This section summarizes the main results obtained from analyzing the interviews. It is structured to highlight the benefits and challenges of using TDD and their causes, along the dimensions of the high level benefit categories of code quality, application quality, and developer productivity.

1. Positive Perceptions

The experience in using a TDD approach as part of their software development process was reported by all five interviewees as generally positive to them personally. They also observed that the view of TDD throughout the development teams had moved from skepticism at the start of the project to general support as providing tangible benefits. One interviewee noted that there was never strong criticism of TDD practices during team retrospectives as evidence of this.

When asked what they thought the main benefit of TDD was the participants’ answers included:

• throughout the development process the engineers think more widely

• higher quality code-it does what it is supposed to do

• increased test coverage

• confidence in the quality of code delivered to the client

• more client and developer confidence that the code will do what we said it will do

These high level benefits show an emphasis on perceiving improvements to code and application quality, resulting in increased confidence in the application for developers and clients. When questioned further about benefits, the interviewees identified a range of what they believed to be beneficial outcomes related specifically to TDD practices. These are summarized in the next three sub-sections. The interviewees attributed these benefits to changes in practice, attitude and behavior as a result of following TDD, compared to their non-TDD experiences.

1. Code Quality

Participants were unanimous in their perception that the use of TDD improved the quality of code compared to their experiences with traditional TL software development. They claimed that the use of TDD encouraged the development of simple, clean and meaningful code, more so than TL encouraged these outcomes. There was the perception that the discipline of following TDD would naturally develop habits that lead to better code as part of developers’ everyday practice. The implication was that it was easier to be “lazy” and get away with developing messy or untested code using a TL approach. As one interviewee put it:

TDD helps developers towards simple designs; keeps things typically OO [Object Oriented] structured; pushes developers towards separated components.

TDD was also viewed as guarding against the pressure of management to deliver code more quickly and compromising code quality, since this was not an option with TDD. Participants identified several underlying factors that are consequences of following TDD practice, which they viewed as strongly contributing to the improvement of code quality. Interviewees perceived that a deeper understanding of the functionality required of a piece of code was a consequence of writing tests first. They stated that often, while trying to write a test for a piece of specified functionality (as part of a user story), they uncovered uncertainty in the meaning or scope of that functionality and sought clarification from the (proxy) client. They viewed this benefit of test writing as being

amplified when the test is written before the functional code because: (1) better understanding triggered by the testwriting did not have the potential to require changes to the

associated functional code, as it did with a TL approach; (2) there was no opportunity to defer (or omit) writing the test, in contrast to TL; (3) the extra effort invested into understanding what the functional code had to do before writing it, compared to TL, more often resulted in a clear idea of the objects and methods required in the functional code, making it easier to avoid cluttered code and have a simple design; (4) the tests and functional units tended to be smaller in scope than those written using a TL approach, making it easier (cognitively) to simplify the design, with fewer factors in the code to consider. Another major contributor to good code design that interviewees identified with TDD was developers’ increased confidence and willingness to put effort into improving the design of “perfectly good working code” through refactoring. This benefit was attributed to two main outcomes of TDD that are less “front of mind” in TL development: (1) refactoring is part of “how we do things” in TDD but was not emphasized to the same degree in the participants’ TL experiences (2) the set of automated tests, also inherent to TDD, increased developer confidence to refactor code or try out new ideas because they could immediately run a set of automated tests to see if they “broke” existing code. They contrasted this with the “if it isn’t broke, don’t touch it” mentality they tended to have with a TL approach. They claimed that this increased confidence to change code, together with the fact that the “chunks” of code tended to be smaller using TDD, increased their willingness to refactor the code to improve design during the implementation process, resulting in better code design compared to using TL practices. Participants also described the code written while using TDD practices as likely to be more “readable” and easier to maintain, compared to code written using TL. They saw this as also contributing to code quality. They explained that in writing the tests first they tended to use meaningful test, variable and class names, and that the use of meaningful names naturally carried over to the production code. Interviewees indicated that the associated tests also contributed to better understanding of code functionality. The interviewees observed that readability was further enhanced because fewer comments in the code were needed, since much of the information and semantics traditionally captured – or not – in comments were embodied in the associated tests. They also noted that the comments that were still required were more likely to be included (and were more likely to be useful), because so few were required. One interviewee makes this point as the following: There is no need to have a large bit of documentation outside of the code or inside it to describe the functionality itself, as the test class would give a good idea of what the relevant class should be doing.