# An Agile Approach to a Legacy System

Chris Stevenson[1] and Andy Pols[2]

[1] ThoughtWorks Technologies (India) Pvt Ltd.
Diamond District, Airport Road
Bangalore, India
CStevenson@thoughtworks.com
http://www.thoughtworks.com
[2] Pols Consulting,
5 Granary House, Hope Sufferance Wharf,
St Marychurch Street, London SE16 4JX, UK
andy@pols.co.uk
http://www.pols.co.uk

**Abstract.** We describe how a small, successful, self-selected XP team approached a seemingly intractable problem with panache, flair and immodesty. We rewrote a legacy application by delivering new features, a radically different approach to those previously applied. This proved to be a low cost, low risk proposition with a very high payoff for success. Most importantly it provided users with new functionality quickly that could never have been retrofitted into the legacy system. In the longer term it may give a migration strategy for replacing the legacy system.

## 1 Background

InkBlot is a large financial legacy application feeding dozens of other applications and supporting up to 100 in-house traders on a daily basis. The system was originally developed in the 1990s and the original team disbanded long ago. There are many external apps that talk directly to the database.

There are no clean external interfaces, which means that we have no idea who is connecting to the system and what they are doing in the system. In fact all external access uses the same well-known username and password.

Business logic is distributed across 1600+ stored procedures, some of which are 3000+ lines of SQL, and exist in multiple versions. Whenever a stored procedure was changed, a new version was added because no-one knew who was using the old version.

There are no primary or foreign keys on the 250+ tables, and triggers are used to maintain data 'integrity'. Code was not under source control and written in a mixture of 4GL, C, SQL and unix shell scripts.

## 2 Our Evolving Strategy

There had been several previous initiatives to improve the system. The most recent was an attempt to rewrite a key part of the system in a language that we

knew (Java) on the assumption that this would increase understanding of the system and make it more amenable to refactoring.

This strategy did not work.

With hindsight the reasons for this failure are clear: we were attempting to change the legacy application itself, therefore we were likely to break it before we fixed it. Also since we were rewriting what was already legacy code, we were by definition writing legacy code.

For example if (say) 75% of the code is unused or irrelevant (probably a lower bound in this legacy application) then working on that proportion of the code is wasted time. Even worse, since there are bugs in the legacy application that other parts of the system assume or work around, we would need to reproduce the bugs in our new code.

The refactoring effort was started as a spike, and morphed into a multi-month project - it should have been time-boxed to prevent this. Also note that this approach, even if it could have worked, was slow, demoralising and would have provided (again by definition) zero benefit to the business.

The drive for the original rewrite came from development management and developers not business and users. Management drive was to improve reliability of the system. This was incorrectly interpreted as a mandatory requirement to rewrite existing code. At one stage we were told specifically 'no business value work is allowed'.

When the rewrite failed we decided a new strategy was needed.

*Therefore our rule of thumb is:* Don't reproduce legacy code

Next, we got a good customer proxy[3] who was able to identify the key problem in the use of the system by front office staff. The system was being used as a reporting tool, using stored procedures that would run for up to 5 minutes before producing a result. This frustrated the users who ended up running the reports continually, adding to the load on the system.

When we asked the users what they wanted, they had very clear ideas about what was wrong. Of course none of them mentioned the part of the system we had been working on – to them it was invisible. If we had persisted on the original rewrite we would not have solved their problem. The area of the system we were focussed on was not even the one causing their problem.

*Therefore our rule of thumb is:* Always ask the users what the problem is

Deciding that our users knew better where they were hurting, our next approach was driven by user requirements. We wrote a greenfield application to extract data from the legacy application database and display it to users in a flexible and timely manner. An already successful team was given the task of

[3] Customer proxy: Someone from the development organisation acting as a proxy for the customer, when an onsite customer is not possible.

writing a quick one-week spike to prove that the required data could be extracted in real time. This gained the trust of our customer proxy and gave the team the confidence to continue.

This proved to be very easy and *'How difficult can it be'* became the team's motto.

Our approach was low risk in that we didn't change the legacy application, and so the potential cost of failure was small. However the payoff for success was extremely high. The new system obsoletes legacy application functionality incrementally while delivering regular new features to users.

The new system only extracts the key relevant data from the legacy application, ignoring the irrelevant code and database tables, and makes no changes in the legacy application. This means that the team gains an understanding of the important parts of the legacy system while ignoring those parts that do not matter.In fact we use only 10 of the 250+ tables in the legacy application.

*Therefore our rule of thumb is:* Refactor a legacy application by delivering new business value

## 3 How we Built the Team

We started the Greenfield application with a team of two people who had just successfully completed an unrelated project. They were bored and looking for something challenging to sink their teeth into. The legacy application team's credit bank was at zero, so there was no confidence in their ability to deliver using this new strategy. The new team believed in the new strategy and lobbied project management hard to get a chance to try it. The new team was given permission to start work unofficially to validate the approach.

The first story was to prove to our customer proxy that we could extract the data in real time, without impacting the performance of the legacy application. This proved to be easy, and basic functionality was implemented rapidly, leading to much more confidence in the approach. As time went on and our and confidence grew, we raised our heads above the parapet more and more, until we were able finally to demonstrate the app to our users. Up to that point all conversations with the users had been theoretical.

*Therefore our rule of thumb is:* Incrementally build trust - prove that you can do the hardest part of the system

After the initial spike[4], people around the team were infected by their enthusiasm, and lobbied to join the team. The team grew to 6 and then fought

[4] Spike: An experiment to explore a possible solution to an unfamiliar problem. So-called because a spike is "end to end, but very thin", like driving a spike all the way through a log.

hard to keep that size. This self-selection meant that the team had a unique ethos and passion. Since everyone wanted to be there the team was committed to the success, and members took collective responsibility for the success of the project.

*Therefore our rule of thumb is:* Build a small, self-selected team

Unlike some XP teams that we have seen, we did not allocate cards to specific programmer pairs. Instead the cards were placed on a whiteboard near the team, and we would take them when we had finished another story. Interestingly even the boring cards were picked up early, as the team's pride was at stake.

We initially planned to have one-week iterations, and most of the time that was fine. Occasionally a piece of work would come along that would block other avenues of development. When this happened we planned for a short 'blitz' to complete the work as soon as possible. This meant that some of our iterations turned out to be quite short - some as short as three days. We sometimes finished all of the work planned for an iteration early, and again in this case we would have an early planning meeting. We found that variable iterations worked well and helped us keep the development focused.

*Therefore our rule of thumb is:* Don't get hung up on process

We would regularly call each other on bad code or small mistakes. When the build broke, we would very quickly call out to the culprit. In fact we did not use automatic integration, because we were integrating ourselves about every 10-15 minutes, and would just shout out if the build had broken.

Team discussions were ego-less but opinionated, and we were all willing to be wrong. Discussions about the system were very robust, but once we had thrashed out a solution, the group would invest in the idea. Ideas were always owned by the group, not individuals.

Single pairs felt very uncomfortable with architectural refactorings that would affect a large proportion of the code base. So we would spend half an hour around a whiteboard to thrash out the details, and then the whole team would work on that refactoring only, until we could commit and move on to something else. Before the first release we refactored the back end architecture completely 4 times in this way, approximately once every couple of weeks. This meant that the architecture stayed flexible and easy to adapt.

*Therefore our rule of thumb is:* Involve the whole team with larger refactorings so the team can move on as quickly as possible

We built our own culture and rituals as the project progressed. For example every afternoon about 3pm we would disappear to the local coffee shop for a half hour. Discussions there were often (but not necessarily) about code problems,

but the primary benefit was that it gave the team a known break point, so that we could maintain a higher pace. Some of our best work happened after these breaks, as the brainstorming and fresh air gave us more energy.

The team socialised together outside work hours as well. When we released we went to a local bar for rounds of Flaming Absinthe - a ritual that we occasionally regretted the next day.

*Therefore our rule of thumb is:* Effective teams need break points

## 4 Delivering

There were people who had no confidence in the team's ability to deliver. Others feared that failure would reflect on them, or the solution compromise the existing legacy application. We approached these antibodies in the same way we would approach a customer - teasing out their fears and requirements and building them into our process as carded activities. We anticipated these sorts of problems and brainstormed the expected antibodies and our response to their concerns. All members of the team were aware of politics surrounding the system and able to 'sing from a common hymn sheet'.

One particularly effective strategy was 'don't say no, say later'. We would take the fear on board (literally carding it and putting it on the whiteboard) for a later iteration, by which time we would have proved that it was no longer an issue, or the initial reason for the request had changed or been forgotten. Fears could then be prioritised in the same way as any other piece of work.

*Therefore our rule of thumb is:* Treat politics as a user requirement

Our initial increments were tested using static data loaded into a test database. We were able to simulate some of functionality of the legacy application. However, we did not appreciate the complexity of the real system's behaviour until we connected our system to the live database.

We could not rely on our unit tests and simulations because these only reproduced what we thought the legacy application did, not what was actually happening. In particular, some external systems that we did not know existed, were directly manipulating crucial tables in ways we were unaware of.

Within minutes of connecting to the live database we noticed inconsistencies and bugs that had been in our code for months. This meant that we had to rethink a large part of the back end of the system. We ran the system with live data for a month before delivery, and built tools to automatically compare the results of our system with those produced by the legacy application. These became our most important integration/user acceptance tests. We did have some of our own 'stress tests' but these were not used for functional testing of the system.

After our first release we no longer left the test system connected to live, and we lost a lot of our reliability. In fact our second release had to be rolled back as bugs were exposed within minutes of going live. Ironically the main piece of functionality of this release had been to allow us to record and play back the events generated by the live system, so that we could improve testing. We had relearned our lesson - and when we connected our test release to the live system again, we managed to do a successful release.

*Therefore our rule of thumb is:* A System that connects to a legacy system must be tested using live feeds.

Even though we had live data feeds to tease out business rule bugs, we still had gui bugs that eluded our unit tests. With hindsight we would like to find ways of introducing robust and flexible acceptance tests much more early in the process. Gui testing is still an open issue for us. We have been bitten by it on several occasions, but have yet to find an effective solution.

Three months into the project (around Iteration 12) we showed the system to key business users and asked them to try it for a while. The system had been running on live data for a month by this stage. The system was so popular that we had problems removing access to the system - when we released the final version, there were still 20 users running the original, some of whom we had not actually given it to. We had designed the system to automatically deploy new versions, so the upgrade was not painful.

We never told the users that they must use the new system. Nor did we remove access to the old system. We relied on making the system so compelling that there was no reason to use the old. This also meant that we stayed focused on the users real requirements.

Because we had actually been 'live' for a month, the first release was an anti-climax. The Project Manager of another team commented that he could not believe we were releasing that week - none of us were staying late and no one worked weekends. In fact our coffee breaks in the afternoons continued.

*Therefore our rule of thumb is:* Engage users and they not only won't they turn it off, they will fight some of your battles for you.

After delivery of the first release we suffered from a bout of 'post-delivery depression'. We concentrated on technical infrastructure problems and refactorings without adding any business value. The team became bored and unmotivated, and the team lost its spark.

Once we got back on business value, the team's demeanour lifted and we sparked again, but we had lost a couple of iterations. A dynamic team like this needs problems and challenges to remain motivated.

*Therefore our rule of thumb is:* Keep giving a good team motivated by giving them new hard problems - don't waste a good team

## 5 Reflection on the Experiences

The project has now been going for seven months. We gave some users a test version to try after two months. They continued to use this version for two months until we delivered the first official release four months into the project. We are about to deliver the fourth release.

We are still running the new system in parallel with the legacy system. We are currently adding major new functionality that is missing from the legacy application and has been an outstanding feature request for some time.

We are also working to enable the new system to operate independently from the legacy application, so that eventually we can switch off the legacy application.

Our project was initially kicked off as a strategic short-term fix. The organisation was planning a long-term project to replace the legacy system, for delivery in 'a couple of years'. Due to the success of our project, this rewrite has now been put on hold.

The team has been asked by other parts of the business to spike solutions to hard problems. This enhanced the motivation of the team.

Looking back on our experiences, we find that our "rules of thumb" paid off well on this project, and we intend to try them out on future projects to see how well they stand up under different circumstances.

## Acknowledgements

The InkBlot team for letting us put our ideas into practice and for making development of the system a pleasure.

Special thanks to Alistair Cockburn for encouragement and advice; to Martin Fowler, Joe Walnes, Gregor Hohpe, Tim Bacon for thoughtful feedback; Ben Authers for de-geeking our prose; and to the London Extreme Tuesday Club (XTC) for their continuous stream of good ideas and discussions.