Feature Driven Development: Enterprise Evolution to Agile

Feature Driven Development (FDD) is not as well-known as many other agile frameworks.

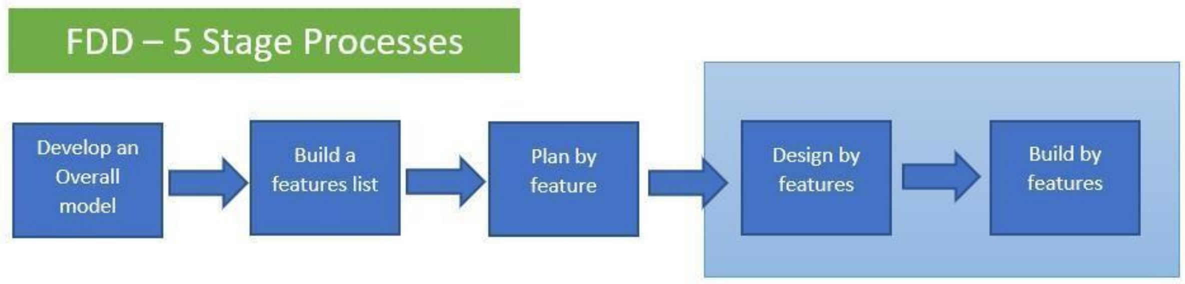
But when you're dealing with a **large, long-running project**, especially in an organization where agile is still mostly confined to software development, FDD may be your friend.

Feature Driven Development bridges the gap between traditional controlled waterfall approaches and the emergent processes found in agile approaches like behaviour driven development, extreme programming, and scrum.

Work starts with building a domain object model and identifying all the features and feature sets (groups of features). After prioritizing the features, work proceeds iteratively and incrementally.

The combination of more traditional planning with agile build practices helps FDD fit in with traditional enterprise structures.

It works well for large teams and long, complex projects.



Planning Phase

The planning phase is about getting to a shared understanding of the scope of the product (a new software system) between the customer and their users, and the development organization.

Step 1: Develop an Initial Model

The first step in FDD is to outline an initial domain object model.

Teams of developers and domain experts (e.g., future users in the customer's company) each define a model that reflects their shared understanding of the important objects in the new system. The chief architect supports and guides them in this work.

Step 2: Develop a Feature List

Next up is the feature list — all the functionality the product needs to offer.

To do this, you split the domain model into subdomains that each represent a business function. For each subdomain, you then list all the features you require for the product to support your customer in that business function.

Features that are closely related are often grouped into feature sets. A feature set can also be a feature broken down into smaller ones because it would otherwise take more than two weeks to complete.

Step 3: Plan by Feature

The third step is where you plan the order in which to develop features and feature sets.

Factors to consider, are:

Which features your customer's users need most urgently.

Which features will deliver the most value soonest.

Development time estimates — you don't want to leave the biggest features for last.

Technical dependencies between features.

Risks involved with features — for example compliance testing, using new technology, learning new frameworks and libraries.

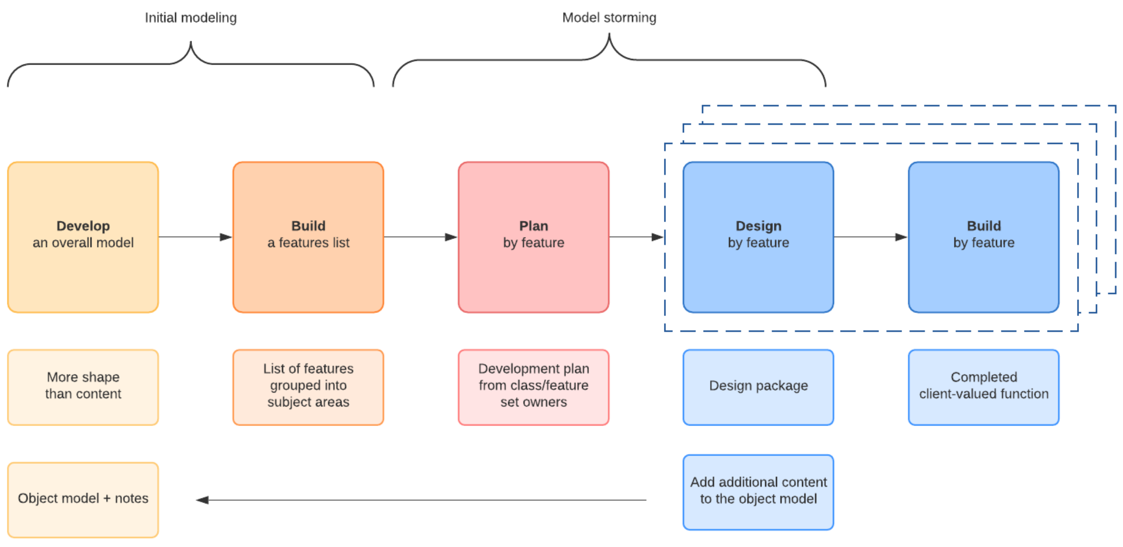
Available people, their skills and expertise.

Workload.

Construction Phase

In the design and build phase of FDD, you work through the feature list on a feature-by-feature basis.

Work proceeds in two-week cycles, or iterations and is thus inherently iterative and incremental. The strict two-week cycle is why FDD wants you to break up features to fit that.



Step 4: Design by Feature

Each feature team works on the detailed design of the features assigned to them for the current iteration.

Developers and domain experts, helped by the chief architect and chief programmer as needed, use UML modeling techniques such as class diagrams, sequence diagrams, and state transition diagrams to specify what the feature will do and how it'll do it.

Step 5: Build by Feature

Each feature team then works to turn their design into working software, test it, and gather feedback from domain experts to verify that the feature works as intended. When all's okay, they'll integrate their work with everything that's been built in this and previous iterations.

It's agile in its emphasis on meeting the needs of the end-users, collaboration with customer's domain experts, and breaking down requirements into small features designed to deliver business value.

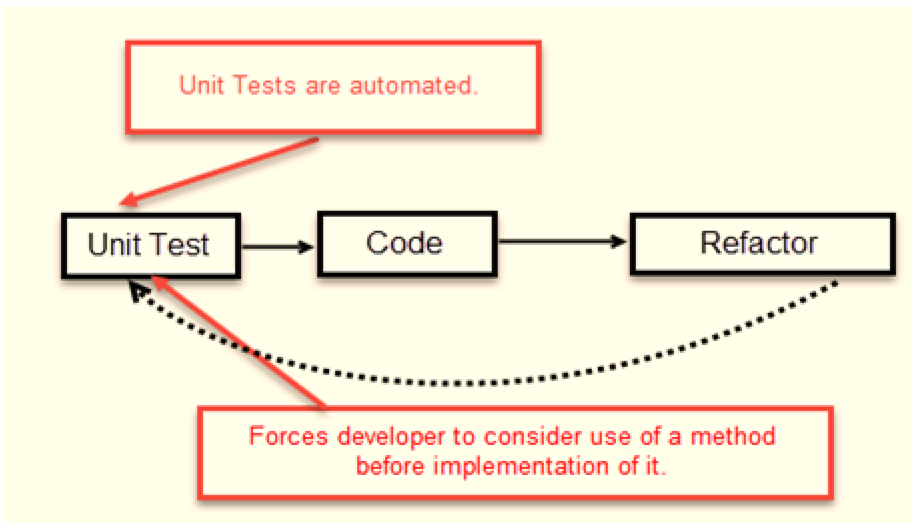
In other respects it's much less agile.

All in all, I'd dare to posit that FDD is a good step away from the waterfall approach, but it falls short of many accepted agile practices that are better at encouraging everyone to take responsibility and activating and engaging everyone's creativity, and deferring decisions to the last responsible moment.

\

Test Driven Development

Test Driven Development (TDD) is software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and making code simple and bug-free.
Test-Driven Development starts with designing and developing tests for every small functionality of an application. TDD framework instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The TDD full form is Test-driven development.


Unit Tests are automated.
Unit Test
Code
Refactor
Forces developer to consider use of a method before implementation of it.

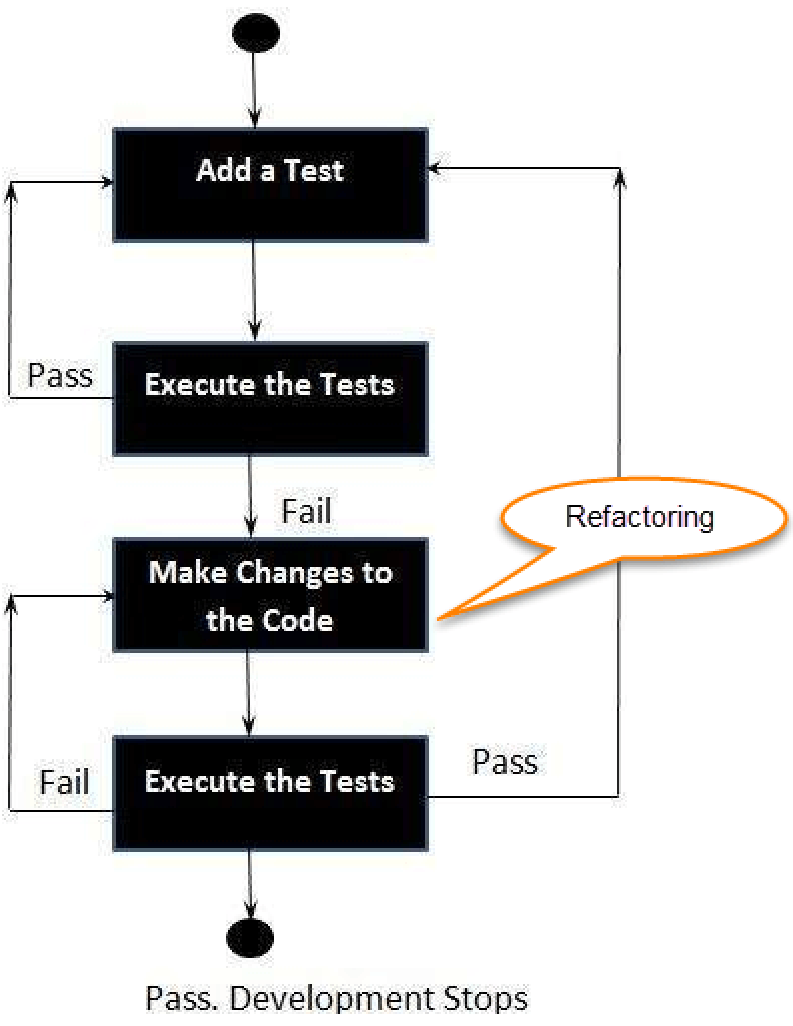
The simple concept of TDD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests. (Tests are nothing but requirement conditions that we need to test to fulfill them).

Test-Driven development is a process of developing and running automated test before actual development of the application. Hence, TDD sometimes also called as **Test First Development.**

How to perform TDD Test

Following steps define how to perform TDD test,

1. Add a test.
2. Run all tests and see if any new test fails.
3. Write some code.
4. Run tests and Refactor code.
5. Repeat.



TDD Vs. Traditional Testing

Below is the main difference between Test driven development and traditional testing:

TDD approach is primarily a specification technique. It ensures that your source code is thoroughly tested at confirmatory level.

- With traditional testing, a successful test finds one or more defects. It is same as TDD. When a test fails, you have made progress because you know that you need to resolve the problem.
- TDD ensures that your system actually meets requirements defined for it. It helps to build your confidence about your system.
- In TDD more focus is on production code that verifies whether testing will work properly. In traditional testing, more focus is on test case design. Whether the test will show the proper/improper execution of the application in order to fulfill requirements.
- In TDD, you achieve 100% coverage test. Every single line of code is tested, unlike traditional testing.
- The combination of both traditional testing and TDD leads to the importance of testing the system rather than perfection of the system.
- In Agile Modeling (AM), you should “test with a purpose”. You should know why you are testing something and what level its need to be tested.