ITERATIVE DEVELOPMENT: KEY TECHNIQUE FOR MANAGING SOFTWARE DEVELOPMENTS

Dwayne Read
Strategic Systems (WA) Pty Ltd
dwayne@ss.com.au

Abstract

Iterative development provides the fundamental structure that enables the effective management of software development. Ultimately it helps to avoid the "big bang" issues and spreads the typical pain of development (integration, major re-writes, insufficient testing, deadline pressure, overtime, etc) over several, easier to manage, units of work. Scoped correctly, development iterations provide a concrete feedback mechanism, which enables better visibility and control of the key elements for successful development. This paper explores these key elements and the benefits that iterative development provides to each of them. In particular, we look at risk management, progress monitoring, requirements management, software architecture and testing.

Keywords

Iterative Development, Software Development, Risk Management, Progress Monitoring, Requirements Management, Software Architecture, Testing, Resource Utilisation, Milestones, Release Cycle, Agile Methodology, eXtreme Programming, Feature Driven Development, Project Velocity

INTRODUCTION

By structuring your project or product development into a combination of releases (external deliveries) and iterations (internal builds) of the system you establish a concrete[1] feedback mechanism. These are true milestones (as apposed to the highly ceremonial document delivery and review milestones) that you can calibrate progress against.

Iterative development ensures the correct focus throughout development by addressing areas of technical concern (esp. architecture) early on, developing the key features/requirements first, obtaining real-world/customer/user feedback on early releases, calibrating effort on an ongoing basis and enabling the technical solution to evolve and be adjusted with minimal overhead. This is one of the best forms of risk management, as you have early feedback on the effectiveness (or otherwise) of risk mitigation strategies.

Iterative development also provides the vehicle to enable evolutionary development (being able to change the direction with minimal impact), facilitates easier resource utilisation/balancing and faster delivery times to the market/customer.

CONCRETE FEEDBACK MECHANISM

The fundamental structure of iterative development divides a project/product development up into several releases, each of which is further divided into several iterations. The objective is to create "mini-projects" each of which has a defined scope, resources, duration, deliverables, testing, etc.

[1] The term "concrete" is used throughout this paper to represent a tangible software development output in the form of an executable and working piece of software (e.g. component, application, sub-system, etc).

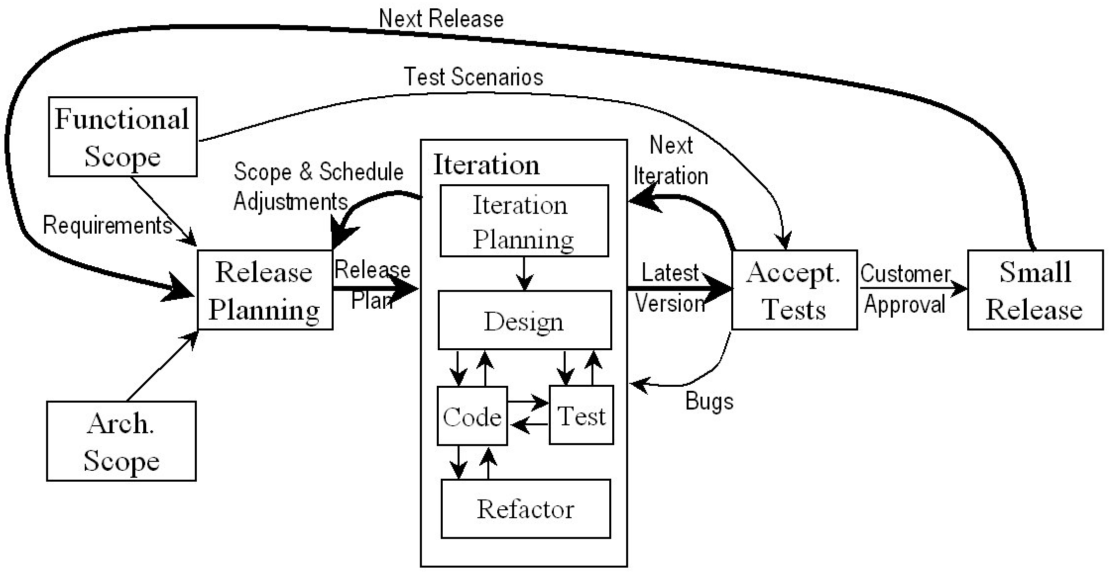

Figure 1: Iterative Development Workflow

Each iteration takes a slice of development from start to finish ensuring closure on the scope of work undertaken. Each iteration is focused on delivering a running, fully integrated and fully tested solution, albeit against a subset of the scope of the release. The focus of an iteration is to provide concrete feedback to the development team on the design, implementation, integration, useability, etc of the system and is considered more an internal build. Figure 1 shows the development activities undertaken within an iteration and also portrays a link to the acceptance testing activity, which represents the testing of the subset of the scope relevant to the iteration. This itself helps to build up the complete acceptance testing of the release. Each iteration is ideally between one to four weeks duration (two weeks being the sweet spot) and will only have a handful (three to ten) requirements that defines its scope. Obviously there are advantages for the manager and customer in knowing the progress of each iteration, with some iterations even being suitable for review by the customer/end users for feedback on the look-and-feel, business rules, etc, however an iteration is predominantly for technical feedback.

A release is comprised of several (typically three to seven) iterations and has an agreed/approved functional and architectural scope, as shown in Figure 1. Each release provides another level of concrete feedback, this time from the user/customer perspective. This is the “reality check” – does the system actually meet the users’ requirements? We want as many “reality checks” as possible, as we have the opportunity through the later release/iteration cycles to adjust the solution. This approach is in line with the principles behind all agile methodologies (Beck et al. 2001) and is quite central to the Feature Driven Development methodology (De Luca 2002) and even the Spiral model (Henderson-Sellers 1994). Scope control is an important aspect that requires diligent management and the iterative development approach gives a vehicle for allocating requirements against a release (and its iterations). At the release level, the requirements need to relate to the minimum set of highest priority features that make business sense to release to the customer/market. The duration of a release is more variable, however two weeks to three months is considered the acceptable range, with a release every four weeks seen as the ideal timing if possible. There is a balance to be struck here, as the concrete feedback obtained from a release is required as soon as possible, however, the “business sense” of what is released often means the system needs to have a whole range of features available to be useful/saleable, plus consideration of any overheads with deployment and upgrading need to be considered.

BENEFITS OF ITERATIVE DEVELOPMENT

Risk Management

Risk management is all about identifying the risks, implementing risk mitigation strategies and then monitoring the effectiveness of these. With a grand/waterfall style development where there is a single pass through each of the phases of development (analysis, design, code and test), you only find out the effectiveness of your mitigation strategies at the end of the project – too late. With the likes of the Spiral model (Henderson-Sellers 1994) approach, a project is broken up into several spirals/releases, however even this has shortfalls as it has no concrete internal feedback mechanism (only phases within the spiral). It comes down to the size (number of features) and duration between the concrete feedback loops.

For software developments, the typical risks can be categorised as follows:

- Not meeting the user requirements
- Cost/schedule blow-outs – the estimates were wrong and/or there is more to be developed
- Integration issues - third party hardware/software components, performance, concurrency, etc

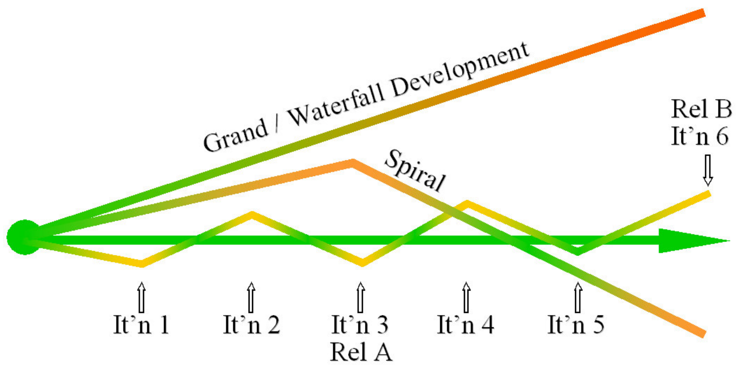

Figure 2: Concrete Feedback Loops Reduce Risks

Iterative development enables effective risk management as every iteration provides concrete feedback on the effectiveness of the risk mitigation strategies. In Figure 2, the Y-axis represents the discrepancy from the real requirements. The potential for a grand/waterfall approach to produce a system that is not fit-for-purpose is significantly higher than an iterative development approach as it lacks the concrete feedback loops. The Spiral model only has the feedback with each release, which is useful, however, true iterative development has more and shorter term concrete feedbacks through the iteration builds.

Each release (and some iterations) enable user feedback on the developments team's understanding of the requirements, with the ability to correct this with the next iteration/release. Often the users' themselves actually adjust their requirements (sometimes referred to as "requirements creep"), however this is a positive situation, as there is no point delivering a system that meets the original specification, if the original specification is actually wrong (or incomplete). Review of the requirements should be promoted after each release to set the scope for the following release. To this extent, one would not plan or undertake any additional effort for intended releases beyond the next release (as it will only change). This way, you can alter the direction with minimal impact.

As soon as you have your first completed iteration you have all the data you need to calibrate your development. You have the estimated effort, the actual effort and the actual duration. With these numbers you can calibrate the estimates for future iterations and releases to give a predictive view of the expected effort and duration. This calibration process is known as Project Velocity in the eXtreme Programming methodology (Beck 2000), and uses the ratios shown in Figure 3 below.

$$\frac{\text{Effort}_{\text{estimated}}}{\text{Effort}_{\text{actual}}} : \frac{\text{Effort}_{\text{forecast}}}{\text{Effort}_{\text{forecast calibrated}}} \quad \textbf{and} \quad \frac{\text{Effort}_{\text{estimated}}}{\text{Duration}_{\text{actual}}} : \frac{\text{Effort}_{\text{forecast}}}{\text{Duration}_{\text{forecast calibrated}}}$$

Figure 3: Ratios Used to Calibrate Effort and Duration

This calibration effectively takes into account every factor that is affecting the development team for that iteration or release, from the complexity of the system, degree of interrupts, skills of the team, number of days sick, etc. As these factors are all accounted for in the "actuals" for the release/iteration(s), they are the most realistic "model" to use to calibrate the next iterations effort. One should not adjust the ratios even if they know they can influence them (eg. by recruiting more staff), as you will know for sure with the next iteration (hopefully within two weeks) whether that influence was actually positive or negative. Ultimately, you should maintain a sliding calibration view, where the metrics from the last three iterations are used to calibrate the next iteration(s), and the metrics from the last one or two releases are used to calibrate the next release(s).

Iterative development significantly reduces the risks of integration issues, simply because it starts the integration very early in the project lifecycle. In fact, the very first iteration should bring together all the key components and architectural design elements and test them as an integrated whole – referred to as the "executable architecture" (refer to the section below on Software Architecture).

Progress Monitoring

How do you know if you're on track or not? Typically, this is achieved through the delivery of documents and associated reviews of the same for the initial (and probably most critical) phases of a project. Unfortunately documents are not a good reflection of progress per se. They may communicate a better understanding of the problem and/or solution, however they can (and usually do) hide or miss all the risks associated with the real development. It is too easy to "fake" progress through the delivery of documentation.

Iterative development provides the Project/Product Manager with the concrete feedback mechanism of the iterations and their cumulative releases. These are true progress milestones that actually cover all aspects of the development process and lifecycle (requirements, design, architecture, coding, testing, acceptance, etc) albeit for a small and deliberate subset of the overall scope. So in effect, iterative development promotes the "suck-and-see" approach (more formally referred to as evolutionary or incremental development). To this extent, the associated documentation is no longer suitable as a progress milestone, as they are changing on a per release and even per iteration basis – they are "living" documents. The integration and acceptance testing of a subset of requirements through a demonstrable and executable subset of the system is a definitive progress indicator. With iterations every 2 weeks or so, and releases every 4-8 weeks or so, we now have a very regular and reliable progress indicator. You just need to ensure good scope control (i.e. don't fool yourself – see next section).

Requirements Management

Requirements management is all about ensuring the requirements are actually met by the delivered system, through:

- Clearly defined and testable requirements
- Agreement by all stakeholders that these are the requirements (not always an easy one)
- Change control (of the scope)
- Traceability of the requirements through the development and ultimately to the delivered system
- Acceptance testing confirming the agreed requirements have been satisfied

Iterative development adds two key factors to the above elements – namely the prioritisation of requirements, and the ability to change the requirements with minimal/no impact.

In iterative development, each release is scoped using the next highest priority set of requirements that makes business sense. Here, we use the prioritisation of the requirements to drive this scoping activity. They are allocated to the release and its iterations, which is now the primary/only traceability (i.e. we don't really need to trace it to the system/sub-system as is traditionally done).

A project/product may ultimately have a couple of hundred requirements to satisfy; however each release should only be allocated 20 to 80 (maximum) requirements. By focusing on the highest priority ones first, we are still open to new or modified requirements coming into the overall scope for the subsequent releases. How can we actually expect the customer/user to know all of their requirements upfront? We have not spent any additional effort on the remaining requirements. If (more when) the requirements change then we can easily accommodate this, by allocating them to the next release, possibly trading them off against some previously allocated requirement.

As such, Iterative development has now given us the vehicle to manage (if not promote) changing requirements, provides traceability through to the concrete release, enables earlier feedback on the correctness of the implementation and promotes more regular runs of the acceptance testing as we evolve the solution to actually meet the user's requirements. This is good requirements management.

Software Architecture

Whatever anybody says, software architecture is hard. It just doesn't happen as a by-product of developing a system (many hope this is the case) and you can't define an architecture up front and expect it to be suitable (although many try). An architecture has to evolve – it needs testing, feedback, refactoring[2] and extending. Iterative development provides many feedback loops on the architecture and the opportunity to adjust it where it fails, is too limited or just too complex.

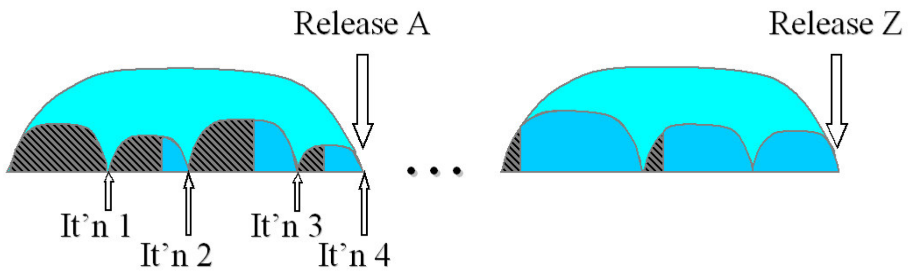

Figure 4: Software Architecture is Implemented and Extended Early in Each Release and Iteration

Being "architecturally driven" (Jacobson 1999) is a key focus for software developments, as it provides the overall structure for all the more detailed design and promotes/enables reuse. However, we need to evolve and adjust (on many occasions) the architecture. With iterative development, the first one to three iterations of each release (and especially the first one or two releases) have a particular focus on extending or adjusting the architecture. The early iterations within the first release are often referred to as an "executable architecture". This reflects the primary focus of these iterations where a skeleton implementation of the proposed architecture is developed, all third party component software and hardware are integrated and a series of executable tests are undertaken. This is the first real concrete feedback we get on the system design. We then refactor as required in preparation for the development of the additional end-user features.

Testing

Testing is all about finding the defects. The typical software application is quite complex, integrating third party components, implementing business rules/logic, undertaking mathematical computations, handling real-time (or near real-time) data feeds, etc. There will be defects, that's a given, and there are a range of techniques to help find these defects, such as code inspection, automated unit testing, integration testing, acceptance testing, etc.

All of the testing techniques are useful, however it is often the case that the testing is "the last thing to be done" and with project pressures the one that often has to give. Delaying/reducing the testing is false-economy at all levels. Firstly, during development the benefit of having a stable code base to further develop from can not be under-stated. The focus of a software engineer is easily broken when they are developing if they encounter a defect in a class/module they are dependent upon. If they need to then go and troubleshoot this itself, then we are already paying a maintenance overhead for code which is still being developed. Secondly, integration testing is often poorly done (in some cases skipped) yet is the key to ensuring a stable application as its core focus is on stress testing the combined hardware and software solution. Thirdly, acceptance and operational testing is the "reality check" that provides feedback on whether or not the system is actually fit-for-purpose.

[2] Refactoring is a mini-process that improves a design (and implementation) without affecting the rest of the system. A strict test before, test after approach ensures a controlled update is undertaken.

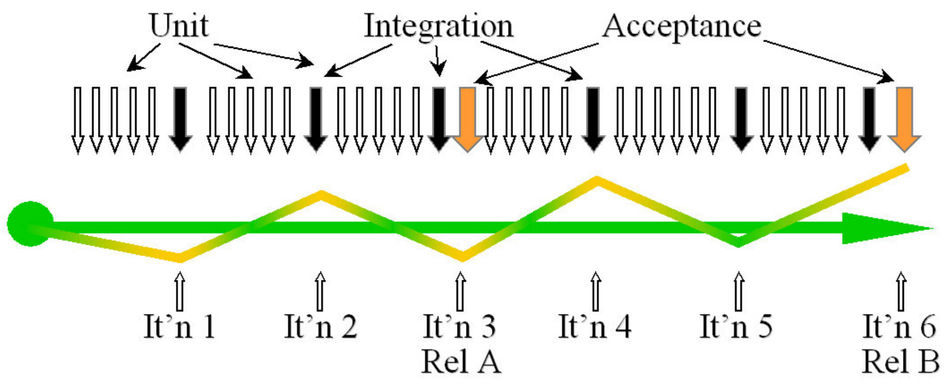

Figure 5: Testing Throughout Iterations and Releases

Iterative development completely changes the timing of the testing activities. As portrayed in Figure 5, the testing activities are spread out over all of the development, not bunched together as a later phase (open to pruning). The unit testing is undertaken on a daily basis and should be fully automated, the cumulative suite of integration testing is conducted at the end of each internal iteration, and the cumulative suite of acceptance testing is conducted at the end of each release. This approach spreads the testing out across the entire project schedule at a relatively even level, giving the benefits of a more stable code base, regularly stress tested and with several user/operational feedbacks. All of this combines to produce a higher quality solution and typically with a more cost effective and risk reduced approach. The iterative development facilitates the continuous testing approach popular with the agile methodologies such as eXtreme Programming (Beck 2000) and the sooner you test the sooner and easier it is to rectify.

CONCLUSION:

Fundamentally, iterative development is a development style that focuses on delivering concrete visibility of the technical, managerial and user aspects of the system/product in small, short-term packets. This enables more regular feedback and the ability and time to adjust the solution accordingly.

As discussed in this paper, iterative development can deliver the following benefits:

- Better risk management (scope, cost/schedule control, integration issues)
- Clearer and more meaningful progress indicators (concrete deliverables per iteration/release)
- Easier and more meaningful requirements management (allocate to a release, promote change, etc)
- Better focus on the software architecture (executable architecture, evolving, refactoring)
- Earlier, more focused and higher quality testing (cumulative, per iteration, per release, automated)

These combined benefits result in a more effective, efficient and even enjoyable development lifecycle as a result of the regular feedback.

REFERENCES:

Beck, K 2000 'Extreme programming explained: embrace change', Addison Wesley

Bech, K, et al. 2001 'Principles behind the Agile Manifesto', www.agilemanifesto.org/principles.html

De Luca, J 2002 'Feature Driven Development Processes', www.featuredrivendevelopment.com/files/fddprocessesA4.pdf

Henderson-Sellers, B & Edwards, J 1994 'Booktwo of object-oriented knowledge: the working object: object-oriented software engineering: methods and management', Prentice-Hall

Jacobson, I, et al. 1999 'The Unified Software Development Process', Addison Wesley