Applying Test Driven Development to Embedded Software

James Grenning

Test Driven Development (TDD) is increasing in information technology applications and product development; however, it has not been widely applied in embedded software development. Embedded developers face many challenges. TDD can help overcome some of these challenges, but TDD has to be adapted for embedded systems development.

Test Driven Development Cycle, the Essence of TDD

The TDD workflow consists of the following steps [1]:

- Create a new test
- Do a Build, run all the tests, and see the new one fail.

(Build is the creation of a program's executable. In this context, we are building the executable that runs all the unit tests.)

- Write the code to make the test pass
- Do a Build, run all the tests, and see the new one pass
- Refactor to remove duplication.

(Refactoring is the activity whereby the structure of the code being worked on is changed without changing its behavior. This is done to clean up any messes that are made in the heat of battle trying to get something to work. For more information see http://www.c2.com/cgi/wiki?WikiPagesAboutRefactoring.)

- Repeat

Prior to executing this core cycle of TDD, the developer establishes the context for the work being done. When the developer decides to work on a specific module, that module has a set of responsibilities and behaviors that allow the module to provide what is needed to the embedded system. Modeling can set the context of the module being developed and is helpful for establishing this context. Most TDD practitioners follow an iterative cycle and do not provide highly detailed design models. The models are typically used to partition responsibilities, while leaving the details in the code.

An encompassing workflow for TDD consists of these activities:

- Identify the module to work on.
- Identify its collaborators.
- Brainstorm a set of tests this module must pass. (This is not an exhaustive list.)
- Choose a subset of the tests to explore the interface of the module.
- Incrementally apply the core TDD cycle to define the interface and its first client—add more interface tests as needed.
- Order the remaining tests such that more complex tests build on the successes of the earlier passing tests.
- Incrementally apply the core TDD cycle to those tests—add more tests as needed.
- Check for completeness.

TDD provides benefits to the developer and the development team, including improved predictability, repeatability, and reduced debugging time. Predictability is improved because TDD breaks programming into a series of small verifiable tests that can be used as a measure of progress. Each test is either passed or failed, giving an unambiguous indication of completeness.

Test repeatability is critical because the software will continue to evolve throughout the product development cycle and potentially in future products. Typical manual testing techniques rely on the developer to anticipate side effects and then manually test for those side effects. Automating tests once and running the tests with every change can save considerable time and relieve the developer of some of the burden of anticipating side effects. Also, because it is not practical to run manual regression tests with every code change, manual testing lengthens the time between defect injection and defect detection. With TDD's rapid feedback, on the other hand, an injected defect can be detected within moments of its introduction. The code can be rolled back to confirm that the last change actually caused the failure. With this form of repeatable cause-and-effect test-

1094-6969/07/$25.00©2007IEEE

ing, the source of the problem is usually evident and quickly corrected. This can greatly reduce debug time by eliminating many problems within seconds or minutes of their introduction and avoiding the bug reporting and repair overhead.

Unit Test Versus Acceptance Test

TDD can be applied as unit tests and as acceptance tests. Unit tests are tests that provide feedback to the developer about whether or not the code written does what it is expected to do. Most embedded development is done in the C or C++ languages. For C, there is no single language construct that specifies a unit. For applications written in C, a lot of care must be taken to make the C code modular. For our purposes, we will consider a C module to be made up of a C header file and its associated source file. The header contains function prototypes defining the interface to the C module. The source file would implement the functions and also use file scope declarations to hide the internal details of the module. For C++, a set of unit tests are written to fully exercise a class. In this article, I will refer to a module that includes both the concept of a C module just described and the C++ class. Unit tests are behavioral tests and attempt to fully exercise the module under test. Ideally, unit tests are run every few minutes with every code change. This can be a significant challenge when we have long compile and download times.

Each module has a suite of unit tests. As already mentioned, unit tests provide feedback to the programmer that the code works as expected. That is not all they do though. They also provide regression tests and relieve the programmer from having to repeatedly manually test the same code. This is very powerful as it helps to greatly reduce one of the software developer's biggest problems, side effect defects. The unit test suite for a module also provides an unambiguous detailed specification of the module under test.

In contrast, acceptance tests operate on integrated groups of modules to show that the software meets its requirements. Modules are bound together and various test scenarios are fed into them, demonstrating the system or subsystem behavior. Ideally, nonprogrammers write these tests in an application-specific test language. These tests become part of the engineering specifications. This article is mainly concerned with unit testing.

Embedded developers face many challenges. TDD can help overcome some of these challenges, but TDD has to be adapted for embedded systems development.

TDD Skepticism

People often react to TDD with skepticism. I know I did. On first encountering TDD, a developer in the insurance industry might say, "Yeah, that looks pretty good, but we're developing insurance software, which with all the constantly changing rules and regulations is really hard. Maybe TDD could be used on easier problems, like nuclear physics." A seasoned product engineer building desktop software might say, "Very interesting, but our code is really hard, it has to interact with the operating system. I could see TDD working for easier problems like insurance software, but not in a domain as complex as ours."

Embedded engineers often raise one or more of these objections:

- The code can't be tested without the hardware.
- We're programming in C.
- It takes 8 min to download our executable program files.
- A full build takes 6 h.
- The target hardware is not ready; any tests not in the target are a waste of time.
- The target hardware is an expensive (or scarce) resource and is shared by the engineers.
- The code needs to be small and fast.
- We write device drivers.
- We don't have time to write those tests.

Clearly for something simple, like nuclear physics or insurance, TDD would be fine. Embedded development is different and has additional challenges. Fortunately, some of these challenges can be solved using TDD.

Winning Over Some Skeptics

The first time I experienced TDD on a real project was during the development of an embedded application six years ago. The system had custom hardware that was not going to be ready for months. There were issues with timing, concurrent processing, and throughput. There were many unknowns, as we had not settled on many of the architecture issues, including the operating system. The management and many of the team members thought that these were crippling uncertainties, and hence thought that the only productive work to be done was to write design documents in preparation for when we could start the implementation. To their surprise, we were able to start design and implementation within days on the areas in which the requirements were understood. We employed TDD to test core behavior and used the best practices of object-oriented design to isolate areas of uncertainty. Work on requirements continued in parallel, reducing the uncertainty in the process. After six months of development, we had only two or three bugs that required more than a few minutes of debugging. The total time spent debugging was only about a week over the six-month project!

The Target Hardware Bottleneck

Concurrent hardware/software development is a reality for many embedded projects. If software can only be run on the target, developers have to wait until late in the development cycle to test, building up a backlog of uncertainty and risk. If target systems are too expensive for each developer to have 100% access to a target, they will have to wait to test their code and will

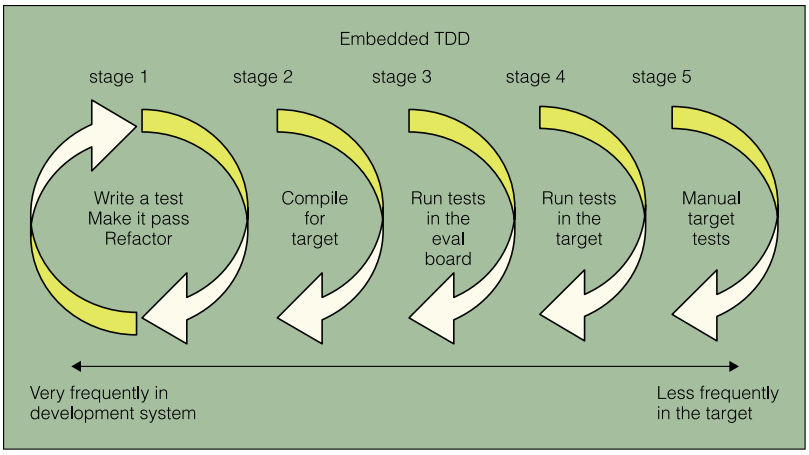

Fig. 1. Embedded TDD.

have to spend valuable system time debugging. Target systems may also have defects. Because of these issues, software progress is often blocked until a reliable execution platform is available.

Use of evaluation hardware, sometimes called eval boards, is common in the industry. An eval board is a development board with the same processor configuration as the target system and ideally some of the same I/O. Sometimes eval boards and target systems may be too expensive to have one dedicated to each developer, or they may have inferior or expensive debug tools.

Embedded Challenges in Adopting TDD

Embedded software development presents a number of challenges to adopting TDD. The hardware bottleneck is just one of a number of challenges faced in adopting TDD for embedded software development. Other challenges include

- Development system and target compiler compatibility,
- Testable design,
- Testing with the hardware,
- Testing in limited-memory environments, and
- Test cycle time and tool chain.

The embedded TDD cycle can help with these problems.

Embedded TDD Cycle

The embedded TDD cycle is an extension of the core TDD cycle described earlier. Figure 1 represents this cycle, which is made up of smaller stages. The first stage is the traditional TDD cycle [1] and is performed on the development system with the goal of rapidly developing clean code with very few defects. In the second stage, the target compiler is used to ensure that we have no compile and link problems with the target. In the third and fourth stages, the automated unit tests are run on the evaluation board and then the target. Finally, end-to-end manual tests are performed to ensure that the system is correctly wired, both logically and physically. At different points in the project life cycle, some of the stages described may either be impossible or not as critical. The team should try to get as far as possible through all stages as early as possible, encouraging early integration; the later the integration with the target, the greater the risk to project success. Let's look at each stage in more detail.

You run the first TDD stage most frequently, followed, as appropriate, by the other steps. For example, a daily build for the target may be adequate while getting new functionality working in the development system. Using the development system allows software development to keep moving forward, without being dependent on hardware uncertainties and unavailability. Executing code on the development system requires a compiler that is compatible, at the source code level, with the target compiler.

Stage 2 concerns the risk of incompatible compiler features. You perform the "compile for target" using the target's cross-compiler and provide an early warning of porting problems. You should do a target cross-compile whenever you use some new language feature, include a new header file, or make a new library call.

Stage 3 concerns the risk that code compiled for the development system will execute differently on the evaluation board. Ideally, you run every test on the eval board (or the target), but download times may make this impractical. At a minimum, you should run the eval test suite daily.

Stage 4 uses the target platform once it is available. This also allows you to run I/O-specific tests. One goal is to make these tests easy to run and repeat, so automating the process keeps the barriers to running the tests low, and hence they will be run more often.

Stage 5 is where you perform full end-to-end testing. Many development efforts only comprise target-based testing, making the developer test and debug in the most challenging environment. The prior stages are designed to reduce the number of defects in the system prior to end-to-end testing. End-to-end testing is essential, and embedded developers know that integrating early is one key success factor. We do not expect to use stage 5 for unit testing; rather, it is where you find manual hardware/software integration problems.

Tailoring the Cycle

At different points in the project life cycle, some of the stages described might be either impossible or not as critical. For example, when there is no hardware early in the project, stages 4 and 5 are not practical. Another example is that once eval hardware is available, and if download times are fast, stage 1 might be skipped in favor of doing the fast feedback tests on eval boards. Keep in mind that the incremental code and test cycle needs to have a short cycle time to keep the developer in the rhythm. If the build and test time starts to get longer than a minute or two, partial builds should be used to keep the code and test cycle short.

Hardware Is Not Available

The ideal situation is to run all automated tests on the actual hardware. This is often not possible as a result of concurrent hardware and software development and constrained memo-

ry size on the target. The embedded TDD cycle helps to remove some of the roadblocks associated with hardware scarcity. To be most effective, build and test times need to be short. This practice leverages both development systems and the target hardware by writing automated tests that run in both the development system and in the target system.

Testing in the development system is an important accelerator for projects because the development system is a proven and more stable execution environment. It often has a richer debugging environment than the target, and each developer has one or can get one tomorrow. The development system is the first place to run any hardware-independent test.

Development System/Target Compiler Compatibility

Using a testing approach on a development system limits the risk of doing all the testing at the end of the project, but it can introduce other risks. One risk with this approach relates to when the compiler, runtime library, or processor for the development system differs from the target system. These risks can be broken into the following parts:

- The target compiler/runtime is missing language features or library calls.
- The target compiler generates code that behaves differently from that generated by the development system compiler.
- Runtime support is not compatible.

Stage 3 addresses these risks. We are not doing general compatibility testing for compilers here. We are making sure that the tests for our application run the same way in the eval board as they do in the development system.

A comprehensive test suite may help find compatibility problems, just as it did during one of my projects. The test suite for CppUTest [2], a unit test harness, uncovered problems when porting to a popular embedded processor. We quickly discovered that the test harness skipped the execution of all the tests. After a little head scratching and focused debugging, we discovered that strstr(), a C library function, operated differently on the target than on the development system. Each system handled zero length strings differently. We modified the usage of strstr() to be compatible on both systems and all tests passed.

Testable Design

To make embedded software testable, you must isolate the hardware dependencies by designing the embedded software for this purpose. Software development starts from the inside with the solving of the application problem and then works its way to the outside, where application code meets the hardware. Interfaces should be designed to describe how the core application interacts with hardware-provided services (see the Unified Modeling Language (UML) diagram in Figure 2). One or more interfaces specify a *hardware abstraction* or *hardware isolation* layer. This abstraction layer shows the intention of the interface, not its implementation. The interface specifies what the hardware can be told to do, not how to do it. For example, the interface would specify "TurnOnAlarm( ), rather than "SetPort(uint address, uchar value)". The core of the application code should not reach down to the hardware and write to some port and cause some interaction with the real world. By telling the application what it wants done, not how to do it, code and tests become more intuitive, and future hardware changes are made easier.

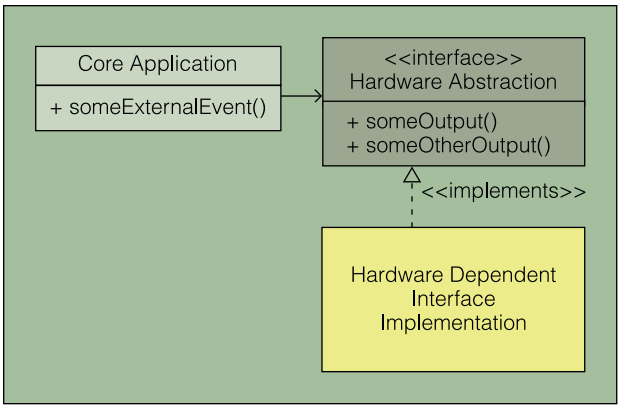

Fig. 2. Hardware abstraction

In the simplified system diagram in Figure 2, the core application accepts some external event in the form of a function call and instructs the hardware abstraction to generate an output. The core application can interact with anything that implements the interface. In C, these interfaces are described by function prototypes in a header file. In C++, an interface is a class declaration, possibly with pure virtual functions.

The UML diagram in Figure 3 illustrates part of a simple home security system. The HomeGuard module implements the essential functionality of the system. It accepts incoming events (such as window intrusion); it reacts to instruct the alarm panel. The AlarmPanel interface defines the allowed interactions with the hardware. HomeGuard is not allowed to know that when you write a 1 to address 0xFDAF00, bit 3, that the alarm will start sounding nor that the front panel for this particular system will use a Model4200 Alarm Panel. With interfaces isolating the core software from hardware details, we can start work immediately on enhancing the core system functionality.

The diagram shows how to take advantage of this design for test purposes. To test HomeGuard's core behavior a HomeGuardTest module is introduced, as is the MockAlarmPanel. The MockAlarmPanel is a test stub known as a Mock Object, which is an object that stands in for a production object during tests [3].

HomeGuardTest orchestrates the test sequence, as shown in the following C++ code example. The test creates a Mock AlarmPanel and feeds it to a newly created instance of a HomeGuard module as an AlarmPanel. The test simulates the event that arms the system and the event that indicates a window has been opened. The test then checks to confirm that HomeGuard interacts properly with the AlarmPanel by interrogating the MockFrontPanel for its current state. The armed indication should be on, the siren blaring, and the strobe flashing. HomeGuard thinks it is fielding events from sensors and commanding the I/O. For example, "hg. arm()" is a simulation of the arm button being pressed.

```
TEST(HomeGuard, WindowIntrusion)
{
  MockAlarmPanel* panel = new MockAlarmPanel();

  HomeGuard hg(panel);

  hg.arm();
  hg.windowIntrusion();
  CHECK(true == panel->isArmed());
  CHECK(true == panel->isAudibleAlarmOn());
  CHECK(true == panel->isVisualAlarmOn());
  CHECK(panel->getDisplayString() == "Window
Intrusion");
}
```

There should be many tests like this focused on specific scenarios. Much of the core functionality of a system can be tested well before the target hardware is available. Furthermore, when the target becomes available, the bring-up time and subsequent integration and testing can be substantially reduced because the core system functionality has already been well tested.

The isolation approach just described is not new; it simply comprises the application of the well-known practices of abstraction and encapsulation and the separation of concerns that are essential for TDD. This technique is independent of any computer language or tool; it is simply a matter of applying good modular design practices. In C++, polymorphism is exploited for creating interfaces between modules. In C or C++, we can use the linker to substitute in test stubs. In C, if run time substitution is needed, function pointers, though hard to read, provide the necessary mechanism.

Testing With Hardware

The use of interfaces and isolation allows us to develop and test significant system behavior without access to the real target. Automated tests to exercise the hardware are valuable to both hardware and software developers because they perform independent tests of hardware subsystems. Testing with hardware falls into the following categories:

- Automated hardware tests,
- Partially automated hardware tests, and
- Automated hardware tests with external instrumentation.

Automated Test

An example of hardware-dependent software that is automatically testable is the flash memory driver. Flash memory is randomly accessible for reading but has restrictions for writing. Previously unwritten locations can be randomly accessed for writing, but once a bit is changed from a 1 to a 0 it cannot be set to a 1 again except by resetting the flash memory block to all 1's. The flash memory driver has to operator according to these rules. The hardware-dependent test of the flash memory can be automated because no external interaction is needed to perform the test.

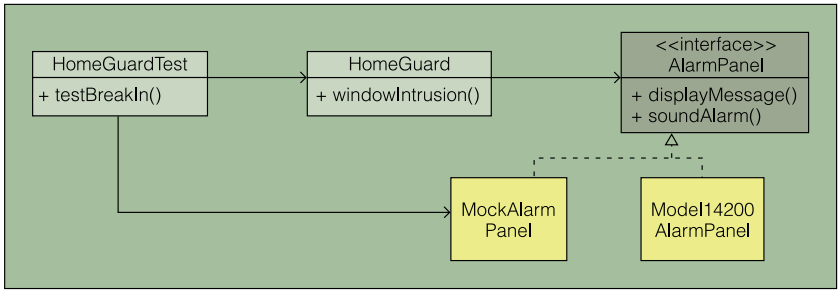

Fig. 3. Alarm system's testable design.

Partially Automated Hardware Tests

Consider testing an LED driver. LEDs are represented in the target system by a word in memory. A 0 in a specific bit in the word may turn off the LED, while a 1 in that bit turns on the LED. This partially automated test would display a message prompting the tester to manually verify that a specific LED is on or off for each LED. Manual acceptance tests like this will likely not need to be rerun very often.

Automated Hardware Tests With External Instrumentation

Special-purpose external test equipment can be used to automate hardware-dependent tests. In the late 1980s, we developed a digital telecommunication monitoring system that monitored 1,544 Mb/s (T1) signals. A major part of the behavior depended on a custom application-specific integrated circuit (ASIC). The ASIC monitored the T1 signals in real time; our embedded software interrogated the ASIC and reported performance information on demand and alarm conditions as they occurred. Testing this system required specialized test equipment to generate T1 signals and inject errors. After tiring of the manual tests, we dug into the instrument's capabilities and discovered it could be controlled through a serial port. Our test engineer wrote automated acceptance tests for the system that drove the T1 line with specific situations and interrogated our system to see if it performed to specification. This practice allowed us to see defects within 1 d of their introduction and resulted in zero defects reported from our installed base of thousands of units. An interesting experience report was given at the Agile2007 conference. [Matt Fletcher, William Bereza, Mike Karlesky, Greg Williams, "Evolving Into Embedded Development", Agile 2007].

Sometimes Code Is Not as Hardware Dependent as You Think

Often engineers think something is hardware dependent when it is not. Recall the LED driver. The LEDs are represented as a word in memory. The application thinks of LEDs not as bits but as LED number. When the LED driver is told to turnOn(LED4), it shifts a 1 bit into bit 4 then ORs that bit into the memory location representing the LEDs. This sort of bit-twiddling code is error prone (one-off errors are common). As it turns out, however, the only hardware dependency in this LED driver is the address of the LED word in the target memory, which can be easily configured for test and production.

The flash file system might sound like it is hardware dependent as well.

It should be acceptance tested with the real hardware, but the flash file system can (and should) be unit tested independent of the hardware by using a mock flash driver.

Testing in Limited-Memory Environments

Embedded systems often have severe memory constraints. Unit tests require space in memory; hence, it may not be possible for all the tests and the code to fit into the target. An eval board with plenty of memory should be used for running all tests. This enables running tests in a binary-compatible target processor. The tests can be organized such that they can be done in batches, taking less memory for each batch, but this may not be practical.

In very constrained environments, special-purpose thin fixtures may be attached through the debug interfaces to allow the tests to stimulate the real target from the eval board or development system.

Test Cycle and Time Tool Chain

TDD requires a rapid feedback loop during unit testing; sometimes the target or evaluation hardware cannot be built or loaded fast enough to make TDD practical. A one-line change in code should be able to be tested in less than 30 s. Faster is better. This fast turnaround allows the developer to remain focused on a task. A critical success factor in embedded TTD is finding ways to shorten the compile-test loop. The best practice is to have the eval board directly connected to the development system so that program-code builds can be as fast as the development system.

I recommend the following tool chain to support a team of embedded developers using TDD. Each developer should have the following:

- Development system for unit tests and hardware-independent acceptance tests,
- Eval system for unit tests and target-independent acceptance tests,
- Build for target and eval acceptance tests,
- Build for target deployment, and
- Access to target for end-to-end testing.

Unit tests are implemented using the appropriate variant of xUnit unit test harnesses [4]. Most xUnit frameworks are modeled after JUnit, a unit test framework for testing Java code [5]. xprogramming.com has a catalog of unit test harnesses [6]. In the example described earlier, we used CppUTest [2]. There is usually only a small amount of work involved in moving a unit test harness into an embedded project's tool chain. FitNesse is an acceptance-testing framework that supports writing executable specifications [7].

Final Remarks

TDD is an important software development practice that can help embedded developers deliver higher quality products. The embedded TDD cycle can help take hardware availability off the software critical path, enabling steady progress with or without hardware. TDD can be used for embedded development in C and C++. Java may also be an option for some embedded systems, and Java is better suited for TDD, as the tools for Java support are much more advanced.

Applying TDD improves test coverage and makes it possible to find defects earlier in the development life cycle. Highly coupled designs and implementations, as found in many embedded systems, make automated test a huge challenge. To be testable, modules have to be independent because automated tests operate on independent modules. Independent modules have advantages, such as improved maintainability and potential for reuse. A commitment to TDD will result in lower coupling and higher cohesion, which are key attributes of solid designs.

Embedded systems expert Jack Ganssle says "The only reasonable way to build an embedded system is to start integrating today…. The biggest schedule killers are unknowns; only testing and running code and hardware will reveal the existence of these unknowns" [8]. Jack goes on to say that "Test and integration are no longer individual milestones; they are the very fabric of development."

I think that TDD is a way to weave test and integration into the fabric of embedded development.

References

[1] K. Beck, *Test-Driven Development*, Boston, MA: Addison Wesley, 2002.

[2] M. Feathers, J. Grenning, B. Vodde, (2007, Oct 1). CppUTest [Online] Available at http://sourceforge.net/projects/cpputest.

[3] S. Freedman and T. Mackinnon, (2000). Endo-Testing: Unit Testing with Mock Objects [Online] Available http://www.mockobjects.com/files/endotesting.pdf.

[4] Anonymous, (2007, Oct 1) Wikipedia [OnLine] Available at http://en.wikipedia.org/wiki/xUnit.

[5] R. Jeffries, (2007, Oct 1) Catalog of xUnit test frameworks. XProgramming.com [OnLine], Available at http://www.xprogramming.com/software.htm.

[6] Anonymous, (2007, Oct 1) JUnit.org Resources for Test Driven Development [OnLine] Available at http://www.junit.org.

[7] R. Martin, M. Martin, (2007, Oct 1) FitNesse Acceptance Testing Framework [OnLine] Available at http://www.fitnesse.org.

[8] J. Ganssle, *The Art of Designing Embedded Systems*, Woburn, MA: Butterworth-Heinemann, 2000, p. 48.

James Grenning (grenning@objectmentor.com) is the Director of Consulting at Object Mentor, Inc. He has been developing software professionally since 1978. His is experienced in embedded and non-embedded software development, management, consulting, mentoring, and training. He is currently practicing and coaching Agile software development techniques, Object Oriented Design, and Programming. James coaches embedded and non-embedded development teams through the transition to Agile. He participated in the creation of the Manifesto for Agile Software Development.