Future Internet **2011**, *3*, 281-297; doi:10.3390/fi3040281

OPEN ACCESS

future internet

ISSN 1999-5903

www.mdpi.com/journal/futureinternet

Article

Test Driven Development: Advancing Knowledge by Conjecture and Confirmation

David Parsons *, Ramesh Lal and Manfred Lange

Institute of Information and Mathematical Sciences, Albany Campus, Massey University, Private Bag 102-904 North Shore Mail Centre, Auckland, New Zealand; E-Mails: lal_ramesh@hotmail.com (R.L.); manfredmlange@gmail.com (M.L.)

* Author to whom correspondence should be addressed; E-Mail: d.p.parsons@massey.ac.nz; Tel.: +64-0-9-414-0800 (ext. 9138); Fax: +64-0-9-441-8181.

Received: 17 October 2011; in revised form: 6 December 2011 / Accepted: 7 December 2011 / Published: 14 December 2011

Abstract: Test Driven Development (TDD) is a critical agile software development practice that supports innovation in short development cycles. However, TDD is one of the most challenging agile practices to adopt because it requires changes to work practices and skill sets. It is therefore important to gain an understanding of TDD through the experiences of those who have successfully adopted this practice. We collaborated with an agile team to provide this experience report on their adoption of TDD, using observations and interviews within the product development environment. This article highlights a number of practices that underlie successful development with TDD. To provide a theoretical perspective that can help to explain how TDD supports a positive philosophy of software development, we have revised Northover *et al.*'s conceptual framework, which is based on a four stage model of agile development, to reinterpret Popper's theory of conjecture and falsification in the context of agile testing strategies. As a result of our findings, we propose an analytical model for TDD in agile software development which provides a theoretical basis for further investigations into the role of TDD and related practices.

Keywords: test driven development; agile methods; testing strategies; pair programming; design patterns; software architecture

1. Introduction–The Emergence of Agile Testing Practices

Agile methods emerged following a chronic software crisis that threatened the computer industry in the 1990s [1]. Proposed solutions at that time tended towards an ideal of software mass production, enabled by the total formalization of requirements. In practice, however, an alternative world view was emerging from the lessons learnt from lean production in the manufacturing sector. In this new development paradigm, the key features were teamwork, communication and simultaneous development, with multi skilled engineers gaining experience throughout the development process [2]. The concept of simultaneous development was a significant departure from the traditional sequential process of engineering. Different parts of the product could be developed at the same time, rather than in a waterfall style sequence. This required a number of supporting practices. Engineers from different parts of the production process had to communicate with and understand each other, and define simple interfaces between their own aspects of the product and others that were being developed at the same time. The change in worldview that lean production introduced was soon to create a fundamental shift in attitude in the software industry too. Instead of the *clean room*, where solutions supposedly emerged perfectly the first time, a more pragmatic concept of emergent design became the philosophy of agile software development. However, this emergent design approach relies heavily on testing practices, which can inform, certify and provide confidence as the design evolves. A testing practice that is the focus of this paper is Test Driven Development (TDD), which has a particular philosophical approach, explored in the context our experience report.

1.1. Test Driven Development (TDD)

Agile approaches to software development are based on the understanding that software requirements are dynamic, where they are driven by market forces [3–5]. An agile approach incorporates shared ideals of various stakeholders, and a philosophy of regularly providing customers with product features in short time-frames [6]. This frequent and regular feature delivery is achieved through a team based approach [7] and by having automated development, testing and release processes [8].

In many agile development teams, the testing component is based on the philosophy and techniques of TDD, which is an evolutionary approach to development, involving implementing a test before implementing and refactoring the code [9,10]. These tests should be small, address a small atomic piece of functionality and be executable by a test tool. Tests written this way can also be interpreted as specifications or requirements that the system under test has to meet. There are also tools, (such as Fitnesse), that allow expressing business requirements in such a way that they can be used as part of automated testing. These requirements are also referred to as executable specifications.

TDD supports the short development cycles of agile methods while ensuring quality requirements are being met. TDD was first used for software development by NASA in the 1960s, but in the context of structured methods with a single development cycle the practice was infrequently used [11]. It was not until the late 1990s that TDD gained wide acceptance as a common practice within agile methods, particularly eXtreme Programming (XP) [12].

1.2. Benefits of TDD

The main benefit claimed for TDD is an improvement in product quality [13]. The fine granularity of writing tests before coding provides continuous feedback for the developer [14], while the collection of automated tests is a valuable asset for regression testing [15]. TDD provides a greater predictability of development performance, helping to estimate project cost [16]. It achieves this by substantially reducing the uncertainty about unwanted side effects. These are avoided by executing an ever increasing test suite that also increases test coverage. As a result the schedule risk is substantially reduced and buffers included in quotes can also be significantly reduced. On average project costs are reduced as the number of bugs is reduced, the time for testing is reduced and the effort for maintenance is reduced, as TDD combined with merciless refactoring keeps the amount of source code to a minimum while maintaining a simple system architecture and design. TDD also supports changes to the product driven by business needs [17], allowing development to be driven by the requirements, rather than losing the requirements perspective as the development progresses [18]. In extreme cases this can lead to delivering each feature in isolation.

1.3. Success Factors

The support of customers and domain experts is critical for successful use of TDD, to determine and constrain a relevant set of use cases and user stories [19]. A user story is an informal statement of a user requirement that serves as a starting point for a conversation between stakeholders and developers, and the basis for acceptance tests [20]. In addition, an automated unit testing environment is required. Automating acceptance testing, including installation and upgrade testing, has also become a common practice for agile development teams [21]. Tool support for these two types of automated testing is widely available, for example the xUnit family of unit testing tools, tools that enable tests across multiple layers of a system (such as Selenium or White) and the FIT acceptance test tool [22–25].

1.4. Integration and Build Systems

TDD also requires continuous integration and build systems [26] to fully leverage its value. Integration systems enable software engineers to submit their latest changes to their source code into a central source control system. A build system with access to all the latest code regularly runs all the unit tests [27]. These two systems together enable implemented features to be tested and packaged with others and errors to be detected at a very early stage of development [28,29]. In combination with thought-through source code branch management these various test-related practices enable agile development to achieve its goal of delivering working software in short development cycles [30].

In summary, then, TDD is an agile practice that provides a number of benefits in terms of code quality and customer responsiveness, but requires the commitment of stakeholders and tool support for automated testing, integration, building and releasing.

1.5. Test Driven Development as Conjecture and Confirmation

Test Driven Development (TDD) shifts the quality assurance process to the fore of software development, requiring test cases to be written before implementing the proposed solution. TDD makes it possible to have short development cycles within a shorter overall time frame, encompassing various tests and quality assurance activities in each iteration. In contrast, traditional engineering methods driven just by a single development and release cycle have the testing and quality assurance activities after implementation.

An understanding of testing activities using both agile and traditional development methods can be provided through a reinterpretation of Popper's theory on *conjecture* and *falsification* [31]. From this perspective, traditional methods of software testing have followed a philosophy of conjecture and falsification in the advancement of knowledge. As a result of tests and quality assurance being backend activities within traditional methods, the proposed solution to a programming problem is the conjecture, and the *post hoc* tests are (potential) falsification. In contrast, Test Driven Development (TDD) shifts the process so that the test cases are written before implementing the proposed solution, so that the tests themselves become the conjecture. This has the effect of redirecting the emphasis from negative *falsification* to positive *confirmation*. This suggests that the tests in a test driven approach are qualitatively different than those written for *post hoc* testing. From a broader perspective, this provides us with a new interpretation of how knowledge advances, by describing tests directly from requirements, and systems directly from tests. This theoretical proposition of positive *confirmation* suggests that in practice the related and complementary practices of TDD must provide certification and confidence in the agile product development environment such that regular and frequently delivered features will meet market expectations.

1.6. Introduction to the Study

In this article we explore the experiences of an agile development team, and investigate how TDD and its supporting practices are applied within a well-defined agile product development environment. Our study seeks to identify the contributions to software quality of the test driven approach in advancing knowledge by confirmation rather than falsification. More broadly, it seeks to explain how the philosophy of conjecture and confirmation underlies successful agile strategies that combine a number of complementary practices. We have adapted Northover *et al.*'s [31] four stage model of how knowledge advances in an agile software development process to guide our investigation.

This experience report has been developed through close collaboration with the engineering manager of an agile team (who has also co-authored this report), through observations of TDD practice as it was carried out in his team's agile production lab and by conducting interviews with the engineers of his team to gather an in-depth understanding of their TDD practice.

In the remainder of this paper we consider a philosophical perspective on TDD, the drivers of the emergence of agile methods, and the historical precedents for early testing, including the relationship between tests and requirements. We then reflect on the experiences of an agile team using TDD, identifying illustrative cases where we can apply our adapted four stage model. We summarise our findings by illustrating the relationships between testing and associated practices as part of a cycle of conjecture and confirmation in agile development. We conclude with a summary and suggestions for future work.

2. Falsification or Confirmation? Agile Testing Philosophies

It is common to address the evaluation of software development practices from a technical perspective. Similarly, evaluation of quality, too, is typically reduced to a quest for empirical data. Hence, applying concrete metrics (such as numbers of bugs reported and how these numbers develop over time) to code quality is a common software engineering practice. The act of creating software is, however, also open to qualitative interpretations. For example, the process of refactoring [32], once it gets beyond the basic rules, does not offer simplistic, context free solutions. Knowing where and why to apply apparently contradictory refactoring such as 'replace inheritance with delegation' and 'replace delegation with inheritance' requires a layer of judgment that goes beyond numerical assessment. Whilst there are a number of simple heuristics that can support refactoring, such as the number of lines of code, or the number of parameters, ultimately a successful refactoring is one that reduces the complexity of any future reworking process. However, the point at which refactoring begins to reduce, rather than enhance, maintainability is something that cannot be objectively measured. Thus, while we might seek to evaluate the outcomes from a software development process using quantitative measures, evaluation of the process itself requires us to also consider qualitative measures. In this article we focus on the use of the TDD practice within an agile team, and attempt to contextualize its role within a larger agile strategy using a qualitative perspective based on a philosophical model.

2.1. A Four Stage Model of Agile Development

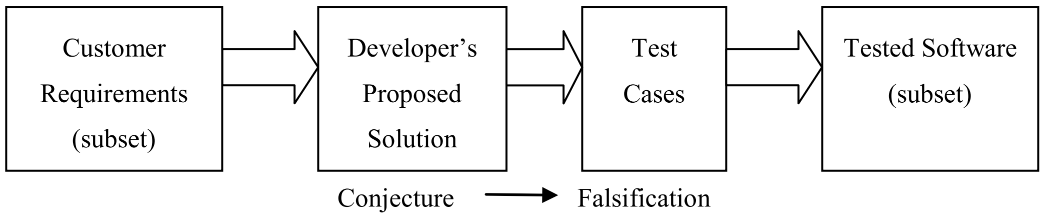
In the context of considering agile testing strategies, it is interesting to begin with Northover *et al.*'s attempt to apply Popper's critical rationalism to the 'small-scale, piecemeal methodological approach of Agile methodologies' [31]. Citing Magee, Northover *et al.* present a representation of Popper's four stage model of how knowledge advances (adapted here for clarity).

Initial Problem → Trial Solution → Error Elimination → Resulting Solution (with new problems)

This model assumes that falsification is the critical method of scientific discovery, that is, testing hypotheses in a rigorous attempt to prove them false.

Northover *et al.* map this four stage model of how knowledge advances to the agile software development process (Figure 1). This relates to Popper's concepts of *conjecture* and *falsification*, so in this model we could regard the proposed solution as a conjecture, and the test cases as (potential) falsification. The sequence of this model assumes that the proposed solution is constructed prior to the test cases, a traditional testing strategy.

Figure 1. Northover *et al.*'s 4 stage model of agile development.



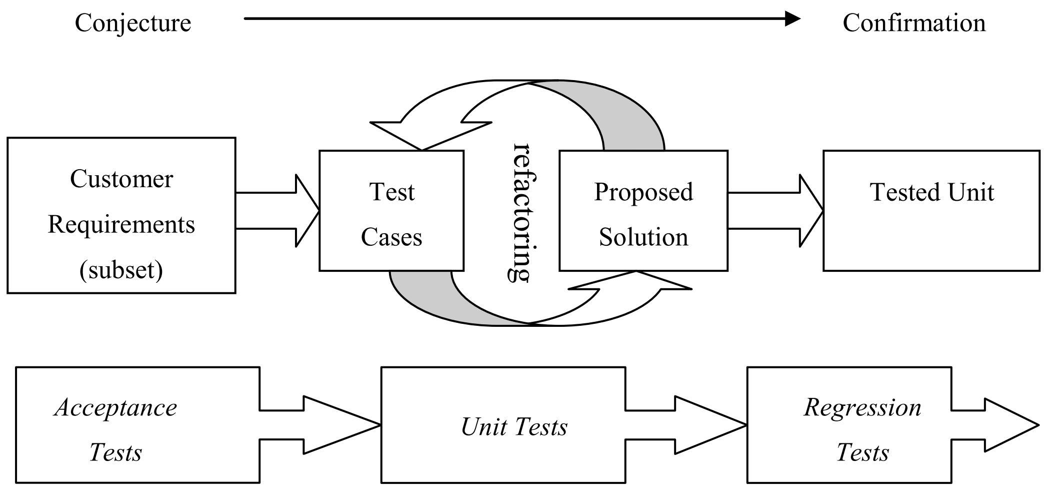
In a traditional unit testing approach, exemplified perhaps by the support provided for *post hoc* unit tests by many integrated development environments (where test cases can be generated from units under test), a predefined class or interface, with a given set of methods, is given an associated unit test (a practice that is the antithesis of TDD). The programmers are required to consider the existing interface (implemented code) of the class and decide how these are to be tested. Preconditions and postconditions, the contract under which the unit under test operates, are implied rather than explicit. In Northover *et al.*'s model, the 'conjecture' is the pre-existing design as embodied in the units under test. The programmers writing unit tests are limited in terms of role to 'falsification', in the sense that they can only attempt to find flaws the unit of code may already contain. Hence they are not in a position to make a positive contribution to the design. Proponents of TDD argue that this approach to testing is not only less interesting to the programmer, but prevents the activities of unit testing from contributing to the design of the system [9,10].

This model may be appropriate to some testing methods in agile development. However it does not appear to be an appropriate model for TDD. Therefore, we propose a revised model, incorporating the philosophy of TDD. This model provides a paradigm shift that takes us away from conjecture and falsification, to a more positive approach, to conjecture and confirmation. Hence the conjecture must be embodied in tests rather than in the proposed solution. In the next section we will outline the proposed model.

2.2. A Test Driven Model of Agile Development

In contrast to the *post hoc* testing model described above, TDD subverts this process such that the test cases come (or, at least, begin) before the proposed solution, so that the tests themselves, rather than the units under test, become the conjecture. They also become the embodiment of requirements. Hence they are no longer just tests, but are also part of specifications. This has the effect of redirecting the emphasis from negative *falsification* to positive *confirmation*. In other words, *post hoc* testing emphasizes the negative desire to break the proposed solution, whereas TDD emphasizes the positive desire to implement a solution that passes the test. This supports the contention that the quality of tests in a test driven approach is better than the tests in a *post hoc* testing approach. In addition, this provides us with a new interpretation of how knowledge advances, by describing tests directly from requirements. This step can be seen as literal if the requirements are expressed as customer acceptance tests, and embodied in tools such as FitNesse. In addition, the process of confirmation is global if a continuous integration tool is used, enabling regression testing to be part of every build. The overall effect is that all the tests are embodied in the whole process, beginning with customer requirements. Figure 2 presents a test driven model of the development of a unit of code, driven from customer requirements (expressed as acceptance tests), iteratively developed as a set of test cases (embodied in unit tests) driving a solution (embodied in a tested unit of code). This code is then repeatedly revisited in regression tests. This adapted four stage model (Figure 2) moves from conjecture to confirmation, underpinned by test processes at different levels of abstraction. However it should be noted that this iterative development is presented for a single unit of code, but each iteration would include many of these sub processes.

Figure 2. Test driven model of the development of a unit of code.



2.3. Post Hoc *Testing and Test Driven Development by Analogy*

Given that we propose a different perspective on the nature of testing in agile software development, it is appropriate to consider why confirmation might be preferable to falsification in a testing strategy. There are a number of arguments for the benefits of TDD over more traditional *post hoc* unit testing that may be analyzed in this light. TDD encourages formalizing the component interfaces at the design level prior to coding. It also ensures that every feature is tested from at least one perspective, while supporting a refactoring process that maintains the quality of code as it evolves, thus avoiding entropy, where the original design gradually loses its coherence as new requirements are incorporated.

Some authors have used historical analogy to throw light upon contemporary software practices. Love [33], for example, used the sinking of a 17th century ship (the Vasa) as an analogy of failed design, and applied it to a number of aspects of software development. One of his examples relates to testing the Vasa's stability after it was built. During these *post hoc* tests, it was discovered that the stability was very poor, but by then it was too late to fix this problem effectively without completely rebuilding the ship. For the purposes of expediency the ship was deemed to have 'passed the test' so that it could be delivered to the customer. Needless to say, the Vasa later capsized, not an uncommon occurrence in the shipbuilding industry prior to the late nineteenth century. The essence of this analogy is that design by trial and error can lead to catastrophic failure, if realistic testing takes place too late in the development cycle. Such analogies suggest that *post hoc* testing is of limited value, as it may be too late to remedy any underlying design flaws. The solution to this has long been recognized as early testing; following the naval architecture analogy, Froude's introduction of empirical design testing for naval ships in the 19th century using models laid the foundations for current tank testing practice. The key advance was not, however, the use of models, which shipbuilders had used for centuries. It was the application of rigorous and accurate testing tools to these models, early in the design process, that made the difference. The lesson we can draw from such examples is that the emergent design required of an agile software development process needs suitable proactive (rather than reactive) testing tools, used in a systematic way, to be successful. In addition this emergent design embodies both functional and non-functional requirements, both of which need constant testing, for example performance testing cannot be isolated to single points in the development process.

3. The Experience Report

This article is based on an experience report written by two independent researchers and a former member of the software development team being discussed. The case organization used in this study (which we will call NZD) is an international software vendor for customer management in the energy sector. Repeated interviews were carried out over an 18 month period with the product manager, senior and principal engineers, and the engineering manager. The team was also observed in their software product lab, and follow up interviews were carried out to validate our conclusions.

3.1. Refactoring Teams, Roles and Quality Assurance

In 2005, NZD refactored their software development process by creating a new agile team, alongside several other autonomous development teams, with the longer term view of improving its development process company wide. The start-up agile team had in total 24 individuals, comprising 18 developers, three performance experts, two product analysts and one usability engineer. The team was headed by an engineering manager who was empowered by the senior management to do whatever was necessary to ensure the success of the team. The developers and performance experts were all given the new job title of *software engineer* and assigned to undertake all tasks relating to product development, not just coding. Within 3 years, as a result of its success, this agile team had almost doubled in size.

Development work is allocated to small sub-teams of between four and six engineers. Sub-teams are created based on how a project can be split into independent parts, enabling each sub-team to implement different parts of a project (simultaneous development). Responsibility for system design falls on the whole team, driven by adoption of TDD as a standard practice, rather than on only a few individuals.

Central to this team change were changes to the quality assurance function. Previously, NZD had a separate dedicated quality assurance team for undertaking manual testing. The quality assurance team (about 10 testers) provided certification for the functionality, usability, reliability, performance, and scalability of their products. With adoption of TDD, the separate quality assurance team was removed; instead, TDD is used by software engineers during code implementation, and the systems (acceptance) tests are used prior to any market release of new features of their product.

3.2. Stories as Acceptance Tests

As TDD became standard practice, product analysts began to write tests for each user story on the back of each story card, requiring engineers to implement them as acceptance tests. Their requirement for a story is that it must be discrete for implementation, that it stands alone, is testable, can be used as a function and provides business value. What is written for a story is only what fits on the front of a card when writing with a marker, a trigger for later conversations about the details of the story once its development has started. The product analyst also writes additional details such as algorithms, field names and other validation information including story tests on the back. Story tests provide worked examples for the engineers.

An acceptance test requires each story to be in its smallest form since small stories with their tests are much easier for engineers to implement. The team collectively decided to have a rule for a story to be no more than 3 points, where 1 point is equivalent to 8 implementation hours of the total hours available for each sprint. As a result, the team adapted to having one week sprint cycles and also began to implement more stories in a sprint.

3.3. Emergent Design in Tests

NZD believes that big up front design leads to preconceptions and false assumptions. Software engineers reflecting on their prior practice found that often detailed designs created up front proved hard to implement in practice. The longer it took for coding to start, the longer it was before the real problems started to get solved. In contrast, problems and risks are exposed early in test driven agile projects, giving management the opportunity to respond by adjusting scope and reorganizing teams.

Of course adopting an agile approach does not mean that there is no design up front at NZD. A certain amount of modeling always takes place before implementation and a skeleton service-oriented architecture consisting of 3 layers was used as well. However, regardless of the formality or otherwise of the modeling medium, a central tenet of agile modeling is 'prove it with code', *i.e.*, a putative design is not proven until it is encoded, and the final design is expected to emerge through the iterative evolution of that initial model. Thus any up front design is necessarily an initial proposal for one aspect of the system, which will only be proven once it becomes embodied in code.

For the code to embody a viable, robust and flexible emergent design, TDD is a key technique. The agile team at NZD became aware that if there is no big design up front, and no detailed up front requirements, then there is a large potential vacuum that cannot just be filled with code, otherwise the lack of process will lead to chaos. Something methodical is needed to replace the concepts of upfront design and specification. For this reason, the agile team at NZD adopted TDD as a critical agile development practice to support emergent design.

The software engineers understand that unit testing alone does not promote good design, because writing tests after the code does not enable them to drive the emergent design through those tests, while TDD enables them to validate if the design requirements have been well thought out and even to discover new requirements. The tests written by the engineers using TDD are used to iteratively define and refine both the requirements of the units under test and the tests themselves. Unit testing tools enforce a set of preconditions for a set of tests; TDD adds a layer of test refactoring that enables these preconditions to be more thoughtfully evolved. Equally important is the use of an automated acceptance testing framework that enables TDD to be used at the customer (or customer proxy) level. Customers work at a level of abstraction that tests the emergent design from the outside in, complementing TDD at the unit level, testing from the inside out.

As one measure of the difference between TDD and *post hoc* unit testing, two sets of tests were measured using the Jumble mutation testing tool. Mutation tests were incorporated by NZD to improve their TDD practice, since mutation tests can provide higher statement coverage [34]. Jumble measures the quality of unit tests from 0% (worthless) to 100% (angelic). When measured by NZD, their unit tests developed as part of TDD scored 95%, whereas a comparative set of earlier tests that had been developed in a *post hoc* manner scored between 10% and 60%. Another perhaps more qualitative measure of the value of these tests is that they are regarded as a valuable part of their intellectual property by the organization. The tests are an investment that represent an incarnation of requirements, and therefore are regarded as equal to the actual software product in terms of their value. In this sense we can regard the tests themselves as the embodiment of emergent design.

3.4. Conjecture and Confirmation in Supporting Techniques

While this article focuses on TDD, this technique cannot be used entirely in isolation. Two supporting techniques for TDD are pair programming and the used of design patterns. In this section we briefly describe how the process of conjecture and confirmation is also evident in these techniques when they are being used to support TDD.

Pair programming is an overarching practice for TDD, since its constant code review ensures that the tests are of high quality. High quality tests ensure high quality code. In the context of the theme of this paper, pair programming plays a control role in the 'conjecture first' approach, in that pairing requires one member of the pair to make a conjecture, subject to immediate confirmation by the other member. It is important to note here that the level of discourse between pairs focuses more on design than on the detail of code [35].

Design patterns [36] have had a high profile role in software design for a generation, but is their role one of conjecture, or confirmation? The presence of patterns in catalogues may imply that that they are some kind of conjecture, however the essence of these patterns is that they grow out of the experience of recognizing similar generic design problems across different applications and domains. The design approach at NZD is not based on the conscious application of patterns from a catalogue. Code developed using TDD will not de facto embody any design patterns but they will emerge (or indeed, be removed) if and when required. Similarly, refactoring to and from design patterns can only be effective where there is a pre-existing suite of tests, enabling a robust refactoring effort. From this perspective we can see that design patterns fit into a role of confirmation; they provide confirmation that an emerging design solution conforms to best practice.

One example from the NZD case study that helps to elucidate this difference between patterns as conjectures and patterns as confirmation, was a project in which Data Transfer Objects (DTOs) [37] were introduced into a system design at an early stage, then removed, then reintroduced, as design understanding deepened within the team. Initially, DTOs were introduced based on an assumption that these were a 'good' design pattern for transferring data between the UI and business logic layers of the application. This is an example of an antipattern; applying a given design pattern as conjecture. As the understanding, experience and knowledge of the project grew, the software engineers identified that these DTOs were very similar to the domain objects that were already being created in the business logic layer. Therefore, DTOs were refactored out of the design. However, further down the timeline of this project, as the local development experience of this agile team was further enhanced, it became clearer to the engineers that DTOs did after all have a role in the design.

Thus DTOs were brought back into the design for the 'right' reasons (a clear understanding, and confirmation, of their design contribution) as opposed to the 'wrong' reasons (applying a design pattern as a conjecture because it was seen as the way something is normally done). The important feature of this process was that it was made possible through test driven development, and the ability to refactor the design safely. By experiencing the different design options, the engineers gained deeper understanding than if they had simply picked a design and had to live with it. Throughout this process, the tests drove the design rather than *vice versa*.

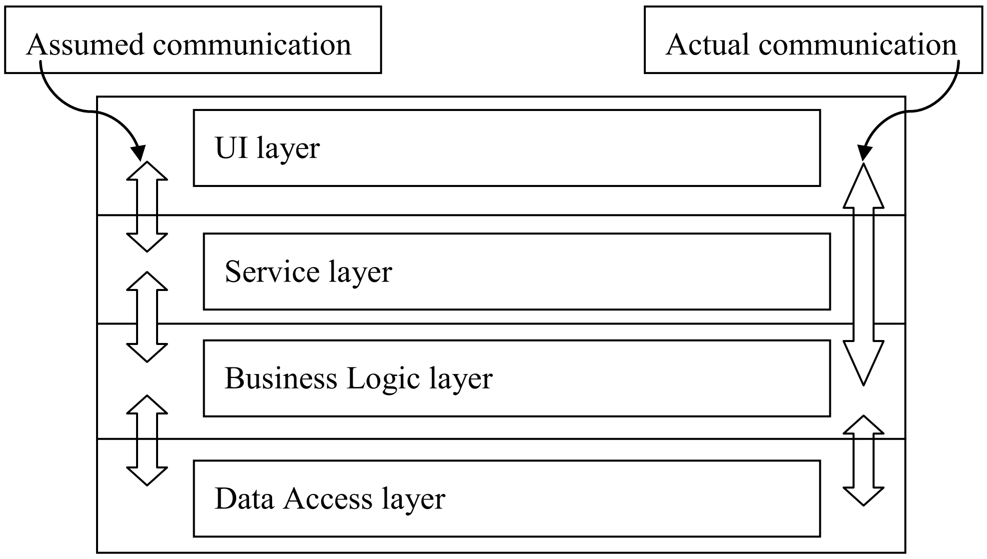
Refactoring changes the design emphasis from flexibility to simplicity [32]. Upfront design is still important, but its emphasis changes. Design is malleable, and is not complete upfront. This may seem to run counter to the philosophy of design patterns, which is that we should be designing for change. However, these views are not opposed. Many patterns give us tools to design for change when the need for that change is thrust upon us. Our case study identified the Strategy pattern [36] as being one that frequently emerges during refactoring because it enables, among other things, customization based on different customer requirements.

3.5. Testing Layered Architectures

While design emerges in an agile project, architecture grows. The agile software engineering process at NZD begins with an architectural vision based on a layered architecture [38]. This implies that there is a minimum level of conjecture required in creating the initial architecture. Nevertheless this is subject to confirmation in practice, and may well evolve over time. In a 'big design up front' approach, all the layers of the final architecture are planned during the initial design. However in an agile project, these architectural layers are more likely to emerge in stages. Initial layers may be in the skeleton architecture that is built early in development, but others may be added later as the code is refactored. From this agile team's perspective, the architecture planning is an up-front activity. However, the team firmly believes that the architectural design must not be more than a bare skeleton, and must also be presentable on a single page, as opposed to a more traditional document of several hundred pages. Agile development is mostly driven by emerging architecture as highlighted by this agile team's philosophy for minimal architectural planning and allowing the architecture to emerge as development proceeds from one sprint (short development) cycle to the next sprint cycle. One NZD project started with a single layer architecture, which consisted of only a single screen with a single test. This architecture expanded to two layers the following day, and then to three layers a week later. This example of emerging architecture from their agile development was with a small project involving a single sub-team on a small-sized project. However, the number of initial architecture layers decided by the sub-teams to start their first sprint cycle in projects may vary. For example, this agile team also had two sub-teams sharing different parts of a larger-sized project, where they started their first sprint cycle with three layers. In this project, starting with a three layer approach was a collective decision, made involving both sub-teams after an initial architecture plan that had five layers. After this initial architecture comes the emergent design, which evolves in a less linear fashion. In the spirit of refactoring, design features may be added only to be removed at a later date (as in the DTO example).

In NZD's agile development environment, the initial nature of the skeleton architecture is dependent upon a number of factors, not just on the size of the project or the team. The level of local development experience in sub-teams and the ability to draw upon their tacit knowledge acquired through the individual and collective learning from the past architectural decisions made in their previous projects, are the other important factors. The engineering manager reflected that in an earlier project they had started with a four-layer architecture (Figure 3). They had several sub-teams on this project. Therefore, they collectively agreed the need to do in-depth architecture planning from the outset. However, after several sprint cycles in the project timeline the engineers discovered that they could not find any reasonable code to be put into the service layer, which in practice seemed to only have the role of forwarding messages and providing pathways for interaction between its surrounding layers. Hence, the sub-teams collectively made the decision to remove the service layer, knowing that they could always re-introduce it if it was required at a later stage. However, even after three years, this layer was still not required for the features of that product. Once again, early conjecture about design, rather than about tests, led to inappropriate decisions.

Figure 3. Assumed and actual communication between architectural layers.



The significance of layered architectures to TDD is that agile development teams require different types of tests, and testing tools, to address different aspects of a layered architecture. Unit tests are different from integration tests, so to drive design through unit tests it is necessary to isolate the layers from one another. Integration tests are equally important; they traverse multiple layers. Thus in an overall test driven strategy, the software engineers have to apply different tools and techniques for TDD when working on different aspects of the system. For example, with this agile team at NZD, Selenium is used to test the UI layer, FitNesse to test the service layer and JUnit to test the business layer.

3.6. Test Automation and Roles

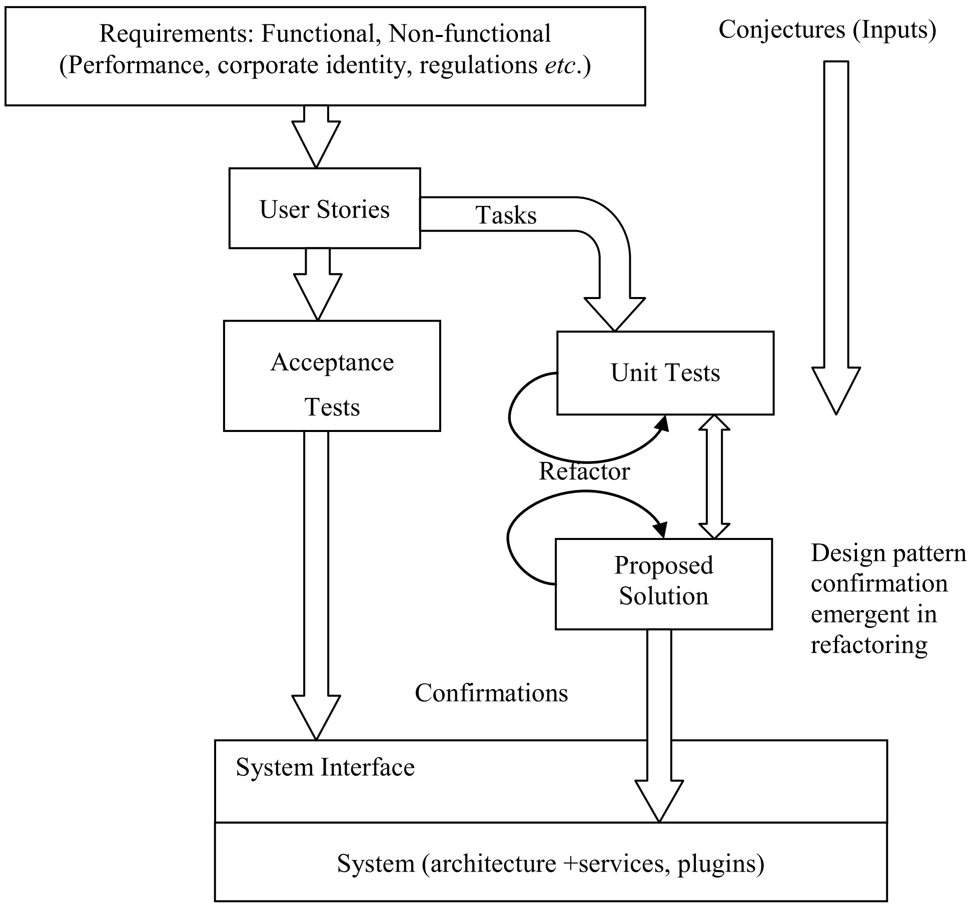
An important aspect of agile development is test automation. At NZD, this automation enables in-depth, thorough and wide test coverage and almost instant feedback on implementation during the agile team's sprint cycles. With their emergent architecture and design approach, test automation is a critical means to quickly certify and to gain confidence by the agile team that they are on the right path and the new features they have implemented have properly integrated with the product. Hence, the overall architecture of the system, as well as the fine grained units of code, with its various layers of testing need to be automated in a managed process. To meet this need, the agile team adapted to include a Build Master role that monitors all the failed unit tests and ensures that they are immediately fixed. This Build Master role at NZD was proposed by a senior engineer since the agile team was experiencing frequent broken builds in a sprint as a large number of automated tests were being accumulated. The broken builds required immediate fixing for the effective functioning of their continuous integration system. Hence, the team collectively agreed to have a pair of engineers to volunteer during sprint planning meetings to work as Build Masters for a sprint. Previously, no one tracked unit tests during sprints. Now, with their extremely large number of test cases that run constantly with their build process, this role ensures that bugs are not allowed to accumulate. In addition, an agile tester role was adopted. The idea to have agile testers was proposed by the principal engineer and was accepted by the team including their engineering manager. The team had discovered that the engineers were limiting their thinking for writing the unit tests to story cards only and the product quality was being compromised. They needed this role to provide development support so that a wider view is considered by the engineers on issues such as performance and usability. Engineers now have support to write unit tests. The idea to have this role was also supported by the product analysts since the agile tester would also provide them with support for writing acceptance tests. Throughout this process, the underlying need was to support conjecture through tests at multiple layers of the architecture, not just at the unit test level. Although small, tested components can be reliably assembled into larger, testable modules that support subsystems and layers it is still necessary to test the interfaces which establishes the interactions between the layers. A supporting approach to conjecture that can be taken is to reuse third party components that support abstraction and encapsulation. An example of this approach taken by this agile team is the use of an object relational mapping tool for the implementation of the data access layer. Using this mapping tool enables them to implement well-defined interfaces to that layer. In a similar manner, legacy features are specified by a set of interface tests, and then these are extracted independent services. From another perspective, testability is enhanced by an architecture that allows interoperability with plugins for integrating external services.

4. A Model of Test Driven Development

Figure 4 shows our model of how conjecture and confirmation play their roles in an agile process. This is an adapted model based on the concepts outlined in Figures 1 and 2 and our key findings on test driven development in an agile process from our experience report organization. Conjecture begins with inputs to the system; requirements and user stories. These conjectures translate into tests; unit tests for systems (including services and plugins), and acceptance tests for the system interface. An initial architecture is also created from conjecture. As the development teams iteratively move through the knowledge building process, confirmation is provided by test results, emergent patterns and the evolving architecture.

It is interesting to contrast this model with the well-known V-Model, which also operates at different levels of abstraction and categorizes various testing strategies against aspects of the development cycle. Of course the critical distinction between them is that the V model follows a sequential waterfall process, whereas the processes described here are agile and iterative.

Figure 4. Conjecture and confirmation in an agile process.


Requirements: Functional, Non-functional
(Performance, corporate identity, regulations *etc.*)
Conjectures (Inputs)
User Stories
Tasks
Acceptance Tests
Unit Tests
Refactor
Proposed Solution
Design pattern confirmation emergent in refactoring
Confirmations
System Interface
System (architecture +services, plugins)

5. Summary and Future Work

The experience reported in this paper provides a new perspective for the software development community on the underlying philosophies that motivate certain approaches to agile development. While this report on TDD is based on a single organization, its agile development experiences in growing and enhancing its highly innovative software product, which sells in a global marketplace, provides us with significant amount of rich data to help explain our model of advancing knowledge. In drawing together the concepts of TDD as a process of conjecture and confirmation, as elucidated by our study, we can characterize conjecture as being made up of several components. First, the architectural skeleton embodies conjecture about the overall structure of the systems to be built. Second, the user stories embody conjectures not only about the stories themselves but also their acceptance tests. Third, the unit tests are written before implementing the units under test. Each of these conjectures has its confirmation. The overall architecture is confirmed by its utility in practice, as validated by integration tests. The user stores are confirmed by acceptance tests, while the unit tests are confirmed by the units under tests in an iterative cycle of refactoring that may inject emergent patterns into the overall design as a confirmation of design quality.

This model of conjecture and confirmation may prove useful as a way of considering various emerging agile practice and tools. Various types of behavior driven development are becoming increasingly popular, while an increasing focus on the user experience brings in new challenges for test driven development. Embodying concepts of design in a test framework is a major challenge in, for example, designing UIs for mobile devices, or other interfaces driven by non-conventional means. The agile development experience, as outlined in this paper, confirms the value not only of a test driven approach but of an overarching philosophy that ensures each step in the design is based on confirmation, rather than falsification. Agile teams considering new tools or approaches might usefully take into account whether those tools or approaches are being adopted in ways that truly support the conjecture and confirmation process.

References

1. Gibbs, W. Software's Chronic Crisis. Available online: http://selab.csuohio.edu/~nsridhar/teaching/fall06/eec521/readings/Gibbs-scc.pdf. (access on 13 December 2011).
2. Womak, J.P.; Jones, D.T.; Roos, D. *The Machine That Changed the World*; Harper Perennial: New York, NY, USA, 1990.
3. Fowler, M. The New Methodology. 2002. Available online: http://www.martinfowler.com/articles/newMethodology.html (accessed on 19 January 2006).
4. Cockburn, A.; Highsmith, J. Agile software development: The people factor. *Computer* **2001**, *34*, 131–133.
5. Beck, K. Embracing change with extreme programming. *Computer* **1999**, *32*, 70–77.
6. Southwell, K. Agile Process Improvement. In *TickIT International*; BSI-DISC: London, UK, 2002; pp. 3–14. Available online: http://www.tickit.org/ti3q02.pdf (accessed on 7 December 2011).
7. Coram, M.; Bohner, S. The Impact of Agile Methods on Software Project Management. In *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, Greenbelt, MD, USA, 4–7 April 2005; pp. 363–370.
8. Holmes, A.; Kellogg, M. Automating Functional Tests Using Selenium. In *Proceedings of the Agile 2006 Conference*, Minneapolis, MN, USA, 23–28 July 2006, pp. 275–281.
9. Astels, D. *Test-Driven Development: A Practical Guide*; Prentice Hall: Upper Saddle River, NJ, USA, 2003.
10. Beck, K. *Test Driven Development: By Example*; Pearson: New York, NY, USA, 2003.
11. Larman, C.; Basili, V. Iterative and incremental developments. A brief history. *Computer* **2003**, *36*, 47–56.
12. Beck, K. Aim, fire [test-first coding]. *IEEE Softw.* **2001** *18*, 87–89.
13. Bhat, T.; Nagappan, N. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil, 21–22 September 2006; pp. 356–363.
14. Kaufmann, R.; Janzen, D. Implications of Test-Driven Development: A Pilot Study. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, USA, 26–30 October 2003; pp. 298–299.
15. Williams, L.; Maximilien, E.; Vouk, M. Test-Driven Development as a Defect-Reduction Practice. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Denver, CO, USA, 17–20 November 2003; p. 34.

16. Canfora, G.; Cimitile, A.; Garcia, F.; Piattini, M.; Visaggio, C.A. Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil 21–22 September 2006; pp. 364–371.
17. Sukkarieh, J.Z.; Kamal, J. Towards Agile and Test-Driven Development in NLP Applications. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, Boulder, CO, USA, 5 June 2009; pp. 42–44.
18. Park, S.S.; Maurer, F. The Benefits and Challenges of Executable Acceptance Testing. In *Proceedings of the 2008 International Workshop on Scrutinizing Agile Practices or Shoot-Out at the Agile Corral*, Leipzig, Germany, 10 May 2008; pp. 19–22.
19. Fraser, J.; Mattu, B. Test Driven Design Challenges for Faster Product Development. In *Proceedings of the 1st Annual Systems Conference*, Honolulu, HI, USA, 9–13 April 2007; pp. 1–5.
20. Williams, L. Agile software development methodologies and practices. *Adv. Comput.* **2010**, *80*, 1–44.
21. Martin, R.C. The test bus imperative: Architectures that support automated acceptance testing. *IEEE Software* **2005**, *22*, 65–67.
22. Hansson, C.; Dittrich, Y.; Gustafsson, B.; Zarnak, S. How agile are industrial software development practices? *J. Syst. Softw.* **2006** *79*, 1295–1311.
23. Alwardt, A.L.; Mikeska, N.; Pandorf, R.; Tarpley, P. A Lean Approach to Designing for Software Testability. In *Proceedings of the AUTOTESTCON*, Anaheim, CA, USA, 14–17 September 2009; pp. 178–183.
24. Do, H.; Rothermel, G.; Kinneer, A. Empirical Studies of Test Case Prioritization in a JUnit Testing Environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering Conference*, Saint-Malo, France, 2–5 November 2004; pp. 113–124.
25. Holmes, A.; Kellogg, M. Automating Functional Tests Using Selenium. In *Proceedings of the Agile 2006 Conference*, Minneapolis, MN, USA, 23–28 July 2006; pp. 275–281.
26. Bowyer, J.; Hughes, J. Assessing Undergraduate Experience of Continuous Integration and Test-Driven Development. In *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 20–28 May 2006; pp. 691–694.
27. Germain, R.; Robillard, P.N. Engineering-based processes and agile methodologies for software development: a comparative case study. *J. Syst. Softw.* **2005**, *75*, 17–27.
28. Rosenberg, D.; Stephens, M.; Collins-Cope, M. *Agile Development with ICONIX Process*; Apress: Berkeley, CA, USA, 2005.
29. Fruhling, A.; Vreede, G. Field experiences with extreme programming: Developing an emergency response system. *J. Manag. Inf. Syst.* **2006**, *22*, 39–68.
30. Hanssen, G.K.; Haugset, B. Automated Acceptance Testing Using Fit. In *Proceedings of the 42nd Hawaii International Conference on System Sciences Conference*, Big Island, HI, USA, 5–8 January 2009; pp. 1–8.
31. Northover, M.; Northover, A.; Gruner, S.; Kourie, D.; Boake, A. Agile Software Development: A Contemporary Philosophical Perspective. In *Proceedings of the Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing countries*, Port Elizabeth, South Africa, 2–3 October 2007; pp. 106–115.

32. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison Wesley Longman: White Plains, NY, USA, 1999.
33. Love, T. *Object Lessons: Lessons Learned in Object-Oriented Development Projects*; SIGS Books: New York, NY, USA, 1993.
34. Smith, B.; Williams, L. Should software testers use mutation analysis to augment a test set? *J. Syst. Softw.* **2009**, *82*, 1819–1832.
35. Canfora, G.; Cimitile, A.; Visaggio, C.A. Working in Pairs as a Means for Design Knowledge Building: An Empirical Study. Presented at *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, Bari, Italy, 24–26 June 2004.
36. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1995.
37. Fowler. M. *Patterns of Enterprise Application Architecture*; Addison-Wesley: MA, USA, 2002.
38. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern Oriented Software Architecture: A System of Patterns*; Wiley: Hoboken, NJ, USA, 1996.