

The Adaptive Development Tenets

The Adaptive Development Tenets	1
Introduction	1
Our tenets	3
An iterative development process	3
Testing, testing, testing	4
Customer interaction	8
Prototyping	9
Continuous integration	10
Lean and effective design, documentation and code	11
Small teams	13
Motivated, empowered developers	13
Collaboration and collective ownership	14
Accountability	15
Automation	16
Judicious branching	17
Diagnostics	17
Tools	18
Depth of thought	18
Agile decision process	18
Egoless development	19
Attack problems and not people.	19
Effective customer support	19

Introduction

This document describes the tenets that we at Adaptive Development have found to be very beneficial over the last 10 years of our software development experience. This document assumes that the reader is familiar with the different agile and iterative methodologies since it will build on those.

By definition, 'Adaptive Development' is not prescriptive. It is not a 'one size fits all' approach where following a set path guarantees success. Instead, as the name suggests, the development process needs to adapt to the current environment and project. One of the principles of the Agile Manifesto is 'At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.' The tenets in this document are a set of extremely useful rules, policies and practices that we regularly employ in software development.

Since we adopt an agile development approach we are able to adapt to change and mitigate risks for our customers. We build upon the Agile manifesto and principles.

The agile manifesto: -

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The agile principles: -

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Working software is the primary measure of progress.
- Deliver working software frequently.
- Welcome changing requirements, even late in development.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity is essential.
- Emergent solutions from self-organizing teams.
- Regular reflection and improvement.
- Business people and developers must work together daily throughout the project.
- The most effective method of conveying information to and within a development team is face-to-face conversation.

Our tenets

Each tenet is described in more detail further down in the document.

- An iterative development process
- Testing, testing, testing
- Customer interaction
- Continuous integration
- Prototyping
- Lean and effective design, documentation and code
- Small teams
- Motivated, empowered developers
- Collaboration and collective ownership
- Accountability
- Judicious branching
- Diagnostics
- Automation
- Tools
- Depth of thought
- Agile decision process
- Egoless development
- Attack problems and not people
- Effective customer support

An iterative development process

We follow a development process that is similar to the one described at <http://www.extremeprogramming.org/rules/iterative.html>, with a few of our own customizations and modifications. Our development process is broken into three layers: -

- The product development cycle
- The iteration cycle

- The daily development cycle

Please see the document ‘Adaptive Development Software Model’ for an in-depth description of our software development model .

Testing, testing, testing

Testing is a crucial part of any development project. It used to be an afterthought in the waterfall development process, but is now an integral part of the Software Development Life Cycle (SDLC). Our rules for testing are: -

- We assume that if something is not tested, it is broken.
- When a new bug is found: -
 1. Add a new test and make sure that it reproduces the bug.
 2. Fix the bug.
 3. Run the test and make sure that the test now passes.
- Make sure that there is a good spread (depending on the project type) between the following tests: -
 - Automated repeatable regression tests. These tests are vital in that they should map to the user stories (i.e. the original requirements). Generally these tests follow these basic steps: -
 1. Provide input to product.
 2. Receive output from product.
 3. Validate that the output is correct.
 - Automated stress tests. These tests are important because they greatly improve the overall quality and robustness of the product. These tests should run as part of the nightly build. Stress tests generally stress the system with random and chaotic behaviour and the measure of success is that the system does not crash and it is in a sane state at the end of the test. It is generally more difficult to track down bugs found by stress tests due to the random and high load nature of the stress tests. Generally we would

depend heavily on logging and other diagnostic tools to track down bugs found by these tests.

- Automated unit tests. These tests validate that a smaller part of the system (e.g. a class or component) is working as expected. These tests should run as part of the nightly build. These tests are important since they verify that each cog in the system is robust. Due to the reduced scope, it is much easier to track down a bug in the unit test than it is to track down a bug in a regression or stress test.
- Manual tests. Manual tests are generally tests that cannot be automated. Since they are not automated they are done at the end of the iteration rather than nightly. Also, it requires developer time to run these tests, which is not the case for automated tests which run by themselves. For these reason we always try to automate our tests. A good reason for a manual test is as a quick sanity check before the product is sent to a customer. It is possible that the automated test system missed something critical that would show up very early on in using the product. Having a developer open up the software and do a 5 minute sanity check before shipping to the customer avoids those embarrassing questions, such as ‘Did you even try opening up the start screen before shipping?’
- Test first / test early. Each new feature requires a test. Sometimes it is not feasible or sensible to write the test before implementing the feature, but as a rule we do not check in a new feature unless it has a regression / unit test.
- Test extensively. The more coverage that the testing has the more quality the product will have. The best way to ensure test coverage is to add a test for each feature as the feature is being developed. It is a lot easier to add tests from the beginning of a product than as a big push at the end. Profiling tools can provide some measure of coverage, bearing in mind that there is more to coverage than a high score on a code-coverage profiler. Coverage also means testing different values for parameters and testing different paths which touch the same piece of code. That is, profiling tools are only one indication as to how well your code is being tested. It is often better to test a critical method with multiple different

parameter values than it is to test an obscure and seldom used method simply to increase a profiler score.

- Testing is not a one-off process and there are no easy shortcuts. Often, there is a 'quality panic' with a buggy product that has been developed over several months / years. The product is falling over every couple of minutes on-site and suddenly there is a strong push to do a big testing effort to add quality. Everyone stops what they are doing and spends a week writing tests and within a week, a bunch of tests are added. Everyone feels great and then goes back to sleep and adds code without testing. A far better approach is to develop a practice of adding 10 or so new tests a week for the areas that most need tests and make sure that everyone adds a test for each new feature. Quality will improve over time and continue to improve into the future. This is not to say that there is anything wrong with a big testing effort, just that a testing culture is more important than a one off testing sprint.
- Failing tests go rotten over time. The longer that tests are left unfixed the harder it becomes to fix them as the knowledge of that part of the code diminishes.
- Existing (known) failing tests hide new failing tests. Don't check-in code that you know will break tests or add new tests that you know will fail. Very often you will hear 'Yes, we know about those tests, we expect them to fail'. What they failed to notice is that a critical test has also started failing recently. Also, expecting tests to fail diminishes confidence in the test suite.
- Failing tests mean that the build is broken. Development should not consider a failing test any differently from a bug that causes the compiler to fail. A failing test indicates that the build is broken and should be fixed before any further development is done on the code.
- Fix failing tests before implementing new features.
- Don't release software where tests are failing. The arguments you hear for releasing software with known defects are: 'That is an unimportant test', 'it is a known bug that won't affect anyone', 'that is a bad test'. There are several reasons why software should not be released when tests are failing: -

- It becomes OK to release software with failing tests. 3 failing tests become 5 failing tests, become 10 failing tests and before long it is hard to determine which tests are OK to fail and which are critical bugs and the n everyone loses confidence in the test software.
- The failing tests are never fixed.
- The released software does not meet the customer requirements. Each regression test should test a scenario that matches a customer requirement. If the test case is truly unimportant or minor then it should be removed along with the customer requirement and added for the next release.

Using continuous integration with nightly automated tests, there is no reason why a product needs to be released while tests are failing. You should always be close to a working build since one the rules of test-first is that tests are run nightly and failing tests are fixed before new features are implemented. So you should never get into the situation where software needs to be released while there are failing tests since you are always one day away from a working build. If the software absolutely has to be released while tests are failing (bearing in mind that this indicates a failing in your process that should be addressed), then make sure that :-

1. The customer is fully informed about the failing tests and the implications. Often the pressure to get the software out is due to internal pressures (such as Employee Bonus schemes and commitments made by management) rather than being customer driven. If the customer would rather wait until a stable build is available then there is no reason to give them the buggy product.
 2. All other stakeholders are informed.
 3. Remove the tests and the requirements (user stories or tasks) that the tests are related to (if they come from a direct customer requirement) and add them back in for the following iteration.
- Decide on a test-development balance. This will vary depending on what is being built, for example if we are building a Mars rover then we would give more time

to testing and if we were building a search utility for an internal intranet, we would spend less time testing. A good rule of thumb is to spend a minimum of 50% of the time on testing and 50% of the time on development. As the testing falls below 50%, the quality of the product degrades badly.

- Don't be a test tyrant. It is easy to get carried away with test coverage and the number of tests. The focus for testing is to provide a stable system that meets the customer requirements, i.e. brings the customer joy. Adding meaningless tests just to make test reports look good doesn't help. Sometimes it is better to only implement a regression test, sometimes a unit test will do, and sometimes the functionality can only be tested manually. In some cases the test will have to be delayed while the requirements, design and code are nussed out. Some changes are already covered by a test case and some are so minor that they do not warrant a test.

Customer interaction

- Regular feedback from customer is encouraged.
- The customer knows the requirements better than the developers.
- The customer is the ultimate user of the product and will know is in the best situation to judge whether the product is working as expected or not. Customer issues should be given priority. Keeping teams small and focusing on accountability ensures that developers are close to the customer and provide effective support when issues arise.
- Field trials. Over and above verbal customer feedback software should be deployed at a customer site at the end of each iteration. In some cases, the field trial will be limited: -
 - The customer is very sensitive to new software going on site.
 - The software only satisfies a small subset of the requirements so it would not make much sense running it live.

There are several options in these cases: -

- Do a limited field trial. Instead of having all 100 intended users running the new software, restrict the trial to just 10 users.
- Create a simulated field trial that is as close the real thing as possible where the intended users (rather than the main customer contact) use the software.

- The customer should be on site, or if the customer has to be off-site, a developer should communicate daily with the customer and act as a proxy for the customer.
- Beta trials and beta customers. Beta customers are worth their weight in gold for rattling out bugs before GA. When the beta release is running at beta customer site work closely with the customer to address any issues. Since the beta customer will experience some discomfort running the latest software, they should be compensated in some way, e.g. a discount on the software or free tech support. The beta customer should realise that they are a 'beta' customer. This may seem obvious, but it is tempting to try to avoid resistance from a customer or avoid cutting a deal with the customer by telling them they are getting production software that is stable and used all over when it is really beta software. Don't do this, it will end in tears.

Prototyping

Prototyping is an essential part of our process. It falls under a concept that we have of the Romans versus the Greeks in our development process. The Romans are more pragmatic and active versus the Greeks who are more philosophical, steeped in theory and open discussion. At some point after adopting the Greek approach of discussing the theory, philosophy and rationalization we will adopt the Roman approach and just bang up a prototype. For example, the Greeks would argue and pontificate at length about whether to use a catapult to siege a city based on material stresses, angles, ratios and political implications while the Romans would build a catapult and see how far it could fling a rock.

The above paragraph has absolutely no historical accuracy, but it illustrates the point. A prototype is quick, pragmatic way to: -

- Determine whether an approach is even possible.
- Quickly discover unknowns.
- Compare different technologies.
- Provide more confident estimates.

There is a lot of discussion as to whether it is better to throw away a prototype or to keep the prototype and develop it into a product. The rule is that the prototype only proves or investigates a small unknown and does no more. The prototype shouldn't grow larger than a week's work into an unwieldy piece of code that implements several features and then grows into the final, untested and hacked product. As long as this rule is satisfied, it is OK to keep a prototype as long as: -

- The design is not fundamentally flawed.
- The first step after proving the prototype is to add extensive tests to test each feature as well as stress tests to make sure it is stable. As long sufficient testing is done, the resulting design and code will be of a high quality.

Continuous integration

This means that source code is checked into the source control regularly throughout the day. Avoid source code on developer machines from getting out of date with the code in the repository. This results in difficult merges that are likely to introduce bugs.

Automated builds are done several times throughout the day (preferably after each check-in). The automated builds are clean (i.e. source code is checked out and built from scratch - there are no intermediate object or project files left lying before the build). The build machine should be reserved just for building the source code. The only way to get a build for a customer site is through the build machine. A build from a developer machine shouldn't go out to a customer site since developer machines are often riddled with strange libraries and dependencies.

Nightly builds should be run to make sure that the source code builds and all tests pass.

If a build fails due to a compilation error or a failed test, it should be fixed as a high priority. This is generally done by the person who broke the build.

Each build should be versioned and the source tree should be tagged so that it is possible to go back to the source code for any build.

The product that is run at the customer site should always come from the formal build.

Lean and effective design, documentation and code

One of the biggest challenges facing developers is to avoid unnecessary complexity.

There are a number of root causes for overly complex systems: -

- We are too enthusiastic. We get carried away and go far beyond the intended requirement. The problem is that this results in monolithic, poorly performing systems that often do not meet the original requirements.
- We are too clever. We over-architect systems or create non-standard code that is difficult for other people to follow.
- The problem space is tricky. It is hard to see the wood for the trees and it is tempting to jump in and start developing before fully understanding the problem space.

Just keeping things simple is not enough, for example a coding standard that can fit on one page is great so long as it actually achieves the intended goal. This is why we focus on lean and effective development – the simplest system that gets the job done. Some examples of where we use ‘lean and effective’ development follow:

Lean and effective design

- High-level diagrams. Diagrams are great for conveying a high level design or process flow. Beware of getting carried away, when diagrams become too low-level they become difficult to maintain and quickly get out of sync with the source code. We use block diagrams, work flows, collaboration, sequence or activity diagrams depending on what is being conveyed.

- Favour good design over performance. A lot of unnecessarily over-complicated poorly performing code has been written in the name of performance. Don't try to guess where the bottlenecks will be, rather favour good design and then refactor for performance.
- Favour simplicity over cleverness (obfuscation) in design and code.
- Favour understandable, readable code over terse code.
- Existing design patterns. In many cases there is no need to reinvent the wheel. There are a number of elegant and proven design patterns that can be employed in various scenarios.
- Object oriented design is a well proven methodology. The concepts of encapsulation and abstraction aid in creating code that is resilient to changing requirements and extension.

Lean and effective documentation

- Brainstorming documents. This is a great way to bash out ideas and proposals in the early stages of a project. It is a free-form document that uses natural English, diagrams and pseudo code to discuss the concepts in a friendly not overly technical manner.
- Self documenting code / code generating documents. A major problem with low-level documentation such as low-level specifications is that it gets out of sync with the code. For example, a message specification that documents the message flow between two components can easily be used to auto generate the message layer code. The specification document is parsed for each build and the code is auto-generated from the spec and then compiled. This means that if the spec is ever changed, e.g. a new message is added, the code will be automatically be changed. If the implications of such a change were not considered it may cause the build to break, which is exactly what you want, otherwise the specification quickly gets out of sync with the code.
- Lean coding standards. Coding standards have been a hot topic since the first text based code editor existed. A lot of time and effort can go into coding standards

that define very strict layout / naming conventions. People have different styles and tastes and, like handwriting, every person's code differs slightly. So, we favour higher level guidelines than can fit on an A4 page rather than overly restrictive coding rules. When working within a block of code that is different from our own, we will try to follow the coding style that was used in that block.

Small teams

Many of the criticisms of agile methodologies stem from the fact that it does not work overly well with large teams. In reality, it has been found that smaller teams are more effective than larger teams for the following reasons: -

- More focused scope.
- More of a sense of ownership and responsibility.
- The processes required to manage the team effectively are more lightweight.
- Closer interaction.
- Lower cost of change.
- The customer is closer – everyone is in the front line.

Motivated, empowered developers

A major focus of our model is to motivate and empower developers: -

- They should be encouraged to decide on the tools they need to get the job done.
- They should be encouraged to consistently up-skill and keep in touch with developments in the industry.
- They should be able to work in an environment that they are best suited to.
- They should feel empowered to make important business, design and coding decisions.
- There should be no overtime. In the exceptional case where developers have to work over time, they should be compensated and the managers and customer need to realise that any overtime results in an equal amount of undertime.

Collaboration and collective ownership

As discussed below, the merits of peer programming are debatable, but we do believe in open and honest exchange. We collaborate heavily no matter what the area - requirements analysis, design, coding, emails, documents, customer interaction and the Adaptive Development process. Daily stand-up meetings are an integral part of our development process. This is why we favour a small team of expert engineers, since they need to be able to work at any level in the SDLC. Also, any developer should be able to work on any piece of code to get their job done, there should be no code territories under the rule of a dictatorial developer who can't tolerate anyone else touching (or in some extreme cases, seeing) their code.

While we encourage collective code ownership, we have found that the concept can be misunderstood and misused especially at a managerial level. Collective code ownership does not mean juggle developers around quickly so that any developer can work in any area or piece of code as easily as any other developer. There is an incorrect notion that if developers are juggled around enough, they become plastic people that can be slotted in anywhere and there is no real loss when an experienced developer leaves because the system is known equally well by all developers.

The problem with this approach is that it takes a lot of time and dedication for a developer to become familiar with a part of the system and even more time and dedication for them to become an expert in the area. Systems benefit greatly from an expert, they know the rationale behind certain design decisions, they can implement a fix very quickly when time is of the essence and they know the best way to extend the existing design. Quickly juggling people between projects means that no member of the team reaches this depth of understanding, the effectiveness of the team is simply diluted and the quality of the system as a whole suffers.

Another problem with the 'plastic people' approach is that it reduces the accountability that any engineer feels when something goes wrong. See the next section on accountability for more information.

What started as a managerial attempt to solve the large scale information loss that occurs when an expert leaves the team leads to a team whose effectiveness is greatly diluted.

- There is no escaping the fact that when an expert leaves the team, it is a major blow. Several months or years of dedication to learning and building source code in a particular field cannot be replaced.

In brief, it is always a good idea to have more than one expert in each part of the system being developed, especially the critical parts. This can't be achieved by swapping people in and out of projects quickly, all that will happen is that nobody will be an expert in any area. A better approach is to: -

- Have more than one person work on a single project, e.g. splitting the design, implementation and coding and slowly transition people into different areas as new projects come up in those areas.
- Developers transition to new areas at strategic times, for example after a major release.

Pair / peer programming. There is a lot of debate raging about the merits / pitfalls of pair programming. Much comes down to the individuals, the project and organization. There is a lot of literature on this topic, see http://en.wikipedia.org/wiki/Pair_programming as a starting point.

Accountability

Too often, when a bug report comes in from a customer, the parties that were responsible from the product have moved onto new development or nobody takes accountability for the problem – everyone points the finger at someone else. More often than not, some junior developer is left responsible to clean up the mess.

While we do not believe in singling people out or blame, but we do believe in personal responsibility. We will always be responsible for our code from initial conception, design, testing and use in the field. Also, we realise that often the most crucial and difficult part of a project occurs when the code is used by the customer. Customer support

requires a host of skills, from knowledge of the rationale behind design and test decisions, through to communication skills. So it does not make a lot of sense that most companies move the original experienced developers onto a new project once it is released and have a newbie or permanent support person handle the customer support.

Furthermore, personal responsibility is the only way to truly learn and grow as a software developer. Everyone makes mistakes, the only way to learn is to take responsibility and fix your own design or coding bugs. If you consistently move on to a new project without 'feeling the hurt' you will never grow as an engineer.

This does not mean that we will be the ones to physically fix every bug in our code. However, if a bug does turn up in our code or design, we won't run for the hills, feign ignorance, blame someone else or claim insanity as so often occurs. Instead we will direct our effort to solve the problem as effectively as possible, whether this involves solving the bug ourselves or assisting someone else.

This may seem to be at odds with collective code ownership, but it is not. Any developer should work on any piece of code. There should be no cases where a developer does not allow other peer developers to touch their code or for developers to use the excuse of 'I am not familiar with that code' to not fix a bug. Instead, we are saying that if a bug does arise all parties that may be responsible for introducing the bug voluntarily take ownership of and assist in fixing that bug.

An extension of this is that we eat our own dogfood. Where possible, we try to use our own software in-house. We put our methodologies into practice in our own software development process.

Automation

Automate anywhere you see humans doing repetitive time consuming tasks that are likely to continue into the future and that could be done by a computer. Determine that cost of implementing automation and the saving that would be reaped. Be wary that it is

possible to over automate or end up with unwieldy automation systems that require a lot of maintenance time.

Judicious branching

The main purpose of a branch is to isolate customers from the instability of a changing product. Excessive branching does more harm than good. When possible we have two branches: -

- A mainline branch that has all the latest features and code.
- A stable working branch that has minimal changes and customer bug fixes, and has run through a beta trial. This is the version that is deployed at customer sites.

Diagnostics

Auditing and diagnostic are beneficial for: -

- Time tracking. We track our time usage on a daily basis. This helps us to determine as a group where our time is being spent.
- Application logging. Application logging is invaluable when tracking down problems that occurred at a customer site. It is like the evidence that is left behind at a murder scene. The more logging there is the more evidence there is and the quicker it is to track down the cause of a problem. Logging shortens our response time to customer issues which enhances our agility.

Intelligent feedback. Often it is useful to go further than basic trace or error logging so that the product can determine that something has gone wrong and alert the interested parties or take some corrective action. For example, the product sends an email to the support desk when a crash occurs or the memory allocations get too high. A word of warning is that the diagnostic mechanisms should not be overly complex otherwise it becomes likely that the diagnostic code will introduce its own problems (crashes, memory leaks, etc).

Tools

We consistently review and use tools that will help us to get the job done. Example of the tools we use are: -

- Source control.
- Bug tracking.
- Automated builds tools.
- Time tracking tools.
- IDE's.
- Coding languages.
- Profiling and memory leak detection tools.

Depth of thought

Pair programming and face to face communication are not always the most appropriate practices. Face to face communication promotes a rapid and efficient exchange of ideas, but there are times when a carefully thought out email (especially when broaching a sensitive topic) is a better approach. Also, it is sometimes better to think through a problem in isolation before bashing it out with a peer.

Also, open-plan offices or cube farms are not conducive to deep continuous thought on a design or problem. We recognize the importance of thinking about or working on something in a quiet setting, so we encourage developers to work from home or away from the open plan office when appropriate.

Agile decision process

The development process often gets bogged down when a decision has to be made and there is no clear consensus on the best way forward. In these processes we adopt the following approach: -

- Try to find a consensus. Allow each party to put their argument forward.
- If after a few minutes no consensus has been reached and there is a clear majority, the minority can opt to 'disagree and commit'. In this case the minority indicate

that they don't clearly agree, but they will agree to commit to the solution so that the project can keep moving forward.

- If no parties will budge, then sometimes (time permitting) it is better to shelve the decision and work on something else for a day or two. It is clearly a thorny problem that has no easy solution, often after a couple of days a new solution or compromise will be found, or another party will 'see the light'.

Egoless development

There is often a sense of pride that comes with learning a technique, thinking of a clever idea, formulating a design or coding a masterpiece and it is not fun to have holes poked in our creations. However, we aim to be open to other ideas and disagreements and to encourage other's opinions. This encourages a more agile approach; if a solution does not hold up to scrutiny, then it should be modified or canned.

Attack problems and not people.

Leave performance appraisals to the performance review process and keep it out of the day to day development process. This ensures we are agile since we focus on solving the problem and making the customer happy rather than internal politics.

Effective customer support

It is great fun to negotiate, design and develop a product, but nobody is particularly keen on supporting and maintaining the product once it has gone out of the door. We believe that effective customer support is a critical and difficult part of any software development process and requires the input of the most skilled developers. Specifically, when a customer reports an issue: -

1. The customer should receive a response immediately from a real person to indicate that their issue has been received and is being looked into.
2. The response should be followed up within a day or two (ideally this should occur straight away since priority should be given to customer support over new development) by a developer who was responsible for that part of the product. The developer should provide an explanation of why the problem might have

occurred and should give an estimate of when the problem will be fixed by (if possible).

3. Ideally, the customer should be able to track the progress of the issue, e.g. through a public issue tracking system such as Mantis.
4. The customer should be involved in testing the fix as soon as it is available.

Support tools that capture the customer information will be a great help to developers for tracking down and fixing customer problems, for example, a utility that grabs a copy of the application logs and the applicable configuration settings and zips them up ready for an email.