Planning to be Agile?

A discussion of how to plan agile, iterative and incremental developments

Mark Collins-Cope

Ratio Group Ltd.
17/19 The Broadway
Ealing W5 2NH
London

Email: info@ratio.co.uk
Web: www.ratio.co.uk

We Know the Object

Table of Contents

1. Introduction

Planning an agile, iterative and incremental software development involves many concerns, trade-offs and judgement calls. In this paper I discusses a number of agile planning principles to help you get it right.

2. Summary points

When approaching an agile software development:

- plan in appropriate detail – in depth for the short term, in broad strokes for the long term
- negotiate with scope – when faced with an imposed deadline
- customer dictates priority – of what should be implemented when
- plan follows reality – using detailed increment plans
- feedback is vital – plan to get feedback to mitigate your risks
- use three types of release – internal, investigative and production
- plan to refactor when necessary – to stop design rot setting in
- trade-off the costs and benefits of incremental development
- consider high impact design decisions during early increments

3. A short story

A few years ago I undertook a long car journey, driving from London to a friend’s wedding in middle Finland. It was an interesting journey, by land to Newcastle, by sea to Norway, by land through Norway and Sweden, then by ferry across to Finland. As it was a wedding, I had to plan the journey in advance to arrive the night before the festivities -- which are somewhat raucous in Finland!

Accordingly I planned out my route in great detail, estimating the mileage and putting approximate arrival times at various mid-points, based on an average speed of about 50 miles per hour. All went well, until we hit Norway. Not being familiar with the Norwegian road system, I hadn’t anticipated just how slow moving traffic would be along the single carriageway winding fjord roads -- it was literally impossible to overtake the many slow moving lorries.

Well behind schedule we arrived in Oslo to stay the first night. The next day we got up especially early, and continued driving through to Uppsala in Sweden, where we were due to catch a ferry. Unfortunately, events overtook us again, and my co-driver managed to fall asleep whilst driving. A wide eyed old Swedish couple looked out of their windows in amazement as my car went through their roadside hedge and came to a crashing halt in their garden. Perhaps more surprising was the fact that they came out and immediately offered we come into their house, where they provided us with tea and biscuits as we overcame our shock! Needless to say, the additional twelve hours it took to arrange car towage, deal with the local police and finally get a hire car meant we missed our ferry and the wedding. The best laid plans of mice and men…

The moral of this little story is that no matter how hard you plan, reality always intervenes to mess things up. As with any serious undertaking, there were many unknowns on the journey, and some events that just could not have been predicted in advance. And the further into the future you try to plan, the less likely it is that your planning will be of any real value.

Which brings me nicely onto my first principle of agile project planning: *plan in appropriate detail:* in depth for the short term, in broad strokes for the long term.

4. Three types of plan

When planning an agile software development project, three levels of planning and estimating suggest themselves:

The first of these, the *initial scoping plan*, is basically about getting an overall feel of where your project is going (this is easy when undertaking a car journey, but somewhat more difficult when developing software). In an OO context, it will likely involve identifying -- but not elaborating -- potential use cases (or user stories or features) and coming up with some broad estimates for these.

The second level of planning, the *broad increment plan*, is about prioritising and scoping. The customer -- who, following the principle *customer dictates priority,* is responsible for determining what is going to be implemented in what order with the project team and allocates use cases to increments (aka iterations) according to business priority and available time and effort.

At this point in the planning process, it is important to realise that many projects are doomed to failure by the imposition of completely unrealistic timescales and deadlines. There is always pressure to get software written quickly -- that's life -- but many projects are thrown into an immediate state of panic from which they never recover. When the pressure mounts, and the deadline is fixed, apply the agile planning principle *negotiate with scope.* We'll discuss issues surrounding the ideal duration of an increment, and the degree to which one should undertake upfront analysis and design, later in this paper.

The final level of planning is the *detailed increment plan*. This is a detailed plan for the next increment, and will contain a list of tasks (based on how to implement the chosen set of use cases) and who is responsible for them. Adopting the principle: *plan follows reality,* later detailed increment plans will take into account feedback on estimates, feedback from the customer, potential design improvements, new requirements and bug fixes.

5. Feedback is vital

Perhaps the most important principle of agile planning is that *feedback is vital*. Feedback is vital in mitigating the many potential risks that face software developments: risks of misunderstandings in functional requirements; risks of an architecture being fundamentally flawed; risks of an unacceptable user interface; risks of analysis and design models being wrong; risks the team don't understand the chosen technology; risks that demanding non-functional requirements aren't met; risks the system won't integrate properly; and so on.

To reduce risk, we must get feedback. The way we get feedback is to create a working version of the system at regular intervals -- per increment in terms of the earlier planning discussion. By doing this we ensure we know what risks the project is *really* facing, and are able to take appropriate mitigating action. At the same time incremental development forces the team through the full project lifecycle at regular intervals -- thus enabling them the opportunity to learn from their mistakes before it is all too late.

6. Three types of release

By this point, you may be beginning to wonder about the overheads associated with all these software releases. It's important here to understand that all releases are *not* full production releases. Following the principle of *three types of release,* an increment may culminate in: an *internal* release, seen only by the development team, which is used to mitigate team and technology related risks; a customer visible *investigative* release, given to the customer to "play with", which is used to mitigate usability and functionality related risks; and a customer visible *production* release -- intended to be used in anger.

Whilst a production release may have higher overheads associated with it -- full system and acceptance testing, production of user manuals and user training, etc. -- the overheads associated with other types of release can be kept to a minimum. Not that there isn't some cost involved -- as we'll see later -- it's just that we consider this cost to be acceptable given the benefits.

7. Rotting software

More traditional (non-agile) software developments lifecycles generally end with a large and somewhat undefined phase called maintenance (which developers often strive to avoid like the plague). Maintenance projects tend to focus on two types of work: bug fixing and new requirements. These bear not a passing resemblance to some of the activities that will be identified during detailed increment planing.

One of the major problems associated with software maintenance is design decay. Software design is fundamentally an optimisation problem based on a defined set of functional and non-functional requirements. During design, we try to come up with the software structure that best implements these -- making a multitude of trade-offs along the way. The “best” design solution is dependent on the requirements we’re trying to implement. If the requirements change, so the best solution changes. During maintenance, slowly but surely, software “rot” often starts to set in, until eventually the software ends up so brittle is becomes impossible to change without major additional costs being incurred.

Coming back to agile development, new requirements may have to be dealt with on a per increment basis. These may have been identified in the *initial scoping plan*, but deferred for later consideration due to their instability or low business priority, or perhaps they’ve only just been thought of. By adopting an agile approach we are run the risk of software rot setting in at a far earlier stage, unless we adopt some practises to stop this happening.

We stop software rotting by following the principle: *plan to refactor when necessary*. Refactoring [8] is a development technique involving changing the structure of a software system without changing its functionality. We use refactoring to minimise software rot as new requirements come up:

- firstly, get a good understanding of the new requirement and the existing design
- analyse the design and work out what changes need to be made to accommodate both the old and the new requirements
- *without* adding the new functionality, implement the design changes, undertaking (hopefully automated) regression testing to make sure the system hasn’t been inadvertently broken
- implement and test the new requirement

Things get a little more hairy when populated databases are involved -- and whilst a detailed discussion of these issues is beyond the scope of this paper, you might like to look at [2] for more information on this.

8. No free lunches

Although not discussed greatly by the more vociferous proponents of agile software development, it is perhaps apparent from our discussion of the need for refactoring that agile development comes with some costs of its own.

Some years ago, Barry Boehm published a famous paper, which demonstrated that the cost of software change increased by an order of magnitude during each stage of the development lifecycle (requirements, analysis, design, integration, testing, and live implementation). Proponents of methodologies like XP (eXtreme Programming) [6] claim that this “cost curve” has been flattened over the years by improved software development techniques. Whilst it is true to *some* degree that good design practises can and should be used to reduce the cost of change (and I think the jury is out on exactly how much), there is clearly a cost associated with refactoring: if we'd designed in the functionality upfront, we wouldn't have to refactor our code at all!

Having said that, some refactoring is almost always necessary: even if you did all analysis and design upfront, you'd still need to refactor as the inevitable change requests appeared, and refactoring as a technique clearly mitigates the risk of accepting change -- something we try to do during agile developments. So we're faced with a trade-off -- which is expressed in the agile planning principle: *trade-off the costs and benefits of incremental development*.

9. All changes are not equal

Following on from this, it is all the more important to understand that some changes will have a *high* cost associated with them. These changes are most likely related to implementing non-functional requirements, such as the need for concurrent multi-user access, security, auditing, etc., late in the day.

The costs associated with such changes are high because they cut right across the software, unlike pure functional changes that are likely to be localised. Changing a flat file, single user system into a multi-user relational database system is no trivial task, and will affect the majority of components in a system. The more of a system that has been written, the greater the cost of these changes will be -- which leads us to the our next principle *consider high impact design decisions during early increments*. Note that as *customer dictates priority*, you will likely need to discuss trade-offs with them.

10. Price fixing

Many software houses are used to undertaking large, fixed price developments based on a set of requirements their customer has produced in advance -- and this is a common objection to adopting an agile approach to development. From a customer perspective, the motivation for this is not so difficult to understand, competitive tenders can be issued, and costs can -- apparently -- be ascertained up front.

Closer inspection, however, reveals that this approach assumes that the cost of the final development will actually be what was quoted upfront -- the presupposition being that the customer has complete and detailed understanding of what they want. There are two major points to considered here.

Firstly, whilst project risks such as architectural inadequacies and staff ability are genuinely borne by the supplier -- it is the customer who will ultimately pay if things do go wrong. They will pay both in terms of the time they have put into the project, and in terms of not getting the system they need. The latter point translates directly into lost business benefit -- otherwise why did they want to develop the system in the first place?

Secondly, in every project of this nature I've been involved in, the initial requirements are incomplete and very often unclear. In this situation, the customer effectively takes on the requirements and usability risks themselves. Be aware that "clarifications" or "additions" to requirements will be subject to rigorous change control, resulting in additional project costs. This is all the more likely to happen if aggressive competitive tendering has driven down the quoted price of development.

A surprisingly simple solution presents itself to this dilemma. Using an agile approach, the cost of each *individual* increment can be fixed in advance. This both gives the customer flexibility and also ensures that costs are kept under a degree of control. The customer also gets feedback they can *really* trust at regular intervals -- in the form of investigative or production releases of the software (how many customers do you know who really understand analysis and design documents?)

11. Size matters?

During my earlier discussion, I mentioned that the ideal increment size is still the subject of some debate. Agile proponents suggest increment durations of a couple of weeks to a couple of months -- with a preference for the shorter timescale. Closely related to this is the issue of just how much design we should do upfront and balancing the cost and benefits if incremental development.

To blur the picture further, we could undertake, say, a month of pure up-front analysis and design whilst simultaneously doing some architectural investigation, and deliver our working software in two weekly increments thereafter. But the questions still remain: just how much upfront design should we do, and how long should our increments be?

There is, unfortunately, no simple formulaic answer to this question. One has to consider the particular risks a project faces, the stability and business importance of a given set of requirements, external constraints like the ideal time to market, and so on.

However, pulling together some of the issues discussed in this paper:

1) undertake a project risk analysis, and ensure that early increments are targeted at getting feedback on these risks, noting that these increments are likely to culminate in either *internal* or *investigative* releases of the system. Determine which risks you really need to mitigate using incremental delivery, and which you're prepared to "take a punt" on.
2) ensure you have a good understanding of the high-impact decisions your project faces, and try to schedule work on these aspects of the system during the early increments

3) consider, with your customer, which requirements are the most stable and business critical; try to schedule the first production release to accommodate these; try to avoid dealing with unstable requirements early on.
4) try, in parallel with the activities in (1) and (2), to do just enough upfront analysis and design of *stable* requirements to reduce the cost of refactoring to a minimum; if this starts to look like its going to take too long (say over one month) schedule a number of functionally cohesive analysis and design activities as the project progresses.
5) try to keep the time to each production release to a minimum, if possible undertaking a production release at least every three months.
6) try to keep increment duration as short as possible -- from two to six weeks -- but not so short as to deliver nothing of concrete value. Where possible keep consistent increment duration to establish a project rhythm.

So on a low technology risk, single user windows application with about 10 use cases, two experienced developers, and a customer bright on ideas but is bad on detail -- we might do a week or so of upfront analysis and design, followed by two week increments each delivering say 2 or 3 use cases worth of functionality for the customer to review, with a full production release every couple of months.

On a higher-technology risk, larger project, with a dedicated and capable customer, a stable set of requirements, and a team of bright but inexperienced developers, we might undertake two technology verification increments of about two weeks. In these increments we'd also round trip the whole development process, delivering some functionality at the end of each, to make sure the team understood just how their analysis fed into design and then into code. As we became confident that they understood this, we could undertake some larger chunks (say one month) of analysis and design to reduce the need for refactoring, then deliver the system in functional chunks every month, targeting a production release for every three months.

12. Summary

In this paper, I've discussed how planning an agile software development involves understanding and mitigating risk, whilst trading off the benefits of this against the costs it incurs. I've suggested a number of agile planning principles to assist in this process, and outlined the broad approach to adopt when considering how to plan your project.

The question that now remains is are *you* planning to be agile?

13. References

1. Cockburn, A, *Agile Software Development*, Addison-Wesley, 2002.
2. Ambler, S., www.agiledata.org/essays/databaseRefactoring.html
3. Rosenberg, D. "Use Case driven modelling using UML"

4. Schwaber, K, Beedle, M. *Agile Software Development with Scrum.* Addison Wesley, 2002.
5. Ambler, S. *Agile Modelling*, Wiley, 2001.
6. Beck, K. *Extreme Programming Explained,* Addison Wesley, 2000.
7. Ambler, S., Rosenberg, D., Collins-Cope, M., Stephens, M., *Agile Development with the Iconix Process*, Apress, Due for release Summer 2003.
8. Fowler, M *Refactoring – Improving the design of existing code,* Addison Wesley, 1999.
9. Larman, C *Applying UML and Patterns – An introduction to OOA/D and the UP,* Addison Wesley, 2002.
10. Other related papers: see www.ratio.co.uk/techlibrary.html.

14. Author

Mark Collins-Cope has been working in the Software Engineering for over 15 years. He has undertaken a variety of roles within the software industry, including: project management; analysis of requirements; technical architecture, software design and development, etc. Mark is now Technical Director of Ratio Group Ltd, a UK based training, consultancy and development company specialising in object and component technologies (see www.ratio.co.uk for further information), where he combines his management role with ongoing involvement in complex technical projects. Mark speaks regularly at a variety of international events, and can be contacted on markcc@ratio.co.uk.

The contents of this paper are based on the one-day Ratio Group course - " A Management Introduction to Agile, Iterative and Incremental Development using UML" Contact Ratio on 0208 579 7900 or info@ratio.co.uk.