**Professional Engineering**

James Hamilton, 2003.03.13

Update: 2007.03.09

* **Security and data Integrity:** The data business is about storing, managing, querying, transforming, and analyzing customer data and, without data integrity and security, we would have nothing of value to customers. No feature matters more than data integrity and security and failures along either of these two dimensions are considered the most serious. Our reputation with our customers and within our company is dependent upon us being successful by this measure, and our ability to help grow this business is completely dependent upon it.
* **Code ownership**: Every line of code in the system should have a non-ambiguous owner and there should be a back-up within that department. Generally, each team should be organized roughly along architectural boundaries avoiding components spread over many or multiple teams. There should be no files "jointly" owned by multiple teams. If necessary, split files to get non-ambiguous ownership.
* **Design excellence:** Utilize the collective expertise of the entire team and, where appropriate, experience from other teams. Design for the long haul. Think through cross-component and inter-feature interactions before proceeding. Never accept quick hacks that we can't maintain over the long term and don't rush something out if it can't be made to work. A good general rule: "never promise what you don't know how to do." Of course, all designs must be peer reviewed.
* **Peer review:** Peer review is one of the most important aspects of the engineering process and it's through code and design review that we get more IQ involved on all important and long lasting decisions. All designs and all code changes must be reviewed before being checked in. Make sure your reviewer has the understanding of the code needed to a good job and don't accept rushed or sloppy reviews. Reviewing is a responsibility and, those that do an excellent job deserve and will get special credit. All teams should know who their best reviewers are and should go to them when risk or complexity is higher than normal. When there are problems, I will always ask who reviewed the work. Reviewing is a responsibility we all need to take seriously.
* **Personal integrity:** It's impossible for a team to function and be healthy if we're not honest with each other and especially with ourselves. We should be learning from our mistakes and improving constantly and this simply isn't possible unless we admit our failures. If a commitment is made, it should be taken seriously and delivered upon. And, when things go wrong, we need to be open about it.
* **Engineering process clarity:** The engineering process including Development, PM, and Test should be simple and documented in sufficient detail that a new team member can join the team and be rapidly effective. It should be maintained and up-to-date and, when we decided to do something differently, it'll be documented here.
* **Follow-through and commitment to complete:** In engineering, the first 10% of the job is often the most interesting while the last 10% is the most important. Professional engineering is about getting the job done. Completely.
* **Schedule integrity:** Schedules on big teams are often looked at as "guidance" rather than commitments. Schedules, especially those released externally, are taken seriously and, as a consequence, external commitments need to be more carefully buffered than commitments made within your team. One of the best measures of engineering talent is how early scheduling problems are detected, admitted, and corrected. Ensure that there is sufficient time for "fit and finish" work. Ensure that the spec is solid early. Complete tests in parallel. Don't declare a feature to be done until at least 70% of the planned functional tests are passing (a SQL Server specific metric that I believe was originally suggested by Peter Spiro), and the code is checked in. Partner with dependent components for early private testing. When a feature is declared done, there should be very few bugs found subsequently, and none of these should be obvious.
* **Code base quality**: Code owners are expected to have a multiple release plan for where the component is going. Component owners need to understand competitors, current customer requirements, and are expected to know where the current implementation is weak and have a plan to improve it over the next release or so. Code naturally degrades over time and, without a focus on improvement, it becomes difficult to maintain over time. We need to invest 15 to 20% of our overall team resources in code hygiene. It's just part of us being invested in winning over multiple releases. We can’t afford to get slowed or wiped out by compounding code entropy as Sybase was.
* **Contributing to and mentoring others:** All members of the team bring different skills to the team and all of us have an obligation to help others grow. Leads and more experienced members of the team should be helping other team members grow and gain good engineering habits. All team members have a responsibility to help others get better at their craft and part of doing well in this organization is in helping the team as a whole become stronger. Each of us have unique skills and experiences -- looks for ways to contribute and mentor other members of the team.
* **QFEs**: must be on time and of top quality. QFEs are one of the few direct contacts points we have with paying customers and we take them very seriously prioritizing QFEs above all other commitments. Generally, we put paying customer first. When a pri-1 QFE comes in, drop everything and, if necessary, get help. When a pri-2 or Pri-3 comes in, start within the next one or two days at worst. Think hard about QFEs -- don't just assume that what is requested represents what the customer needs nor that the solution proposed is the right one. We intend to find a solution for the customer but we must choose a fix that we can continue to support over multiple releases. Private QFEs are very dangerous and I'm generally not in support of them. Almost invariably they lead to errors or regressions in a future SP or release. The quality of QFEs can make or break a customer relationship and regressions in a "fix" absolutely destroy customer confidence.
* **Shipped quality**: This one is particularly tough to measure but it revolves around a class of decision that we have to make every day when we get close to a shipment: did we allow ourselves enough time to be able to fix bugs that will have customer impact or were we failing and madly triaging serious bugs into the next release trying to convince ourselves that this bug "wasn't very likely" (when I spend time with customers I'm constantly amazed at what they actually do in their shops – just about everything is likely across a sufficiently broad customer base). And, there's the flip side, did we fix bugs close to a release that destabilized the product or otherwise hurt customer satisfaction. On one side, triaging too much and on the other not enough and the only good way out of the squeeze is to always think of the customer when making the decision and to make sure that you always have enough time to be able to do the right thing.
* **Check-in quality:** The overall quality of the source tree impacts the effectiveness and efficiency of all team members. Check-in test suites must be maintained, new features should get check-in test suite coverage, and they must run prior to checking in. To be effective, check-in tests suites can't run much longer than 20 to 40 minutes so, typically, additional tests are required. Two approaches I've seen work in the past: 1) gauntlet/snap pre-checkin automation, or 2) autobuilder post-checkin testing.
* **Bug limits:** Large bug counts hide schedule slippage and the bugs count represents a liability that must be paid before shipping and large bug counts introduce a prodigious administrative cost. Each milestone, leads need to triage bugs and this consumes resources of productive members of the team that could be moving the product forward rather than taking care of the bug base. We will set limits for max number of bugs carried by each team and limits that I've used and found useful in the past are: each team limits active defects to less than 3 times the number of engineers on the team and no engineer should carry more than 5 active defects.
* **Responsibility:** Never blame other teams or others on your team for failures. If your feature isn't coming together correctly, it's up to you to fix it. I never want to hear that test didn't test a feature sufficiently, the spec was sloppy, or the developer wasn't any good. If you own a feature, whether you work in Test, Dev, or PM, then you are responsible for the feature being done well and delivered on time. You own the problem. If something is going wrong in some other part of the team and that problem may prevent success for the feature, find a solution or involve your lead and/or their manager. “Not my department.” is not an option.
* **Learn from the past:** When work is complete or results come in, consider as a team what can be learned from these results. Post mortems are a key component of healthy engineering. Learn to broadly apply techniques that work well and take quick action when we get results back that don't meet our expectations.
* **Challenge without failure:** A healthy team should be giving all team members new challenges and pushing the limits for everyone. However, to make this work, you have to know when you are beyond your limits and before a problem is no longer solvable, get help. Basically, everyone should step to the plate but, before taking the last strike, get your lead involved. If that doesn't work, get their manager involved. Keep applying rule until success is found or the goal doesn't appear to be worth achieving.
* **Wear as many hats as needed:** On startups, everyone on the team does whatever is necessary for the team to be successful and, unfortunately, this attribute is sometimes lost on larger, more mature teams. If testing is behind, become a tester. If the specs aren’t getting written, start writing. Generally, development can always out-pace test and sometimes can run faster than specs can be written. So self regulate by not allowing development to run more than a couple of weeks ahead of test (don’t check in until 70% of the planned tests are passing) and, if works needs to be done, don’t wait – just jump in help regardless of what discipline is in short supply.
* **Treat other team members with respect:** No team member is so smart as to be above treating others on the team with respect. But do your homework before asking for help – show respect for the time of the person whose help you are seeking.
* **Represent your team professionally:** When other teams ask questions, send notes, or leave phone messages ensure that they get quality answers. It’s very inefficient to have to call a team three times to get an answer and it doesn’t inspire confidence nor help teams work better together. Take representing your team seriously and don’t allow your email quotas to be hit or phone messages to go unanswered.
* **Customer Focus:** Understand how customers are going to use your feature. Ensure that it works in all scenarios, with all data types, and supports all operating modes. Avoid half done features. For example, don’t add features to Windows that won’t run over Terminal Server and don’t add features to SQL server that don’t support all data types. Think about how a customer is going to use the feature and don’t take the easy way out and add a special UI for this feature only. If it’s administrative functionality, ensure that it is fully integrated into the admin UI and has API access consistent with the rest of the product. Avoid investing in a feature but not in how a customer uses the feature. For example, in SQL Server there is a temptation to expose new features as yet another stored procedure rather than adding full DDL and integrating into the management interface.
* **Code Serviceability & Self Test:** All code should extensively self check. Rather than simple asserts, a central product or service wide component should handle error recording and reporting. On failure, this component is called. Key internal structures are saved to disk along with a mini-dump and stack trace. This state forms the core of the Watson return data and the central component is responsible for sending data back (if enabled). Whether or not Watson reporting is enabled, the last N failures should be maintained on disk for analysis. There are two primary goals: 1) errors are detected early and before persistent state is damaged and 2) sufficient state is written to disk that problem determination is possible on the saved state alone and no-repro is required. SQL Server helped force this during the development of SQL Server 2005 by insisting that all failures during system test yield either 1) a fix based upon the stored failure data, or 2) a bug opened against the central bug tracking agent to record more state data to allow this class of issues to be more fully understood if it happens subsequently. If a customer calls service, the state of the last failure is recorded and can be easy sent in without asking the customer to step through error prone data acquisition steps and without asking for a repro.
* **Direct Customer feedback:** Feedback directed systems like Watson and SQM are amazingly powerful and are strongly recommended for all products.
* **Ship often and incrementally:** Products that ship frequently stay in touch with their customers, respond more quickly to changes in the market to changes in competitive offerings. Shipping infrequently, tends to encourage bad behavior in the engineering community where partly done features are jammed in and V1 ends up being a good quality beta test rather than a product ready for production. Infrastructure and systems should ship every 18 months, applications at least every 12 months, and services every three months.
* **Keep asking why and polish everything**: It’s easy to get cynical when you see things going wrong around you and, although I’ve worked on some very fine teams, I’ve never seen a product or organization that didn’t need to improve. Push for change across the board. Find a way to improve all aspects of your product and don’t accept mediocrity anywhere. Fit and finish comes only when craftsman across the team care about the entire product as a whole. Look at everything and help improve everywhere. Don’t spend weeks polishing your feature and then not read the customer documentation carefully and critically. Use the UI and API even if you didn’t write it and spend time thinking of how it or your feature could be presented better or more clearly to customers. Never say “not my department” or “not my component” … always polish everything you come near.