MicroTools' Software / Systems Development Policy Manual Rev B

Introduction
This document provides the general guidance to be used at MicroTools (MTI) for developing the kind of systems and software that we design, develop and test. This can range from small embedded systems, software only embedded systems (MTI provides only the software) and medium size embedded systems.

Scope
This document applies only to MicroTools' projects. It is our desire that MTI achieve a SEI Maturity level of 4. This document is intended to help us achieve that.

Proposal / Acquisition Policies
Everything starts at the proposal stage. Most of our project failures (missed schedules, blown budgets, technical failures) can be traced back to inadequate planning at the beginning of the project when the proposal was written. This is understandable at one level since we do a lot of proposals and the amount of time that we can spend on each is limited. But with that in mind, we need to get better at that.

The Proposal Process
When a request for a proposal comes in, it is the responsibility of Program Management to assign this. If a request does not come into Program Management, it should be forwarded to Program Management as soon as possible. The assignment needs to include:

a. Who will be the lead on this proposal?
b. Who may need to provide input?
c. When is the proposal due?
d. How much time do we want to spend on it?
e. What are the proposal deliverables?
f. Is the proposal time billable?

Logistics
All proposals should be produced in Word format and delivered to the customer as a PDF. Email proposals should be avoided. The proposal (and all the supporting documentation) should be stored on the server under the name of the customer and the proposal directory (e.g /Company/proposal). The proposal should be modeled after the proposal template found on the server /mti/Doc Templates/proposals.

All proposals shall have a date and revision associated with them.

All proposals shall include:
Cost
Schedule

Specification
Statement of Work
Assumptions
Billing Milestones

Costing

Estimating the cost of a project during the proposal stage is a difficult process. All MTI employees who do proposals should be trained in high level estimating principles as well as the four techniques used at MTI. Appendix A and Appendix B provide the beginning of that training process. Appendix A looks at estimating from 5,000 feet. Appendix B is an excellent outside source of information about estimating. In addition it describes four of the techniques that we use at MTI for estimating project costs.

For all MTI proposals, two methods of costing shall be attempted. If a separate project developed the specification, an easy check is to use the Phase Distribution method.

Specification

Every proposal shall have a specification associated with it. With as much detail as possible in the time allotted, the proposal writer must document the “haves” and (if necessary) the “have nots” of the project. “Have nots” are explicitly listed when it may be something that is expected but will not be provided. For example, the ability of the software to be updated over the air may be assumed with a device with wireless capability. But if it is not being covered, it should be explicitly excluded. Here are some of the things that should be considered as part of the specification within a proposal:

System Specific Specification Items

a) Identify the extent and scope of any specifications either implied or explicitly called out in the RFP. These can be customer specifications or external industry standards. For example, medical devices can invoke a specific alarm standards (60601-1-8).
b) Identify system safety concerns. If a medical device, identify the class that you are going to assume (per 62304).
c) Identify the certification requirements for the project. Does the customer require UL/safety. What FCC unintentional emissions class does it fall under (residential or business)? If wireless (WiFi, Bluetooth, Cell modem, proprietary RF) what FCC standards will the unit need to be designed and tested to.
d) If the customer has a specification(s), the proposal shall include a compliance matrix.

Hardware Specific Specification Items

a) All of the I/O (analog / digital / serial). When there are unknown quantities, cap the number (e.g. up to 10 analog channels)

b) All networking interfaces
c) Special interfaces: serial flash, serial number chip, I/O expander, etc
d) Power sources
e) PCB board configuration resistors
f) If a battery is part of the system, the charging characteristics (e.g. trickle charging is simple but not always acceptable; rapid charging is more complex); the type of battery technology
g) The physical constraints of the circuit board (must it fit into a dime sized circuit board?)
h) The number of circuit boards
i) A block diagram of the system

Software Specific Specification Items

a) The number of separate software configuration items (separate builds)
b) The boot time (time required to be operational)
c) Application software update requirements
d) If applicable (kernel and u-boot) update requirements
e) Number and complexity of screens for a user interface
f) When defining a user interface, users are used to power features on their phones so you need to explicitly list “have nots” like “No auto resizing” unless you are planning to support that.
g) Networking protocols
h) Networking response times
i) When interfacing with other undefined computers, bound the data by specifying for example: “No more than “No more than 10 alarms”
j) Alarm and or alert mechanisms (be aware of special requirements for medical alarms in 60101.8)
k) Error logging requirements

During the specification part of the proposal, it is wise to highlight the document exchange mechanism. Some customers have been surprised when they invoke “Doors” and wonder why we cry “Change of Scope.”

Statement of Work

All proposals shall include a statement of work that clearly identifies the tasks that will be performed by MTI. Where there is shared responsibility with the customer or other partners, that should also be indicated in the statement of work. For internal purposes it may be useful to break down the work into as many separate tasks as you can identify. The more tasks that you can identify during the proposal, the more accurately you will be able to cost the system. You may not choose to provide this to the customer, but it should be in your pre-delivery versions of the proposal.

For example, designing a circuit board may consist of 14 steps in your work break down. You may want to list these separately to allow you to cost them separately.

If the project includes communications to a server, special care must be given to how debugging will be handled and who is responsible. Building a test server to specification is often easier and can eliminate finger pointing at integration.

Assumptions

A list of assumptions shall be identified. These are critical to our success in making fixed price proposals. Often a customer has one set of assumptions (e.g. "I assumed that you would cover the costs of EMI testing.") and we have another. As many of the specific assumptions that can affect cost and schedule should be listed.

The proposal boiler plate has the list of standard assumptions that should be reviewed and included as appropriate.

Deliverables

The proposal shall list all of the deliverables (schematics, software documentation, hardware prototypes, source code and executable code).

Schedule

The proposal shall have a schedule with billing milestones. For projects that exceed $10,000, the billing milestones should be front loaded and be spread out in such a way to facilitate cash flow.

Billing milestones that depend upon the customer should have a fall back should the customer fail to deliver. For example, if the customer fails to test the delivered software, within 90 days of delivery, the billing milestone should be met by default.

Time and Materials Projects

Many projects are performed on a Time and Materials basis. A statement of work agreed to in writing by the customer shall be put in place for all T&M projects. Appendix C provides a sample. A template can be found on http://www.microtoolsinc.com/Articles/Statement%20of%20Work%20Template.docx

Software/Systems Development Policies

Project Turn-On Phase

The normal way projects are started at MTI is with a project turn-on with a PO or verbal (email acceptable) from the customer. No projects can start without something in writing. For select and existing customers, project turn-on can come from a verbal. Once the PO or turn-on is received [and for new customers the first down payment], the following things kick off the project.

Appoint a project lead - All projects initiated at MTI shall have a project lead appointed by Program Management.

Define the team – Additional members of the team are added at this time with an estimation of % of time expected to work on the project and when.
Billing ID - Program Management shall establish a project ID for purposes of time card tracking that includes Customer:project. Program Management shall determine if subprojects shall also be identified, Subprojects IDs can be added at any point to assist in obtaining metrics.
Kick-off meeting – Program Management shall meet with the project lead and any team members to assure that the proposal guidelines are met.
Develop a Project Plan – All projects shall have a project plan modeled after IEEE Std 1058-1998. A template for the MTI project plan can be found at http://microtoolsinc.com/articles/ProjectPlan.docx

Develop a high level Test Plan – At the start of every project, the project lead shall assess how the system is going to be tested. Although much of this work should be done during the proposal, more detail can be added at this phase.

Specification Requirements

All software systems designed at MicroTools shall have a software requirements document. If there is hardware, a systems requirements document is also required. These documents are to be written in accordance with the guidelines provided on the web site:

Software and Systems Requirements Specification
http://microtoolsinc.com/Howsrs

Requirements Review

All projects shall have at least one requirements review. Incremental requirements reviews should be encouraged. Minutes and action items shall be taken. The proposal will indicate if the customer needs to be invited. In general, the customer should be invited to attend.

Architectural Design

All projects shall create an architectural design document. All project leads shall be familiar with IEEE 42010. This document describes the architecture as:

> Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution

Most of our code should be self documenting. The purpose of the Architecture design document is to provide a road map to the system/software. For example, what is the relationship of the system to its environment? How many concurrent threads / tasks are included in the design (the elements)? What is the mechanism for Inter Process Communications (IPC) used in the system (the relationships)? What are the principles of design incorporated into the system (e.g. the requirements dictate a hard dead-line real time thread which we have

met by using the real-time extensions of Linux). A pictorial diagram of the system is often useful in describing the architecture of a software system.

Appendix D provides a sample Architectural Design diagram.

Design Review

Generally prior to coding and prior to laying out the board, software designs and hardware designs shall be reviewed by at least one other senior member of the MTI team. Multiple design reviews are encouraged. Appendix E provides a check list for these design reviews.

Coding Standards

All projects shall identify a coding standard in the project plan. These coding standards will be self-enforced. Since the bulk of MTI's software is written in C, there existing a C Coding standard that can be adopted for most projects. See http://www.microtoolsinc.com/Articles/MicroToolsCodingStandard.pdf

Code Walkthrough

All software shall be walked-through by at least one other technical member of the MTI team prior to shipping the software. The exact placement of code walk-throughs in the schedule can vary from project to project.

Testing

All projects shall, at a minimum be tested to a written test plan for final test. This test plan should be written based on the Systems and Software requirements. See http://www.microtoolsinc.com/howstp for general guidelines. The project plan may call out that the testing will be performed by the customer.

Module testing is a requirement for Class B and C medical devices. For other projects, it is up to the project lead to decide on the applicability of module testing.

Module testing may be performed by the same person who wrote the module unless specifically restricted by a higher standard. Final testing shall not be performed by the same person.

Test Readiness Review

Prior to the start of final test, the project lead shall call a Test Readiness Review where the following items shall be addressed and / or reviewed:

1. Known bugs shall be disclosed to the tester(s)
2. Unimplemented requirements
3. Test resources required
4. Test plan

Release

All software that is made available to a customer requires the following:

1. All of the source code is backed up
2. Release Notes are provided to the customer

Additional requirements shall be put into a check list for each project. See http://microtoolsinc.com/articles/ProjectPlan.docx for an example (also Appendix F)

Additionally, if it is a final release a Version Description Document shall be provided. See http://www.microtoolsinc.com/Articles/VDD2.pdf for the guidelines that shall be followed.

Change Board

Once software has been released to a customer (a formal release) that the customer deploys to the field, any changes to the software shall be presented to a Change Control Board. The personnel make of the CCB shall be defined in the project plan.

Configuration Management

All released software shall have a means of identifying the version and if applicable a sub version. The form may vary from project to project. Informal alpha and beta releases may take the form of a dated release. Dated releases shall include the target version and a date that includes day, month, and year in some form.

MicroTools projects should take advantages of source control tools like git whenever possible. Although not required, it is strongly recommended.

File Naming Conventions

All files that make up a project should have meaningful names. The directory name can be part of that meaningful name. When a directory has some tool or test driver or something that would require some user instructions, there shall be a readme.x file where x is an appropriate extension for a text file, a word document, a PDF, etc.

Backup Requirements

All software, schematics, project files, makefiles, etc shall be backed up locally at MicroTools and off sight. The project lead shall check the status of this backup at least monthly.

Meeting Minutes

All meetings where decisions are made or action items assigned shall include minutes. Minutes shall be emailed to all participants and interested parties. The

minutes should summarize the decisions made during the meeting. Any action items should be assigned to as specific person at MTI and as a minimum a client company (if it is up to the company to decide who handles the action item) or the client's employee. All action items shall have a date as to when it is required.

Lessons Learned

At the end of each project, the project lead shall call a lessons learned meeting. All participants in a project should be invited. The goal of this meeting is to improve our future projects.

Hardware Development Policies

Specification Requirements

When developing hardware, a system block diagram shall be created to provide a high level description of the board. If applicable, this diagram should be in the System Requirements Specification.

The hardware designer shall maintain a folder accessible to all members of the team where all of the applicable data sheets are stored.

All schematics shall retain a rev level and change history. The board rev shall if at all possible be included in the silk screen.

A Bill of Materials (BOM) shall be generated for all hardware designs.

The schematics shall included the part number of the component wherever possible.

If the electronic hardware has software on it, the board should have id resistors unless there is no need for the same software to run on multiple designs.

Any traces that will carry more than TBD ma, shall include the maximum current capacity required.

All schematics shall be backed up locally and off-site.

PCB

TBD

Appendix A - Estimating Your Embedded System's Project The Challenges of Estimating Software Projects

Introduction

This morning at our 8:30 staff meeting, I asked one of the project leads if we were still planning on shipping a software update for an embedded system by 11:00 AM. In essence he reported that barring the sky falling or the final test failing, it should be released. The release was shipped at 11:14 AM. His response reflected the challenge we all face in making estimates for completing tasks we are assigned.

Accurately estimating my arrival time for dinner can be challenging. Accurately estimating embedded software development is almost impossible. Don't let anyone fool you into thinking that this is achievable in the real world. But we are not without hope. We will look at some of the general problems associated with estimating software development. In the next section we will look at specific problems unique to embedded software development. We will close out the paper discussing what we can do to continuously improve our estimating skills.

Learning how to estimate the time and dollar cost of designing an embedded system is very important to our business. We make our living by designing and building embedded systems for other people. We do this as fixed priced projects. We live and die by our estimates. Perhaps we are like one of Nassim Taleb's black swans[1] and the fact that we are still in business is just a statistical anomaly. But I think we have learned enough about estimating that, as imperfect as it is, we are able to estimate accurately enough to stay in business. Learning to estimate the time it takes to design embedded systems may not be that critical to you staying in business, but it is an important skill even if you work for someone else. Let's dive in by trying to identify some of the problems.

The General Problem

Asking you to estimate software development time is like asking you to estimate the number of pens I have in my desk. You could probably bound the problem on the lower side. "Well it is not less than zero." You could probably put an upper bound on it based on the estimated size of the average desk drawer and the size of an average pen. You could probably assign some reasonableness factors to it. "Bob would not have 1,000 pens in his desk." But how accurate can you be with so little information. In software, without actually doing the design, you are working with even less data when you estimate. Software systems are the most complex things we design on the planet. And the complexity increases

1 The Black Swan – the Impact of the Highly Improbable Nassim Taleb – This is essential reading for anyone who wants to be able to predict the costs of developing embedded systems

exponentially as the systems get bigger. So our first axiom is that estimating the cost to develop software is extremely difficult.

As engineers we are well aware of an old axiom when applied to electronic systems design:

You cannot control what you don't measure.

If you are asked to control the temperature of a room but cannot measure the temperature you will fail. Tom DeMarco in his book Controlling Software Projects applied this well worn chestnut to software estimating. It was 1983. He introduced me to the concept of software metrics. Unless I was measuring what it actually took to design and develop software, I would not be able to estimate what it would take to design and develop my next project. This was a pivotal step in my professional growth. Two of the major points of his book were that we poorly estimate software development time because: we don't develop estimating expertise; and we don't base our estimates on past performance. I set out to correct that. I wanted to develop expertise in estimating. I wanted to understand past performance.

With Tom's principles as my mentor, over the next seven years, my organization generated all kinds of metrics (some of which we will touch on in the last article) and used them in estimating software development costs. But there was a problem. After all that, I wasn't getting any better at estimating even though I had a lot of data. I developed my own corollary to DeMarco's axiom:

And even if you do measure it, it doesn't mean you can control it.

I needed a new mentor. I found one in W. Edwards Deming. Deming arguably was the father of statistical quality control who is credited with single handedly bringing Japan's economy out of its World War II disaster to become the 3rd largest economy in the world. Concerning quality control, he taught that:

"The most important things cannot be measured."

I began to perform post-mortems on projects (a highly desirable practice for anyone who wants to continue to improve their software development process – including estimating). I tried to understand where the estimate went wrong. I had lots of examples. What I found was that Deming was correct and that the things that were sinking the schedules and making the estimates look bad were, for the most part were things I could not measure.

Deming was also famous for saying that in quality control:

"The most important things are unknown or unknowable."

Specific Problems with all Software

Let's look at just a few of the unmeasurables, unknowns and unknowables that caused my estimates to go wildly wrong:

Not Clarifying or Misunderstanding Requirements

In 1993, we started the largest software project in our history at that time. There was one little line in the statement of work that said: "Provide reports for the data." This was an embedded system running on an Intel 80188. It had 512k of EEPROM. How many reports can this be? We estimated that it would take us 80 hours to create these reports. It took us over 200 hours to generate the 100 different reports that the customer had in mind. Here was a requirement that was knowable but we didn't clarify it. That made it unknown to us.

Misestimating Memory Requirements

If you have to start playing games to shoe-horn your code into the memory allowed, you can blow your schedule completely out of the water. In 1974 a professor in one of my classes said:

> "In the very near future, you will never have to worry about memory constraints again."

Now this professor was smart. But he was dead wrong. Hardly a project goes by my desk that still doesn't concern itself with either RAM or ROM memory usage. In 1997 we were porting an embedded system to a DOS PC. Because we misestimated the memory requirements, we were forced to convert the program to use overlays (basically keeping only some of the program in memory at a time – something that happens automatically now). That part was easy. The problem was that the overlays didn't work in a multi-tasking OS that we put on top of DOS. It took two engineers a month working close to 80 hours per week to solve.

The memory required for a software project is an unknowable unless you have designed it or something similar before.

That Elusive Bug

I have more stories than there are pages in this magazine about one tenacious bug that took almost as long to find and fix as the entire estimate. This happens a lot. In 2006, we were about to release an embedded system on time and within budget estimates. Then during the last weeks of testing, a problem surfaced in the Linux C library that was difficult to duplicate and even more difficult to fix. We spent more than a calendar month and several man-months to correct this bug. How could we have estimated for this unknown?

The Other Guy

There are a class of estimate defeaters that I call "The Other Guy." The other guy's API doesn't work the way we expected. The other guy's web server doesn't respond quickly enough. The other guy's device doesn't meet

specifications. On a 2010 project, we were using a flow sensor to measure oxygen content and flow. This sensor worked in most cases but sometimes didn't. Some sensors worked flawlessly. Some did not. This was unknowable to us when we performed the estimate for the proposal. We spent many times our original estimate in getting it to work.

Hidden Complexity

Very often when an estimate is made, there are lurking under the surface a vast array of hidden complexities that can only be uncovered by implementing and testing. Fred Brooks puts it this way in The Mythical Man Month:

> "… the incompleteness and inconsistencies of our ideas become clear only during implementation"

This week I was reviewing the specification for an interface to a new cell modem module we were including in one of our designs. All of our designs work on very low cost data plans. This requires us to use very little data bandwidth every month. One of the cell carriers has a very strict and very clear set of requirements about handling retries. Retries can be very costly in data plans. My specification called out for the software to meet these very strict and very clear requirements. It should be easy to estimate how long it will take to implement that algorithm. But during the review, one of the designers asked: "How do we manage the retries handled by the TCP/IP stack underneath us?" We all knew that the TCP/IP logic in our embedded Linux systems performs retries. How will we make that work with the carrier's retry requirements? This is an unknown with lots of hidden complexity and could easily add an order of magnitude to the effort required to implement this. In this case, we happened to think about this complexity but that doesn't always happen.

Programmer Efficiency

We all know that different people take different amounts of time to do the same job. Joel Spolsky in his book Smart and Gets Things Done claims that his research shows that there can be more than an order of magnitude difference is programmer efficiency. Okay, so you take that into account when you estimate a job based on the programmers you have. But a programmer leaves or gets sick. You have to use your least efficient programmer. Now you could easily see your estimate go out the window by an order of magnitude. The most important things are unknowable and unmeasurable.

Optimism and Hubris

These non-identical twins are one of the reasons we fail to accurately estimate our software projects. We always think we can get better. Because of our pride we can easily fall prey to wishful thinking. I know I can do it better this time. Research has demonstrated that this is a pervasive problem in software estimating. Related to this is what is called "Hindsight bias." When combined with optimism and hubris we just think that we did better in the past than we actually performed.

Conclusion

Recognition of the problem is the first step at correcting it. We have named a few of the trees in the forest of problems with estimating software development costs. Tom DeMarco has said:

> *An estimate is the most optimistic prediction that has a non-zero probability of coming true*

The problem for us is that the probability of most of our estimates of coming true is very close to zero. We have to learn how to do it better.

Specific Problems with Embedded Software

Introduction

Recently, we were asked to estimate the cost to develop a system that would interface with every device of a particular type manufactured in a particular country. Our job was to design a system to extract data from these devices. There are 10 international standards applicable to these devices in this country. These standards define the protocol for accessing this data. The data is available on one of three possible hardware data busses. Some of the data available on these busses is in the public domain and some is only available from the manufacturer. There are over 150 different types of these devices sold each year in this country. Each year, another 150 new or similar types are sold. The specification is perfectly clear. By the way, can you have the estimate to me by Monday? Hmmm! I saw an advertisement the other day that said that their one day seminar would teach me to accurately estimate firmware schedules. Maybe if I took that class on Friday, I could get the accurate schedule by the end of business on Monday.

How can one estimate something like this? Here is axiom number one. Don't believe anyone who tells you he can teach you to accurately estimate your firmware schedule in a one day seminar. Or in a one week seminar. Or even in 10 years of doing it every day. Accurate? No! But we can get better. And the best way I know how is by first defining the problems. That has been the focus of these first to articles in our series.

Last time we looked at the general problem of estimating software development costs. This time we will look at the challenges that are unique to embedded software development. Certainly there are things that make embedded software more challenging to develop than other types of software. But what makes embedded software that much harder to estimate?

The Surface Area is Much Bigger

I reviewed section one with our team and asked the question: Why is estimating embedded systems more difficult than estimating other kinds of software? One engineer said: The surface area is much bigger. What he was saying is that all of the standard problems with estimating just got multiplied. Let's just review what we said in the last section and see how some of these issues are more complicated for embedded systems.

Not Clarifying or Misunderstanding Requirements

The accuracy of our software estimates can only be as good as our understanding of the requirements. This difficulty is multiplied with embedded systems because of the complexity of the interfaces. In addition, there are a lot of requirements that only become clear after you implement. The data sheet of a small microprocessor we use on one project is 1400 pages long. There are just a lot more requirements that can be unclear or misunderstood. We approved a re-work once to one of our designs that required the manufacturer to add a wire to one end of a capacitor. After the first few thousand were shipped, the capacitors started shorting [especially problematic for by-pass capacitors!]. Buried in the capacitor's data sheet was the requirement to not touch the capacitor with a soldiering iron. The re-work needed to be performed with a hot air process. It was very clear on page 78 of the capacitor's data sheet!

The specifications can also be wrong. Many times errata come out after you have started your design. We once missed an errata in an 800 page microprocessor data sheet that said "Oh by the way, this device has a 256 megabyte address range but can only address 16 megabytes of NOR flash!"

That Elusive Bug

Embedded real-time systems and systems with concurrency make debugging much more difficult. That we can plan for. But those elusive bugs that take 2 weeks in non-embedded systems can take 2 months on embedded systems because your tools are not as powerful and the complexity of the design is that much greater.

Hidden Complexity

The scale of complexity is greatly multiplied in embedded systems. We are supposed to write software that interfaces with other very complex devices. Take this simple requirement from a data sheet of chip we interface with.

> To reset the chip, hold RESET_N low for 300-500 ms.

On the surface that seems straight forward. But what is hidden and not written in the manual is that if the RESET_N is held low for more than 1000 ms, the chip powers down and will not start when the RESET_N line is brought high. If for some reason your function that releases RESET_N gets delayed, the chip would not become operational as you expected. This requirement of raising RESET_N becomes a hard deadline that you might not expect to be as such. These kinds of hidden complexities are legion in embedded systems.

Programmer Efficiency

Two years ago I sat with one of the best embedded designers I know. He was running out of real time on a project. The problems were so complex that it took two of us with a combined experience of 60 years of designing embedded systems to figure out what was going on and how to fix it. Where a less efficient programmer might be 4 times less efficient than your best designer, in an embedded environment that same programmer might be 10 times less efficient.

Optimism and Hubris

A couple months ago one of our customers asked us to add a splash screen and a progress bar to the start of a device. One of our best designers saw that u-boot had hooks for sending an image to our display. Linux had a progress bar app (psplash) that worked with our display. The system was built on a BeagleBone architecture so others must have done this before. The on-line community support for this architecture is huge. We knew we could get lots of help. In addition, we have done similar projects in about 4 days without this kind of support. We know how to do this. We can deliver this fully tested in 4 days. At the end of 4 days we found that the hooks in u-boot didn't work. No one in the on-line community knew how to make them work. At the end of 2 weeks we discovered that the u-boot image was inverted from what the Linux driver was expecting. At the end of 4 weeks we discovered that the progress bar did not play well with this particular display. At the end of 6 weeks we discovered that the customer did not provide us with the right code base to start with. We were optimistic. Embedded systems will amplify the negative effects of your optimism and hubris enough to put you out of business.

Customer schedule creep

Customer schedule creep is a specific instance of "The Other Guy" problem we talked about last time. But it has a unique feature to it. We are now 6 months behind schedule releasing a new version of embedded software for a product we designed for a customer. One of the driving factors in the delay is that the customer still doesn't have their portion of their web server operational. Every day it slips, our team has to work on other things instead of completing the testing. Each day the team might spend a half hour coordinating with the customer. None of this 60 hours was estimated. The inefficiency of this schedule creep is even more costly. Fred Brooks in The Mythical Man-Month puts it this way:

> Disaster is due to termites, not tornadoes

Some of this is common to non-embedded software. But embedded software by its very nature is embedded in stuff. And often, stuff that is being designed in parallel. As a minimum it must talk with hardware that is often not completely designed. It may also talk with other machines that are being developed in parallel. How well those are designed and when they are delivered can be a multiplier in the schedule and cost of an embedded system.

Quality of our partners

Another instance of "The Other Guy" problem is with your partners. Some of the partners we interface with are the hardware we run on; the busses we communicate on; the networks we connect to; the other devices we talk to; the hardware designer who designed our board, the hardware layout team that laid out the PCB and the hardware build team that actually built the board. How well they do their job has a direct bearing on how much it will cost you to develop your embedded system.

Let me share two examples. We have a supplier who builds our printed circuit boards and assembles them during our development stage. We love this supplier because their work is impeccable. Sometimes our customers require us to get the boards built someplace else or by them. Invariably parts are put in backwards. Ball Grid Arrays (BGA) parts are not x-rayed to verify their connections. Flow soldering techniques cause modules to reflow and not re-center on their footprint. When we get the boards it might take us 2-3 days more to debug and trouble shoot these problems because of the supplier. Remember that we are checking out a new design which can have flaws in it as well. How does one estimate for that extra 2-3 days. You don't know the quality of that supplier until you have used them.

Another problem we have is with other hardware designers. When we design the boards, we know the quality factor of our designers. They may not be perfect but they are a known quantity. We know by experience how long it will take to integrate the boards designed by our own people because we have metrics and experience. But what if you are designing embedded software that runs on a board that is designed by "the other guy?" Our experience shows that it can ruin a schedule in two ways. First is the extra time it takes to "bring the board up" because there are more errors in the design than you are used to. This can easily add several weeks to a schedule. But often we find that it takes more turns of the board than it normally takes you to get an operational board. During that extra 2-5 weeks your team is much less efficient. Do you assign them to a new project? That is not practical. So the software team becomes less efficient. They work on "cleaning up the code" and "doing some documentation." Sounds good but these are schedule killers. And for estimating, the problem is: how do you know in advance what the quality factor is.

Testing difficulties

Embedded systems are much more difficult to test than conventional software systems. That additional difficulty can be planned for and the estimate adjusted to take that into account. The problem comes when we don't think through these difficulties when we estimate the project. We developed a tiny embedded device that was implanted into a human body. This device communicated to the outside world via infra-red. The device sent 8 bytes every millisecond. We accurately estimated the time it would take to design the hardware and the software necessary to accomplish these requirements. However when it came to test it, we did not have a means to easily do that. There were no off-the-shelf tools to read the IrDA and provide an integrity check to it. How does one know that all 8000 bytes are correct every second? A special test tool was needed to display and analyze that it was meeting its requirements. But special test tools take time and money to design. They can drastically expand the effort required to design and develop an embedded system.

Another thing that can affect our ability to estimate embedded system is the time delay inherent in many designs between making a change, testing the change and reprogramming the device. When the time delay is very small (as in non-

embedded systems), iterative designs can be created much more quickly. Where this impacts our estimates is that we often don't know what the time delay is and exactly how it will impact the schedule. For example, let's imagine that over the course of the project you make 1200 changes to your software requiring a compile and load. If the compile and load time takes 70 seconds compared to 10 seconds, this can add 3 extra days to your project. Often, during the time we estimate, we don't know with that precision the compile and load time.

Conclusion

The surface area of complexity in estimating embedded systems is many times more complex than designing non-embedded software. Knowing what some of the problems are can help us get better at this impossible task.

Approaches to Estimating Embedded Software Projects

Introduction

"People who spend their time, and earn their living, studying a particular topic produce poorer predictions than dart-throwing monkeys."

This quote from a Nobel prize winner who knows his stuff is a jarring introduction to our final installment in our article series dealing with estimating the cost and schedule of our embedded software systems. Is this whole process of software estimation no better than what dart-throwing monkeys could come up with? In the opening article I said that accurate estimating is extremely difficult. But I also said that there is some hope. This month I would like to provide some thin slices of help for anyone who is asked to estimate how many man hours it will take to create an embedded system or even a part of an embedded system. And the help will come from the author of that quote.

Daniel Kahneman is a psychologist who won the Nobel Prize in Economics in 2002. The above quote is taken from his 2012 book entitled Thinking, Fast and Slow which summarizes his decades of research on the psychology of judgment, decision-making, and behavioral economics. Kahneman has provided some key insights into how we approach the impossible task of estimating costs of developing embedded software systems.

This article will not delve into function points, use-case points, software metrics, COCOMO models, SEER-SEM or any of the other methods for estimating software. These are good, useful and well documented in the literature. We use function points, use-case points and software metrics in our company. I would heartily recommend that you understand function points[2], use-case points[3] and

2 A place to start with function points is the International Function Point User Group http://www.ifpug.org/ But you can also just google something like "function points per hour."

3 The following article http://www.methodsandtools.com/archive/archive.php?id=25 will provide a useful introduction to using use-case points in estimating.

develop software metrics. In particular, develop software metrics that relate to your experience of function points or use-case points. In other words, count the function points or use-case points during the estimation phase of 3-4 projects and see what you learn about yourself, your estimating process and your projects. Go back to completed projects and determine the number of function points or use-case points and factor that into a metric.

In this section, I want to look at estimating from 5,000 feet. I want to see how Kahneman's research can help us become better at estimating software. The stated purpose of his book was, in his words, to "learn to recognize situations in which mistakes are likely and try harder to avoid significant mistakes when stakes are high."[4] If we can glean that from his book, we will become better at estimating the costs of embedded software systems.

Kahneman proposes almost 50 heuristics in his book. A heuristic is a method or process that enables us to learn something on our own. Only a few of them are applicable to us in estimating software. But if we can master them, they will enable us to become better at estimating embedded software.

Four Heuristics

Priming

Sometimes one of our customers will tell us that a project needs to be completed in three months. Or that it needs to be completed for under $15,000. These numbers can have a very bad effect on our ability to accurately estimate a software project. I am amazed how often my estimate closely parallels the customer's estimate. Can it be that the customer really knows how long it is going to take or how much it is going to cost? Or is something else going on?

A heuristic discussed by Kahneman in his book is called priming. He and other researchers have demonstrated that our behavior can be primed by what goes immediately before us. For example, he sites one study, where two groups of young people (aged 18-22) were asked to form some sentences from a set of five words. One group had a set of five words associated with the elderly. After the exercise, the young people were asked to walk through a corridor. Those who worked with words about the elderly walked slower than the other group! Experiments like this have been repeated many times with a wide variety of different priming mechanisms. The evidence seems to indicate that we are deeply influenced by priming.

How are we to use this knowledge about ourselves to become better at estimating? First, as much as possible, we need to avoid obtaining from our customer or bosses expected costs and schedule before we make our estimates. Estimating is difficult and I really want to know what the customer or my boss

4 Thinking, Fast and Slow, Daniel Kahneman 2012 page 28

expects me to estimate. But resist the urge. Priming is a powerful and proven effect and we must avoid it as much as possible. Watch out when your boss tells you that he needs this in two weeks and then asks you to estimate it.

If however, the cat is out of the bag, we need to make an extra effort to not let that number influence us. This is by far the harder of the two options. Once primed, even when I take herculean strides to not be influenced, the priming has its effect. But at least I am aware of the effect. Develop your estimate with your usual method of function points or use-case points or whatever, and if it comes in the same ball park as the "primed" number, be wary of your numbers and extra-vigilant – run your numbers again.

Anchoring effect

I have noticed that a lot of my estimates seem to be remarkably similar to previous estimates. Could it be that my projects are so similar that it always takes 40 hours to write the software specification for all projects? Or that User Interface designs always take 160 man hours. Or is something else at work. ?

One of the heuristics Kahneman has identified is what he calls the anchoring effect. The anchoring effect "occurs when people consider a particular value for an unknown quantity before estimating that quantity."[5] With serious academic rigor, Kahneman demonstrates how we are influenced by previous numbers. For example, he tells us that if we were asked if we thought that Gandhi was 114 years old when he died, we would immediately say "No." If we were then asked how old we thought he was when he died, our number would be higher than if we were first asked if we thought Gandhi was 35 years old when he died. That first number acts as an anchor to pull our estimate in its direction. Kahneman's claims that this phenomenon is "one of the most reliable and robust results of experimental psychology: the estimates stay close to the number that people considered [previous] – hence the image of an anchor."[6] This means that we will be affected by it when we do our estimating.

Anchoring is closely related to priming. I would make the distinction that priming involves numbers related to the estimate (the customer's estimate for the same project). Anchoring happens when I take numbers from an unrelated project into account before I make my estimate.

How can we take this knowledge and become better at estimating? Here is where software metrics come into play. Look back over the last several estimates of unrelated projects. Were the estimates similar to each other? How did the actuals compare to the estimates? If you see a correlation with the estimates but not in the actuals, anchoring is a possible cause. Develop a range

5 Ibid Page 119

6 Ibid Page 118

of estimates based on actuals and use these when making a new estimate. For example, imagine that the user interface took 120 hours on project A, 130 hours on project B, and 250 hours on project C. Attempt to identify the similarities and the differences and place a weighted value to each. How many menu items or screens were involved? Was it a graphical interface? Was a working driver provided? Was it a touch screen or keypad or both?

To avoid the anchoring effect on new estimates, we need to ruthlessly dissect previous projects into their subcomponents using software metrics. In other words, keep track of how long it took to write the specification, design the user interface, design the manufacturing test fixture, etc. Then when we approach a new project we need to make a quantitative comparison between the smaller elements. For example, the user interface is about twice as complex as project B and half as complex as project C. This quantitative approach can help us avoid the anchoring effect.

Optimistic bias

Kahneman describes many biases that affect our ability to estimate. He posits that the optimistic bias may be the most significant. In my earlier articles in this series, I discussed optimism as it relates to estimating but it bears repeating. I would recommend reading chapters 23 and 24 of Kahneman's book in an attempt to hammer home how pervasive this heuristic is and learn to make adjustments.

How do we counter this optimistic bias? I would say that a thorough knowledge of our optimistic bias is a good start. Software metrics can help if you develop actual numbers and then compare them to your estimates during a post-mortem. But human nature such as it is, unless we learn to develop a sort of "humility before the data" we can ignore the stubborn facts the metrics show us. A simple mantra that could be said after you have completed your estimate and before you submit it, is to repeat these words:

> All evidence shows that I am repeatedly over optimistic in what I think I can do. How should this estimate change based on that?

Small sample size

We all know that using a small sample of data sets us up for errors in estimating or drawing conclusions. But Kahneman takes us to a new level of awareness of the danger of using small samples. For example he cites a study of the incidence of kidney cancer in 3,141 counties in the United States. The counties with the lowest incidence per capita are "mostly rural, sparsely populated, and located … in the Midwest, the South and the West."[7] Upon hearing this most of us immediately start jumping to conclusions. But he goes on to also state that counties with the highest incidence per capita are also mostly rural, sparsely

7 Ibid Page 108

populated, and located in the Midwest, the South and the West. I leave it to you to figure out why sample size is the reason for this apparent contradiction.

In using our software metrics to estimate embedded software systems, we have to recognize that we have an extremely small sample size that we are drawing upon. I have been in this business since 1973. I have estimated almost a 1000 such projects. Yet even with all that experience, that is an extremely small sample to accurately predict how the next project is going to go.

So what can be done in light of this last heuristic? I recommend that you doggedly pursue from other companies the results of actual projects. Most companies are not willing to part with this information. But I have found that it doesn't hurt to ask. In non-competing situations, we can learn a lot as we expand our sample. There are a lot of numbers floating around the web. Take the time to create your own data base of "the other guy's" actual development time.

On one project, we had expended an immense amount over our original estimates. After the project we found that another company designed a very similar product and took 2-3 times as much calendar time and cost. Had we had that number in the beginning, we may have been more accurate in our original estimates.

Although your environment is unique to you and your company, industry standard metrics like hours/line of code and hours/function point or hours/use-case point can help expand your sample. These metrics are prone to many errors and are widely variable. Nonetheless they are another data point for your estimate. They help us limited engineers do the impossible: expand our sample without actually doing the work.

Conclusion

Accurately estimating embedded software systems is impossible. Don't let anyone tell you otherwise. Hopefully with some input from this series, you will become a little better at it than you were before. And that is no small accomplishment.

Appendix B – Mechanisms for Estimating

The following article provides a good baseline introduction to the methods that we use at MTI:

- Function points
- Phase Distribution Estimation
- Intuitive task estimating
- Participative estimating

http://www.ksinc.com/itpmcptools/EstimatingGuidelines.pdf

Appendix C Template for a Time and Materials SOW

Statement of Work

Bob Japenga
MicroTools, Inc.
714 Hopmeadow Street Suite 14
Simsbury CT, 06070

Contact Person
Company Name
Address
City, State, Zip code

Scope of Services: Provide _____ for the software development for the software used in the _[Describe product]_____ including:

Main Controller
Safety PIC
Motor Controller
Pendant
Primary User Interface
Datalogger

___[Describe any time limitations and schedule constraints. Describe any caps on the amount of time]_____ This SOW may be extended with an amendment with approval of both parties.

Fees and Payment Schedule:

- This is a time and materials agreement. MicroTools shall be paid $150 per hour for services rendered here under. Hours not to exceed the following estimates without prior written (email ok) approval of Client. Any additional work will be authorized via email or _____.
  - Describe any milestones

MicroTools Inc	CLIENT: Company Name
_____	_____
SIGNATURE	SIGNATURE
ROBERT JAPENGA	_____
PRINT NAME	PRINT NAME
PRESIDENT	_____PRESIDENT_____
PRINT TITLE	PRINT TITLE
_____	_____
PRINT DATE	PRINT DATE

Appendix D Sample Architectural Drawing

Appendix E – Design Review Check List

Appendix F – Release Check List

Release Date: Click enter text.
Product name: Click enter text. Product version: Click enter text.
Platform: Click enter text.
Software category: (all that apply)
◯ OS/kernel; ◯ kernel driver; ◯ utility/tool; ◯ installation; ◯ user executable; ◯ library
Release level: (select one) ◯ pre-release; ◯ formal release; ◯ final release;
Release purpose: (select one) ◯ prototype; ◯ for review; ◯ for testing; ◯ for operational delivery

Complete	Incomplete	N / A	CHECKLIST ITEMS	Comments
◯	◯	◯	Bugs: Reviewed and updated bug list	Click here to enter text.
◯	◯	◯	Bugs: Resolved all bugs of high priority or required for this	Click here to enter text.
◯	◯	◯	Code: Performed build-clean and rebuild of source code	Click here to enter text.
◯	◯	◯	Code: Removed debug-code / configurations	Click here to enter text.
◯	◯	◯	Code: Resolved Incomplete / unimplemented code (e.g.	Click here to enter text.
◯	◯	◯	Controls: Archived release executables / installation image	Click here to enter text.
◯	◯	◯	Controls: Updated and tagged source code in source code	Click here to enter text.
◯	◯	◯	Document: Updated Code headers / comments	Click here to enter text.
◯	◯	◯	Document: Address Operational security	Click here to enter text.
◯	◯	◯	Document: List Unsolved / outstanding bugs	Click here to enter text.
◯	◯	◯	Document: Updated all required documentation	Click here to enter text.
◯	◯	◯	Review: Reviewed code with peers	Click here to enter text.
◯	◯	◯	Test: Checked for memory leaks and usage	Click here to enter text.
◯	◯	◯	Test: Completed the test plan	Click here to enter text.
◯	◯	◯	Test: Tested software following installation instructions	Click here to enter text.
◯	◯	◯	Test: Completed unit testing	Click here to enter text.
◯	◯	◯	Test: Checked version reported by software is correct	Click here to enter text.

Check list competed / submitted by
_____
Reviewed by
_____

Release has been:
◯ Accepted
◯ Rejected