

Agile Software Development Practices

at

Wastewater Management Division

Information Services

Larry P. Sherrill

February 10, 2004

1. Executive Summary	3
2. The Practices.....	4
2.1 War Room.....	4
2.2 Unit Testing	5
2.3 Emergent Design.....	6
2.4 Weekly Iteration Goals	7
2.5 Acceptance Testing.....	9
2.6 Deliver Value Early	9
2.7 Onsite Customer.....	9
2.8 Continuous Integration.....	10
2.9 Collective Code Ownership	10
2.10 Coding Standards	11
3. The Role of the Manager	11
4. Antipatterns.....	12
4.1 Developers are separated from each other.	12
4.2 Developers are separated from the customer.	12
4.3 The team is not test-infected.	12
4.4 The immediate supervisor doesn't get it.....	12
5. Conclusion	13
References.....	14

1. Executive Summary

Developing software is risky and hard to estimate. Projects fail due to time and cost overruns or by simply building the wrong thing. One guaranteed recipe for a high failure rate is the project that consists of one lone developer, isolated in a cubicle, following no process at all. To counter this, organizations often go to the other extreme by mandating a one-size-fits-all, document-centric, waterfall process. Waterfall describes a project that flows from analysis to design to implementation to test in a sequential, one-way direction. These projects may produce volumes of paper, complete with milestone signature lines guaranteeing that the documents for a given milestone are correct and unchangeable.

Unfortunately, changing requirements, analysis paralysis, rigid communication channels, Gantt chart fiction, and the chaotic nature of software development become risk elements for the waterfall process. In the 1980's, the Department of Defense required its contractors to follow the waterfall life-cycle specified in DOD-STD-1267. It has since turned away from this approach and now recommends MIL-STD-498, which describes a multi-build approach (i.e., iterations). It states that "requirements may not be fully defined until the final build" and that "design may not be fully defined until the final build".

To counter the risks in developing software, the IS Department of Wastewater has implemented a set of agile software practices. The following section discusses these practices in detail.

2. The Practices

Agile teams focus on practices, not artifacts. Artifacts are the documents developed in the course of a project (e.g., requirements documents and class diagrams). People and conversation are valued over process and paper. The agile practices described in this section are interrelated. For example, the practice of group design cannot succeed without the immediacy and spontaneity provided by the common room environment. The practice of emergent design cannot succeed without the practice of developing unit tests to assert system viability after aggressive refactoring.

2.1 War Room

A team composed of a limited number of developers (e.g., four at Wastewater) work in a common room, also known as a war room. This has many benefits. First, all major design decisions are discussed as a group. Impromptu design discussions occur as the need arises. A good war room requires plenty of whiteboard space on which to draw UML class diagrams, use case diagrams, and other relevant artifacts. Group design and brainstorming eliminate the problem of the lone, myopic developer working in isolation. The departure of one team member will not cripple the team because everyone shares the design knowledge.

A second advantage is that developers become more generalized as expertise is shared. Developers bring different areas of expertise to the table. In the common room, developers can easily ask for help. There is a healthy buzz in the war room as everyone hears the answers to questions.

A third advantage occurs when the customer visits the war room. The customer, in the agile context, is the end user or person requesting the software. The customer

should be in a position to determine when the software is acceptable. The customer helps the team by clarifying requirements and by providing usability feedback as the project proceeds. When the customer visits the war room, all the developers have a chance to join in or listen to the requirements and usability discussions.

A fourth advantage of the war room is the subtle peer pressure felt when developers check off their completed iteration goals. Weekly iteration goals, written on a war room whiteboard, provide focus to the team and serve as a constant reminder of what each team member should be doing during the week.



Figure 1. The Wastewater War Room

2.2 Unit Testing

In unit testing, we write tests that exercise the code that we are developing. For example, if we are writing a Java class called Account, we will also write a test class called TestAccount that exercises Account. Using the open source JUnit class library, developers can easily write suites of tests. Each test contains assertion statements that verify the postconditions after exercising the target class. Over time, the number of tests for a project grows. All the tests are executed together. The results can be printed out, saved as an HTML file, or emailed to interested parties.

Unit testing is a critical piece of agility and serves several purposes. First, unit tests are a design tool. The phrases Test-First-Design or Test-Driven-Development (TDD) refer to the practice of developing tests before, or concurrent with, the development of actual code. The developer uses the new API under development and gets a feel for its programmer usability. It also forces the developer to write code that is testable in the first place. In other words, making code testable mutates the design, typically making it more flexible and maintainable. Good unit tests will actually drive the design.

Second, unit tests serve as documentation. By exercising individual pieces of the system, unit tests provide reminders of how the internal API should be used. New developers on the team can study the tests to better understand the system.

Third, unit tests capture requirements. For example, if account numbers should always be 16 characters long, a unit test can make sure that bad account numbers are rejected. Reading the unit test will inform the reader of the correct requirements for an account number. Unit tests serve as an executable form of requirements validation. With good test coverage, when the tests succeed, the requirements have been met.

Finally, unit tests verify the correctness of the system. New code should be viewed only as a hypothesis and the unit test as the proof of that hypothesis. Code delivered without its corresponding proof-of-correctness (i.e., a unit test) should be considered suspect until proven otherwise.

2.3 Emergent Design

Large, up-front design is discouraged on agile projects. During the first iteration, the developers will meet with the customer and gather enough requirements (through

conversation—not paper) to get started. A preliminary, short design session follows and GUIs are prototyped on the whiteboard. Developing infrastructure for future iterations should be avoided (within reason). The phrase “you aren’t gonna need it” (YAGNI) describes this practice.

Left to their own devices, developers can expend great amounts of time and energy anticipating future needs, which may or may not materialize. Agile projects stress doing the simplest thing possible for a given iteration. As the requirements grow, the system is refactored, resulting in a design that emerges over time. At the end of the project, the architecture will be just robust enough to meet the requirements. Unit tests play an important role in maintaining system viability as the design emerges from repeated refactoring.

2.4 Weekly Iteration Goals

At Wastewater, the developers meet every Tuesday morning to decide on the next week’s tasks. This provides focus for the developers and gives the immediate supervisor a window into the team’s activities. Estimating software projects months in advance is an exercise in futility. Software is not a building, and blueprints are the wrong metaphor. Developers can, however, estimate one week fairly well. Each week, developers write iteration goals on a whiteboard. As the week progresses developers check off what they have completed.

Weekly goal-setting establishes a rhythm for the project, and enables the team to turn on a dime and address new business priorities and new requirements that may arise. This ability to change course quickly is one of the reasons why the word “agile” is appropriate.

After the goals are written on the whiteboard, they are emailed to the manager.

Figure 2 shows a sample email listing the new goals for the coming week and how well we did in the previous week:

```
Iteration Tasks 58 (completion date: 2/3/2004)
1. Clean up Service | AKA data in storm - Jim, Joanne, GERALYN
2. Add functionality for new Panel buttons in StormMerge - Jim, Joanne, GERALYN
3. Dynamic-ize stormmerge server data sources - Ryan, Larry
4. Create prototype dbf loader - Dan H, Larry
5. Load storm taps into dev hansen - Dan H
6. Make parcels tab work in sanlien - Santiago
7. Make parcel filtering work in sanlien - Santiago
8. Modify inventory reports - Jim, Joanne, GERALYNN

-----Last Week -----
Iteration Tasks 57 (completion date: 1/27/2004)
1. add 'add panel' button to parcel pane - Jim, Joanne, GERALYN - COMPLETE
2. gis2storm acceptance test - Larry, Jim, Ryan - COMPLETE
3. check sanlien into cvs - Larry - COMPLETE
4. sanlien - import txt into sanlien_run and sanlien_parcel via web page - Larry,
Santiago, Ryan - COMPLETE
5. sanlien - display lien summary page with real data - Santiago - COMPLETE
6. deploy eztime - Ryan - COMPLETE
7. figure out test data approach for stormmerge - All - COMPLETE
8. stormmerge - remove old report methods and ivars - Larry, Santiago - COMPLETE
9. setup Hansen test schema - Jacob, Dan K - COMPLETE
10. manually add one record to stormmain to verify approach - Dan H - COMPLETE
11. prototype oracle to oracle for hansen for one stormmain record - Dan H, Larry -
COMPLETE
```

Figure 2. Weekly Iteration Goals

Short iterations will expose a project that is in trouble much quicker than a project with long requirements and design phases. High-risk items are attacked in early iterations and showstoppers will hopefully appear much earlier than with a big-bang implementation at the end. If goals are consistently left uncompleted at the end of iterations, this may be a warning sign that the team is not estimating well, is not decomposing tasks correctly, or is stuck.

2.5 Acceptance Testing

What will convince the customer that the system is acceptable? This should be one of the first questions the team asks the customer. Putting a tester on the team can greatly benefit an agile project. Working in conjunction with the customer and the developers, the tester develops a set of test cases. The tester can act as an advocate for the customer and give the customer a consistent means of evaluating the system. The test cases can be used as training tools for new end users and new developers. They can also crystalize requirements, complementing the ability of unit tests to capture requirements. Test cases are one of the few persistent paper artifacts produced on agile projects at Wastewater.

2.6 Deliver Value Early

Aggressively look for ways to deliver business value to the customer early. Identify items that can be carved out and delivered quickly. This gives the customer the opportunity to begin using pieces of the new system as soon as possible. Typically, this will also elicit new requirements as customers begin using parts of the system. For example, if you are working on a lost and found system for the airport, an early iteration could simply list all the articles in the database on a web page. The second iteration could add the ability for the user to specify search criteria.

2.7 Onsite Customer

The customer needs to be close-by and available. The customer is the person that requests the project, can specify the requirements, and has authority to accept the final product. Project risk increases greatly when the customer is inaccessible or distant from

the developers. The customer will need to come to the war room to discuss requirements, review prototypes, perform acceptance testing, and identify potential valuable items for early delivery. When it is impossible to have an in-house customer or when there are multiple customers, a customer proxy can represent the customers.

2.8 Continuous Integration

A well-defined build process provides valuable knowledge about how to build a system. This is especially important when the original developers have left the project. At Wastewater, we use Ant (similar in purpose to UNIX *make*) to specify compilation, testing, and deployment tasks. We also use CruiseControl in conjunction with Ant. CruiseControl is a build program that runs continuously. Every hour it checks for source code changes in the version control system, checks out the new code, compiles it, runs the unit tests, and emails the results to interested parties. Continuous integration eliminates big-bang integration. All code is integrated automatically every hour. Developers find out very quickly if integration has failed.

2.9 Collective Code Ownership

In an agile project, everyone collectively owns the code and has the right to refactor it. This prevents a lone developer from owning a subset of the system and taking it in a bad direction. By using coding standards, developers can easily switch to new areas of the code without adjusting their thinking or their editor. Unit tests support collective code ownership by serving as documentation and giving reassurance that the system still works.

2.10 Coding Standards

During each iteration meeting, developers may choose to work in different areas of the code. Without code ownership, it is important that the team uses a consistent style. The style should be a team decision. At Wastewater, we use a ready-made set of Java coding standards developed by Sun Microsystems.

3. The Role of the Manager

The immediate supervisor of the developers is the most important element in the success or failure of the team's move to agility. No team can become agile if the manager does not understand or support the goal of adopting agile practices. The role of the manager is to:

- Set project priorities
- Monitor the team's progress
- Determine if the team is stuck (and take action to break the log jam)
- Find and furnish a war room
- Promote the use of coding standards
- Encourage unit testing and group design
- Encourage developers to work on different parts of the system to ensure redundant knowledge and to counter lone-developer myopia
- Educate the customer of their vital role in the team's success
- Act as a firewall to protect developers from outside distractions that deter the team's ability to meet iteration goals.

4. Antipatterns

The following antipatterns can prevent a team from successfully adopting agile development.

4.1 Developers are separated from each other.

Group design suffers. Developer conversations with customers are not overheard by other developers. Developers who are stuck cannot simply ask the group for help.

4.2 Developers are separated from the customer.

If the customer is in a different part of the city, the project is in trouble. This can be mitigated partially by using the tester or someone else as a customer proxy. Customers must be actively involved with the project.

4.3 The team is not test-infected.

The team does not write unit tests. Aggressive refactoring is required to achieve an emergent, lean design. Without unit tests to verify system integrity, the project will slow down or get out of control as new bugs are introduced into the system during refactoring.

4.4 The immediate supervisor doesn't get it.

A team cannot succeed with agile development without the full support of the immediate supervisor. The team needs its supervisor's trust, vision, and involvement.

5. Conclusion

Agile software development can help the city mitigate the risk of software failure. Developing software is too complex to separate the activities of analysis and design from the activities of coding and testing. Separating these activities limits feedback. Requirements will never be frozen, nor should they be. Anticipating the correct design without being in the code is unrealistic. The Rational Unified Process (RUP) and the Unified Modeling Language (UML) are often misused in a waterfall fashion. This is ironic, given the fact that Philippe Kruchten authored RUP as an iterative response to a failed waterfall project (the Canadian Automated Air Traffic Control System project).

The iterative approach recommended by the Rational Unified Process is generally superior to a linear, or waterfall, approach for a number of reasons.

Project managers often resist the iterative approach, seeing it as a kind of endless and uncontrolled hacking.

If the sequential process is ideal, however, why aren't the projects that use it more successful?

Philippe Kruchten, (1998). The Rational Unified Process: an Introduction.

Viewing a project as a series of iterations requires a different mindset than viewing the project as a Gantt-chart-driven waterfall. The success rate of iterative projects, however, makes it a mindset worth acquiring.

References

The Agile Alliance. (2001). *Manifesto for Agile Software Development*. [On-line]
Available: <http://agilemanifesto.org>.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Boston:
Addison-Wesley.

Burke, E., & Conyer, M. (2003). *Top 12 Reasons to Write Unit Tests*. [On-line]
Available: <http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html>.

Kruchten, P. (1998). *The Rational Unified Process: an Introduction*. Reading,
Massachusetts: Addison-Wesley.

Larman, C. (2003). *Agile & Iterative Development: a Manager's Guide*. Boston:
Addison-Wesley.

Martin, R. (2003). *Agile Software Development*. Upper Saddle River, New Jersey:
Prentice Hall.

What is Extreme Programming? [On-line] Available:
http://www.xprogramming.com/what_is_xp.htm.