Iterative and Incremental Development (IID)

Robert C. Martin

Engineering Notebook Column

April, 1999, C++ Report.

In my last column I took a satirical look at the waterfall process, trying to expose some of its faults. This month we are going to look at an alternative that tries to deal with those faults. But before we begin I'd like to say: Thank goodness that waterfall showed up in the early seventies! Back in those days, there was very little thought put into the structure of a software application prior to coding it. If waterfall did nothing else, it got us to accept the need to think before we code.

A Summary of the Problems with Waterfall.

The symptoms discussed in my last column can be attributed to the following three attributes of the waterfall model of development:

1. No feedback between phases. Once Analysis is done, it is done, and everyone should be working on Design. Once Design is done, it is done, and everybody should be working on Implementation.
2. Hard dates on phases. The dates set for completing Analysis and Design are major milestones that developers are measured against.
3. No completion criteria for either Analysis or Design.

Lets examine each point in turn.

No feedback between phases.

Although Dr. Winston Royce, the inventor of Waterfall, clearly showed that there must be feedback between the phases, many development managers have ignored this. Ignoring the feedback between Analysis and Design is tantamount to saying that we can analyze the entire application up front without learning anything from the structure or behavior of the resultant software.

Though seductive, this seldom turns out to be true. To emphasize this point, Booch[1] quotes Gall[2]:

> "*A complex system that works is invariably found to have evolved from a simple system that worked. . . . A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.*"

We poor humans are not capable of holding entire complex systems in our minds. To truly analyze them, we have to be able to touch them, fiddle with them, get inside them, and evolve them. Analysis that is deprived of the feedback from later phases, is nothing more than a hypothesis, an educated guess. And the more complicated the application, the more that Analysis is like building castles in the air.

Hard dates on phases.

Of all the horrible features of Waterfall, this causes the most pain. When managers saw that software development could be broken up into phases, their natural reaction was to track these phases by putting

---

[1] *Object Oriented Analysis and Design with Applications*, 2d. ed., Grady Booch, Addison Wesley, 1994, p. 13

[2] *Systemantics: How Systems Really Work and How They Fail*, 2d. ed., J. Gall, The General Systemantics Press, 1986, p. 65

milestone dates on them. Though this seems reasonable from a project management point of view, it is actually one of the worst possible ways to abuse a development process.

Teams do not make linear progress through analysis. They don't start with requirement 1 and analyze it, then move to requirement 2 and analyze it. One cannot measure how much analysis has taken place by looking at how far the analysts have read through the requirements document.

Analysis tends to be revolutionary. After thinking about a significant fraction of the requirements, some creative sole has an AHA! reaction, and gathers deep insight into the application. This insight may be right, or it may be wrong. If wrong, following it will lead to the next deep insight.

That's the way the creative process works. We learn more from our mistakes than from our successes. We stumble along in a partially random, partially directed walk from mistake to mistake until we finally get to something that looks reasonable.

Putting a date on the completion of this process is ludicrous. It can neither be measured nor scheduled. Does that mean that software cannot be scheduled at all? No! We can schedule software, but not by scheduling intangibles like analysis. More on that later.

No completion criteria.

The real insidious thing about putting dates on the Analysis and Design phases, is that those dates are often met! This is not because sufficient work has been done to analyze or design the application. It is because the only completion criterion for the Analysis and Design phases is – the date.

There is no way to tell when Analysis and Design are complete. It is always possible to continue Analysis. And Design is truly complete only when the code itself stops changing. There is no meaningful way to judge the completeness of an analysis or of a design.

Thus, managers are often given good news when the Analysis and Design phases meet their dates. However, this good news is a lie. It is mis-information. And managers are basing their decisions on it.

Feedback: The universal way to control.

How do our bodies regulate our breathing? As CO_2 levels rise in our blood, signals are sent to our brain to increase respiration rate. Why do stars shine for billions of years with such consistency? The amount of energy expended provides just enough heat to hold up the mass of the star. How can cruise-controls keep our car running at the same speed whether we are going uphill or downhill? The cruise-control senses the speed of the car and regulates the flow of fuel to the engine to keep the speed constant. All these system, some natural, some evolved, and some man-made, use the same mechanism for maintaining control. That mechanism is feedback.

To control a software project, we also need to use feedback. The only way to truly control a software project is to continuously measure its progress, compare that progress against the plan, and then adjust the development parameters to correct any deviation from the plan. Simple.

Unfortunately, in practice, measurement cannot be made continuously; and there are very few development parameters we can adjust. Lets deal with each issue in turn.

Approaching continuous measurement.

Since we cannot measure the progress of a project continuously, we must measure it at certain milestones. The more milestones we have, the more we approximate continuous measurement, the better. However, we must be careful about *what* we choose to measure. If we choose something like the completion of Analysis or Design as measurements, the measurements will be meaningless.

Tom Demarco[3]said of project management:

---

[3] *Controlling Software Projects*, Tom Demarco, Yourdon Press, 1982, p137

"Your project, the whole project, has a binary deliverable. On scheduled completion day, the project has either delivered a system that is accepted by the user, or it hasn't. Everyone knows the result on that day."

The object of building a project model is to divide the project into component pieces, each of which has this same characteristic: Each activity must be defined by a deliverable with objective completion criteria. The deliverables are demonstrably done or not done."

So a binary deliverable is a deliverable that has one of two states: done or not-done. Clearly Analysis and Design are not binary deliverables. They can never be shown to be complete. On the other hand, what better binary deliverable could we have than a piece of code that executes according to stated requirements?

Vertical slices.

When we begin an iterative and incremental development (IID), our first goal is to subdivide the project into binary deliverables. Those binary deliverables are defined as executing segments of the overall project.

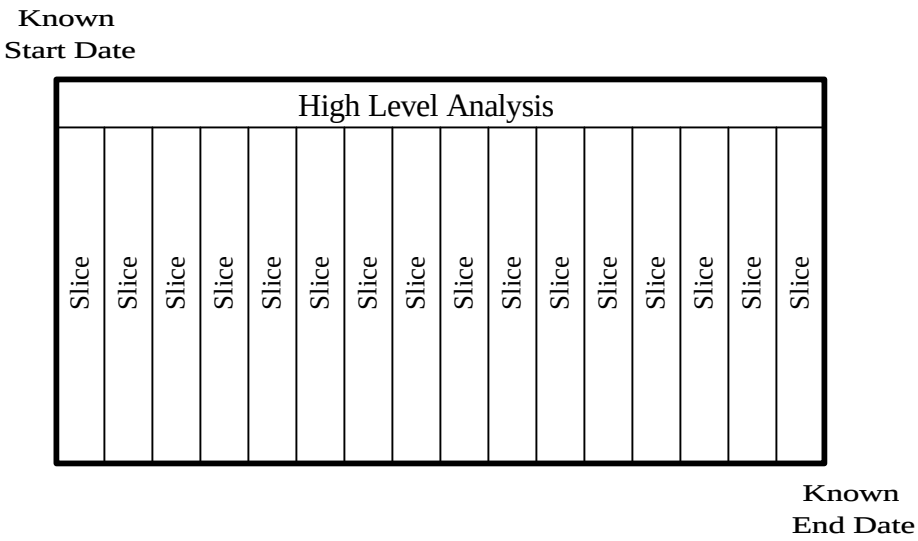

Figure 1. Subdivision of the project into vertical slices.

Figure 1 shows a schematic of how this is done. The project begins with two known pieces of data. The start date and the end date. Indeed, *the end date is usually the first datum known about any project*. Market pressures frequently establish the end date long before there is a reasonable set of requirements. You can complain about this evil fact all you like, but it's the way the real world works. We, as engineers, have to deal with it.

The next ugly fact we have to deal with is that the requirements are going to be volatile throughout the entire development phase, and probably well past the end date. Indeed, the requirements document is likely to be the most volatile document in the project. Again, though we wail and moan, this is just how things are. We, as engineers, have to deal with it.

To create a set of binary deliverables, we first do some very high level analysis. This analysis has a very specific purpose, and it has a well-specified end date. The purpose is solely to subdivide the project into a set of vertical slices that are binary deliverables. That is, the slices will execute and show conformance to requirements. The end date for this analysis depends upon the size of the project, but should be a matter of a few days or weeks. The thing to remember here is that the slices you choose during this period will not be the slices you eventually end up with. Like everything else that is done up front, the slice partitioning is a guess, and it will be wrong. So don't bother spending lots of time perfecting it.

The criteria for the slices are:

1. They should be vertical. That is, they are not subsystems. Rather they cut across as much of the functionality of the system as possible. For example, an IO driver subsystem, or an IPC framework would not be a valid slice. Rather, the slice should implement a use case, or part of a use case. It should be something you can show to a user, that the user would attach meaning to.
2. They should represent features. Later we may have to decide to eliminate parts of the project in order to meet schedule. It would be good to be able to eliminate whole slices, rather than have to eliminate part of many slices.
3. They should be executable and demonstrable. They should have completion and acceptance criteria. Users should be able to evaluate whether the slice is complete or not.
4. A slice should take a matter of days or weeks -- not months -- to complete. I prefer an average of two weeks. To the best of your ability, slices should be roughly equal in manpower.

These criteria are ideals, and cannot always be met for every slice. However, the up front analysis should try to make the majority of slices fit them. The success of the iterative process depends upon this fit.

Iteration.

Upon completion of the slice partitioning, and the writing of the acceptance criteria, the project managers choose the first slice to develop. This slice should be the one with the most risk in it. It should be the most complex and error prone of all the slices. The reason for this is twofold:

1. We want to know as early as possible whether or not this project will succeed. Thus we attack the high-risk areas first.
2. We want to calibrate the estimation model as conservatively as possible. Thus we want the first slice to take a relatively long time to complete.

It may be, of course, that as you work though the project you find that there are more complex and risky slices than the one you chose. That's all right. The idea is to choose a slice that represents the highest risks.

You may also find that choosing the highest risk slice is impossible because it depends upon other, simpler, slices. In that case you will simply have to do the other slices first. But get the high risk slices in as soon as possible.

It may be that you have more manpower than is required for the first slice. As the project proceeds you will certainly be running more than one slice at a time. However, I prefer to run the first slice or two serially. The team will learn a tremendous amount from these first slices, and that knowledge will save lots of time later.

The slice is developed in isolation. That is, the developers do not concern themselves with the contents of other slices. They analyze, design, implement and test the slice. This sounds like waterfall inside the slice. In a way, it is. However, it is waterfall as envisioned by Winston Royce, not as perverted over the years. The feedback mechanisms between analysis, design, and implementation are all operating.

The developers will analyze a little, design a little, code a little, in a tight loop. Booch calls this process of tight iteration the *Microcycle*. Their deliverables at the end are:

1. An executable that meets the acceptance criteria.
2. Analysis and design documentation of the executable.

Slices are not milestones.

The whole purpose of cutting the project into slices and then developing them iteratively is to *produce* data. We want to know how long it takes to build a slice. Therefore we do not put dates on the slices. Resist this

temptation! If you attempt to schedule the completion of individual slices, you will corrupt the data that you are trying to produce, and will lose control of the schedule.

This is not to say that objectives should not be set, or that the project cannot be managed. As we'll shortly see, project management takes place at a level above the slices.

Continual estimation.

When the first slice is complete, several pieces of data are produced.

1. We have a piece of code that we *know* works. I.e. a tangible part of the project is truly complete.
2. We have the analysis and design documentation for this working portion of the project.
3. We know how long it took to complete the slice.

Using this last datum, we can perform a very simple calculation. We can multiply the duration of the slice by the number of remaining slices to get the approximate end date of the project.

This is the crudest of estimates, and the error bars are huge. However, this estimate of the completion date is based upon real data; and is therefore far better than previous estimates that were based upon guesswork.

Clearly we don't want to make any dramatic decisions based on this first estimate. The error bars are too large to justify any immediate action. However, that estimate could start helping us to evaluate alternate plans.

Having finished the first slice, we start the second. And, indeed, we may decide to increase the manpower on the project to start two or three slices simultaneously. We don't want to ramp the manpower up too quickly though. These early slices are going to be producing foundational structures in the code. Those structures won't have a chance to form if too many slices are implemented concurrently.

During the development of the second slice, the engineers will find that the structure of the first slice was incorrect. So, the first slice is repaired as part of the development of the second. All analysis and design documents are brought up to date also. Thus, the second slice may take a bit longer than the first. Indeed, each subsequent slice will be repairing structural errors in the preceding slices. This is how the structural foundation of the project will form. By including the time for forming this structure in the slices, it becomes part of the overall project estimate.

Some might complain that this "rework" could be avoided with a little forethought. Perhaps. However, forethought is seldom very accurate. For a large project it is not very likely that we could truly anticipate the final structure of the software.

Look at it this way. The work we are doing on the early slices *is the forethought*. We are using design and code as a way to *verify* that this forethought is correct instead of trusting a set of tentative ideas. More on this in my next column.

As slice after slice is completed, the error bars on the estimate get smaller and smaller. After a few slices we can be pretty confident in what it is telling us. And it may not be giving us good news. But, it's better to hear the bad news sooner rather than later. The earlier we hear it, the more options we have.

Exerting control over a project.

There are three different control dimensions that project managers can use to manage a project: Staff, Scope, and Schedule. If our estimated completion date is past the due date that was selected at the beginning of the project, there are three things that managers can do. They can increase staff; they can change the project scope by reducing features; or they can alter the schedule by changing the due date. No project can succeed without flexibility in at least one of these dimensions. If all three dimensions are fixed when the project begins, then the project will be unmanageable, and managers will have no options, other than prayer.

Remember, the estimate produced by the slices is appearing *early* in the project lifecycle. If we decide to increase staff, it's not at the last minute. We'll have time to integrate and train new team members. Or, if

we decide to limit the project scope, we'll have time to negotiate that limitation with our customers. Or, in the *extremely* unlikely case that we decide to change the project due date, we'll have time to negotiate that change with our customers.

If you don't like any of these options, you can always storm around the project floor alternately threatening and encouraging the engineers. But I guarantee that this will demotivate most of them, and slow the project down even more.

The iterative process is *producing data* about the schedule. The completion times of the slices are giving us unambiguous empirical measurements that help us predict when the project will be complete. There is a wonderful thing that mangers can do with these measurements – with this data managers can… manage. They can stop dictating, stop motivating, stop losing control.

Conclusion

Don't let this article mislead you. If you follow the advice above and begin developing projects in an iterative and incremental way, bluebirds will not fill your sky. Schedules will still be missed, there will still be bugs, problems, and mis-aligned expectations. Software is, after all, software; and software is hard. However, what you *will* be doing is using a process that produces data; whereas waterfall produces none. With that data, managers can try to manage the project.

In my opening paragraph I was thankful that waterfall taught us to think before we code. Have we now given this up? Not at all! It's just that, now, we think it through in small increments, and use code to verify that our thinking was correct.

In my next column we'll talk about what to do if you have a corporate policy that is entrenched in waterfall. We'll also talk about early customer feedback, throwing slices away, and building reusable frameworks.