Using Agile Practices in a Maintenance Environment

Steve Shaw
Intelliware Development Inc.
steve@intelliware.ca

Abstract. The use of agile practices has largely been confined to the development of new systems. This paper details the use of several key agile practices in the context of maintaining a complex system. Each practice is rated in terms of usefulness and uptake in terms of the maintenance of the system.

1. System description

The system described here was developed by Intelliware for an external client. The application is a next generation replacement of a key legacy system used to handle prescription sales in several hundred pharmacies. The system is much more complicated than a traditional retail system because of such things as regional legislation differences, drug interactions and many other requirements.

In terms of deployment, each store runs its own instance of the system, with an on-site server and a number of client workstations scattered around the pharmacy area.

Intelliware took the project over from another outside contractor, and eventually fully redeveloped the next generation application. The system has been in development for over three years, with new functionality being developed to allow deployment to different regions and to stores with different needs than traditional retail pharmacies. The development team consists of a number of developers (usually in the range of 10-15 deployed over two or three teams), several end-users (pharmacists and pharmacy technicians) and two to three dedicated quality assurance (QA) specialists. The application is currently deployed to several hundred stores across Canada.

After the original system was deployed, Intelliware and the client agreed that any issues that arose in a store should be handled by a specialized maintenance team, allowing the majority of developers to concentrate on adding new functionality. I was involved on the maintenance team for a period of two years, spending a significant part of that time as the team lead.

2. Scope of maintenance activities

Responsibilities of the maintenance team can be broken into three categories: defects, enhancements, and general support.

Defects are generally identified as bugs: the system was not behaving as expected in stores. This could be caused by errors in code, by bad data, or by any number of external factors.

Enhancements are considered to be those “nice to have” requests by end-users and other business people. These requirements are not in the original story cards, but it has become apparent that the behaviour of the system is sub-optimal when deployed in a real-world store environment. Often these requests are too small to be considered for a story of their own, or are consolidated into stories and handled by the maintenance team instead of the development team (for budgetary reasons).

General support issues are those things that aren't covered by the above categories. The client's information systems support group calls us when their front-line support team can not answer questions about the system. This category includes things like some performance analysis, database queries, and responses to requests for information from the client’s systems group.

An important but infrequent part of the support team’s duties is managing the delivery of new versions of the system to the client. These code drops are usually scheduled well in advance, but an emergency code drop may be required in some circumstances. The support team is responsible for ensuring that the current code base passes the automated build, and packages it in a form that the clients can easily install.

Initially, about 30% of the maintenance team's activity was fixing defects, with about 65% of its time going to enhancements and the remaining 5% in the general support category. As the system matured and a concerted effort was made to reduce the number of outstanding defects in the system, the breakdown shifted so that now defects account for less than 5% of the maintenance team's effort, with 5% still in
general support and the rest in enhancements.

3. Origin of defects

One of the key benefits of agile development is that the number of defects in the delivered product is supposed to be minimized\(^1\). When it became apparent that we had not delivered a defect-free system, some analysis was done to improve the quality of the code that was going out the door.

The initial rush of defects that occurred after we delivered the initial system was due largely to time pressures. When Intelliware took over development of the system from the initial developers, we reused as much of their code as we could. Due to the complexity of the system and the business rules involved, it became apparent that we would not be able to redevelop everything from scratch and so we concentrated on the problem areas. The initial port of the project from the old code base to the new one still had large chunks of code that weren’t up to our standards. After it became apparent that a lot of the problems were being caused by that code, an effort was made to rewrite those portions of the system.

Even after we rid the system of most of the problematic code, defects were still occurring. There were several reasons identified for the continued failure rate.

- **Data.** Because of the nature of the system, each store has its own database. When the system is deployed to the store initially, the server is seeded with a converted version of the legacy database. When we started developing the system, we used a sample database that had been generated by running an extremely primitive version of the conversion process. Within several weeks, our test suites depended on certain data being in the database, and the developers got comfortable with the data that we had.

  Unfortunately, the conversion process (which is managed by the client) changes rapidly. Because of scheduling pressures, the newest version of the converted data is never ready until weeks after we deliver the newest version of the application. And even if we had been able to adapt to the latest conversion process, our test suites depend on the very specific data that we had received at the very beginning.

  The result is that the data that we test with is different in content than that used in the stores. This problem has largely abated with more recent versions of the system, but some quirky interactions of the conversion process and new code occasionally still cause problems in some stores.

- **Store environment.** Aside from the data, there are several aspects of running the system in a live store that differ from the development environment or even the QA environment. As an example, Manitoba requires all prescriptions to be reported to the provincial health ministry. However, no access to the health network is allowed from outside the province. So it was not until the system was actually dropped into a pilot store in Manitoba that we were able to fully test the functionality of the reporting system.

- **External agents.** The system depends on a number of external agents, notably several in-store systems (inventory tracking and that sort of thing) and some out-of-store entities (chiefly insurance companies and government agencies). As is often the case with external interfaces, what works in the test environment does not necessarily work in the production environment.

- **User disagreement.** A number of defect reports are made in cases where the system is behaving exactly as described in the story card. Occasionally end-users do not agree with our on-site users about the desired behaviour of the system.

- **Non-reproducible bugs.** A number of reported defects can not be reproduced consistently. Over half of the outstanding defects fall into this category. Some of these are due to the unique situation of a certain store, some are likely due to user error, and some are due to the “real-world” performance of the system (usually having to do with the sheer volume of prescriptions with which a large store must deal with). If a problem can not be reproduced in a store setting, it is quite difficult to diagnose and fix in the development environment.

- **Nature of agile development.** By concentrating on what the client requested up front (“always work on the most important thing”), several minor requirements and edge cases have been missed in the past. Sometimes it is not until the application has been rolled out to the majority of stores before a missed use case it noticed because different stores do things their own way.

---

\(^1\) Or eliminated, if you believe the hype. And if you have hit your head against a hard surface recently.
4. Team organization

The maintenance team usually consists of 1-2 developers at any given time. Earlier in the project, there were as many as four developers on the team, but as maintenance pressures decreased, fewer resources were needed.

An interesting discovery was that the maintenance team is not a good place to introduce developers to the project. At first, it seems an ideal choice: the work is fairly contained, and the new developer gets to see many different parts of the system instead of being focused on whatever the development team is doing. It turned out that new developers do not pick up enough if they're paired with a senior who is quickly cranking out a bug fix. Generally, it would take so long to explain the context of a certain piece of work that productivity was hampered.

No matter the size of the team, at least one senior developer is needed to oversee the work being done, and to spot-check the code that the less experienced developers were checking in.

5. The maintenance experience

The maintenance team generally doesn't use stories to manage its work. Instead, the developers select something to work on, fix the bugs, write some tests to make sure the bugs don't reappear, and commit the code.

Selecting bugs. All bug tracking is handled through Bugzilla\(^2\), an open-source bug tracking program. All defects and enhancement requests are entered into Bugzilla, where they are referred to as “bugs” no matter if they are a defect or an enhancement.

Bugs can be reported by several different departments. The client's QA department is the source of most bug reports, but developers, on-site QA people, on-site analysts/end-users and store support specialists can all enter bugs. Once a bug is entered, its severity and priority is determined by the client lead, who often consults with the maintenance team lead to get some idea of the technical impact of a bug.

When a developer is selecting a bug, the priority and target milestone set by the client lead are the primary factors used in selection. The developer does have some leeway in selecting bugs – for example, it's generally not a good idea to start a multi-day effort on a Friday afternoon, and it might make more sense to pick something that can be knocked off in a couple of hours.

From Intelliiware's point of view, it is always better to fix the defects before adding enhancements to the code. It is a continuing goal of the project leads to reduce the number of defects in the system, and several hundred defects have been removed from the system in the first two years of the maintenance cycle.

Fixing bugs. Actually fixing bugs in an agile environment does not differ that much from a traditional development environment. The defect must be verified, the problematic code tracked down and then fixed. One deviation from standard agile practices is that the maintenance team does not always work in pairs. Scanning code, jumping between classes, and looking for the one little problem are not necessarily good uses of two people's time. Some developers work better in pairs and continue to do so on the maintenance team. Of course, if someone is having trouble tracking down a problem then a partner can be assigned to them to give a hand.

One difference in the maintenance environment is that the code that is being changed is often the production version of the code (as opposed to the development version). When this is the case, a lot of the standard agile practices are discarded since we want as little impact on the code as possible. Specifically, refactoring code for clarity and functionality is avoided. We would much rather put another if statement in the code in an awkward spot rather than refactor a new method out. As well, any database changes are forbidden due to the effort involved in rolling those changes out to hundreds of stores.

Writing tests. One thing the maintenance team excels at is writing tests to ensure that the defect or enhancement actually works. The defect case is a situation where test-first programming works well, and several of the developers use that. Others simply write tests afterwards, but revert their code changes to see the test fail with the old code and pass with the new code.

Enhancements are tested in much the same way that new story code is tested. However, since the scope of enhancements (and defects) is usually much smaller than a story's scope, the code is often tested more completely.

---

\(^2\) www.bugzilla.org
Committing the code. The actual process of committing the code back into source control is not any different than the development environment, and is much the same on projects everywhere. However, there is the added complexity of branch management.

An example: if a bug is fixed on the production branch, it also needs to be fixed on the development branch (or the bug fix will be lost when the current development branch is dropped into production). Since the code around the bug may have changed significantly between branches, it is often necessary to solve the problem in a different fashion. As well, if a bug fix was a bit messy due to the constraints on changing production code, it can be redone in a more maintainable fashion, since there are no limits on modifying code on the production branch.

After the code is committed, the defect is updated in Bugzilla to state the nature of the change. The developer usually creates a patch for the fix and attaches it to the bug, which can be useful for future analysis. The patch can also be examined by a senior developer to ensure that code quality standards are being met.

6. Use of agile practices

On-site customer (followed). We are lucky enough to have users on site, mainly to support the development team. They have become a key part of the defect/enhancement management process and are the final arbiters of scheduling and prioritization of fixes.

Bugs or enhancements are sometimes approached as mini-stories, and the developers involved will often present several solutions. This allows the users to make informed decisions, weighing the costs and benefits of each proposed solution.

The other benefit of having the users immediately available to us is that enhancements can be fully explained instead of having to rely on an enhancement request written by someone in a store using jargon that might not be immediately familiar to the developer.

Short iterations/integrations (followed). The maintenance team usually has the shortest iterations of any team, due to the short-term nature of the work. Actual delivery of the code to the client is usually tied to development drops, except in the case of important fixes that have to be delivered immediately to stores.

Stories and story cards (generally not followed). In a “standard” XP environment, the work is guided by what appears on the story card. In most cases, we treat the bug report or enhancement request as a task of a conceptualized yet non-existent larger story. However, for large units of work, the developers are responsible for tasking out the fix as they would task out a story.

The reasoning for this is straightforward: ideally, at Intellware, a story will take a week or to complete, and each task should be done in under a day. Unless the bug or enhancement is much larger than this mean, each unit of work is already task-sized or smaller and there is no need to break it down further.

Testing (followed). Maintenance developers have the same responsibility to test their code as any other developers. Since the code changes are usually fairly narrow in scope, the testing is usually quite thorough.

On occasion a bug will be fixed and it becomes apparent that it could have been caught with adequate testing or analysis. In these cases, the developers assigned to a defect will often examine the tests that were originally written to see if the entire team's test-writing skills can be improved to avoid defects slipping through in the future.

Pair programming (generally not followed). Partly because much of the work in tracking down bugs is analysis-centric, and partly because some of the developers eschew pairing in general, pair programming is not enforced. If developers want to pair, that is encouraged, but it is rare to force people to pair off.

It is important to realize that even though pair programming was generally not used, there is still a lot of back-and-forth in the project room. People ask for help in tracking something down or get opinions about a proposed fix.

Prioritization of tasks (followed). The maintenance team is usually confident that they were “working on the most important thing”, since the client is the one that set the priorities for each defect or enhancement.

3 That would be me.
**Measurement of velocity (eventually followed).** After several months, we started tracking how long it was taking to fix a defect or complete an enhancement. Our initial analysis put each task at around eight developer hours, and that has stayed fairly constant.

This tracking has allowed for surprisingly accurate estimates of how many defects could be fixed or enhancements could be delivered with the next scheduled code drop.

**Daily stand-up meetings (not followed).** Partly because of the small size of the team, and partly because the tasks being worked on are so short, daily stand-up meetings usually do not happen. The developers usually have a good idea of what's going on in the room without them, and some developers remain unconvinced of the usefulness of stand-ups.

On occasion, a representative will be sent to the development team's stand-up meeting to compare notes and find out what is going on in the other project room.

**Leave optimization until the end (followed, sort of).** While this is not necessarily an agile practice, it fits well into the agile methodology – first get it working, then get it working properly. The maintenance team does follow this, although sometimes the optimization step is skipped. This isn't so much due to laziness, but because it can actually be counterproductive to modify the production code base too much.

**No overtime (followed).** This rule is strictly followed, but that again may be due to the personnel involved.

**Refactoring (followed occasionally).** The refactoring performed by the maintenance team is generally minimized while fixing defects. This has been covered above: generally, the goal is to fix the bug without modifying too much of the core code. Most of the refactoring is left to the development team.

Occasionally the maintenance team will identify a section of the system that is really poorly designed or has accumulated cruft over the years. In that case, the developers will often propose a technical enhancement and get that prioritized by the clients in order to properly refactor a set of classes. This is a fairly important part of the maintenance team's duties since they are working with code that is largely ignored by the development team. If a certain section of code has been found to contain a lot of defects or needs a lot of enhancements, it may be time for the section to be reworked.

**System metaphor (not followed).** The maintenance team usually does not take the time to develop new metaphors for the work that is being done. Instead, we piggyback on the metaphors created by the development team and apply those to the code that we write.

**Common code ownership (followed).** Everyone owns all the code, which is what allows for the creation of a separate maintenance team in the first place.

7. **Summary**

Many of the standard agile practices do work well for maintenance projects. In fact, some of them (such as short iterations) are the default practice for a defect-fixing release. Most of the practices that are not followed either do not make sense for the maintenance cycle (e.g., story cards) or used infrequently because of the personnel on the team.

Admittedly, the deck is stacked in our favour since we already have the clients on-site. It may be more difficult to convince a customer that you need to have a client on-site just for the occasional consultation about a small defect or enhancement.

We feel that breaking the maintenance team out of the standard development team was a good decision, allowing each team to focus on their own tasks without constantly getting distracted by other concerns.

8. **Previous Work**

Svennson and Host find that many of the agile/XP practices have to be modified to work in a mature

---

4 Eight hours per bug may seem like a lot, but that's an average. Some large enhancements might take a week or two, while some of the smaller bugs would be knocked off in twenty minutes.

5 That would be me again.

6 Well, sometimes it is.

7 Yeah, that would be me again.
maintenance environment. Many of the changes made are because of the nature of the organization, which did not have an existing mature agile practice.

Poole and Huisman[2] discuss applying the XP methodology to a mature maintenance environment, and find that the practices they applied were quite useful in improving productivity and morale. They use many of the practices that we have avoided, and provide excellent reasons for doing so. This is indicative of the flexible nature of the agile and XP practices.

References


9. Appendix: Tools

**Bugzilla.** The only tool that the maintenance team uses in addition to the standard suite of development tools is Bugzilla. We use Intellijware's main Bugzilla server and have separated several projects out for our purposes.

Bugzilla is not a bad tool, and it's easy to allow users remote access to update bugs from off-site. But there are more robust tools available. We're also stuck with an older version of Bugzilla, and since the entire company uses the same instance of Bugzilla, we can't upgrade it on our own whims but have to schedule upgrades with several other projects across the company.

Moving to our own instance of Bugzilla, or a different piece of software entirely, is a complicated matter because of the migration hassle. We started using Bugzilla because it's basically the default choice at Intellijware, but it did really lock us in to using it for the foreseeable future unless we want to pay a fairly hefty labour cost to move to a different product.