Iterative and Incremental Development (IID)

Robert C. Martin

Engineering Notebook Column

June 1999, C++ Report.

In my last column, we looked at the fundamentals of an iterative and incremental process. We discussed how to divide a software project up into vertical slices and to analyze, design, and implement each in turn. We showed how to estimate project completion by extending the slice completion time by the number of slices. We discussed how the analysis and design documents become more complete as each slice is completed. We discussed how to use the feedback generated by the process to manage the scope, schedule, and staffing of the project.

This month we are going to continue this exploration into IID by looking at how to deal with a corporate process that is entrenched in Waterfall. We'll also discuss how to manage the project with milestones, how to get customers involved early, what happens if early slices are very wrong, and how to manage the creation of reusable frameworks.

Faking it.

Parnas[1] told us that we can never have a completely rational development process because:

- A System's users typically do not know exactly what they want and are unable to articulate all that they do know.
- Even if we could state all of a system's requirements, there are many details about a system that we can only discover once we are well into its implementation.
- Even if we knew all of these details, there are fundamental limits to the amount of complexity that humans can master.
- Even if we could master all this complexity, there are external forces, far beyond a project's control, that lead to changes in requirements, some of which may invalidate earlier decisions.
- Systems built by humans are always subject to human error.
- As we embark on each new project, we bring with us the intellectual baggage of ideas from earlier designs as well as the economic baggage of existing software, both of which shape our decisions independent of a system's real requirements.

Parnas goes on to observe that "For all of these reasons, the picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic." But Parnas goes on to say that it is both possible and desirable to *fake it*.

Suppose you have a corporate process that is so entrenched in waterfall that there is no short term possibility of changing it. Are you forced into abandoning IID? Not at all! Indeed, IID can be used as an iterative mechanism beneath waterfall.

Design and Code as Analysis Tools.

Consider Figure 1, which shows a project being divided up into iterative slices. During the first several slices, we are primarily exploring the problem domain. Though we are producing code, that code is incidental to the knowledge we are gaining. That knowledge has to do with the problem. Users, looking at

[1] Parnas, D. and Clements, P. 1986. A Rational Design Process: How and why to Fake It. *IEEE Transactions on Software Engineering* vol. SE-12(2) p251 as quoted by Booch in *Object Solutions*, Addison Wesley, 1996, p 9.

the behavior of those first few slices are adding new requirements and changing old requirements as they refine their own view of the problem. In short, this is analysis.

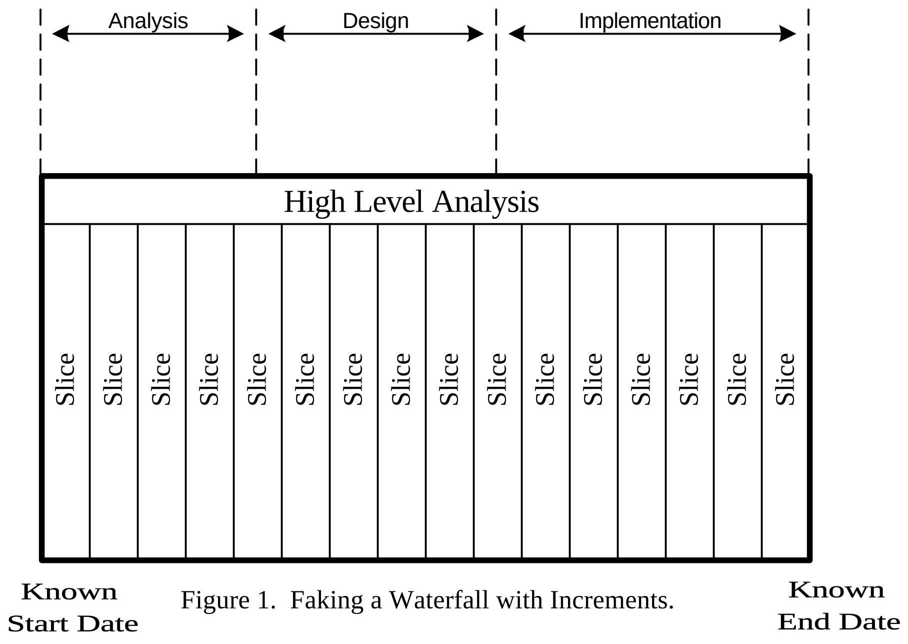

Figure 1. Faking a Waterfall with Increments.

One way to perform analysis is to build models of the problem domain and test those models against users' expectations. But another, very valid way to do analysis, is to build working increments of the project and have the users examine those slices. Either way, we are exploring the problem domain and understanding the problem to be solved.

Code is a perfectly valid analysis tool. It can be used as an extremely accurate instrument to divine the requirements and test them against users' expectations.   So, at some point in the suite of slices, we can declare analysis to be complete… Actually, analysis is never absolutely complete, rather after sufficient time we can say that it is complete enough for purposes of the waterfall.

The same technique can be used for design. Once we are well into the creation of slices, the focus of the development shifts away from the problem and more towards the solution. During this phase we are hunting for the appropriate system architecture. Slices are focussed upon finding the best way to solve the issues of the problem domain. Thus, code is being used as a design tool. Slice after slice adds knowledge to the structure and architecture of the software.

Remember, as each slice proceeds, we discover things about the previous slices that we don't like. We immediately refactor those previous slices as part of the current slice. Thus, the design becomes ever more refined as the iterative process continues.

Eventually, the architecture of the system settles down, and we can declare design to be complete. From this point on the slices start cranking out with little or no affect on previous slices. The problem has been determined, the architecture is stable, and we are just implementing. Thus, the implementation phase has begun.

It should be noted that there is no difference in activity at any point along this process. Each slice is developed according to its own requirements. From slice to slice there is no difference in the process. Yet, as the slices pile up, the engineer's knowledge of the project changes exactly as anticipated by waterfall. The first questions to be resolved are those about the problem domain. Later the solution domain becomes ever more clear. Finally, the implementation just starts to crank.

So, if you have a corporate environment that is entrenched in waterfall, you can still use IID to fake a waterfall. Indeed, you can even produce all the necessary work products. For example, at the end of analysis, you can produce a problem domain model. Having done the slices that explore the problem domain, you will be in a very good position to create that model. There will be very little guesswork involved. The model will simply be a restatement of what the slices have taught you.

Similarly, at the end of design, you can produce a design document that shows the design of the project. This document will be very accurate and will simply be a translation of the design that evolved while building the slices.

From the top, this process can look exactly like a waterfall model. The only difference is that you are using code as a tool to explore the problem and solution domains. You are using the slices as a means to gather the problem domain and solution domain knowledge needed for the waterfall phases.

Setting Milestones

Of course, one of the major attractions of waterfall is the ability for managers to set milestone dates. This can still be done, even using an IID. However, milestones have a very different flavor in IID than in waterfall. In IID, a milestone is not a commitment. Rather it is a decision point.

In IID we use the process to produce the data that tells us how we are doing against the schedule. We do not commit to finishing certain slices at certain times. Rather we use the time that it takes to produce a slice as input to calculate when all the slices will be done. This gives us a way to measure the progress of the projects and to predict the completion date.

However, it is unrealistic to suppose that a project should not have milestones. Indeed, it should! Those milestones might have comforting names like “Analysis Complete”, or “Design Complete”; so that from the top it looks like we are doing waterfall. Those milestones might be the completion of a certain work product, or they might be a date. Whatever the milestone is, it is some kind of event.

The difference in IID is the way we treat that event. We do not treat it as a commitment. The project team does not promise to make all its milestones. The project manager promises to manage the project through the milestones to obtain an acceptable end result. The project manager uses the milestones as events that trigger decisions.

And those decisions should be written down ahead of time, and reviewed frequently. No one on the project should be surprised at what happens when a milestone is passed; it should all have been written down ahead of time.

For example, lets say that we have a project with a milestone at the September 1st, 2000. This milestone does not have a completion criterion. Rather it is simply a date at which a decision will be made. That decision might look like this:

- If slice 7 is complete, continue according to plan.
- If slice 6 is complete, then eliminate slice 12 from the plan.
- If slice 5 is complete, then eliminate slices 12 and 15 from the plan.
- If Slice 4 is complete, then eliminate slices 12 and 15 from the plan and add another engineer by Oct 1st.
- If slice 4 is not complete, then cancel the project.

Getting Customers Involved Early.

During the analysis “phase” of IID, it is critical to involve customers, or their representatives in the evaluation of the evolving project. When the customers are actually able to touch and feel the project, they will invariably decide that what they are seeing is not what they really wanted. They will submit changes.

This is a good thing! We want the changes early. We want the customer to be confident that we are building the right things. So the project plan will thrash a bit during this initial phase. There are likely to be slices added, other slices removed, and many slices significantly altered.

As time progresses, and the analysis “phase” draws to a close, the users will gradually reduce the number of changes they want made. It is unlikely, however, that they will every completely stop. Indeed, as the users’ environment and business problems change, they are likely to come increase the number of change requests.

Every such change affects the array of slices in the project. It will affect some that have been completed, and some that are yet to be completed. It may add or remove slices too. But since we have been measuring our “speed” in terms of slice completions, we will be able to estimate the schedule effect of each change. Users, when confronted with the schedule ramifications, can be given the option of accepting the change to the schedule, or postponing the change.

All of this has the effect of putting the users in control. The users never view the project as a black pit of manpower from which nothing but arcane documents emerge. Rather the users actively participate in the development and evolution of the project.

When Slices go Wrong.

Of course getting the users involved in this way means that there will be some substantial amounts of rework as those users make changes. Indeed, there will also be rework as the problem domain and solution domains settle down. Sometimes the rework can be so significant that the team decides to reimplement whole slices. So long as this is infrequent, this is not a bad thing.

Once it is determined that a slice is just plain wrong, it should be discarded and reimplemented. This is especially true if the slice is early in the project and is defining critical architectural elements. We don’t want that incorrect architecture poisoning the rest of the project. Indeed, it is much more likely that the project milestones will have good outcomes if such errant slices are discarded, than if they are kept. Keeping them will force everyone to have to work around the problems and slow the whole project down. Refactoring and repairing them eliminates the roadblocks.

One could argue that an approach that focussed more on up front analysis and design would eliminate or mitigate this kind of rework. This is doubtful. When a slice goes wrong, there is no doubt that it has gone wrong. But when an analysis model or design model goes wrong, we often don’t find out about it until we are implementing it. Thus, while errors in slices are discovered early and repaired before they can do much harm; errors in analysis and design models aren’t discovered until late after many other models and decisions have been predicated upon them.

Managing Reusable Frameworks.

This leads us to the topic of reusable frameworks. Such a framework should never be a slice, or a suite of slices. A reusable framework that is developed by itself will probably not be very reusable. Rather a reusable framework needs to be built in the context of application slices. And more than one if possible.

If you need to build a reusable framework, then take care choosing your slices. As always, make sure that they are slices that are rich in user features, and that cut across the whole system as opposed to implementing subsystems. Have the slice developers collaborate on creating the reusable framework by insisting that nothing be put into the framework unless it is reused by more than one slice. Never put anything into the framework because it “seems like a good idea”. Rather, insist that the only things that go into the framework are things that would otherwise need to be repeated in two or more slices.

The engineers can negotiate on the correct abstractions that allow the reusable elements to be correctly accessed from each of their slices. This ensures that the abstractions in the framework are suitable for reuse by at least two slices. And where two are able to reuse it, more are likely to follow.

Conclusion

This brings to a close our discussion of iterative and incremental development. In this series of articles we have explored the failures and fallacies of the waterfall model, have shown how to divide a project up into increments that are binary deliverables. We have shown how to estimate project completion and how to manage the project to have a good outcome.

The recommendations and techniques that have appeared in this series of articles have worked for me, and for others. But they are not perfect. They will not solve all software development problems. Software will still be late, still be intractably complex, still be perniciously difficult to develop. Software is, after all, Software. However, the process we have outlined does have the benefit that it *produces* data. And a process that produces data can be managed. A process that does not produce data, cannot be managed.