

Software Development 1

This handout gives my take on how to build a successful software project. This handout focuses on the basic axioms and issues that define the problem. Later handouts will focus (more cheerfully!) on techniques for success.

The Three Variables

Features - Time - Quality

As time runs out, either features or quality will have to give.

Classic error

Denial. As time runs out, nobody admits how low the quality is for the chosen feature set. The deadline arrives, and you end up shipping a project with features that are 80% done.

Would have been better to accept the reality halfway through the project, cut some features, and end up shipping with a smaller number features 100% done.

Alternate phrasing

It can be done on time. It can have all the features you want. It can be done well. Pick any two.

Conclusion

Take testing and milestones seriously to evaluate the current state or the project. Avoid denial. Make informed decisions early that balance the three variables.

Note: People

You could argue that "people" is a fourth variable -- you could increase the number of people to do better with the other three variables.

I don't present it that way, since it is generally accepted that adding people to an in-progress project is often counterproductive. Quote: "adding people to a late software project makes it later." We'll talk about this later -- it roughly true, but not some kind of law.

The Spirit of "Tradeoff"

Empty Statements

It's too, too easy to mention benefit without expressing the relationship to costs.

The statements appear true, but actually don't add anything to the discussion.

Statements that avoid stating costs express nothing. This is a peeve of mine in political debates, where candidate may think it's safer to say nothing than to take an actual position that is sure to offend some people.

Candidate 1: "education is important"

Candidate 2: "education is important"

Candidate 2: "education is important"

Tradeoffs

Everything costs something. A statement (or decision) with real content will take a position on both the cost and benefit.

Making decisions is about making hard choices about what to favor and at what cost.

Candidate 1: "we should tax gasoline more, and use the money to fund education" (my personal favorite)

Candidate 2: "parents should fund the education of their own children. We should cut taxes, reduce government education spending, and let parents pay for it themselves with they extra money they keep. If families are poor, too bad."

Candidate 3: "we should require all 21 year olds to spend an unpaid year volunteering in elementary schools" (this is actually the Candidate 1 position, but with a different form of tax)

Point: these three positions are actually different and there's debate to made between them.

Prioritization List

A simple way to express tradeoffs is to have a list of priorities.

A prioritization list has real content in it.

So for cars, Ferrari might think...

1. Fast
2. Reliable
3. Inexpensive

While the original VW bug might think...

1. Inexpensive
2. Reliable
3. Fast

Everything takes longer than you think

-or-

"Iceberg" Principle

-or-

100% foresight is naive

External/Abstraction View

The way you think about something from the outside -- how it interfaces with other things.

The abstraction it presents.

Internal/Implementation View

The external view + all the details and quirks of how it is actually implemented.

The internal view is much more complex than the external view.

Iceberg Analogy

20% of the iceberg sticks above the surface, but most of the iceberg is below the waterline.

With most systems, we observe the obvious part sticking above the water line and use that view in our mental models. Most of the complexity is not visible.

Tend To Plan in terms of External View

When thinking about a system, we tend to use the external view for the various components. (I believe this is an inherent feature of using a brain which must use conceptual simplifications to think about a complex world.)

Underestimate

As a result, estimates invariably understate the total complexity. This happens every time.

Allow Extra Time

There will always be unexpected details and problems -- allow extra time for them. This applies to any project (a paper, a widget, a movie, a vacation). There's always more work in the details than is apparent from the external view of the finished product.

When making schedules, allow an extra 50% to deal with the inevitable "unexpected" problems. "Underpromise, overdeliver"

When Can you have X done?

The answer to the question "when can you have X done?" is not "what is the earliest date by which you just might be able to have X done?", it is "what is a reasonable date, allowing for the usual unexpected problems, by which X can reliably be finished?" Why build a schedule around dates which have only a 50% chance of being met? On the other hand, when you have committed to a deadline, it's a real commitment.

e.g. Making List of Tasks

Suppose you were to try to list out ahead of time all the tasks in a software project.

- 1) Make a list ahead of time of all the methods and issues which a piece of software will need to deal with
- 2) Do you really think you've thought of everything in (1)? It's guaranteed that there will be issues which are only revealed during the implementation. Could add a blanket "things we haven't thought of" item to the list in (1)

100% Foresight is Naive

These "unexpected problems" are not due to poor design -- The systems involved are sufficiently complex that 100% foresight is naive.

You could try to spend a lot of energy in your design so as to make your predictions more and more accurate. In my opinion, it's better to accept that predictions are fairly inaccurate, and just move forward building in extra time for the unexpected, instead of spending a lot of time trying to make more and more accurate models and predictions.

80/80 Rule

80%

The first 80% of the functionality takes 80% of the time.

20%

The last 20% of the functionality also takes 80% of the time.

Details take time

Details and polish take more time than you think.

Budgeting

Realize this when budgeting your time

Watching Demos

Be less impressed with a demo which shows major functionality, but has a "a few details left to clean up."

If the core works, but the details don't, it's 50% done at most.

Examples

There are many details which are not big or obvious, but which end up requiring just as much work as the big obvious "core" features...
 paginating when printing, online help, foreign file conversions, fixing bugs created by fixes to other bugs, remaining backward compatible with your old files, interfaces, opening your window in various monitor sizes and color depths correctly, preferences file maintenance, sample documents, being OS level scriptable, internationalization (th, st, monday, tuesday, ...), working under old versions of the OS, file systems,..., Multiplatform issues (say hello to your friend #if ...), working with weird printer devices, color models..., low memory handling, dealing with random error conditions correctly everywhere— math input errors, disk read errors.

Parallelism Matters

3 people, 3 computers -- parallel

Writing and testing in parallel -- separate schedules, separate testing, separate computers -- not only are there three of you, but the logistics are much easier. A luxury most likely seen in the beginning

Depends very much on the design and separation between the components
 — this is why abstraction and ADTs are stressed so much in CS1/CS2.

3 people, 1 computer -- integrated

Always reduced to this eventually

Careful design to allow you to enjoy the parallel model as long as possible

Careful design and testing allows the integrated phase will go well

Point

Having a design and testing plan which enables the "parallelism early" to work is key topic we will consider.

Linux

How is it that all the people working on Linux are able to work independently? The person working on `gcc` vs. the person writing the `make` implementation? Careful abstractions and interfaces between the two allow them to be developed quite independently -- separate developers, separate release schedules, separate bug databases. Most software components aren't as separate as `gcc` is from `make` (although perhaps they should be!).

Only exposure to the CPU makes something close to correct

Compiled vs. Debugged

Too easy to think of something that compiles as being nearly done. It may be far from it.

CPU Bugs

Realize when a method compiles, but has never had the CPU run over it with plausible data. Any such method will contain some bugs.

Only when a component is really run — "exposed to the CPU" — will bugs fall out. There will be significant bug fallout when a component is first exposed to the CPU. There will be additional fallout when it is mixed with other components under the CPU.

Test-Bed

Build throwaway test-bed code to try to catch some bugs early. It's much cheaper then. Before CPU exposure, it's impossible to know if the code is worth anything.

When working independently/in parallel with other stuff, build throwaway test bed code to really exercise the code **before** integration. You don't want to be testing the independent components after they are combined -- testing then is 3x more difficult technically, and more expensive in terms of person-hours because everyone is present..

AKA the "Serious Early Deadlines" rule -- you do not want to use your last (expensive) deadline to find errors. Better to take your early (while still

parallel) testing seriously -- build throw-away tests to really exercise things.

Integration Pain

Suppose you have two components which are each somewhat debugged. When you mix them together, new problems will show up.

Therefore...

- 1) Try to design enough before coding so that huge design holes do not show up at integration time.
- 2) Try to debug the things independently before integration.
- 3) Allow time for integration

At The End, It's Too Late

You cannot make things proceed quickly at the end, no matter how badly you want to or how hard you push yourself, or how many people you have to throw at the problem

Therefore, serious coding effort early in the process is the only way to have something workable at the end.

Self Motivation

This is purely an issue of self-control and motivation -- everyone is motivated at the end, but it's too late. The skill is: work with that "night before it's due" intensity two weeks before it's due.

"The will to win means nothing without the will to prepare." - Juma Ikangaa (marathoner, also attributed to the basketball coach Bobby Knight)

Anatomy Of a Slipping Schedule

Everything is fine until the last 20%. Then there's a sickening realization that the project is much farther from completion than anyone thought.

Why Does It Work That Way?

1. Human nature to want to interpret and report things optimistically

Alice: I think there's huge monster behind this door."

Bob: "Let's not look in there."

2. Quality of code is not immediately apparent (CPU rule)

You have to actively test code it to see if it works.

With lack of time, it's very easy to not test hard. So the quality of the code is not really known. This path is less work and emotionally convenient.

3. When things are really getting integrated, it becomes undeniable what works and what does not.