Chapter 7: Software Development Methodologies

A software development methodology is a way of managing a software development project. This typically address issues like selecting features for inclusion in the current version, when software will be released, who works on what, and what testing is done.
No one methodology is best for all situations. Even the much maligned waterfall method is appropriate for some organizations. In practice, every organization implements their software development project management in a different way, which is often slightly different from one project to the next. None the less, nearly all are using some subset or combination of the ones discussed here.
Choosing an appropriate management structure can make a big difference in achieving a successful end result when measured in terms of cost, meeting deadlines, client happiness, robustness of software, or minimizing expenditures on failed projects. As such, it is worth your time to learn about a number of these and make your best effort to choose wisely.

Agile family - Agile methods are meant to adapt to changing requirements, minimize development costs, and still give reasonable quality software. Agile projects are characterized by many incremental releases each generated in a very short period of time. Typically all members of the team are involved in all aspects of planning, implementation, and testing. This is typically used by small teams, perhaps nine or fewer, who can have daily face-to-face interaction. Teams may include a client representative. There is a strong emphasis on testing as software is written. The disadvantages of the Agile methods are that they work poorly for projects with hundreds of developers, or lasting decades, or where the requirements emphasize rigorous documentation and well documented design and testing.

- SCRUM - is currently the most popular implementation of the agile ideals. Features are added in short sprints (usually 7-30 days), and short frequent meetings keep people focused. Tasks are usually tracked on a scrum board. The group is self-organizing and collaboratively managed, although there is a scrum master tasked with enforcing the rules and buffering the team from outside distractions.
- Dynamic Systems Development Model (DSDM) - is an agile method that sets time, quality, and cost at the beginning of the project. This is accomplished by prioritizing features into musts, shoulds, coulds, and won't haves. Client involvement is critical to setting these priorities. There is a pre-project planning phase to give the project a well-considered initial direction. This works well if time, cost, and quality are more important than the completeness of the feature set.
- Rapid Application Development (RAD) - is a minimalist agile method with an emphasis on minimizing planning, and a focus on prototyping and using reusable components. This can be the best choice when a good prototype is good enough to serve as the final product. RAD has been criticized because the lack of structure leads to failed projects or poor quality products if there is not a team of good developers that feel personally committed to the project.
- Extreme Programming (XP) - is a frequent release development methodology in which developers work in pairs for continuous code review. This gives very robust, high quality software, at the expense of twice the development cost. There is a strong emphasis on test driven development.
- Feature-Driven Development (FDD) - is an iterative development process with more emphasis on planning out the overall architecture, followed by implementing features in a logical order.
- Internet-Speed Development - is an iterative format that emphasizes daily builds. It is tailored to the needs of open source projects where volunteer developers are geographically distributed, and working around the clock. The project is built from a

vision and scope statement, but there are no feature freezes. Development is separated into many small pieces that can be developed in parallel. The down side of this process is that the code is constantly in flux, so there are not necessarily stable release points where the code is particularly well tested and robust.

Agile: Strengths and Weaknesses

Strengths

- … Iterative-incremental process
- … Based on modeling the problem domain and the system
- … Requirements are allowed to evolve over time.
- … Traceability to requirements through the Product Backlog
- … Architecture of the system drafted before the development engine is started.
- … Iterative development engine governed by careful planning and reviewing planning and
- … Active user involvement
- … Simple and straightforward process
- … Simple and straightforward process
- … Early and frequent releases, demonstrating functionality at the end of each iteration (sprint) of the development cycle.

Weaknesses

- … Integration is done after all increments are built
- … Lack of scalability
- … Lack of scalability
- … Based on the assumption that human communication is sufficient for running projects of any size and keeping them focused
- … Not necessarily seamless (details of tasks are not prescribed)
- … No clear-cut design effort
- … Model-phobic
- … Models are not prescribed, leaving it to the developer to decide what model can be useful
- … Lack of formalism

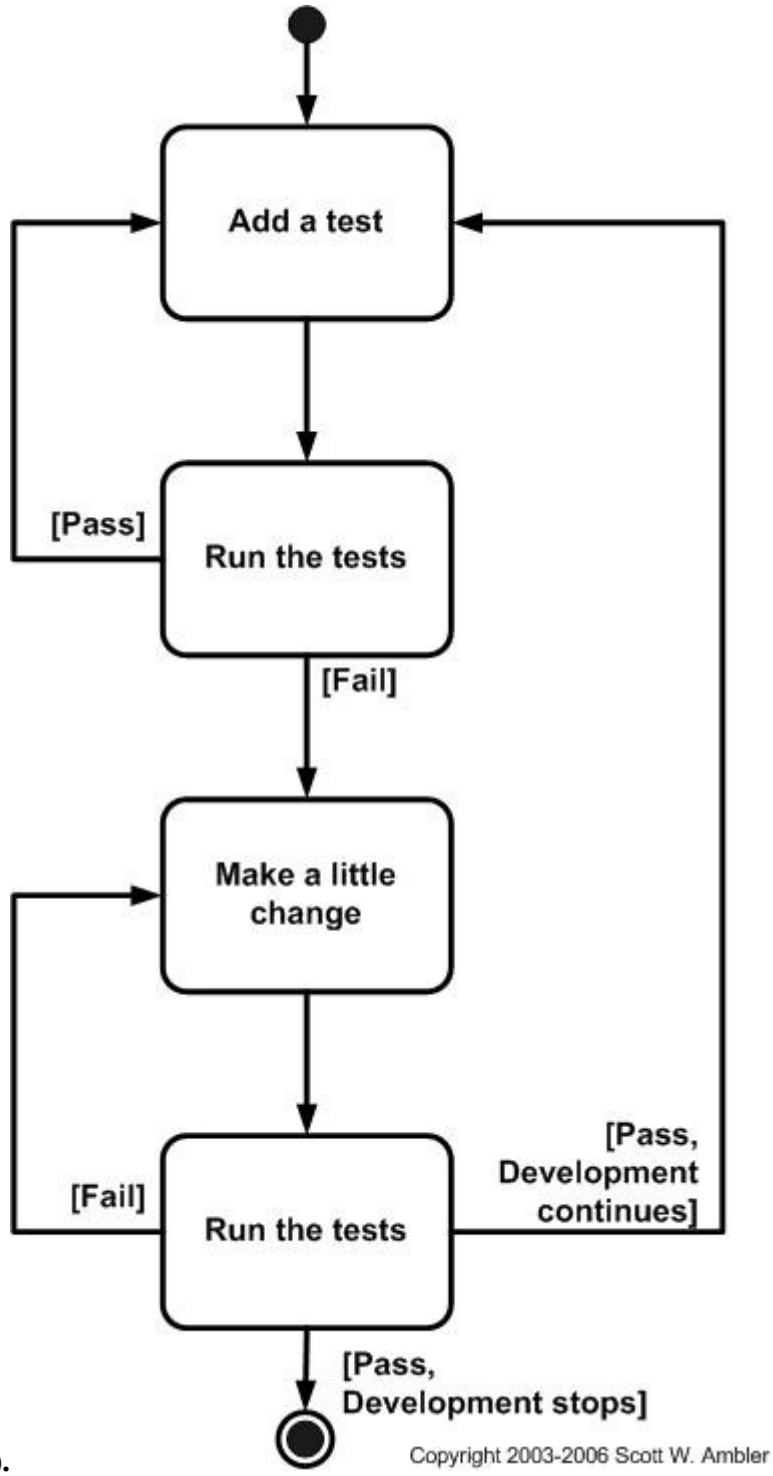
Test Driven Development

Test-driven development (TDD) (Beck 2003; Astels 2003), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. What is the primary goal of TDD? One view is the goal of TDD is specification and not validation (Martin, Newkirk, and Kess 2003). In other words, it's one way to think through your requirements or design before your write your functional code (implying that TDD is both an important agile requirements and agile design technique). Another view is that TDD is a programming technique.

What is TDD?

The steps of test first development (TFD) are overviewed in the UML activity diagram of Figure 1. The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).

Figure 1. The Steps of test-first development



(TFD).

TDD can be describe with this simple formula:

TDD = Refactoring + TFD.

TDD completely turns traditional development around. When you first go to implement a new feature, the first question that you ask is whether the existing design is the best design possible that enables you to implement that functionality. If so, you proceed via a TFD approach. If not, you refactor it locally to change the portion of the design affected by the new feature, enabling you to add that feature as easy as possible. As a result you will always be improving the quality of your

design, thereby making it easier to work with in the future.

Instead of writing functional code first and then your testing code as an afterthought, if you write it at all, you instead write your test code before your functional code. Furthermore, you do so in very small steps – one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn't present. In fact, they refuse to add even a single line of code until a test exists for it. Once the test is in place they then do the work required to ensure that the test suite now passes (your new code may break several existing tests as well as the new one). This sounds simple in principle, but when you are first learning to take a TDD approach it proves require great discipline because it is easy to "slip" and write functional code without first writing a new test. One of the advantages of pair programming is that your pair helps you to stay on track.

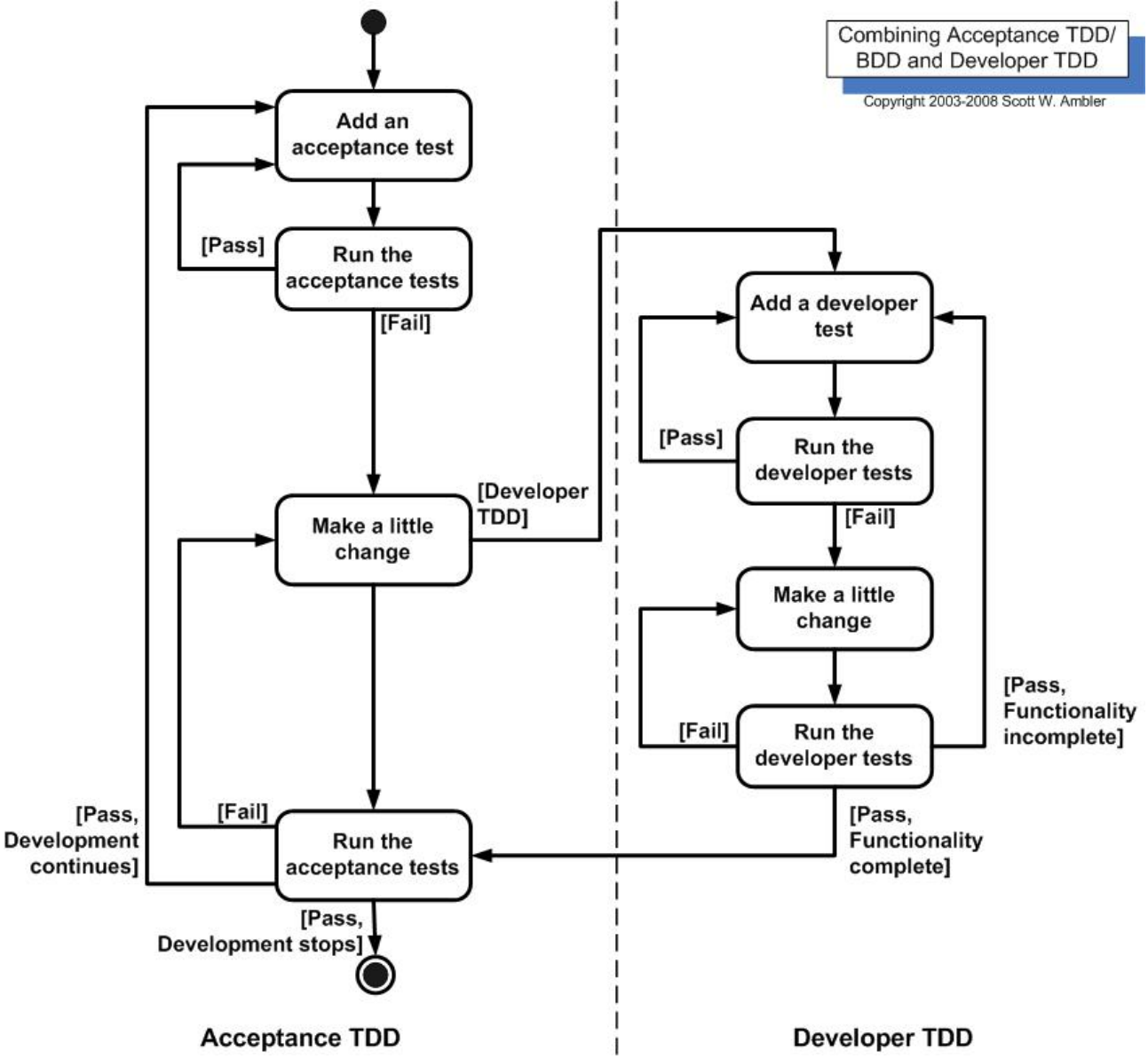
There are two levels of TDD:

1. **Acceptance TDD (ATDD)**. With ATDD you write a single acceptance test, or behavioral specification depending on your preferred terminology, and then just enough production functionality/code to fulfill that test. The goal of ATDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis. ATDD is also called Behavior Driven Development (BDD).

2. **Developer TDD**. With developer TDD you write a single developer test, sometimes inaccurately referred to as a unit test, and then just enough production code to fulfill that test. The goal of developer TDD is to specify a detailed, executable design for your solution on a JIT basis. Developer TDD is often simply called TDD.

Figure 2 depicts a UML activity diagram showing how ATDD and developer TDD fit together. Ideally, you'll write a single acceptance test, then to implement the production code required to fulfill that test you'll take a developer TDD approach. This in turn requires you to iterate several times through the write a test, write production code, get it working cycle at the developer TDD level. -

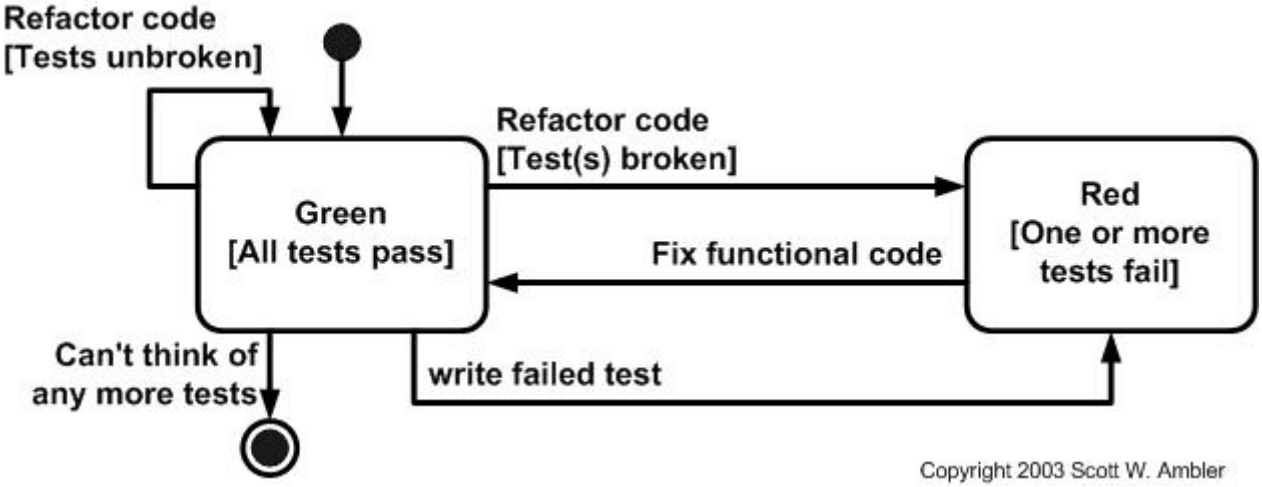
Figure 2. How acceptance TDD and developer TDD work together.



Note that Figure 2 assumes that you're doing both, although it is possible to do either one without the other. In fact, some teams will do developer TDD without doing ATDD, see survey results below, although if you're doing ATDD then it's pretty much certain you're also doing developer TDD. The challenge is that both forms of TDD require practitioners to have technical testing skills, skills that many requirement professionals often don't have (yet another reason why generalizing specialists are preferable to specialists).

An underlying assumption of TDD is that you have a testing framework available to you. For acceptance TDD people will use tools such as Fitnesse or RSpec and for developer TDD agile software developers often use the xUnit family of open source tools, such as JUnit or VBUnit, although commercial tools are also viable options. Without such tools TDD is virtually impossible. Figure 3 presents a UML state chart diagram for how people typically work with such tools. This diagram was suggested by Keith Ray. -

Figure 3. Testing via the xUnit Framework.



Kent Beck, who popularized TDD in eXtreme Programming (XP) (Beck 2000), defines two simple rules for TDD (Beck 2003). First, you should write new business code only when an automated test has failed. Second, you should eliminate any duplication that you find. Beck explains how these two simple rules generate complex individual and group behavior:

- You develop organically, with the running code providing feedback between decisions.
- You write your own tests because you can't wait 20 times per day for someone else to write them for you.
- Your development environment must provide rapid response to small changes (e.g you need a fast compiler and regression test suite).
- Your designs must consist of highly cohesive, loosely coupled components (e.g. your design is highly normalized) to make testing easier (this also makes evolution and maintenance of your system easier too).

For developers, the implication is that they need to learn how to write effective unit tests. Beck's experience is that good unit tests:

- Run fast (they have short setups, run times, and break downs).
- Run in isolation (you should be able to reorder them).
- Use data that makes them easy to read and to understand.
- Use real data (e.g. copies of production data) when they need to.
- Represent one step towards your overall goal.

TDD and Traditional Testing

TDD is primarily a specification technique with a side effect of ensuring that your source code is thoroughly tested at a confirmatory level. However, there is more to testing than this. Particularly at scale you'll still need to consider other agile testing techniques such as pre-production integration testing and investigative testing. Much of this testing can also be done early in your project if you choose to do so (and you should).

With traditional testing a successful test finds one or more defects. It is the same with TDD; when a test fails you have made progress because you now know that you need to resolve the problem. More importantly, you have a clear measure of success when the test no longer fails. TDD increases your confidence that your system actually meets the requirements defined for it, that your system actually works and therefore you can proceed with confidence.

As with traditional testing, the greater the risk profile of the system the more thorough your tests need to be. With both traditional testing and TDD you aren't striving for perfection, instead you are testing to the importance of the system. To paraphrase Agile Modeling (AM), you should "test with a purpose" and know why you are testing something and to what level it needs to be tested. An interesting side effect of TDD is that you achieve 100% coverage test – every single line of code is tested – something that traditional testing doesn't guarantee (although it does recommend it). In general I think it's fairly safe to say that although TDD is a specification technique, a valuable side effect is that it results in significantly better code testing than do traditional techniques.

Comparing TDD and AMDD:

- TDD shortens the programming feedback loop whereas AMDD shortens the modeling feedback loop.
- TDD provides detailed specification (tests) whereas AMDD is better for thinking through bigger issues.
- TDD promotes the development of high-quality code whereas AMDD promotes high-quality communication with your stakeholders and other developers.
- TDD provides concrete evidence that your software works whereas AMDD supports your team, including stakeholders, in working toward a common understanding.
- TDD "speaks" to programmers whereas AMDD speaks to business analysts, stakeholders, and data professionals.
- TDD is provides very finely grained concrete feedback on the order of minutes whereas AMDD enables verbal feedback on the order minutes (concrete feedback requires developers to follow the practice Prove It With Code and thus becomes dependent on non-AM techniques).
- TDD helps to ensure that your design is clean by focusing on creation of operations that are

callable and testable whereas AMDD provides an opportunity to think through larger design/architectural issues before you code.

- TDD is non-visually oriented whereas AMDD is visually oriented.
- Both techniques are new to traditional developers and therefore may be threatening to them.
- Both techniques support evolutionary development.

Which approach should you take? The answer depends on your, and your teammates, cognitive preferences. Some people are primarily "visual thinkers", also called spatial thinkers, and they may prefer to think things through via drawing. Other people are primarily text oriented, non-visual or non-spatial thinkers, who don't work well with drawings and therefore they may prefer a TDD approach. Of course most people land somewhere in the middle of these two extremes and as a result they prefer to use each technique when it makes the most sense. In short, the answer is to use the two techniques together so as to gain the advantages of both.

How do you combine the two approaches? AMDD should be used to create models with your project stakeholders to help explore their requirements and then to explore those requirements sufficiently in architectural and design models (often simple sketches). TDD should be used as a critical part of your build efforts to ensure that you develop clean, working code. The end result is that you will have a high-quality, working system that meets the actual needs of your project stakeholders.

Why TDD?

A significant advantage of TDD is that it enables you to take small steps when writing software. This is a practice that have promoted for years because it is far more productive than attempting to code in large steps. For example, assume you add some new functional code, compile, and test it. Chances are pretty good that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps. I generally prefer to add a few new lines of functional code, typically less than ten, before I recompile and rerun my tests.

Behaviour Driven:

Behavior-driven development (BDD) is a software development methodology in which an application is specified and designed by describing how its behavior should appear to an outside observer.

A typical business application project would begin by having stakeholders offer concrete examples of the behavior they expect to see from the system. All coding efforts are geared toward delivering these desired behaviors. The real-life examples gleaned from stakeholders are converted into acceptance criteria with validation tests that are often automated. The results of these tests provide confidence to stakeholders that their desired business objectives for the software are being achieved. Ideally, the reports are generated in such a way that the average stakeholder can understand the business logic of the application. Living documentation is used throughout the system to ensure that all documentation is up to date and accurate.

In practice, behavior-driven development may be similar to test-driven development when all stakeholders have programming knowledge and skills. However, in many organizations, BDD offers the ability to enlarge the pool of input and feedback to include business stakeholders and end users who may have little software development knowledge. Because of this expanded feedback loop, BDD may more readily be used in continuous integration and continuous delivery environments.

BDD practices

The practices of BDD include:

- Establishing the goals of different stakeholders required for a vision to be implemented
- Drawing out features which will achieve those goals using feature injection
- Involving stakeholders in the implementation process through outside–in software development
- Using examples to describe the behavior of the application, or of units of code
- Automating those examples to provide quick feedback and regression testing
- Using ‘should’ when describing the behavior of software to help clarify responsibility and allow the software’s functionality to be questioned
- Using ‘ensure’ when describing responsibilities of software to differentiate outcomes in the scope of the code in question from side-effects of other elements of code.
- Using mocks to stand-in for collaborating modules of code which have not yet been written

BDD vs TDD

• It is important to keep BDD distinct from TDD. These two practices are equally important but address different concerns and should be complementary in best development practices.
• BDD is concerned primarily with the specification of the behavior of the system under test as a whole, thus is particularly suited for acceptance and regression testing. TDD is concerned primarily with the testing of a component as a unit, in isolation from other dependencies, which are typically mocked or stubbed.
• BDD should talk the language of the business domain and not the language of the development technology, which on the other hand is “spoken” by TDD.