Strategies for replacing systems in agile projects

Niklas Bjørnerstedt
Leanway
Oslo, Norway
niklas@leanway.no / *twitter.com/smalltalk80*

Abstract

In replacement projects one of the biggest questions is what strategy to use in replacing the old system. The simplest strategy is to wait until the new system is "at least as good as the old one" before switching. Although this strategy sounds compelling it has many serious drawbacks. An alternative strategy is based on what I call a "Minimal Deployable Entity". Switch systems when you have just enough to be able to survive with the new system. The concept is similar to "Minimal Marketable Feature" but more focused on deployment strategy. This paper uses two projects as concrete examples of patterns and principles that can be used to define a strategy for a Minimal deployable entity.

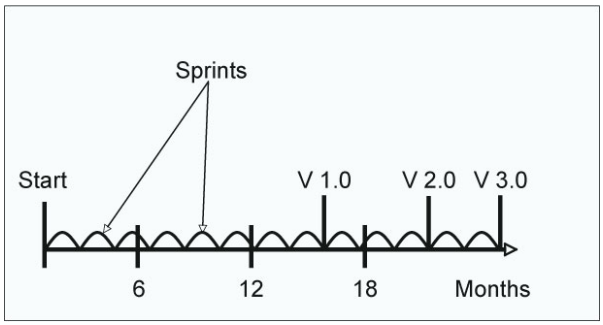
1. Background

Many of the standard textbook examples of agile development projects assume that it is possible to deploy a first version after just a few weeks of development. My experience as a product owner in a number of replacement projects has been very different. In each project we have grappled with how to best sequence development in order to get something into production in a reasonable time frame. The trade offs were difficult and the consequences of our choices often surprised us. I was amazed to find that this problem received so little attention. Excellent books on agile planning; such as "Agile estimation and planning"[1] do not mention replacement projects. This paper describes a selection of the strategies that I have used as product owner in order to arrive at something I will call a "Minimal Deployable Entity" (MDE). The strategies are then distilled into patterns with generalized strengths and weaknesses.

This paper will not attempt to describe why it is so important to deploy early and regularly. I will simply take for granted that the reader understands this.

2. Minimal deployable entity

In a typical replacement project the development team is faced with a large, monolithic and undocumented legacy system. Knowledge of the workings of the legacy system is fragmented and partial. The users of the legacy system have a love/hate relationship with it that can be summed up in two conflicting remarks: "It will be great when we finally can pull the plug on this old system" and "I will not accept that you replace this system until the new one is at least as good". This leads to an initial release plan that looks something like figure 1:



Release plan with many sprints before the first release to production

A **MDE** is the smallest set of features that must be developed before a system can be deployed into production. To count as being "deployed to production" the system must be used by at least one user set to perform real tasks within a particular workflow or set of operations. The term is similar but not identical to **Minimal Marketable Feature** (MMF) as defined in the book "Software by Numbers" [2]. The primary difference is that while MMF focuses on maximizing the creation of value over time, MDE is also focused on deployment strategy and feedback. On a meta level the difference between MDE and MMF is in the definition of value. MMF is focused on the value of

developed functionality while MDE focuses on the value of early feedback and risk reduction.

3. Case 1: Research application system

The first example of how to work with MDE is from a project that was going to replace a system that handled research applications and funding. I was hired as the product owner and had the responsibility of figuring out the optimal sequencing of the system releases. A software company was contracted for the development. The legacy system was pre-Internet and inherently "introvert". The only human interface was designed for internal employees. Researchers had to send in paper applications that would then be punched in to the system. External reviewers of proposals also had to work by receiving and sending paper documents. Progress reporting for projects that received funding was equally primitive.

We decided to use four main strategies in the first phases of the project:

- We would employ a "shared database". The new system would be based on the existing database. This decision was based on the observation that the existing database platform and structure were acceptable for the new system.
- The system would be built according to the sequence of the underlying workflow: submission of application, peer review, funding decision, progress reporting, and finally end report. The first release would be a web-based system that allowed researchers to assemble and submit applications.
- The first release would only be used for a small portion of the applications. We would only handle the simplest type of research proposal and would limit the number of users that were allowed to send their applications through the system. This strategy was facilitated by the shared database.
- Only functionality that was essential was included. Features that were valuable but not essential were excluded from the first release. My authority as product owner was severely tested since every stakeholder had a different definition of “essential”. I found it to be an advantage that I was external to the organization since it made it easier to argue that my decisions were not based on internal politics.

These strategies worked well for us. We were able to deploy the first version after four months and it received about 100 applications in the first months of operation. This gave us valuable feedback and one surprise. We now had to support a system in production that was characterized by very fixed deadlines (submission dates) and highly demanding users (researchers). This made the prioritizing of features in subsequent releases much more difficult. The biggest problem was in verifying exactly how the legacy system used the database. Almost all data validation was in the system and not in the database. A set of data that was accepted by the database would cause the legacy system to fail. As with most legacy systems, there was also a lot of dormant functionality that was either obsolete or just rarely used. Ignoring rarely used functionality could lead to nasty surprises so it was important to differentiate it from obsolete functionality.

4. Case 2: Book club system

In this project the customer wanted to replace a book club system. The goal was to integrate the book clubs into an Internet bookshop in order to be able to increase cross selling between the two channels. As in case 1, I was hired as product owner and a software company was to do the development. The existing book club system was operated as a third party ASP service that was not accessible for any type of integration. There were many different types of book club in production.

Because of the closed nature of the legacy system we could not go for any strategy that required integration with it. The first strategy we considered was to develop support for a completely new book club. In that way the first release would not have to migrate any users or need to have any of the functionality that comes to play over time in a club. This strategy had to be rejected since the customer was not planning to launch any new clubs. There was no point in delivering something to production that no one would use.

The second strategy considered was to take an existing club and split it so that new members would be handled by the new system while existing members would stay in the old one. Again, this would have allowed us to focus on a core set of functionality. Unfortunately this strategy also turned out to be unfeasible. The killer detail this time was customer support. When customers call in they often do not know their customer id (they often do not know the name of their club!). With a club split across systems, customer support would have to look for the customer in both systems.

The strategy we finally chose was to develop enough functionality to be able to migrate one of the smallest clubs. This was a larger MDE than we had hoped for since we now had to support both new and existing customers as well as develop the migration code. This strategy worked well but there were some drawbacks. Customer support still had to cope with two systems but now they knew which system to use as long as the customer remembered which club they belonged to. Some customers were split across both

systems since they belonged to more than one club. Finding and rejecting bad customers was complicated by the fact that records of bad customers resided in two systems.

5. General principles

After working as product owner on many projects I have come to realize that there are a number of principles that should be heeded in most replacement projects. These principles are not always applicable but they should always be considered.

5.1. Is it necessary?

The first thing that one should ask when planning a replacement project is "Is it really necessary to replace?" Augmenting a legacy system is often better than replacing it. My personal rule of thumb is that it will be twice as expensive as estimated to replace a legacy system. Many projects decide to replace legacy systems without really investigating the alternatives. In the research application system we ended up replacing a large part of the legacy system but not all of it.

5.2. The value of delaying

Knowledge of the legacy system is often poorest at the outset of a project. If it is not possible to identify a reasonably small MDE one should consider a delaying tactic. Develop a peripheral function that can be used both with the legacy system and the system to be. This will give the project time to learn more about the legacy system and come up with better strategies to reduce the MDE. The biggest problem with a delaying strategy is selling it to the customer's management. They will have very strong incentives to start immediately with the core parts of a new system. The value of building a peripheral function is primarily indirect (risk reduction). The costs on the other hand will be up front since development of the core functionality will be delayed. There is also a cost associated with developing the function to work with the legacy system and later modifying it to work with the new system. The case for delaying is very similar to that of refactoring. Both entail an up front cost that brings long-term benefits.

5.3. The value of pain

Regardless of how you plan to tackle MDE in a project the first rule for minimizing MDE is that version 1.0 **should** hurt. Only include functionality that is absolutely essential. Functions that are valuable but not essential should not be in the first release. The hard part is getting the customer to accept this (both users and management). This is especially true for functionality directed towards external users. Minimizing functionality does not mean that one should release a system that will disappoint users. Two examples of strategies that work with external customers is releasing to pilot users and compensating users for any inconvenience they experience. The important thing is to make all stakeholders understand why the first release must be limited and give them an idea of how long it will take to get a more complete version deployed. In my experience, once the first version is in production it is much easier to prioritize functionality based on business value.

5.4. Limited releases

Try to identify a small group of users that can be used as pilot users. I have found that as few as 6-8 users can be representative for a whole user group. The MDE has to include just enough functionality so that these users can perform meaningful work (more on this later). By limiting the number of users you gain two advantages:

- it is easier to get acceptance for limiting functionality in the MDE
- issues that are identified by the pilot users will not be as critical since they affect so few people

5.5. Verify your strategy

When defining a MDE strategy in a project make sure to test it against all user groups. A strategy can look promising until you see that it will not work for one user group. This was the case with the book club system and the strategy of splitting a club across two systems. Also remember to check for temporal problems. In the book club system we initially thought migration was a simple matter of copying members and a list of books received for each member. We later realized that when we migrated a club there would be members with outstanding payments. Finding a strategy to handle these payments added complexity.

5.6. Scope control

Agile methods such as Scrum emphasize scope control of iterations. It is equally important to control scope with regards to the MDE. A project that always is one iteration from deploying risks loosing all credibility. I have seen large projects fail due to the MDE growing in this way. Controlling the size of the MDE results in some very difficult trade-offs. The multiple iterations within the MDE will produce some very high priority issues. Balancing these against the

cost of growing the MDE is not easy. Reserving a buffer of about 10% for issues that are identified during iterations is highly recommended.

6. MDE strategies as patterns

My experience as product owner has taught me that finding a minimal MDE is in many ways an art. Each project is unique and it is very dangerous to assume that a strategy that worked well in one project will be a good choice in the next one. On the other hand I have found that the strategies that one can use to minimize MDE can be described as patterns. These patterns have inherent trade-offs that are valid across projects. When looking for the optimal strategy for a new project one should consider each of these patterns to see which will give the best results. It is often possible to combine more than one pattern in the same project. What follows is a list of the patterns I used or evaluated in the two projects described earlier. There are many more patterns but the ones described here will serve as examples.

6.1. Shared database

This is a very powerful pattern but it also has some severe trade-offs. The power comes from the fact that by building on the existing database you are in fact augmenting the legacy system. MDE can often be dramatically reduced and there is a fall back option of only partially replacing the legacy system. The most obvious drawback of this pattern is that you have to accept the existing database design and platform. You can add new tables (and in some cases columns) but you cannot change anything until the legacy system is retired. By that time the new system will have grown to the point where it will be difficult to make major changes to the database. A second potential weakness with this pattern is that even if the structure of the existing database is reasonably good, the documentation often is not. It can be very resource consuming to figure out all the implicit rules that the legacy system has in its interaction with the database. Testing will also be much more complicated since you need to test both systems. There is also the risk that the database has corrupt data. This can add greatly to the complexity of the project. On a different level some user sets may be forced to split their work on two systems. This can be very confusing and lead to errors.

6.2. Workflow by workflow

Large systems often support many workflows. By including only one workflow in the first release the MDE is reduced. This pattern has the advantage of being relatively easy to communicate. Users will quickly understand what is supported in a particular release. This pattern does not work well if there are strong dependencies between workflows. In the book club system we left the handling of returned books as a manual process that was automated in a later release.

6.3. Follow the workflow

This pattern was also used in the research application system. We first deployed support for tasks that were vital to the start of a workflow. Subsequent releases expanded support to cover more and more of the workflow. Besides reducing MDE this pattern made it easier for users to keep track of when to use the new system. The pattern works best if the workflow has a very long duration. We could in some cases manage to expand support on a "just in time" basis as a set of applications moved through the workflow. The biggest draw back we encountered was that release dates tended to harden since we had to follow the time line of the workflow.

6.4. New business

In insurance this is called a run-off strategy. The legacy system handles existing business while the new system takes all new business. Over time the new system will gradually supplant the old one. One major advantage with this pattern is that no migration of data is necessary. This pattern was partially used in the book club project. When a club was migrated, outstanding payments were kept and processed in the legacy system. If a customer defaulted on a payment this information was exported to the new system. As described previously this pattern will often cause problems for some user sets that are forced to split their work across both systems.

6.5. Pilot product

This pattern is useful when it is possible to identify a new product (such as a new book club) that can be created in the new system. MDE can be greatly reduced if the new product is simple or if there is a lot of functionality that only comes into play over time. This pattern is often followed by the “Product by product migration” pattern. The main drawback of this pattern is similar to the “New business” pattern. Some user sets will have to interact with both systems in ways that can be very inefficient.

6.6. Product by product migration

Instead of migrating everything in one "Big bang" the products in the legacy system are migrated in small sets. In the research application system we first migrated only a few small research programs. This pattern works well if there are few dependencies between the products being migrated. Again, the main drawback with this pattern is that some user sets will have to interact with both systems. Researchers that applied to more than one program found it hard to understand why the application procedures were so different.

7. What does it take to identify MDE?

Finding the right patterns to use in a project and combining these into a MDE strategy is one of the biggest challenges a product owner will face at the outset of a replacement project. In my experience you need a combination of technical and business knowledge in order to succeed. Technical knowledge allows you to identify possible strategies and assess the development costs associated with each one. Business knowledge allows you to evaluate which of these strategies will be acceptable for the customer.

8. References

[1] Mike Cohn, *Agile Estimating and Planning*, Prentice Hall, 2005

[2] Mark Denne and Jane Cleland-Huang, *Software by numbers*, Prentice Hall, 2004.