluate products from vendors and develop tests for them. We would receive products and expected to see both the manufacturing documentation, when it was made and what were the process parameters, as well as the results of their own tests. We would then perform our tests, similar to those that the manufacturer had performed, and known to the manufacturer. These might include dimensional inspection, salt sprays, stress tests, etc. Essentially this provided an information sharing mechanism. The manufacturer built their objects using a specified process and provided documentation for the build of a specific batch along with the batch. They tested that batch and provided details of their testing. We then audited that batch with our own testing but would not even audit without the necessary documentation, rather we would simply reject the batch. If the manufacturer had not used a repeatable and agreed on process to manufacture and test the product to ensure it met our requirements, and could not document that they had done so, we would simply refuse to even look at the batch.

I worked in 'heavy metal' manufacturing plants. Every square yard of the plant had something that could kill or maim the casual or the careless. It was in this facility that a foreman, the most experienced man in the plant, got his hand caught in a press. They worked to try and get him out for four or five hours as I recall. Finally, knowing they would have to disassemble the press and that would be a twelve hour job, the foreman told them to 'stroke it through', removing the majority of the fingers on his right hand. Another man's legs were crushed due to the carelessness of a fellow employee. The paramedic was a friend of mine. He said he could hear the man's screams all the way up from the tool and die shop and in the main building as he was responding to the call. Both buildings were heavily insulated. Many of our finished products went onto the drive and trailer axles of over the road trucks. Failures had caused accidents, fires, and had killed numerous people over the years.

In my subsequent career I built machines similar to the one in which the foreman had lost his hand. When I was a Machine Design Engineer and Maintenance Foreman at Western LithoTech in Jacksonville, Texas, a man who was working for me, a friend, had gotten his son on as an electrical contractor. I responded to a 'code red' one day and got out to the location just in time to see the paramedics loading his son into the ambulance. I'd never seen someone that had just been killed. It's not something you forget. His father was downtown in Jacksonville, Texas eating lunch with vendors. This was before cell phones, so I drove down and picked him up and took him to the hospital, sitting with him for the next several hours until they pronounced his son dead. The young man was survived by a wife and two daughters, ages three and five as I recall. I worked as a Machine Design Engineer for Martin Marietta, Middle River Plant just south of Baltimore, MD on the Navy's Vertical Launch System. Failures in these objects could mean that the ship was left without defenses, or it could mean an explosion of a warhead on board that could kill dozens, or disable or sink a ship.

I tell you these things so that you understand that I approach manufacturing and quality from a different perspective than many of my peers who are also Software Engineers. I realize that, if our product fails, no fires will be started, no explosions will occur, and it is unlikely that I will have to explain to the widow and young children of a coworker at a funeral why I implemented a sub standard design. Regardless, we are expected to deliver objects that meet business requirements, to deliver them on time, and to deliver them with a minimum of defects. A failure of my design or deliverable may not (directly) cause the death or maiming of a coworker, friend, client, or innocent user of our application, but I will lay odds that our client doesn't give a damn, and expects on time deliverables that meet business requirements with a minimum of defects.

About fifteen years ago, after about fifteen years of building systems and using progressively more automation, computer aided design, and control systems I went into software engineering full time. I have applied the same methods I learned in some of the most dangerous factories in the United States as a Machine Design Engineer to my career as a Software Engineer. During that early career I wrote software in a variety of languages, used statistics to evaluate processes, and spent two and a half years on an assignment to build an Artificial Intelligence system for the US Dept of the Interior. All of this I apply to my methodology as a Software Engineer. These methods work. They are based at the core on W. Edwards Deming's processes and on hard earned experienced that I gained through my failures, my successes, or through working with mentors during my early career. I refer to [my Wiki](http://sourceforge.net/p/chamomile/wiki/katherines_laws_of_software_design/) because I've already condensed much of my knowledge and experiences there.

What I have written here is primarily not a simple restatement of what is on my wiki. Instead, it is a commentary on that content which I believe to be relevant to what we are doing and how we are doing it. I would suggest you consider reading both the original statement of [*Katherine's Laws of Software Design*](http://sourceforge.net/p/chamomile/wiki/katherines_laws_of_software_design/) along with my commentary here. I do not cover Change Management here, but will do so in the future.

*There was never a single injury to any person working on any machine that I built, modified, or upgraded.*

## [Katherine's Laws of Software Design](http://sourceforge.net/p/chamomile/wiki/katherines_laws_of_software_design/)

Clarity - (First Law) *- Well written software can be re-factored. Poorly written software can only be tolerated.*

I neither design objects nor do I write software that are intended to be enshrined on the bosom of infamy. I write software that is intended to be refactored! I assume going into a design or build that someone will have to come after me and extend, improve, update, or otherwise modify that object. For this reason I try to be as clear as possible in my use of alias names, formatting, etc. I try to build things in 'groups' that are logical, often using what might seem to be spurious begin/end constructs to separate out a block of code as being a 'unit'. I avoid obvious comments such as '--parameter list', but try to place strategic guideposts in code that helps the reader understand what each section is doing. Long or complex merge statements or common table expressions are an example of where I might place documentation at the breaks between statements in a cte chain or in the clauses of a merge statement. I use this 'dead' documentation sparingly (see the Sixth Law). It is not intended to describe the business functionality of the object, only to guide the developer who comes after me (or me, after a few hours or days!).

I do not design for the lowest common denominator. *Well written software* is a statement that is meaningful only to a skilled and experienced engineer, developer, or dba. I will make no excuses for using sophisticated constructs that are in common usage or are needed to abstract complexity away from the public interface. I will make no excuses for fully exercising the methods and capabilities of the platform and language (SQL Server and Transact SQL) to their fullest. Designing and building objects for ignorance is not nor should it be considered to be a result of this law. NASA engineers didn't design the space shuttle so [Gomer Pyle](http://en.wikipedia.org/wiki/Gomer_Pyle) could work on them. They used the complexity necessary. Even then, by [ignoring basic engineering practices](http://sourceforge.net/p/chamomile/wiki/best_practices/) they caused the death of the Challenger crew. (See the *Third Law* and the *Eighth Law*). Simultaneously, and with reference to the *Eighth Law* below, I often refuse to build in performance enhancements or constructs which might be more optimal. Once it meets the existing business objectives I am done. Future requirements can be met in the future, when they are defined.

Robustness - (Second Law) *- There is no such thing as "bad" data.*

I don't design objects with the intent to *let it fail*. That actually was the methodology for a group that I worked with recently. They refused to allow any error handling in the code. Nor would they allow any 'complex' constructs, insisting on building code that could be understood by the most junior developer. It was common to be woken up at zero dark thirty when on call to debug a failure and find out it was something as simple as the inbound data (these were mostly ETL {Extract/Transform/Load} processes) having an alphanumeric where a numeric string was expected. Rather than using common data scrubbing techniques they chose to let the code fail. It should be obvious to all but the least intelligent that this is a 'house of cards'. The more processes that you implement in this manner, the more 'live' debugging you will need to do. This is not a scalable solution.

I commonly use data scrubbing techniques for any data that is manipulated. For an ETL process I might use a PowerShell script that parses the file using regular expressions to evaluate each column of each row, extracting the rows that pass to a 'good' file and the rows that fail to a 'bad' file. For inbound parameters I often use typed XML as a data type. This allows me to use regular expressions to type the data so that, for example, I can require that a string be only forty characters. If I simply define the input parameter as [varchar](40), a longer string will be silently truncated. Using typed XML an inbound string of more than forty characters will cause an exception, notifying the calling application that the data is invalid. Silently truncating business data is simply not something that I will consider. An application, an object, should be robust enough to handle business data safely and securely, reporting intelligent exceptions. Failure due to an unexpected condition or unusual data is never an option.

Complexity - (Third Law) *- If a method is too complex to effectively*[*unit test*](http://sourceforge.net/p/chamomile/wiki/unit_test/)*, it is too complex.*

Many development shops have standards for complexity. Some do not and you will find stored procedures that are three to five thousand lines long and are just a [big ball of mud](http://en.wikipedia.org/wiki/Big_ball_of_mud), sometimes referred to as spaghetti code. Some shops simply state that a method should be no longer than a certain number of lines. This is valid at face value, but you can rapidly find that you are having to create multiple methods to perform accessor and mutator operations on the same object. By using an arbitrary limitation as a complexity rule, you have forced the complexity into the public interface. A complexity measure that is common with the [Agile methodologies](http://en.wikipedia.org/wiki/Agile_software)(and one which I carry over from my time working with dangerous machinery) is that a method is too complex if it cannot be effectively tested programmatically. This is a core and driving factor in the [Test Driven Development methodology](http://en.wikipedia.org/wiki/Test_driven_development).

The logic behind this statement is simple; if you can effectively, repeatably, programmatically, and fully test an object, it can be[refactored](http://en.wikipedia.org/wiki/Refactor)in the future for performance, extension, clarity, etc. If you cannot effectively, repeatably, programmatically, and fully test an object, then even fifteen lines of code is too long. If it can be tested, you can modify or enhance it and run the original tests against it to ensure that the original functionality has not been lost. If you cannot test it, you have to wait for the customer to tell you if you have impacted the original functionality. In 'machinery' terms, that would mean that I would have to wait until someone is killed or maimed to find out if my design change is valid. In business software terms, that means I may have to wait until a hacker steals or corrupts business data, or until a client’s data is compromised or corrupted to know that my design change is valid. Neither option is acceptable.

*Note that nothing is said here about the test environment. That is not an indication that the test environment or the process of change management is of minor importance. To the contrary, the test environment and the change management process can be the deciding factor between success and failure. Nonetheless, both are supporting components of the design and build process rather than primary components.*

Objectives - (Fourth Law) *- If you have no objectives, you cannot meet them; if you have no principles, it will not matter if you do.*

I would like to be able to say that this is one of those concepts that is immediately obvious to the casual observer. I'm not going to go into it at length. You either get this or you don't. If you'd like a semantic explanation please ask.

Normalization - (Fifth Law*) - Normalization is not the normal state of a design, but it is the preferred state.*

A relational database is built on the [relational theory](http://en.wikipedia.org/wiki/Relational_theory) developed by [E.F. Codd](http://en.wikipedia.org/wiki/Edgar_F._Codd) in the 1960's. The relational theory is based on mathematics; [set theory](http://en.wikipedia.org/wiki/Set_theory). Virtually all modern databases are built in relational databases (RDMS) using relational theory. The reason for this is that the design is robust and ensures data integrity. There are a number of ways you can normalize a design. You can use meta-data (catalog) tables or data constraints for example. Typically, I will use a data constraint on a column if there are only a few values and those values are not expected to change. For example; there might be only two types of order, *sell*, or *buy*. In that case, a simple constraint on the column will ensure that the data is constrained and meet the rules of normalization while introducing the minimum of complexity. Conversely, there may be an extended list of fund names, so a meta-data (catalog) table that can be extended is called for.

Meta-data - (Sixth Law) *- Good software should be self-documenting.*

I have worked with mechanics, electricians, and other engineers on machinery that was up to a century in age. Many software engineers and developers have worked on code that was half a century in age, particularly in the effort to address the potential 'y2k' issues. My rule of thumb is that I should be able to give my code to an intelligent but non-developer colleague, and that colleague should be able to read my code, understanding its basic functionality. I minimize the use of abbreviations for this reason, and try to give common names to objects. For example, the following SQL code is comprehensible to even someone who has never written a line of any code;

*declare @today [datetime] = current\_timestamp;*

*select [order]*

*from [sales].[data]*

*where [date]=@today*

*and [customer]=N'Bob Johnson'*.

This is a simple example of course, but when I focus on this as an *intention*I tend to write clearer code. This means that future refactoring (or debugging) will be simpler, furthering the objective of delivering objects on time and with minimal defects that meet business requirements. I also make heavy use of meta-data to document an object. In the case of SQL server, this meta-data is captured in extended properties. This is effectively a carryover from Object Oriented design; the meta-data for an object should be contained within the object and should be accessible by querying the object. This is 'live' documentation and is distinguishable from the 'dead' documentation in a method which is delimited by '-- docs' or '/\* docs \*/' which cannot be queried. 'Dead' documentation should only be used for developer 'sign posts', to improve the clarity of the code. All business documentation should be contained within the 'live' documentation as extended properties. These can be extracted and formatted as required for professional presentations, but the presentations should never supplant the meta-data of the object.

Testing - (Seventh Law) *- If it can't be measured, it doesn't exist.*

As engineers and developers it is easy to build objects that do things we'd like them to do rather than which meet business objectives. One way we manage that is with the *Fourth Law*. Another technique is through the application of the *Third Law*. The *Seventh Law*speaks to the requirement for facts. [Software Engineering is in great part art](http://sourceforge.net/p/chamomile/wiki/engineering_as_an_art_form/) rather than a science. Software cannot be fully defined by either Newtonian or Einsteinian physics. Instead, a software design or object is a [fractal](http://en.wikipedia.org/wiki/Fractal), defined by [Chaos Theory](http://en.wikipedia.org/wiki/Chaos_theory). Often, the only way we can fully constrain and define the result of a design or of an object is through a test. In the case of software this is commonly referred to as a [unit test](http://en.wikipedia.org/wiki/Unit_test).

if you cannot effectively, repeatably, programmatically, and fully test an object, not only is it too complex, you cannot even prove that it meets the business requirement. If you cannot prove that it meets the business requirements, you should not even bother to build it.

Complexity - (Eighth Law) *- Complexity must justify itself.*

Most systems are comprised of multiple tests, constraints, rules, guidelines, or laws. The reason for this is that few systems are so simple that they can be adequately defined on only one dimension. Such is the case with these laws. It is easy to run headlong down the path of others of these laws and build systems or objects that are highly complex. Sometimes this is justified. Building complexity within an object to abstract it away from the public interface is a logical and defensible objective. Regardless, the complexity must always justify itself. There must be a rational reason why this complexity exists. It is not enough that it is cool, trendy, elegant, or fun to write.

You must always be ready to explain why you use a technique, tool, or process in light of other potentially less complex solutions or tools. Referring to other laws in this set, you should also design and build objects that allow refactoring, that are easy to understand, and that are fully tested. Doing so will allow you to remove or replace complexity without compromising the business objectives.

Focus - (Ninth Law) *- When dogma becomes an impediment to the achievement of your objectives, perhaps it is time to shoot your dog.*

Dogma is not a deliverable except when your are teaching a class. Standards are not a deliverable except when your are selling processes. If your deliverable is an object that is expected to meet business requirements and to be delivered on time and with a minimal number of defects, at some point you may have to make the decision to shoot your proverbial dog. This does NOT mean you should throw caution to the winds early in the process! It means that you should constantly evaluate what you are doing and why, being ever critical of the dogma, the standards of your process.

*We've always done it that way* should be a phrase that makes your hackles rise, that makes you question, and that makes you want to understand. If either dogma or standard impedes your ability to meet business objectives on time, and you can provide a product that meets those business objectives with a minimal and acceptable number of defects in the absence of your dogma and can do it repeatedly and on demand, your dogma, your standards need to be reevaluated. Dogma and standards should support the meeting of objectives. They are not the deliverable. Hence, they should constantly be critically evaluated and refined as necessary to assist in the regular achievement of business objects.

When standards or dogma become orthodoxy, they are by definition invalid until proven otherwise.