Introduction to Test Driven Development Methodology

Patrick Nicolas - March 2006

Contents

Introduction

Test-Driven Development (TDD) is a novel approach to software engineering that consists of short development iterations where the test case(s) covering a new functionality are written first. The implementation code necessary to pass the tests is implemented afterwards then tested against the test cases. Defects are fixed and components are refactored to accommodate changes. In this approach, writing the code is done with a greedy approach, i.e. writing just enough to make tests pass, and the coding per TDD cycle is usually only a one-to-at-most-few short method, functions or objects that is called by the new test, i.e. small increments of code.

Test-Driven Development can be implemented at several levels:

- Basic cycle which covers the generation of unit test cases, implementation, build and refactoring.
- Comprehensive development process which extends the methodology to overall black box testing scripts, including performance, scalability and security testing.
- Agile process such as SCRUM

Benefits and Limitations

There are several benefits to have the test cases written before the production code

- The engineer(s) have almost an immediate feedback on the components they develop and tested against the test cases. The turnaround time for the resolution of defects is significant shorter than traditional waterfall methodology where the code is tested days or weeks after implementation and the developer has moved to other modules.
- The implementation is more likely to match the original requirements as defined by product management. The test cases are easily generated from the use cases or user scenario and reflect the functional specifications accurately without interference from the constraints and limitation of the architecture design or programming constructs. The approach guarantees to some degree that the final version fulfill customer' or product marketing requests.
- The methodology prevents unwarranted design or components to crimp into the product. The test cases or unit test drivers define the exact set of required features. It is quite easy to identify redundant code, detect and terminate unnecessary engineering tasks.
- The pre-existence of test cases allow the developer to provide a first draft of an implementation for the purpose of prototyping, evaluation or alpha release, and postponed a more formal implementation through refactoring.
- TDD fits nicely into customer-centric agile process such as SCRUM. For instance, the sprint phase can be defined as the implementation of functionality to execute against a pre-defined set of test cases instead of the more traditional set of specifications or problem statement.
- TDD can lead to more modularized, flexible, and extensible code. The methodology requires that the developer think of the software in terms of small units that can be written and tested independently and integrated together later.

As with any software development methodology, TDD has some clear limitations

- It fits very well with unit test tools but does not scale with web-based GUI or database development, although SQL, ODBC or JDBC clients or HTTP clients can be easily created.
- The methodology is cumbersome if the design or the programming language or IDE does not have an integrate unit test framework such as C++test, jUnit.

Basic TDD Cycle

Unit Test Drivers

The most primitive cycle for TDD involves the automation or semi-automation of the generation and execution of unit test drivers. Programming language such as Java, C++ and Perl as well as IDE such as Eclipse or Visual Studio provide support for integrating unit test frameworks.

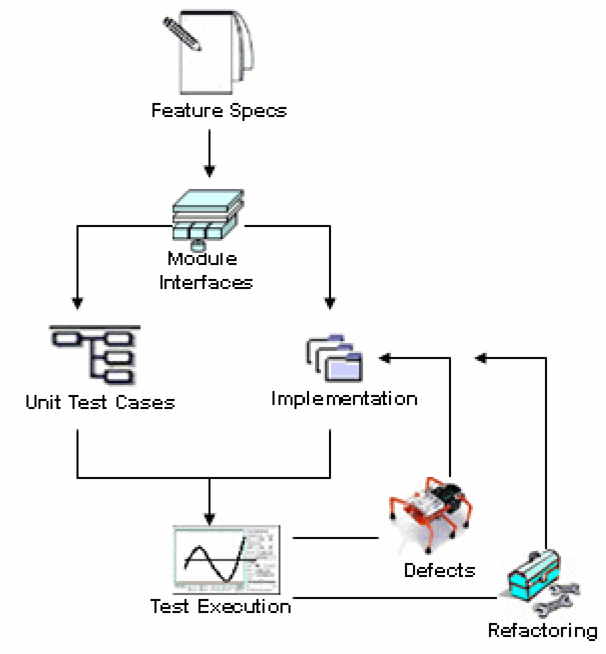

Fig. 1 Representation of Basic TDD Cycle

The key tasks in the TDD methodology are

1. Product management defines the new features or improvement to implement
2. Architect or senior developer define an interface (declarative) that fulfill the use case
3. Engineers create a unit test driver to test the feature
4. Engineers implements a first version which is tested against the test driver
5. The implementation is subsequently refined then re-tested incrementally
6. At each increment engineer(s) may upgrade the original design through refactoring.

Automation

One well documented concern regarding this methodology is the effort and cost required to create detailed unit test code. Consequently, a commitment to automate part of the unit testing cycle is essential to the overall adoption of TDD. The two key elements of an automated TDD strategy are

- Automated test generation: The team should investigate and adopt framework or IDE that support the generation of unit test code from a declarative interface
- Automated test execution: The test code should be automatically compiled and executed as part of the build and regression test process.

Example of a client driver code generated through a framework.

```
    public interface WebServicesEP {
        public  String endpoint(String hostName, int port);
    }

    public class WebServicesEP_Client {
       private WebServicesEP _server = null;
       private String[] _args0 = null;
       private int[] _args1 = null;

       public void init(String[] args0);
       public void init(int[] args1);

       public void WebServicesEP_Main(String[] env) throw TestException {
               _server = new WebServicesEP_impl();

               try {
                       for( int j = 0; j < _args[0].length; j++) {
                               for( k = 0; k < _args[1].length; k++) {
                                       String ePStr = _server.endPoint( _args0[j], _args1[k]);
                               }
                       }
               }
               catch( Exception e) {
                       throw new TestException(e.toString());
               }
               finally {
                       throw new TestException(e.toString());
               )
       }
  }
```

Extending the TDD Process

Product or System Testing

Most of TDD practitioners focus on unit test driver with or without automation. However, the concept can be extended to product testing as long as interfaces are clearly defined and a test scripting engine is available. The engine should be able to generate sequence of actions or

traffic which simulates the user interaction with the overall product or any of its components such as GUI, database or web servers.
The typical client test engine generates

- Graphical actions and events through a record and replay mechanism
- Command line request
- SQL, HTTP or TCP traffic or packets

The diagram (Fig 2) below describes the role of

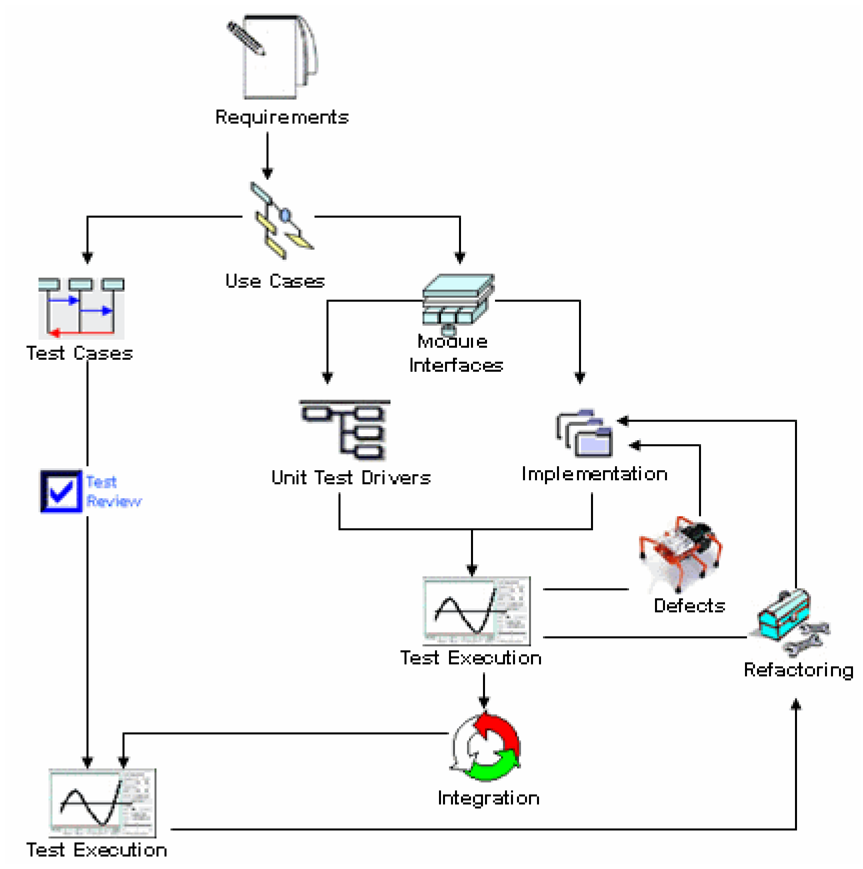

Fig. 2 Representation of Extended TDD Process

Here are the steps

1. Product management defines the new set of features or improvement to implement
2. Use cases are created to describe the basic sequence
3. Architect specifies the set of programming or graphical interface functions
4. QA engineers rely on use case to create test cases and test scripts
5. Unit test drivers are created from the user cases and programming interface
6. Engineers implements a first version or draft which is validated (failed in the first couple attempts) against the test scripts and unit test drivers
7. The implementation is incrementally refined and re-factored as necessary
8. The product is then tested against the test scripts following the integration phase.

Code Coverage Analysis

The value of a test harness depends highly on the path, code or statement covered during the execution of those tests. Code coverage can be easily measured as part of the regression testing during the execution of the test harness.

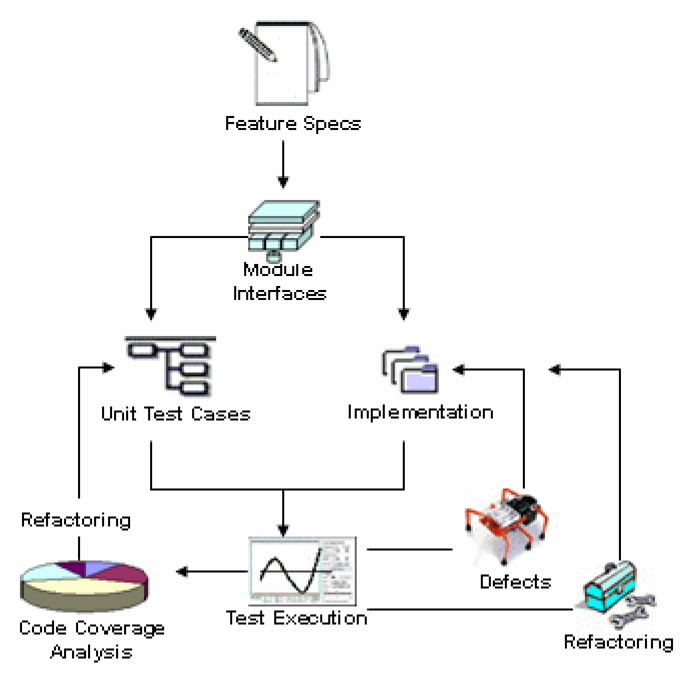

Fig. 3 Integrating Code Coverage Analysis with TDD

As described in Fig. 3, code coverage Analysis allows the team to evaluate the quality of the unit test client code in term of coverage. The unit test code can then be re-factored to improve the coverage of statements, path and code.
Code coverage metrics should also be used to provide a context for the traditional quality metrics such as number of pending defects or ratio of failed regression tests.

Refactoring

Refactoring is a key element of any agile process including TDD methodology. A trend analysis of the evolution of the design and implementation through successive refactoring iteration may give a hint on the probability to reach a stable architecture or component. It is quite conceivable to add design review of the component design during the development cycle to evaluate the stability of the implementation as described in fig 4.

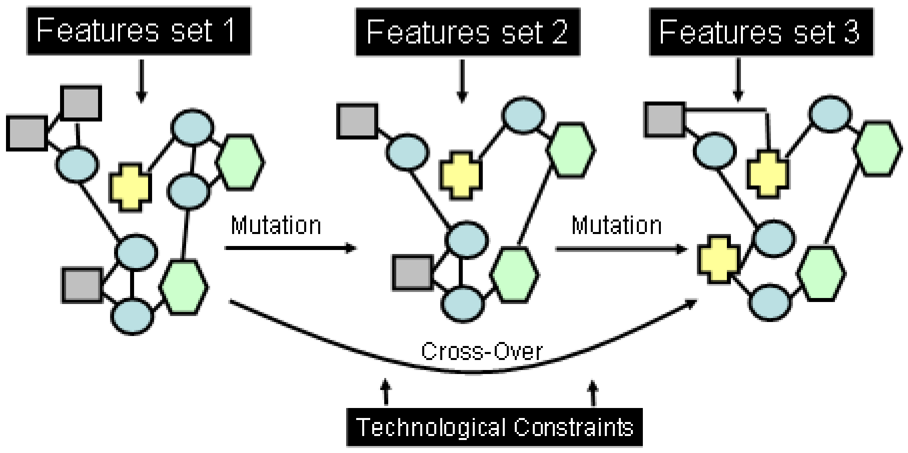

Fig. 4 Mutation and stabilization of design during incremental refactoring

Integration with SCRUM

Test Driven Development is the greatest asset to come out of the agile movement. Significant quality and productivity gains are made by using TDD. Hence it is very rapidly spreading through development circles. However, the engineering teams who already practicing agile process such as SCRUM need to reconcile their existing practice with TDD.
By providing a framework to automatically validate incrementally production code against predefined unit test drivers, TDD allows to extend the purpose of sprint as converging the implementation to pass the test harness and add critical quality metrics (ratio of code coverage and passed test) to the monitoring dashboard.

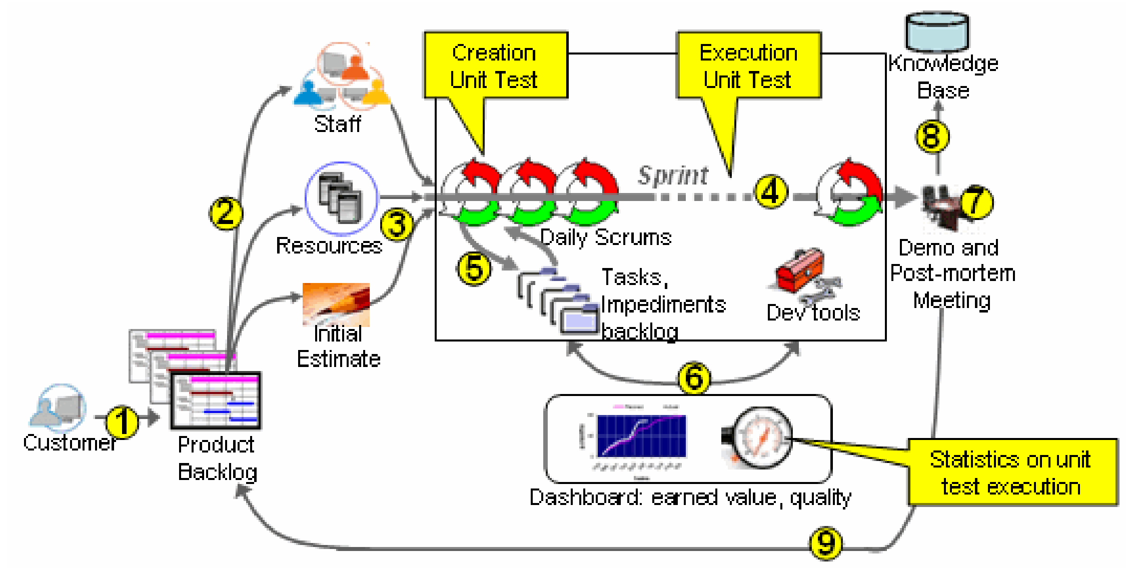

Fig.5 Overall SCRUM agile process with TDD integration points

1. The customers are involved in defining the product roadmap, key user scenario and functionality. Their feedback is used to prioritize the product features and enhancements which are logged into the Product Backlog.
2. The engineering manager defines the initial list of tasks and features to be implemented in product backlog. The manager gathers the necessary resources (hardware, development tools) and the appropriate composition of the cross-functional team to drive the sprint. One of the team member is designated as SCRUM master responsible for maintaining the current list of issues.
3. Management commits to provide the team with all the initial resources and estimates for the project not only at its inception but also during sprint. **The unit test code has to be created before the spring phase starts. As a matter of fact, successfully executing the test harness becomes a key objective of the spring phase.**
4. The Sprint phase lasts usually 4 weeks during which the team works autonomously. Project executed around the clock by distributed teams may last 2 weeks. Management may add or remove features or defects to accommodate the pre-defined completion date. **Test harness is automatically executed on daily basis and its results are added to the performance metrics used by management to steer the project.**
5. Short daily SCRUM meetings are held to report, analyze new impediments and update the tasks backlog. Management attends the meeting to get status and provide, if needed, external resources and skills required by the project. The task backlog can be display on intranet site using Wiki style of presentation.
6. Product quality, burn-down and risks are constantly monitored and reported on the intranet. The statistics are generated from the daily SCRUM meeting as well as the development tools (build, automated test, requirement management, defects tracking system).
7. A presentation or demo of new feature is made to product management, executives and potentially beta customers upon the completion of the project.
8. A post-mortem meeting is to be held to analyze the success and failure of the project. The findings are added to the knowledge base for the future SCRUM team to use.
9. Unresolved issues are added to the generic product backlog, for future improvements. Refactoring (design) of the product is a common tasks to be added for future SCRUM projects.