Agile Fixed Price Projects part 2:
"Do you want agility with that?"

Pascal Van Cauwenberghe
Nayima
pvc at nayima.be

Introduction

Most proponents of "agile software development methods" [Highsmith 2002] [Beck 1999] will tell you not to do fixed-price projects, as they are bad for providers AND customers. There is some truth there, but it's still a cop-out. What if you do fixed-price projects? Can't agile methods help you? Sure they can.
I will describe some "agile" tools that I've used to improve my fixed-price project method. They have allowed me to get better results and increase the range of projects that I'm prepared to handle under a fixed-price contract.

Question 1: Are we all committed?

I can add some measure of agility to fixed-price contracts, but this requires an even greater effort and involvement from the customer than usual. I can only do that if the customer is able and willing to put in this effort.
A good indicator of a committed customer is a hard release date; e.g. "The project must be done on <this date>, because that's the starting date of the marketing campaign" or "The project must be done of <that date> because that's the date in the fixed-price contract between the customer and its customer". A project without a firm end-date is a warning sign, as the customer is under less pressure to do their part of the work.

Question 2: Will I get timely feedback ?

I will need timely feedback from the customer. For example: they will have to perform regular acceptance tests and report any issues within a few days. This enables my team to fix bugs rapidly and keep bug counts low.
All of these commitments are put into the contract:

- How often the customer must be available to answer questions.
- When the customer will receive new releases.
- The response time for acceptance test feedback and decisions to be made.
- The dates certain information must be provided by the customer.

The aim is not to over-regulate communication between the customer and the development team, but to agree on maximum communication latencies. If the feedback and communication latencies become too large, we can't steer the project.

Sales tip 1: Many small projects are better than one big project

It's a well-known fact that project success rates are higher for small projects than for big ones [Johnson 2002]. Small projects are easier to oversee, require fewer people, handle fewer requirements, estimation errors are smaller, and they lead to tangible results faster… I prefer smaller projects, lasting a few months, requiring a handful of people. But what if my customer has a really big need? Do I need to take on that extra risk that a big project brings? Maybe not…
I always try to reduce the size of a project to a level that I'm comfortable with. Does the customer really require all that stuff? First of all, we have to prioritize the requirements: what is crucial, what is important, what is nice to have? If we just do the crucial stuff, could the customer use the product? If not, what do we need to add? What would be enough for a first, useful release?
Customers are often surprised when I do this, but there are many advantages for them:

- Project cost is reduced if we can drop or postpone some features
- The users get the software earlier than expected, as the timing is reduced
- Project risk is reduced as we work on fewer requirements and concentrate on the high value features
- The customer can evaluate the outcome of the project sooner
- They can delay their decision about the requirements that are not in the first release. At that time they will have more information and knowledge to make a better decision. This allows the customer to "Decide Later" [Poppendieck 2003].

I get some advantages too:

- My risk is reduced, as I have to estimate and handle fewer requirements, get feedback sooner and have a smaller team.
- I can prove myself and gain the **trust** of the customer by delivering something worthwhile. Most of the tips in this text rely on a constructive, open and trusting working relationship with my customer. A

first small, successful project is the perfect way of earning that trust and building that relationship.

- If it doesn't work out, both parties' loss is small.

I don't want to do projects that are too small, either. Projects of only a few days are very hard to get right, as there's no room for compensating for missed estimates or problems. There is one disadvantage for me: I've just arranged to earn less or have my income delayed. This is the subject of the section "Warning: The money trap".

Implementation tip 1: Let the customer sort the requirements

I've got this long list of requirements in the specification. As described in "Sales tip 1: Many small projects are better than one big project" they have been categorized as crucial, important and nice to have. I'll tackle them in that order. But in what order do I implement the requirements in each category? I let the customer decide.

When the customer and I lay out the project plan, the customer gets to choose the order. It's a simple process: first you do the crucial features. Just ask the customer "which one is the crucialest of them all?" This one goes first. Then the next most crucial requirement, and so on. Then the important stories. How does the customer choose? By comparing the value each requirement will bring. It's usually possible to compare two requirements and decide which one is more important.

Can I always implement stuff in the order that the customer chooses? Aren't there any dependencies, requirements that have to be handled first to reduce risk or dependencies between requirements? Yes, but not many, if you really try to keep each feature independent from the others. In those few cases where there are dependencies or risks, we can increase the feature's priority and adjust the planning accordingly.

There are several advantages to this technique:

- We reduce the risk because the least important requirements are tackled near the end of the project, where there's most schedule pressure.
- The customer can give feedback on the most important features first, when there's most leverage.
- The customer sees value being added to the system from the early stages of the project. They might even be tempted to use the system before it's finished (see "Implementation tip 7: Frequent releases, incremental delivery"), thereby learning to handle incremental delivery.

This technique is also used in the Extreme Programming "Planning Game" [Beck 1999] and SCRUM's "Product backlog" [Schwaber 2002].

Implementation tip 2: Requirements as stories. Don't sweat the details

I don't specify each requirement in great detail. Just enough detail and no more. Don't I need all these details to estimate and plan correctly? Not always. For example, this is from the planning of a fixed-price project: "The user can view 5 types of reports about orders. These are shown in a separate browser window. Cost 10 days. The customer shall specify the parameters, layout and data to be shown before <the latest date implementation of this feature must start>".

How do I know 5 types of reports is enough? That's what's usually required for customers and projects in this domain. How do I know it will cost 10 days, if I don't even know the parameters, layout, data or queries? I know the kinds of reports that are useful in this domain and I've implemented them several times before. From previous projects I know that it typically takes somewhat less than 10 days to complete these reports. Let's estimate 10 days to have some slack.

What do we gain by this technique?

- The specification becomes smaller, easier to write, easier to understand and verify by the customer.
- As there is less work to do on the specification, we can get on to the implementation part earlier and thus deliver value earlier.
- The customer can delay decisions. At that time they will know more and be able to specify more precisely what they need.

This technique only works under the following conditions:

- I have a pretty good idea of what's required without the details
- I have a constructive, trusting working relationship with the customer. I must trust that they will complete the requirements in time and not ask unreasonable requests. I must trust that there will be no problems when it comes to accepting the implementation based on this incomplete specification. The customer must trust me to implement this requirement following the spirit of the contract, not its letter.
- I have some slack to handle unforeseen problems. For example: they really require

6 reports, the queries are unusually complex…

This is the same technique as "User Stories" in Extreme Programming, where requirements are detailed when and if needed.

Implementation tip 3: Exchange Requests

It's clearly not possible to always define the perfect specification before the start of the project: we make mistakes, we forget things, and we learn new knowledge, the environment of the system changes… One way to cope with this is by using "*Change Requests*", a technique to make changes to the specification. Because each change request increases the time and cost of the project, they are best avoided if you want a successful project and a happy customer.

"Exchange Requests" work like this: each time the customer and I need to change the specification, we make a Change Request and estimate its effort (and thus cost). When the customer approves the change request and its estimate, they can add the new features to the project **if they first remove functionality requiring at least the same effort**. We can only remove unimplemented features that the development team is not working on. For example, the customer can add feature X (cost 5 man days) if they first remove features A (cost 3 man days) and B (cost 2 man days).

What are the advantages of this technique?

- We keep the budget and timing constant (by definition). The development team and the customer have the satisfaction of finishing a job on time, on budget.
- We are able to change the specification flexibly to deliver what the customer needs, not necessarily what they asked for. We do not deliver what was specified, but we do (at least) as much work as agreed. If we've revised the specification, the new features were more valuable than the old ones, or the customer wouldn't have swapped them in. This means we deliver something that's at least as valuable than what was agreed.
- We put more thought into changes to the specification: the customer has to think very carefully if the new feature is really worth more than the feature being taken out. Therefore, we expect fewer, but more useful, changes to the project.
- We shorten the feedback loop between functional changes and their effects upon budget and timing, so that they are immediately visible. The customer project manager becomes more responsible for budget and timing.

Implementation tip 4: Put dropped requirements into a follow-up project

There seem to be no disadvantages to the Exchange Requests technique, or are there? What about those requirements that were dropped? What if they were crucial to the project? What if they were useful?

Well, if they're really crucial or useful, the customer will just have to define a follow-up project to implement these features. This is a new project, with a new specification, a new planning (we can reuse the estimates of the features that were dropped) and a new contract. I can implement it as soon as this project is done.

What's the difference with the project that results from change requests? The customer gets the same features; the provider bills the same amount. These are the differences:

- With change requests we have one project that is late and over budget. With exchange requests we have two projects that are on budget and on schedule. I know which one I would rather be the project manager of.
- Unless really crucial requirements have been dropped, the customer can use the software on the date planned.
- Usually, the customer will learn that some of the original requirements were not really crucial and can be dropped altogether after the exchange. Thus, the project is often shorter and less costly with exchange requests than with change requests.

Implementation tip 5: Let the customer use the software before the follow-up project

If there are no crucial requirements to implement, just some useful or "nice to have" requirements, I advise the customer to use the software before defining a follow-up project. They will get lots of useful feedback, which will allow them to define a far better follow-up project. They will add and drop requirements, based on actual use. They will have learned what works and what doesn't.

Sounds perfect, doesn't it? Except for one thing: we've fallen into the "*money trap*".

Warning: The money trap

What is the "*Money Trap*"? Simply put: "Money received now is worth more than money received later." The 100 Euro in my pocket is worth more than the 100 Euro I'll receive in a year. Why? I can invest the 100 Euro in my pocket to bring me some return, say

6% in a year. Within a year, when I get that 100 Euro, the money I receive now will be worth 106 Euro.

Two of the tips, "Sales tip 1: Many small projects are better than one big project" and "Implementation tip 5: Let the customer use the software before the follow-up project", will delay part of the execution of the project, and therefore its payment. For example, if the customer delays a follow-up project by six months to use the software, I will be worse off, because my income has been delayed for six months. Even worse, the customer might realize that the follow-up project is not really needed! Forget that income.

For the customer it's all benefit: they get to change the specification; they get to drop requirements that are discovered to be unimportant; they can postpone decisions until they have more knowledge and experience; they never pay more than necessary; projects are never late…

All of this makes a happy customer. Happy customers have a habit of awarding projects to providers who make them happy. Forget the small loss you make now, invest in a long-term relationship with your customer.

In my experience, the customer often gets lots of useful ideas for improvement and extension by using the software. Thus, the follow-up project is sometimes larger than it would have been if they hadn't used the software first. So, in the long run I earn more…

Investing in quality and your relationship with the customer pays off, if you can afford the initial investment.

Implementation tip 6: I'm the onsite customer

The top three remarks I get about XP are: "It will never work", "It doesn't work with fixed-price contracts" and "**You will never find an onsite customer**". What is an "onsite customer"? They are the interface between the development team and the organisation whose requirements the team is implementing. Someone the developers can ask questions, who can prioritize requirements and make decisions in name of the customer organisation. And they're supposed to be available at all times, hence the "onsite". For the development team, this is an ideal situation: they only have to deal with one person; they can clarify any requirement when needed and ask for business decisions to be made.

Where can you find someone who has the necessary knowledge and authority? Who can spend all their time with the development team?

I lack the authority and some knowledge, but I have to spend time with the team anyway, so in most cases **I will have to do**. I know the domain, I have the experience and I've gained a lot of information about the customer during the sales and specification process. I expect to be able to answer most of the developers' questions. I've already prioritized the requirements with the customer, so that should not be a problem. I expect to be able to take many decisions, as I've agreed most of these issues with the customer beforehand. And if I'm unable to answer the question or take a decision, I can always ask the real customer.

Thus, we get a situation where both the developers and the customer can work effectively:

- The specification doesn't have to be very detailed (see "Implementation tip 2: Requirements as stories. Don't sweat the details") and thus easier to create and to understand.
- Most of the team's questions get answered quickly. Most of the decisions get taken quickly. Some are deferred.
- The customer doesn't have to be available all the time. They do have to be available regularly to answer my questions or take some decisions.

I can only do this if I have the necessary knowledge and experience of the domain and of this particular customer. **The most important thing is to know when I don't know** the answer or can't take a decision. Better to take some time deferring to the customer than losing the team's time by sending them on the wrong path.

Implementation tip 7: Frequent releases, incremental delivery

I like to release the software often. Typically the software will be released once per week to the customer project manager, as agreed during the sales process. The team gives a demonstration of the new features; the customer uses these releases to do acceptance testing; the customer gives feedback within a defined delay; the team acts upon this feedback before implementing other features. What are the advantages of releasing so often?

- The development team gets the hang of releasing the software, with all the messy stuff related to installation, database upgrades, backward compatibility
- The development focuses on delivering high quality complete features. Each time the customer accepts a feature the team gets a little buzz of satisfaction.

- The customer can test and accept features incrementally. Each week, some new features are available for testing. All the testing work (and its valuable feedback) is not delayed until the end of the project.
- The customer can give useful feedback from the beginning of the project. They learn a lot from seeing and using the actual product. This knowledge can be used to improve the rest of the project.
- The customer has a real sense of progress.
- The finished features you put into your "burndown chart" are accepted by the customer. They therefore better reflect the amount of work done.

I divide the whole project in 1-month increments. I try to define each of these increments so that it delivers some coherent functionality, organised around some "theme". The project is not complete until all the increments have been delivered ("waterfall" style), but this technique helps the customer to learn to handle incremental development. After a while, as the features and the increments roll in regularly, the customer gets more confidence in the provider and the process.
During their testing they will often discover that these increments are good enough and complete enough to be used by end-users. Thus, they might suggest using the increments. This enables them to get the value of their system earlier than expected. On the next project, they will demand incremental delivery.

Implementation tip 8: Looking back to learn

After each project, we need to take some time to look back, to learn lessons and to prepare for the following project. Project Retrospectives [Kerth 2001] provide a useful format to learn from our experiences.
This is also the time to compare the estimates to the actual time, so that we can improve the accuracy of our estimates.
If we've encountered some new risks and handled them, we should preserve this useful knowledge.

What's the cost of this extra agility?

Under a fixed-price contract, the customer has the security that price, timing and scope are fixed. With the agile techniques described in this article they gain even more advantages: they get value sooner from their system, they can delay some decisions, they see progress (or lack thereof) sooner and clearer and they can flexibly adapt the requirements. What more could they ask for?

Of course, there's this one universal rule "There is no such thing as a free lunch". What's the price and who pays it?

- The customer must spend more effort and time on the project: they must test the features regularly and give timely feedback; they must be available to answer the questions of the team and take decisions in the shortest delay possible; they must actively participate in the follow-up and management of the project.
- The project manager is even more involved than usual: being an onsite customer is hard work; the frequent releases must be carefully reviewed and followed up with the customer; the planning must be updated when exchange requests are included.
- The most important requirement is that there is a constructive, honest and trusting working relationship between customer and provider. It takes a lot of time and effort to build up this relationship: trust must be earned. The best way the provider can earn this trust is to be honest and deliver upon their promises.

Strangely enough, few customers are prepared to put this amount of work into their important projects. They think they can let the provider do most of the work. It should be clear that a software project can only succeed if both parties do their part of the job. If the customer is not willing to spend the effort, I'm not willing to risk failure by accepting the project.
Many of the techniques are about handling risk, instead of avoiding it. This requires a leap of faith from the customer who is used to the classical project management techniques. Customer and provider must trust each other and work together to be able to handle and respond to risk. The best way to earn this trust and cooperation is to deliver successfully and increase the customer's involvement gradually.

Thinking about project management

Many of the situations that I've described have the form "*If you do X it will bring benefit in the short term but has disadvantages in the longer term*" (for example the use of "Change Requests") or "*If you do Y you will see negative effects in the short term and positive effects in the longer term*" (for example "Implementation tip 5: Let the customer use the software before the follow-up project"). These forms will sound very familiar to those who practice "**Systems Thinking**" [Weinberg 1997] [Senge 1990].
Experience and examining these situations as systems has taught me that it's important to act upon the causes of problems in such a way that the long-term effects are positive. I'm not just

doing this project, but will be doing many more. For example: I can drive my team really hard and exhaust them to deliver on the current project. But then they'll be in no shape to deliver the next one. By winning some time on this project, I lose more time on the following project.
The tricky problem is that it's hard to "do the right thing" when I'm under pressure to deliver. I'm always tempted to take the shortcut, to do what brings me the short-term gain, to fix the symptom without fixing the cause. The way I counteract this, is to impose "rules" upon myself.

Them's the rules

I have described some of the rules I use on fixed-price projects. These force me to work so that I attack causes and not symptoms of problems; they force me to work towards the long-term good of my team, my customer and myself. When under stress, I follow the rules, which avoids the temptation to sub-optimize for the short term.
All of these rules are guided by some "meta-rules".

Always keep the goal in mind: delivering value for the customer

Everything we do, every decision we take must have this one goal in mind. We do whatever we need to do, within the constraints of the other rules, to reach our goal. The task of the project manager is to ensure that the team never loses sight of its goal.

Choose the rules that fit the game

We have seen rules for fixed-price projects; some only apply on agile projects. Applying rules (for example those from "Extreme Programming") where they don't fit doesn't help me, it will harm me. I choose and tailor the rules to the domain, the customer, the team, the technology, and the environment…

The rules are the rules. You don't play the game, unless you accept the rules.

I'm strict about these rules: if you want to play in or with my team, you have to follow the rules. If the customer can't or doesn't want to follow the rules, we don't do the project. If a team member doesn't follow the rules, they're off the team.

Rules can be broken if that's the only way to solve a problem.

Sometimes I have to be a bit flexible and bend the rules. I only do this if this is the only way we have to solve a problem and if, after examining the situation, we agree that this will help us to reach our goal. For example: one of my rules is "No overtime", because I know the negative effects it has on productivity. I'm quite willing to break this rule and work two hours longer to finish something or to meet some deadline. If that's not enough we have to look at another way to solve our schedule problem: more overtime will not fix the cause of the schedule slip but it will lower team productivity.

Every rule can be changed, but not during a project.

No rule is perfect. Some rules become obsolete, others must be updated, and new rules are learned. The rules should capture the knowledge you gain. But I can't run a project if the rules change out from under me or if there's constant discussion about the rules. Regularly scheduled reviews (like the project retrospectives described in "Implementation tip 8: Looking back to learn") are the ideal moment to evaluate and update the rules.

Be honest with the customer, with the team, with myself.

Honesty is the best and easiest strategy in the long run. If I have a problem, I shouldn't hide it, but solve it. I can only solve the problem if I know and admit I have one. Hiding problems from the customer or the team doesn't work: they find out eventually, most likely when the product must be released. Why not ask for their help in solving the problem?

Why would I want to do fixed-price projects?

We've seen all the advantages fixed-price projects can bring to my customers, at least if they and my team implement the project correctly. What's in it for me, except all this hard and risky work?

- Because of the fixed schedule I can more easily plan different projects: if I know project A will run from January 15th to August 20th, I know that I can commit my team to implement project B from August 25th on (taking some time off between projects for the end of project party and the retrospective).
- Because of the fixed schedule, my costs are predictable.
- Because of the fixed budgets my income is predictable.
- Some customers are more likely to do the hard work of thinking about their real requirements and taking the tough

decisions if there are clear threats to the budget or schedule if they don't.

Conclusion

We can apply "agile" techniques to handle risk instead of avoiding all risks. These techniques can be used to add some flexibility to a fixed-price contract, without losing its advantages. These techniques can only be applied if there is sufficient trust and commitment from the customer. The provider has to earn that trust and commitment, by delivering upon promises and gradually increasing the involvement of the customer.

Building on that trust, we can go from projects where budget, time and value are fixed to projects where budget, time and **minimum value** are fixed.

Customers who have experienced these gains by working agilely don't want to work any other way again. They don't want to work with anyone else again.

References

[Beck 1999] "Extreme Programming Explained", Kent Beck – Addison Wesley 1999

[Highsmith 2002] "Agile Software Development Ecosystems", Jim Highsmith – Addison Wesley 2002

[Johnson 2002] "Collaborating on Project Success", Jim Johnson, Karen D. Boucher, Kyle Connors, and James Robinson – Software Magazine February/March 2001. Online at http://www.softwaremag.com/archive/2001feb/CollaborativeMgt.html

[Kerth 2001] "Project Retrospectives: A Handbook for Team Reviews", Norm Kerth – Dorset House 2001

[Poppendieck 2003] "Lean Development: An Agile Toolkit for Software Development Leaders", Mary Poppendieck - Addison-Wesley To Be Published

[Schwaber 2002] "Agile Software Development with SCRUM", Ken Schwaber and Mike Beedle - Prentice Hall 2002

[Senge 1990] "The Fifth Discipline", Peter Senge – Random House 1990

[Weinberg 1997] "Quality Software Management: Systems Thinking", Gerald Weinberg – Dorset House 1997