Impact of Agile and TDD Implementation in Database

Kalpna Sagar
University School of Information Technology
New Delhi, India

Bindu Goel
University School of Information Technology
New Delhi, India

ABSTRACT
Agile Software Development Methodologies not only provide high level of adaptability but also deal with short iterations for efficient and frequent product delivery, incorporate rapid change in requirements and provide high customer satisfaction. During the commercial development of software and academic project environment, the focus is given on the application but the backbone of the application is the database. And database professionals seem firmly rooted in serial development. So we require a technique with which data professionals can work in the evolutionary manner so that qualitative schema can be maintained. Evolutionary Development of the Database can be achieved with agile methodologies.

As the software development takes place it seems that entire database schema which is developed once and for all, is incorrect and unnecessary. But if we develop the database with agile and TDD, we can get schema with high quality, correctness and with frequent delivery. In TDD we write test cases before developing or making any database changes which minimizes the chances of introducing defects and early detection of defects in our system and database since changes happen as per the specifications of the tests. The paper describes agile database development through focal entity prototyping with TDD.

Keywords
Test Driven Development (TDD), Test First Development (TFD), and Entity Relationship (ER).

1. INTRODUCTION
Today software products are becoming more complex and need to develop more quickly. Furthermore, customers are demanding software with better quality, and requirements keep changing. These features can be achieved with agile software development methods and therefore in the recent years agile has gained significant attention in the software engineering community. Development of the application is adapting mainly evolutionary approach i.e. agile [1] because it can deal with customer's expectations more effectively and respond to changes more quickly. But the backbone of the application is database and database professionals seem firmly rooted in serial development. So we require a technique with which data professionals can work in the evolutionary manner so that qualitative schema can be maintained. TDD plays the major role in the database development although it seems to be a new concept for the data professionals. The paper describes the database aspects of TDD and specifies database behavior via database tests.

Now our main focus is to apply agile development approach via focal entity prototyping approach [2] to the database which will be productive and successful activity when undertaken in well organized manner. Changes in the database are incorporated through frequent short iterative processing table by table. In order to verify and validate the changes in the database, several database tests are to be created to check the database behaviors using TDD [3]. The tests are to be done with sample test data so that consistency and integrity in the database can also be checked. And early defects detection through TDD in the focal entity prototyping approach will result into high quality schema, which will be verified and validated iteratively and incrementally based on database table processing into the hands of the customer. Then feedback from the customer about the changes and additions commences well before the final completion of the system.

2. TEST DRIVEN DEVELOPMENT
Test driven development is an evolutionary approach to development which combines test first development where tests are written first before we write just enough production code to fulfill that test and refactor. The actual development is done in iterations on the basis of those tests [3]. For example when developer goes to implement a new feature, first question he asks himself is "Is this best design possible which enables me to add this feature?" If answer is yes, then he does the work to add feature. If the answer is no, he refactors design to make it best possible then they continue with a Test First Development (TFD) approach. There are four iterative steps of TDD [3] as depicted in the figure 1:

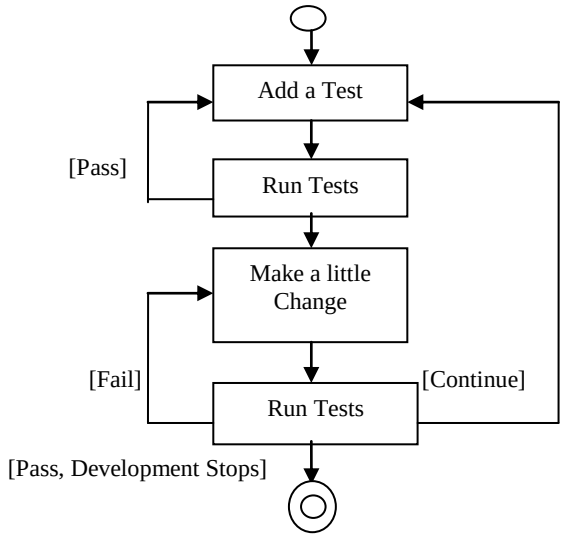

Fig 1: Test Driven Development

1. Quickly add a test: We basically need just enough code to fail (typically a single test).
2. Run tests: We will often need complete test suite although for sake of speed we may decide to run only a subset, to ensure that the new test does in fact fail.
3. Make change: Do just enough work to ensure that your production code passes the new test(s).

4. Run tests again: If it fails we need to update our production code and retest. Otherwise go back to Step 1.

In the recent times, TDD is gaining significant attention in the industry mainly for the application development. Focus is given on the application development but the backbone of the application is the database which should be well tested, validated and corrected. So our main job is to apply TDD to the database development. To extend TDD to database development, we need database equivalents of regression tests, refactoring, and continuous integration. So this test driven database development provides the benefits of TDD, plus a few others which are database related. First, it enables to ensure the quality of data. Data is an important corporate asset, yet many organizations suffer from significant data quality challenges. Second, it enables us to validate the functionality implemented within the database. Third, it enables data professionals to work in an evolutionary manner, just like application programmers.

The first part of Test driven database development is database regression testing [9]. There are two categories of database tests: interface tests and internal database tests. Interface tests (Black box tests) validate what data is flowing in, going out and getting mapped to our database. If an organization is doing any database testing at all, it is usually at the interface level. For example admin can access all the tables and all information in the database but user can access only the specific information whereas some user can't access even specific information. For this we can have the test case i.e. what will be the different access right on the tables let us say a table should have read access to admin whereas user should have no access to table.

Internal database tests (clear box tests) validate the internal structure, behavior, and contents of a database. The present study is highly focused on the internal and interface testing via focal entity prototyping.

A database refactoring [8] is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. Database schema includes both structural aspects such as table definitions and functional aspects such as stored procedures and triggers. Database refactoring is conceptually more difficult than code refactoring. Code refactoring only maintains behavioral semantics, whereas database refactoring also maintains informational semantics.

The continuous database integration deals with the regular integration of the changes to each team member's database instances including structural, functional and informational changes.

2.1 Importance of TDD in the Database

1. It promotes significantly higher-level of unit testing not only in the application but also in the database development i.e. every single line of code and interface and structural requirements (of database table) are tested – something that traditional testing doesn't guarantee. This seems to be far more productive than attempting to code and test it in large steps.
2. The suite of unit tests in the database provides constant feedback that during each change, behavioral and informational semantics remain preserved.
3. It helps in the early detection of defects in the database and can be tested earlier accordingly. Therefore high quality of the database can be achieved.
4. High quality of the database design can also be achieved via refactoring as discussed above.
5. Well tested and more comprehensible low-level database design as the database design evolves iteratively. This helps in building the confidence in the developer by running the tests.
6. Security concerns in the database can also be achieved via regression tests (interface tests) as discussed earlier.
7. It provides executable system specification that act as documentation in order to keep up-to-date unlike the traditional design documentation.
8. Test-driven development forces critical analysis and database design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.
9. Reduced debugging time

3. PRACTICE

A previous study presented in the paper at agile 2005[2] described Focal Entity Prototyping (which is based on an entity relationship concept). The study inspired from the above mentioned paper. Some recent publications on evolving the database can be found at [4, 5, 6, and 7]. Our main focus is to apply TDD to focal entity prototyping approach i.e. extending TDD to the database development via focal entity prototyping as depicted in figure 2.

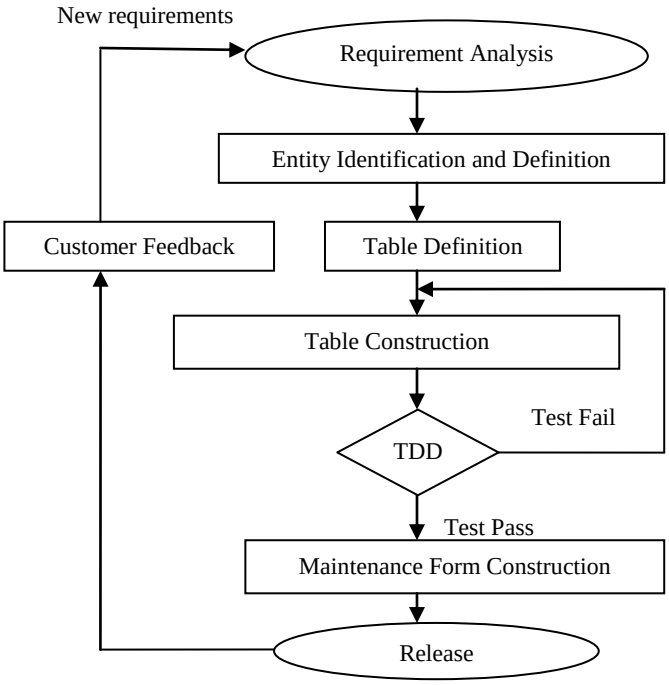

Fig 2: Focal Entity Prototyping with TDD

According to the customer feedback and the functional requirement of application (i.e. banking application which is described in the figure 3), entities (i.e. customer and account) are identified and defined. For each entity, construction of the table is done after which TDD is applied table by table. Then different tests cases need to be created (like interface, internal test, and structural refactoring tests). After running the test, if

the test passes, then maintenance form will be constructed and delivered to the customer for getting the feedback and if test fails changes are need to be done iteratively.

Integrity and Consistency are the most important features required in the database development which is where we can utilize the benefits of TDD.TDD is about checking the integrity and consistency not only in the application but also in database before and after the change. TDD is also about testing the application for predictable and unpredictable changes in the application so that behavior of the application is known and validated including the failure scenarios. TDD revolves around writing 3 types of test cases – with positive, negative and boundary conditions. Positive test cases should always pass and are meant to test the expected results. Negative test cases are intended for testing the unexpected results like passing in value that does not fall within the structural semantics of the system (passing integer when expecting Boolean is an example for this). Boundary test cases are meant for testing the boundary conditions which "just" fall outside the valid ranges like passing -1 when input from only 0 to 10 is expected. In context of the database, the idea of doing the TDD is to ensure the behavioral semantics don't change when introducing the iterative structural changes in the database. So any small change in the database, like adding a column or moving it around in different tables, can be tested for those semantics from the application perspective using test driven approach. For the application, the behavior of a stored procedure should not change - as in returning different data or not returning something that I expect to see as output which you can easily do by writing test cases before and after introducing any schema changes in Database.

Secondly, we can do the data validation, persistence and data execution checks by doing clear-box testing. We write some code using any programming language, in Java or C#, to call stored procedures to ensure what we intend to get as output before changes and after making changes which ensure our test cases run fine. We have some tools available for writing such tests Microsoft's Visual Studio Team System for Database Professionals includes testing tools for SQL server.

When TDD approach is applied, we write test cases to validate each structural or functional change one at a time (mainly table by table and having validation checks on tables or other database objects) and then evolve the database schema to fulfill the test-specified functionality, from the very start of the project continually, incrementally and iteratively, throughout the development activity. Through this we can validate syntax, field type, field length and constraints on the table i.e. we are validating the data and database objects which interact with that data in the database. Such tests might specify following validations:

- Checking primary keys
- Checking foreign keys
- Checking unique key constraints
- Checking data-types
- Checking indexes
- Checking the joins to establish the relations
- Validating column default value rules
- Validating size rules
- Validating value existence rule
- No null constraints
- Checking auto-increment (seeding) attribute.

3.1 Testing Example:

Assume that we are building a banking application as depicted in figure 3.

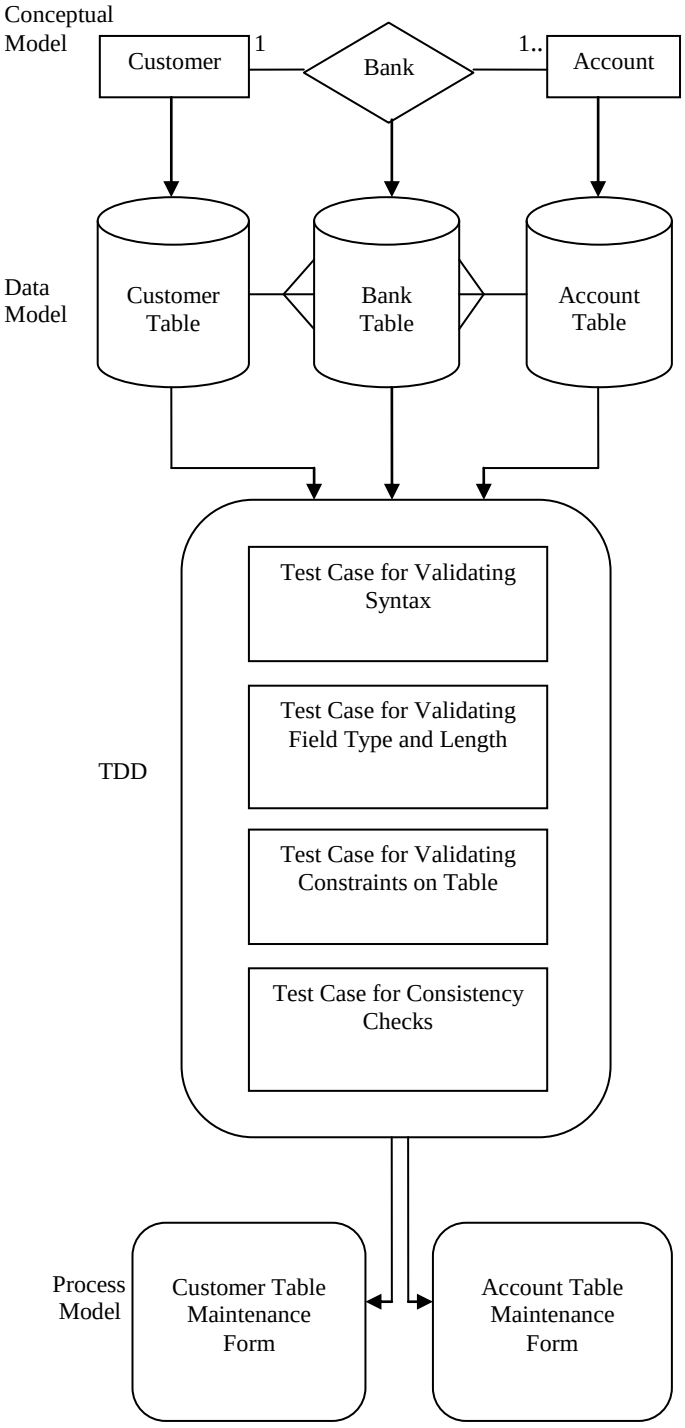

Fig 3: Transformation Model

Using the Conceptual Model (ER modeling) Bank relation is setup between customer and account entities. After which, data model for banking application is deduced which deals with the table creation for each entity.

Now the TDD's Internal (clear box) tests are to be created. We can use Microsoft's Visual Studio Team System for Database Professionals including testing tools for SQL Server. For example for the customer table, we can have the

test case to validate field type, length, some more constraints of the customer table as follows:

Table 1. Customer

	Cust ID	Cust First Name	Cust Last Name	Cust Add	Cust Tel	Bal
Field Type	Int	Char	Char	Char	Int	Int
Field Length	15	25	25	40	10	25
Other Constraint	Primary key, Not null, Auto increment	Uppercase, Not null, Indexing	upper case	Not null	Not null, Unique	null

Next step is doing the database refactoring i.e. structural refactoring. For example, assume the Customer table has a balance column, which depicts the total balance the customer has in his bank account, which should really have been a part of Account table. We have 2 different applications A & B which connects to the same database to do the CRUD operations on these tables. So iterative (TDD) development suggests we should move the column to the account table while also maintaining the integrity of the data. So we add a column (Balance) in Account table while keeping the column in the Customer table which will be phased out after the complete iteration. To keep the data consistent and in sync, we write data object – a trigger in our case. The trigger will update the value of balance in the new table Account when we update primary (Customer) table. Now we modify our applications one at a time (other application will continue to work since the existing rules and semantics did not change in the Customer table) to reflect new changes. Hence the database design will be improved while maintaining behavioral and informational semantics and consistency in the database.

Here TDD will have test cased, for the first iteration, for running a stored procedure to update current balance for a customer and then to display the final balance and since we have a trigger to keep the tables in sync, mainly the columns, after the update both columns will reflect the same value and our test case would always pass as behavior of procedure has not changed which effectively means, our data persistence, retrieval and execution engines have not changed. In addition we also write test cases to ensure negative test cases fail and boundary condition test run successfully.

Another example when we want to preserve the informational semantics so that change does not affect the information of the customer could be when we change the value of the Customer Phone column of customer table from one format" (011)25-678" to another"01125678". Here TDD will have test cases to validate the display and storage of data before and after the conversion of data type of the column from integer to Varchar. Here informational semantics have been preserved and change to database has improved the design. This proves the effectiveness of TDD and Agile development approach in the context of the database.

4. CONCLUSION

In this paper our main focus was on the application of TDD to the focal entity prototyping approach i.e. through this we can validate syntax, field type, field length and constraints on the table and can maintain the integrity and consistency in the database. In other words we are validating the data contained within the database. Through this, confidence in the database development can be gained by developer. For any new system, we can incrementally deliver processing capabilities, based on the database table processing, into the hands of the customer, then feedback from the customer about the changes, validations and additions in the table commences well before the final completion of the system. We are considering not only the evolutionary development of the database schema, but also the changes and refactoring which are verified and validated iteratively. The impact of the agile and TDD implementation in the database development will be such that single iteration can produce a well defined, tested and good designed increment to the database schema mainly table by table and can detect defects earlier in life cycle of focal entity prototyping.

Hence, similar to the agile application developers who take quality-driven, TDD approach to software development, the same approach can also be applied by agile database developers.

5. REFERENCES

[1] Juha Koskela Software configuration management in Agile methods http://www.vtt.fi/inf/pdf/publications/2003/P514.pdf

[2] Roy Morien, 2005"Agile Development of the database: Focal Entity Prototyping Approach" Proceedings of the agile development conference (ADC) IEEE Computer Society.

[3] Scott, W. Ambler, "Introduction to Test Driven Design (TDD)"Available: http://www.agiledata.org/essays/tdd.html

[4] Ambler, Scott (2003), Agile Database Techniques, Wiley Publishing, 2003.

[5] Ambler, Scott (2003a), at http://www.agiledata.org/essays/impedanceMismatch.html

[6] Ambler, Scott, (2004), "Agile Data Modelling", Software Development Magazine, July, August, September.

[7] Fowler, Martin (2003) at http://www.martinfowler.com/articles/evodb.htsml.

[8] The Process of Database Refactoring by Scott W. Ambler

[9] A Roadmap for Regression Testing Relational Databases by Scott W. Ambler

[10] Morien, Roy (1992), "Prototyping Large on- Line Systems: Using a concept of a Focal Entity for Task Identification", Proceedings of the Third Australian Conference on Information Systems, Wollongong.

[11] Morien, Roy & Cant. R (1994), "Specification With Prototypes: Two Case Studies", Proceedings of the 5th Australian Conference on Information Systems, Monash University.

[12] Harriman, Alan, P. Hodgets and M. Leo (2004), "Emergent Database Design: Liberating Database Development with Agile Practices", Agile Development Conference, Salt Lake City, Utah.