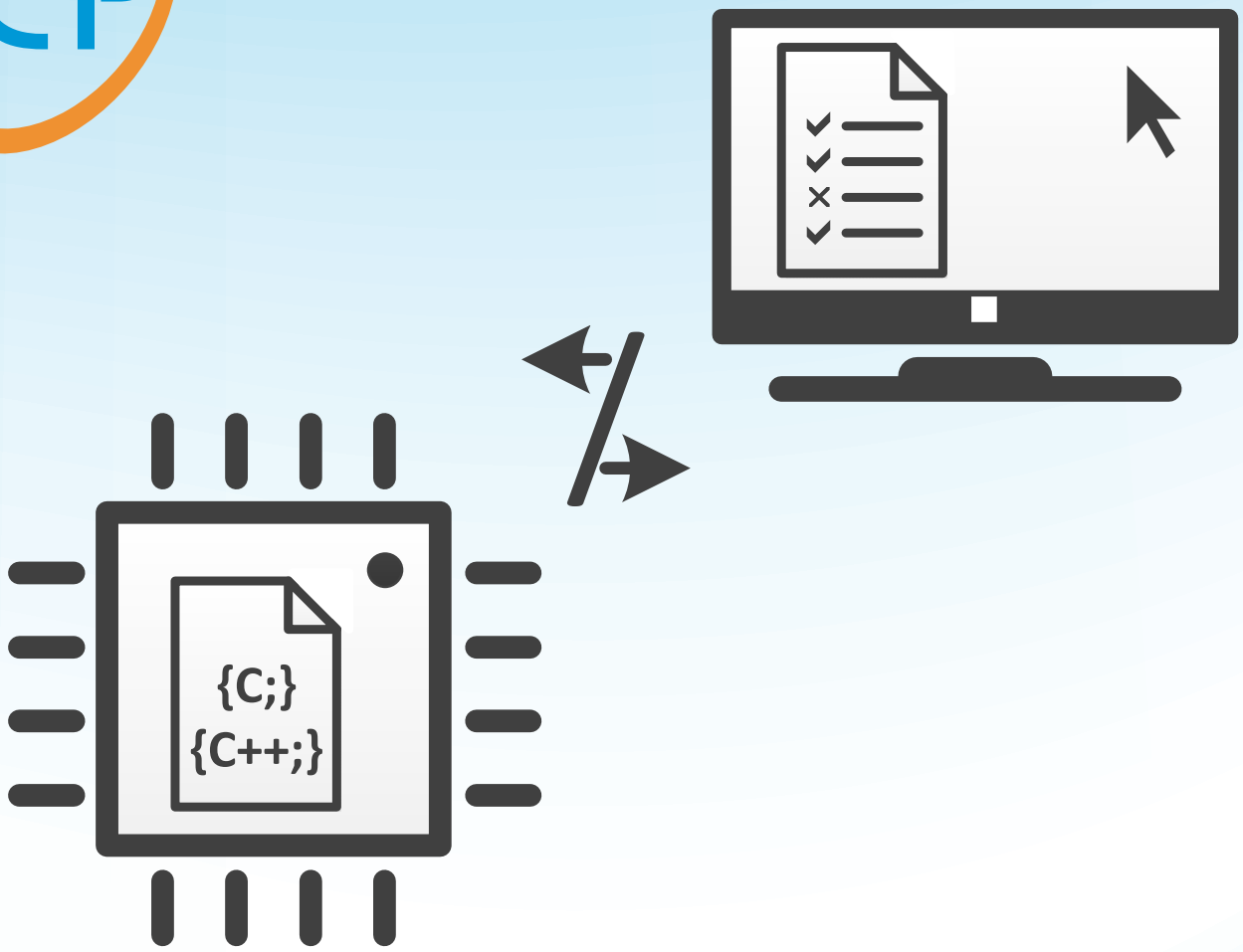

Test-Driven Development for Embedded Software

Manual for industrial developers and academic instructors.

Funded by IWT-090191 (Tetra)

Test-Driven Development for Embedded Software: manual for industrial developers and academic instructors.

Piet Cordemans[a], Sille Van Landschoot[a], Jeroen Boydens[b] [c]

September 29, 2011

[a] P. Cordemans and S. Van Landschoot are scientific staff members at KHBO funded by IWT-090191.
[b] J. Boydens is a professor in Software Engineering at KHBO.
[c] J. Boydens is an affiliated researcher with the Department of Computer Science, K.U.Leuven - Celestijnenlaan 200 A, B-3001 Leuven, Belgium.

Funded by IWT - TETRA 090191

KHBO Dept. Industrial Sciences & Technology, Zeedijk 101, B-8400 Oostende, Belgium

{Piet.Cordemans, Sille.VanLandschoot, Jeroen.Boydens}@khbo.be

version 1.0

http://ep.khbo.be/TDD4ES

User committee: DSP Valley, E.D.&A., FMTC, K.d.G., K.U.Leuven, Marelec, Newtec, Q-star test, Sirris, Summa, Televic, Tesco, Unitron,Van De Wiele

Preface

This manual was produced in the context of the IWT Tetra research project 090191: TDD4ES - Test Driven Development for Embedded Software, which started October 2009 and ended September 2011. Tetra stands for Technology Transfer, which means that research was specifically oriented towards applicability in the industry. To achieve this goal, the project was co-financed by a number of industrial partners who became part of the user committee. Other members included scientific and valorization partners, respectively supporting research and dissemination of results. User committee meetings were held regularly to assess preliminary results and provide valuable feedback.

The strategies, patterns and methodologies described in this manual have been applied in three case studies as a proof of concept. One of these case studies was delivered by a member of the user committee, who provided an industrial embedded hardware system, as well as a legacy software code base. Note that the project and this manual specifically focus on the practice of TDD. Other Agile[55] or eXtreme Programming[15] practices, such as Continuous Integration and pair programming, were not considered. These practices might prove their worth, especially when combined with TDD or considering the context of embedded. Yet for more detailed information, references have been provided in the bibliography.

Furthermore nearing the end of the TDD4ES project, James Grenning published a book on the subject of Test-Driven Development for Embedded C[36]. This book provides a lot of information on TDD for embedded, which is quite similar to our own experiences with the subject. Therefore a choice has been made to give a specific subject, namely TDD for C, less attention in this manual. Instead for those who are interested in the C specific issues for TDD, the book of Grenning is recommended. Nevertheless, this manual covers some topics which are novel ideas. So, even for those who are only developing in C, a number of ideas in this manual might prove to be interesting. Although most examples in this manual are written in C++, they should be simple enough to be comprehensible with only a minimal understanding of Object Oriented features.

For those interested in the general motivation start with 1 Introduction. For those who do not know about Test-Driven Development, start with 2 Test-Driven Development. For those who know TDD and want to start immediately, begin with 3 Test-Driven Development for embedded software. Finally for those interested in the advanced concepts, chapters 4 and 5 deal with advanced topics, such as legacy code and patterns.

Contents

Chapter 1

Introduction

"The relative cost to fix an error exponentially increases in a later phase of the project."

Barry Boehm[16]

The term embedded system covers a wide range of electronic applications. These can be simple systems such as a controller for a microwave oven, or complex systems like a digital camera. However, all these systems have one thing in common, which is that they are designed with one specific application in mind. The embedded system looks like a small computer with dedicated hardware, but the main difference with a computer system is that its goal does not change during its lifetime. An embedded system for a digital camera will always remain the system for a digital camera, it will not change into a system for a microwave oven. It should be noted that multiple embedded systems might be combined in one appliance. A cell phone could for instance integrate an mp3-player, as well as a digital camera. These underlying embedded systems can be fully integrated, or kept completely separate.

Embedded systems have a number of key properties: (1) their restricted price, (2) their restricted size, (3) their required performance and (4) their restricted power consumption. More advanced properties are their (5) reactiveness to events and their (6) time-critical behavior. These properties make embedded system design a very specific co-design of hardware and software. The hardware developer must be aware of the software restrictions and vice versa.

In the embedded world there is a gap between the importance of software testing and the state of testing embedded software. Delivering an embedded product with faulty software is very costly as firmware updates or patches are hard or even impossible to apply. Thorough testing of embedded software is essential to assure the desired functionality has been achieved[17]. Even though testing does not prove the absence of bugs[21], it can confirm certain expectations, which boosts the confidence in the quality of the code.

As indicated the importance of testing is essential for the quality of embedded software, however the current techniques are ad-hoc, end-to-end and debugging, only focusing on the current issue[42]. Also typical for embedded system development, testing is postponed after integration of software and hardware[57].

1.1 Why Test-Driven Development?

As embedded systems are currently becoming more and more complex, the importance of their software component rises. Furthermore, due to the definite deployment of embedded software once it is released, it is unaffordable to deliver faulty software. A thorough testing is essential to minimize software bugs. The design of embedded software is strongly dependent on the underlying hardware. Co-design of hardware and software is essential in a successful embedded system design. However, during the design time, the hardware might not always be available, so software testing is often considered to

be impossible. Therefore testing is mostly postponed until after hardware development and testing is typically limited to debugging or ad-hoc testing. Moreover, as it is the last phase in the process, it might be shortened when the deadline is nearing. Integrating tests from the start of the development process is essential for a meticulous testing of the code. In fact, these tests can drive the development of software, hence Test-Driven Development.

It is crucial for embedded systems that they are tested very thoroughly, since the cost of repair grows exponentially once the system is taken in production, as stated in figure 1.1, which depicts the law of Boehm. However, the embedded system can only be tested once the complete development process is finished. Most embedded systems are developed using the waterfall process for their software. First the user requirements are gathered. Next, these requirements are translated into functional specifications. Once all specifications are formally written down, the global technical design phase can start. After the global design comes the detailed technical design, as a basis for the next phase of programming. Finally, the system can be tested. If the hardware is still not available at this point, simulation tools and instruction set compilers are used to verify the behavior of the software component. Thorough testing can only be done when the hardware is fully configured. In the current strategy for developing embedded systems, the testing phase is generally done manually. This ad-hoc testing is mostly heuristic and only focuses on one specific scenario. At this point debugging facilities are very handy to look at the inside functioning of the software component that is tested. As noted, the testing is done late in the development cycle, with all due disadvantages. When we want to start testing as early as possible, a number of problems arise. One problem being the hardware unavailability, and another being the difficulty to automatically test embedded systems.

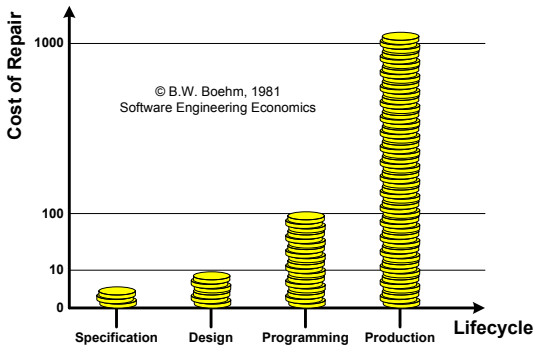

Figure 1.1: Law of Boehm

As illustrated in figure 1.2 the development phase of TDD takes longer than that of conventional development. A reason for this is the extra effort it takes to create tests during development. However, a quality assurance phase is necessary when creating software using conventional development processes. During quality assurance, bugs and misconceptions are fixed in the software. This phase is generally known as the beta test phase. This quality assurance phase is spread out during TDD, hence the longer development time. The investment in testing early on in TDD results in a non-negligible life cycle benefit.

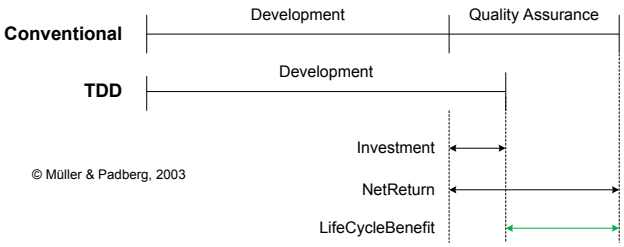

Figure 1.2: Life-cycle benefit of TDD

1.2 Automated testing

Testing manually is time-consuming and error-prone. Test-Driven Development demands that tests are frequently run during development; as such that test automation is indispensable. Without automated tests, testing the code will occur less frequently, which results in larger chunks of code tested at once, which will inevitable contain more bugs, slowing the process even further. Finally this will drag the development cycle to a halt. An automated test suite is a necessity in TDD, however automating tests in an embedded development environment is not trivial. Primarily because hardware dependencies are fundamental to embedded software.

Developing using the TDD strategy should be done with caution. A pitfall generally known as the broken window syndrome[38] states that one must be careful with broken tests. The basic idea of broken window comes from windows in an abandoned building: once one test starts to fail, shortly after the number of failing tests will grow gradually, ultimately leading to all tests being broken. The tests can break for different reasons: the expected behavior is changed, or the implementation of the business code is changed. In the former case the developer adjusts the test code to correctly test the updated expected behavior. In the latter case the implemented business code does not support the original expected behavior. To solve this problem, the business code is adjusted so the test succeeds. Another potential pitfall is test cancer[28]. It is a bad habit to leave out a failing test from the test scenario. One could think that by leaving out this test, the scenario now completely succeeds. But when the failing test is not corrected, more and more tests will fail and be left out from the scenario. This way the failing tests spread like cancer through the developed code.

1.3 Incremental development

To support development of modern embedded systems, the waterfall process and other sequential models fail to deliver. Furthermore, co-design and co-development of hardware and software is a necessity to avoid integration issues early on. As indicated by the law of Boehm, earlier detection of problems reduces their costs. Acknowledging this fact, traditional software development methodologies have shifted from Waterfall to Agile practices.

Engineering in general lends itself towards incremental development, hence software development follows this same lead. But since software engineering is an extremely agile discipline, other development models emerge in the embedded world. On the one hand, this is because of the ease of changing and duplicating software. On the other hand, a fast feedback cycle, supported by automated tests, can be provided, which is needed to guide the development process.

However, regardless of complications, three benefits are obtained when incrementally developing embedded software. First, testing software, on a target hardware platform or the host platform, is introduced early on. Next, developing incrementally pushes the design towards a thorough modularization. Finally, as the hardware platform will inevitably change, hardware abstraction becomes a necessity. This improves software reusability across multiple platforms and across different iterations.

The software development process of Test-Driven Development (TDD) finds its origin in general software development. TDD's methodology originated in the late eighties, but since the general acceptance of cyclic development processes such as eXtreme Programming (XP), SCRUM and the Unified Process (UP), it has received more attention.

1.4 Further reading

George and Williams[30] have conducted a research on the effects of TDD on development time and test coverage. Siniaalto [56] provides an overview of the experiments regarding TDD and productivity. Nagappan [50] describes the effects of TDD in four industrial case studies. Note that research on the alleged benefits of TDD for embedded software is limited to several experience reports, such as written by Schooenderwoert [52, 53] and Greene [32].

Chapter 2

Test-Driven Development

> "By writing a test before implementing the item under test, attention is focused on the item's interface and observable behavior."
>
> Kent Beck[14]

Test-Driven Development (TDD) is a fast paced incremental software development strategy, based on automated unit tests. First, this section describes the core of TDD (2.1), using a simple example. Next, the advantages (2.2) of developing by TDD are discussed and the limitations (2.3) of the strategy are indicated. Finally, an overview of unit testing frameworks (2.4) is given.

2.1 TDD mantra

Test-Driven Development consists of a number of steps, sometimes called the TDD mantra. In TDD, before a feature is implemented, a test is written to describe the intended behavior of the feature. Next, a minimal implementation is provided to get the test passing. Once the test passes, code can be refactored. Refactoring is restructuring code without altering its external behavior or adding new features to it.When the quality of code meets an acceptable level, the cycle starts over again, as visually represented in figure 2.1. Red and green refer to the colors sometimes used in a unit test framework, respectively indicating test failure and success.

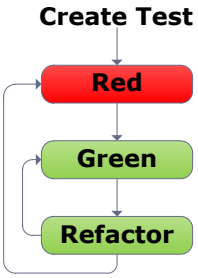

Figure 2.1: TDD mantra

TDD reverses the conventional consecutive order of steps, as tests should be written before the code itself is written. Starting with a failing test gives an indication that the scope of the test encompasses new and unimplemented behavior. Moreover, if no production code is written without an accompanying test, one can assure that most of the code will be covered by tests.

Also fundamental to the concept is that every step is supported by executing a suite of automated unit tests. These tests are executed to detect regression faults either due to adding functionality or refactoring code.

2.1.1 Create test

Creating a test is translating of feature specifications into an executable test, by means of an example. For instance, take the example of developing a Temperature class (or struct), which stores the value of temperature in Kelvin and allows for accessing and updating the temperature in Kelvin, Celsius or Fahrenheit. A first test might look like the KelvinTest in listing 2.1. Note that the test is written for UnitTest++[44].

Listing 2.1: Store and retrieve a temperature value test

```
1 TEST(KelvinTest)
2 {
3    Temperature myTemperature;
4    double aValue = 273.15;
5    myTemperature.setKelvin(aValue);
6    CHECK_EQUAL(aValue, myTemperature.getKelvin());
7 }
```

A test consists of three steps.

1. Setup of the test
2. Executing the function under test
3. Asserting the results

At test setup the necessary objects are created, values are set, connections are made, etc., to arrive in a desired initial state. Next, the function under test is executed. Finally an assertion is made of the results, checking whether the desired final state has been reached.

The properties of a good unit test can be described by the F.I.R.S.T. acronym.

1. Fast: the execution time of the tests should be limited. If it takes too long, a suite of many tests will limit development speed.
2. Independent: the setup and result of a test should be independent of other tests. Dependencies between tests complicate execution order and lead to failing tests when changing or removing tests.
3. Repeatable: the execution of a test should be repeatable and deterministic. False positives and negatives lead to wrong assumptions.
4. Self-validating: the result of a test should lead to a failing or passing assertion. This may sound obvious, nevertheless should the test output something else, like a log file, it cannot be verified automatically.
5. Timely: this refers to the TDD way of writing tests as soon as possible.

When a test is created, it might call non-existing functions, which results in compilation errors. Subsequently these compilation errors must be dealt with before proceeding. However, one should only implement a minimum amount of code in order to get the system through compiling. Effectively TDD states that no code is ever written without a covering test. Writing more code than needed to proceed to the following step should be avoided. In effect, a programmer should not try to anticipate on additional features.

In order to continue with the Temperature example, listing 2.2 provides a skeleton of code to lead to a failing KelvinTest. Note that no implementation is given to the methods.

2.1.2 Red bar

Listing 2.2: Minimal structure of the Temperature class to get through compilation

```
1  class Temperature
2  {
3     public:
4           Temperature();
5           virtual ~Temperature();
6           double getKelvin() const;
7           void setKelvin(double k);
8  };
9
10 double Temperature::getKelvin() const
11 {
12    return 0;
13 }
14
15 void Temperature::setKelvin(double k)
16 {
17 }
```

Once the test and accompanying code get through compilation it becomes possible to run the test and see it failing. A failing test indicates that a new feature is implemented. Writing a failing test is also an indication that the test can fail. If a test never fails, it is worthless and might lead to false assumptions on the code under test. The code already provided should be enough to proceed to the next step.

2.1.3 Green bar

Next, a minimal implementation should be provided in order to get the test passing. The easiest and minimalistic solution is to fake the result, this means returning the result the test expects, as shown in listing 2.3.

Listing 2.3: Providing a fake result in order to pass the first test

```
1  double Temperature::getKelvin() const
2  {
3     return 273.15;
4  }
```

At first this might be conceived as too obvious. Indeed, the following code is too specific to be of any use, but as mentioned before, TDD is a cyclic process. Next step would be to write a new test, which proves the current implementation is not sufficient. This example might be too simple to prove its use, but in a realistic development process, one encounters functionality, which might be too complicated or unclear at that moment to directly write a good implementation. Faking the first test and subsequently writing a second test might provide more information on the problem, ultimately leading to a more realistic solution.

However, faking the result is a very small step in the TDD cycle. When possible, one can immediately provide the obvious implementation. Continuing the example, listing 2.4 adds a private field kelvin and implements the accessor and mutator, accordingly.

Listing 2.4: Providing the actual implementation in order to pass the first test

```
1  class Temperature
2  {
3     public:
4           Temperature();
5           virtual ~Temperature();
6           double getKelvin() const;
7           void setKelvin(double k);
8     private:
9           double kelvin;
10 };
11
12 double Temperature::getKelvin() const
13 {
14    return kelvin;
15 }
16
17 void Temperature::setKelvin(double k)
18 {
19    kelvin = k;
20 }
```

Writing the obvious implementation is the ideal development strategy, but sometimes it is impossible to just do that. In that situation, TDD allows to change gears and work according to the fake or some other strategy. In fact, when starting to implement an feature and failing to immediately provide the correct code, one can easily erase previous changes and start over, taking smaller steps this time.

It is important that code written in this phase only deals with writing code to get the test passing. This means no assumptions should be made towards future features, as future features should be covered by writing new tests.

At this point, one can start over, picking a new feature to implement or should it be necessary to refactor the current implementation. As no refactoring is imminent, the thermometer example can be extended with the feature of setting and retrieving the temperature in Celsius. Following the TDD strategy, a test is created first, next it is made to compile and prove it fails. Finally a minimal implementation is provided to get the test passing, ultimately leading to the following code (for conciseness, listing 2.5 does not show the Kelvin related code).

Listing 2.5: Test and implementation for Celsius

```
1  TEST(CelsiusTest)
2  {
3     Temperature myTemperature;
4     double aValue = 18.5;
5     myTemperature->setCelcius(aValue);
6     CHECK_CLOSE(aValue, myTemperature->getCelcius(), 0.01);
7  }
8
9  class Temperature
10 {
11    public:
12          Temperature();
13          virtual ~Temperature();
14          double getCelsius() const;
15          void setCelsius(double c);
16    private:
17          double kelvin;
18 };
19
20 double Temperature::getCelsius() const
21 {
22    return kelvin - 273.15;
23 }
24 void Temperature::setCelsius(double c)
25 {
26    kelvin = c + 273.15;
27 }
```

Now, the code works, as proven by running the tests and see them passing, thus reaching the so-called green bar state. However, internally duplication has shown up. The constant 273.15, which is the offset between Celsius and Kelvin, recurs twice and does not explain its intent. Therefore, before creating a new test, refactoring is in order.

2.1.4 Refactor

Refactoring is changing code for readability, remove duplication or improve software design, without changing or adding its behavior. This becomes necessary, because providing the minimal implementation will lead to code that is suboptimal, effectively less manageable. Moreover, as TDD focuses on one

feature at a time during the first steps of its cycle, one could lose the general overview. Maintaining focus on a single feature has its merits though, but at some point one needs to tend to the quality of software.

Coincidentally, regularly running automated unit tests will ensure that no behavior is altered while refactoring. Running these tests frequently will indicate when refactoring breaks the system, providing important feedback.

In the thermometer example, listing 2.6, the hard coded value is replaced by a static const. Re-running the tests proves that the behavior remains the same, as expected.

Listing 2.6: Removing duplication through refactoring

```
1  static const double kelvinShift = 273.15;
2
3  double Temperature::getCelsius() const
4  {
5     return kelvin - kelvinShift;
6  }
7  void Temperature::setCelsius(double c)
8  {
9     kelvin = c + kelvinShift;
10 }
```

Fundamental to the concept of TDD is that refactoring and adding new behavior are strictly separated activities. When refactoring, tests should remain passing. Yet should a failure occur, it should be solved in quick order or the changes must be reverted. On the other hand, when adding new functionality, the focus should stay on the current issue, only conducting the refactorings when all tests are passing. Refactoring can and should be applied to the tests themselves as well. In that situation the implementation stays the same and can be reassured that the test does not change its own scope.

2.2 Advantages

Programming according to the principles of TDD has a number of advantages. First and foremost are those which result from incrementally developing an automated test suite. Also TDD allows for a steady and measurable progression. Finally TDD forces a programmer to focus on three important aspects.

As TDD imposes to frequently run a test suite, four particular advantages result from it. First, the tests provide a safety net when refactoring, alerting the programmer when a refactoring went wrong, effectively altering the behavior of software. Next, running tests frequently will detect regression when code for a new feature interferes with other functionality. Furthermore, when encountering a bug later on (TDD cannot guarantee the absences of bugs in code), a test can be written to detect the bug. This test should fail first, so one can be sure it tests the code where the bug resides. After that making the test pass will solve the bug and leave a test in place in order to detect regression. Moreover, tests will ensure software modules can run in isolation, which improve their reusability. Finally, a test suite will indicate the state of the code and when all tests are passing, programmers can be more confident in their code.

Next to the automated test suite, TDD also allows for a development rate, which is steady and measurable. Each feature can be covered by one or more tests. When the tests are passing, it indicates

that the feature has been successfully implemented. Moreover, through strategies like faking it, it becomes possible to adjust the development rate. It can go fast when the implementation is obvious or slower when it becomes difficult. Anyhow, progression is assured.

Finally TDD is attributed to put the focus on three fundamental issues. First, focus is placed on the current issue, which ensures that a programmer can concentrate on one thing at a time. Next, TDD puts the focus on the interface and external behavior of software, rather than its implementation. By testing its own software, TDD forces a programmer to think how software functionality will be offered to the external world. Furthermore TDD is complementary to Design by Contract in this respect, where a software module is approached by a test case instead of formal assertions. Lastly TDD moves the focus from debugging code to testing. When an unexpected issue arises, a programmer might revert to an old state, write a new test concerning an assumption and see if it holds. This is a more effective way of working as opposed to relying on a debugger.

2.3 Limitations

TDD has a number of imperfections, which mainly concern the overhead introduced by testing, thoroughness of testing and particular difficulties when automating particular hard to test code.

Writing tests covering all development code doubles the amount of code that needs to be written. Moreover the tests that are written need to be maintained as well as production code. Furthermore setting up a test environment, might require additional effort, especially when multiple platforms are targeted.

Next a critical remark has to be made on the effectiveness of the tests written in TDD. First, they are written by the same person who writes the code under test. This situation can lead to narrow focused tests, which only expose problems known to the programmer. In effect, having a large test suite of unit tests, does not take away the need of integration and system tests. On the other hand code coverage is not guaranteed. It is the responsibility of the programmer to diverge from the happy path and also test corner cases. Additionally, tests for TDD specifically focus on black box unit testing, because these tests tend to be less brittle than tests, which also test the internals of a module. However for functional code coverage glass box tests are also necessary. Although these are focused towards a specific implementation. Therefore these must be changed each time the internals of the module changes, hence they are called brittle, and incur an additional overhead on test maintenance.

At last TDD is specifically effective to test library code. This is code which is not directly involved with the outside world, for instance user interface, databases or hardware. However when developing code related to the outside world, one has to lapse on software mocks. This introduces an additional overhead, as well as assumptions on how the outside world will react. Therefore it becomes vital to do some extra tests, which verify these assumptions.

2.4 Unit testing framework

In TDD the most valuable tool, is a unit testing framework. Most of these frameworks are based upon an archetypal framework known as xUnit, from which various ports exist, like JUnit for Java and CppUnit for C++. In fact for C and C++ more than 40 ports exist to date and most of them are open source. Regardless of the specific implementation, most of these have some common structure, which is shown in figure 2.2.

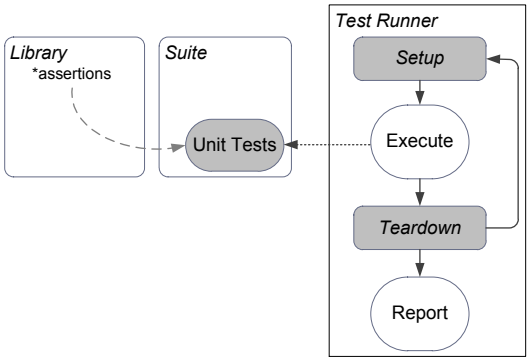

Figure 2.2: xUnit testing framework

A unit testing framework consists of a library and a test runner. On the one hand the library mostly provides some specific assertions, like checking equality for various types. Optionally it might also check for exceptions, timing, memory leaks, etc. On the other hand the test runner calls the unit tests, setup, teardown and reports to the programmer. Setup and teardown in combination with a test is called a test fixture. First, setup provides the necessary environment for the unit test to execute. After test execution, teardown cleans the environment. Both are executed accordingly in order to guarantee test isolation. Instead of halting execution when it encounters a failing assertion, the test runner will gracefully eject a message, which contains valuable information of the failed assertion. That way all tests can run and a report is composed of failing and passing tests. For organizational reasons tests might be grouped into suites, which can be independently executed.

Choosing a unit test framework depends on three criteria.

- Portability. Considering the different environments for embedded system applications, portability and adaptability are a main requisite for a unit test framework. It should have a small memory footprint, allow to easily adapt its output and do not rely on many libraries.
- Overhead of writing a new test. Writing tests in TDD is a common occurrence and therefore it should be made easy to do so. However some frameworks demand a lot of boilerplate code, especially when it is obligatory to register each test. As registration of tests is quite often forgotten, this can lead to confusion.
- Minor features, such as timing assert functionality, memory leak detection and handling of crashes introduced by the code under test, are not essential, but can provide a nice addition to the framework. Furthermore some unit testing frameworks can be extended with a complementary mocking framework, which facilitates the creation of mocks.

Deciding on a unit test framework is a matter of the specifications of the target platform. An incomplete and concise overview of possible candidates is given.

- MinUnit[6] is the smallest C framework possible. It consists of two C macro's, which provide a minimal runner implementation and one assertion. This framework can be ported anywhere, but it should be extended to be of any use and it will take much boilerplate code to implement the tests.
- Embunit[3] is a C based, self-contained framework, which can be easily adapted or extended. However, it requires to register tests, which is error-prone and labor intensive.
- Unity[8] is similar to Embunit and additionally contains a lot of embedded specific assertions. It is also part of a tool suite called Rake, which includes a mocking framework and code generation tools written in Ruby to deal with boilerplate code.

- UnitTest++[44] is C++ based, which can be ported to any but the smallest embedded platforms. Usability of the framework is at prime, but it requires some work to adapt the framework to specific needs.
- CppUTest [1] is one of the latest C++ testing frameworks, it also has a complementary mocking framework, called CppUMock.
- GoogleTest [4] is the most full-blown C++ unit test framework to date. It provides integration with its mocking framework, GoogleMock. However it is not specifically targeted to embedded systems and is not easily ported.

2.5 Test coverage

As mentioned in section 2.3, unit tests written in the TDD process cannot replace the testing phase. Nevertheless it may complement and reduce the amount of testing, when the following remarks are considered.

First unit tests will only cover as much as the programmer deemed necessary. Corner cases tend to be untested, as they will mostly cover redundant paths through the code. In fact writing tests which will cover the same path with different values are prohibited by the rule that a test should fail first. This rule is stated with good reason, as redundant tests tend to lead to multiple failing tests if a regression is introduced, hence obfuscating the bug. Testing corner case values should be done separately from the activity of programming according to TDD. An extra test suite, which is not part of the development cycle allows for a minimalistic effort to deal with corner case values. Should one of these test detect a bug, the test can easily be migrated to the TDD test suite to fix the problem and detect regression.

On the other hand programmers become responsible to adhere strictly to the rules of TDD and only implement a minimum of code necessary to get a passing test. Especially in conditional code, one could easily introduce extra untested cases. For instance an *if* clause should only lead to an *else* clause, if a test demands to do so.

Furthermore, a consecutive number of conditional clauses tend to increase the number of execution paths without demanding to write extra test cases. Similar to the corner case values, an extra test suite can deal with this problem. However, TDD also encourages avoiding this kind of code, by demanding isolation. This will typically lead to a large number of small units, for instance classes, rather than one big complicated unit.

Finally, a unit test should never replicate the code that it is testing. Replication of code in a test leads to a worthless test as bugs introduced in the actual code will be duplicated in the test code. In fact, code complexity of tests should always be less than the code under test.

2.6 Further reading

An extensive introduction to TDD is given in the seminal book by Kent Beck, Test-Driven Development, by example[14]. The F.I.R.S.T. acronym is coined by Robert Martin [45]. Refactoring [25] gives an overview of a great number of refactorings. An introduction to Design by Contract is given by Mitchell and McKim[49] . Hamill[37] provides a language independent overview of unit test frameworks, while Llopis [43] has given an overview of some older C++ unit test frameworks.

Chapter 3

Test-Driven Development for embedded software

> "Quality is free, but only to those who are willing to pay for it. "
>
> Tom DeMarco[20]

Ideally Test-Driven Development is used to develop code which does not have any external dependencies. This kind of code suits TDD well, as it can be developed fast, in isolation and does not require a complicated setup. However, when dealing with embedded software the embedded environment complicates development. Four typical constraints influence embedded software development and have their effect on TDD. To deal with these issues three strategies have been defined, which tackle one or more of these constraints. Each of these strategies leads to a specific setup and influence the software development process. However, neither of these strategies is the ideal solution and typically a choice needs to be made depending on the platform and type of application.

3.1 Embedded constraints

Embedded systems encompass a range of electronic systems, which only have a microprocessor in common. These embedded platforms might range from a simple 6 pins 8-bit microcontroller to a full-blown 32-bit dual core, depending on the application. Moreover, there are some less-defined characteristics, which are common as well. First, the development platform differs from the execution platform, also known as host and target respectively. Next, all embedded systems are designed to be cost-effective, resulting in limited memory and processing power. Finally, at least a part of software on an embedded system is closely related to the hardware.

1) Development speed TDD is a fast cycle, in which software is incrementally developed. This results in frequently compiling and running tests. However, when the target for test execution is not the same as the host for developing software, a delay is introduced into development. For instance, this is the time to flash the embedded memory and transmit test data back to the host machine. Considering that a cycle of TDD minimally consists of two test runs and expectantly will take several more, this delay becomes a bottleneck in development according to TDD. A considerable delay will result in running the test suite less frequent, which in turn results to taking larger steps in development. Moreover this will introduce more failures, leading to more delays. In turn this will reduce the number of test runs, etc.

Furthermore, due to the specific nature of embedded systems, software is developed concurrently with hardware. This implies that hardware might be (partially) unavailable during software develop-

ment. As TDD is a fast iterative cycle, delays introduced in co-designing the embedded system should not interrupt software development. In TDD's case, both the development and testing platform should remain available.

2) Memory footprint Executing TDD on a target platform burdens the program memory of the embedded system. Rather than solely the program code residing in target memory, tests and the testing framework are also added. This results in at least doubling the memory footprint needed.

3) Cross-compilation issues In respect of the development speed and memory footprint issues, developing and testing on a host system solves the previously described problems. However, the target platform will differ from the host system, either in processor architecture or build tool chain. These issues could lead to incompatibilities between the host and target build. Comparable to other bugs, detection of incompatible software has a less significant impact should it be detected early on. In fact, building portable software is a merit on its own as software migration between target platforms improves code reuse.

4) Hardware dependencies External dependencies, like hardware interaction, complicate the automation of tests. First, they need to be controlled to ensure deterministic execution of the tests. Furthermore hardware might not be available during software development. Regardless of the reason, in order to successfully program according to TDD, tests need to run frequently. This implies that executing tests should not depend on the target platform. Finally, in order to effectively use an external dependency in a test, setup and teardown will considerably get more complicated.

3.2 Test on target

In the *Test on target* strategy, TDD issues raised by the target platform are not dealt with. Nevertheless, *Test on target* is a fundamental strategy as a means of verification. First executing tests on target deliver feedback as part of an on-host development strategy. Moreover during the development of system, integration or real-time tests, the effort in mocking specific hardware aspects is too labor intensive. Finally, writing validation tests when adopting TDD in a legacy code based system, provides a self-validating, unambiguous system to verify existing behavior.

3.2.1 Implementation

Fundamental to *Test on target* is a portable, low overhead test framework, as shown in figure 3.1. Secondary, some specific on-target test functionalities, like timed asserts or memory leak detection, are interesting features to include. Finally, it is important to consider the ease of adapting the framework when no standard output is available.

Tests are written in program memory of the target system, alongside the code under test itself. Generally, the test report is transmitted to the host system, to review the results of the test run. However, in extremely limited cases it becomes even possible to indicate passing or failing tests on a single LED. Yet this implies that all free debug information is lost and therefore this should be considered as a final resort on very limited embedded systems.

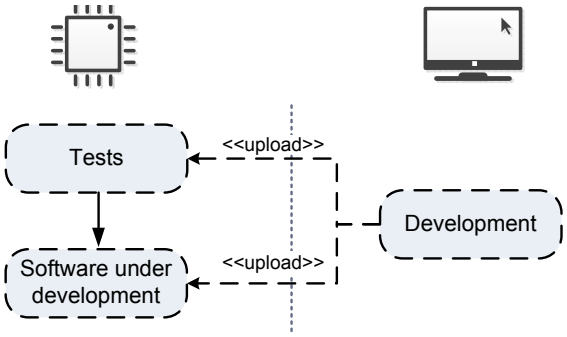

Figure 3.1: Test on target

3.2.2 Process

Test on target is too time-consuming to comfortably develop embedded software, because of frequent upload and execution cycles on target. Still it complements embedded TDD in three typical situations.

First, it extends the regular TDD cycle on host, in order to detect cross-platform issues, which is shown in the embedded TDD cycle, figure 3.2. Complementary to the TDD cycle on host, three additional steps are taken to discover incompatibilities between host and target. First, after tests are passing on the host system, the target compiler is invoked to statically detect compile-time errors. Next, once a day, if all compile-time errors are resolved, the automated on-target tests are executed. Finally, every few days, provided that all automated tests are passing, manual system or acceptance tests are done. Note that time indications are dependent on an equilibrium between finding cross-compilation issues early on and avoiding too many waiting periods.

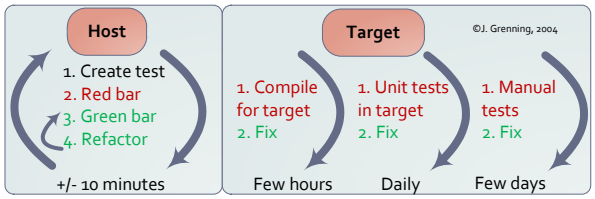

Figure 3.2: Embedded TDD cycle

A second valuable use of *Test on target* is the development of target-dependent tests and code. For instance, memory management operations, real-time execution, on-target library functionality and IO-bound driver functions are impossible to test accurately on a host system. In these situations, forcing TDD on host will only delay development. Furthermore, an undesirable number of mock implementations are needed to solve some of these cases, resulting in tests that only test the mock. TDD should only be applied to software that is useful to test. When external software is encountered, minimize, isolate and consolidate its behavior.

Finally, *Test on target* has its merit to consolidate behavior in software systems without tests. Changing existing software without tests giving feedback on its behavior, is undesirable. After all this is the main reason to introduce TDD in the first place. However, chances are that legacy software does not have an accompanying test suite. Preceding refactoring of legacy software with on-target tests capturing the system's fundamental behavior is essential to safely conduct the necessary changes.

3.2.3 Code example

In the following example, a focus is put on automation of tests for low level hardware-related software, according to the Test on target strategy. In a true TDD fashion, developing new functionality starts with a test (listing 3.1).

Listing 3.1: A test to check the getState logic of a button.

```
1  TEST(RepeatButtonTest)
2  {
3     Button *button = new Button(&IOPIN0, 7);
4
5     button->setCurrentState(RELEASED);
6     CHECK(button->getState() == RELEASED);
7
8     button->setCurrentState(PRESSED);
9     CHECK(button->getState() == PRESSED);
10
11    button->setCurrentState(RELEASED);
12    button->setCurrentState(PRESSED);
13    CHECK(button->getState() == REPEAT);
14
15    delete button;
16 }
```

This test creates a button object and tries to check whether its state logic functions correctly, namely two consecutive high states should result in REPEAT. Now, one way to test a button is to press it repeatedly and see what happens. Yet this requires manual interaction and is not feasible to manually test the button every time a code change is made. However, automation of events related to hardware can be solved with software. In this case an additional method is added to the button class, setCurrentState, which allows to press and release the button in software.

Two remarks are generally put forward when adding methods for testing purposes. On the one hand, these methods will litter production code. This can easily be solved by inheriting from the original class and add these methods in a test subclass[a]. On the other hand, when a hardware event is mocked by some software, it might contain bugs on its own. Furthermore there is no guarantee that the mock software is a good representation of the hardware event it is replacing. Finally, is the actual code under test or rather the mock code tested this way?

These remarks indicate that manual testing is never ruled out entirely. In the case of automating tests and adding software for this purpose, a general rule of thumb is to test it both manually and automated. If both tests indicate the same behavior, consider the mock software as good as the original hardware event. The added value of the automated test will pay back for its investment when refactoring the code under test or extending its behavior.

In the RepeatButtonTest the state of the button is changed almost instantaneously to simulate a glitch caused by the mechanical contact of the button. In this button, such behavior should be ignored. Usually the compiler will complain about some missing definitions, therefore listing 3.2 provides the code to satisfy the compiler and allow the hex file to be uploaded to target.

[a] In a programming language, which is not object oriented, a solution for hiding testing methods requires a bit more work, as will be demonstrated in section 3.3.2.

Listing 3.2: Definition of the Button class to satisfy the compiler.

```
1  enum ButtonState{RELEASED = 0,PRESSED = 1,REPEAT = 2};
2
3  class Button
4  {
5     public:
6        Button(volatile unsigned long* portAddress, char pinNumber);
7        virtual ~Button();
8        void setCurrentState(ButtonState current);
9        ButtonState getState();
10 };
```

The implementation of each method is omitted, but these can be considered to be empty or returning a constant RELEASED value in the case of getState(). Running the test on target will fail on the second assert, which allows to go to green bar. Listing 3.3 provides the correct implementation, which will make the test pass. Note that two private fields are added to the Button class, namely currentState and previousState.

Listing 3.3: A minimal implementation of getState() to pass the test.

```
1  void Button::setCurrentState(ButtonState current)
2  {
3     currentState = current;
4  }
5
6  ButtonState Button::getState()
7  {
8     ButtonState output = RELEASED;
9     if(currentState == PRESSED && previousState == RELEASED)
10       output = PRESSED;
11    if(currentState == PRESSED && previousState == PRESSED)
12       output= REPEAT;
13    previousState = currentState;
14    return output;
15 }
```

Now the test passes, yet the software button is not dealing with any hardware at the moment. Namely, the currentState is not fetched from the hardware register. Listing 3.4 adds a private method, which will fetch the currentState value. This can be a simple accessor for testing purposes, since in that case the currentState will be set by the setCurrentState method.

Listing 3.4: A small refactoring is in order to allow a future coupling to the hardware.

```
1  ButtonState Button::getCurrentState()
2  {
3     return currentState;
4  }
5
6  ButtonState Button::getState()
7  {
8     ButtonState output = RELEASED;
9     currentState = getCurrentState();
10    if(currentState == PRESSED && previousState == RELEASED)
11    /*The rest of getState() remains the same */
12 }
```

This refactoring can be validated by uploading the code to target and run the test. Now, the final step is to actually fetch the value from the hardware register of the port, as shown in listing 3.5. This will of course invalidate the test, so a manual verification of correct behavior is in order.

Listing 3.5: Changing the getCurrenState method to access the hardware register.

```
1  ButtonState Button::getCurrentState()
2  {
3     return (((*((volatile unsigned long *) portAddress)) & (1 << pinNumber))
   ? PRESSED : RELEASED ;
4  }
```

The actual implementation of getCurrentState is irrelevant. However manual replacing this code with the mock implementation of getCurrentState can be regarded as an ill practice. This problem can be solved in a multitude of ways, which will be discussed in section 3.3.2. Still, since automating the test required to develop code which is only loosely coupled to hardware, the need to run tests on target can be reduced even further until code can be tested on host.

3.3 Test on host

Ideally, program code and tests reside in memory of the programmer's development computer. This situation guarantees the fastest feedback cycle in addition to independence of target availability. Furthermore, developing in isolation of target hardware improves modularity between application code and drivers. Finally, as the host system has virtually unlimited resources, a state of the art unit testing framework can be used.

In the *Test on host* strategy, development starts with tests and program code on the host system. However, calls to the effective hardware are irrelevant on the host system. Hence a piece of code replaces the hardware related functions, mocking the expected behavior. This is called a mock, i.e. a fake implementation is provided for testing purposes. A mock represents the developer's assumptions on hardware behavior. Once the developed code is migrated to the effective hardware system, these assumptions can be verified.

3.3.1 Implementation

The *Test on host* strategy typically consists of two build configurations, as shown in figure 3.3. Regardless of the level of abstraction of hardware, the underlying components can be mocked in the host build. This enables the developer to run tests on the host system, regardless of any dependency on the target platform. However, cross-platform issues might arise and these are impossible to detect when no reference build or deployment model is available. Ideally, building effectively for the real target platform will identify these issues. Although, running the cross-compiler or deploying to a development board could already identify some issues before the actual target is available.

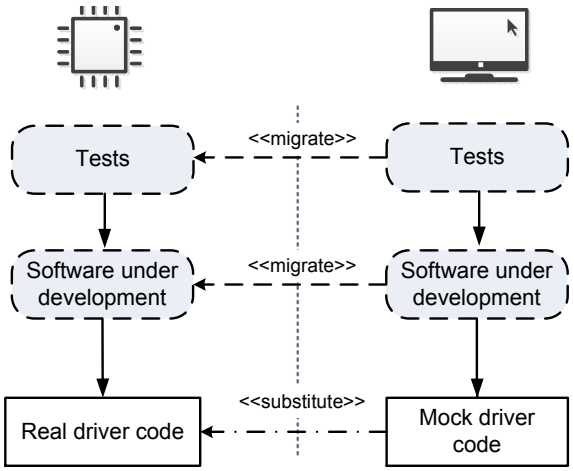

Figure 3.3: Test on host

3.3.2 Mock replacement

Fundamental to *Test on host* is effortlessly switching between host and target build configurations. Therefore hardware mocks must be swapped with the real implementation without breaking the system or performing elaborate actions to setup the switch. Five techniques have been identified, three based upon object oriented principles and three C-based, which facilitate the process.

1) Interface based mock replacement In the interface based design, as shown in figure 3.4,the effective hardware and mock are addressed through a unified abstract class, which forms the interface of the hardware driver. Calls are directed to the interface thus both mock and effective hardware driver provide an implementation. The interface should encompass all methods of the hardware driver to ensure compatibility. Optionally the mock could extend the interface for test modularity purposes. This enables customizing the mock on a test-per-test basis, reducing duplication in the test suite.

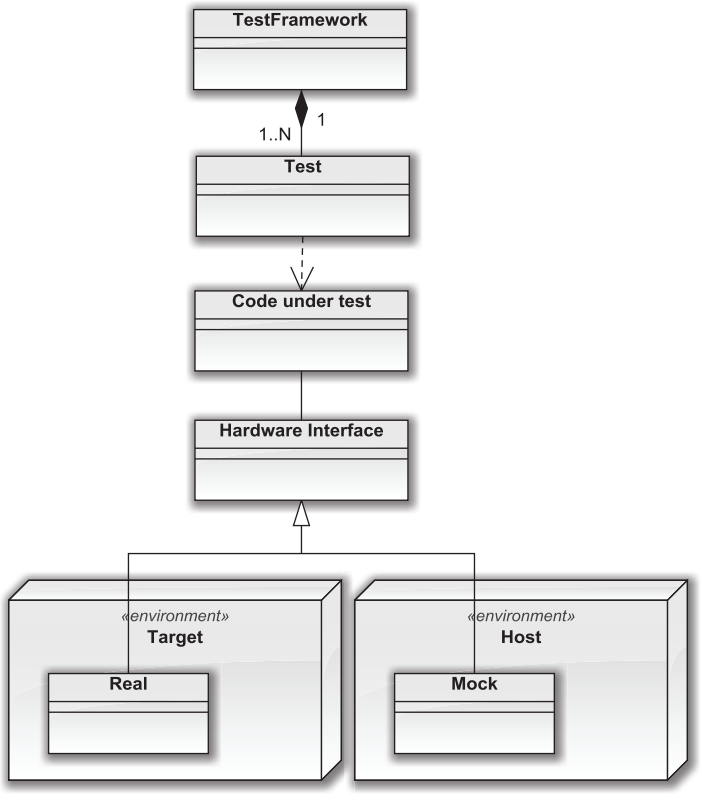

Figure 3.4: UML class diagram of interface based mock replacement

It should be noted that the interface could provide a partial implementation for the hardware independent methods. However, this would indicate that hardware dependencies are mixed with hardware independent logic. In this situation a refactoring is in order to isolate hardware dependent code.

Inheriting from the same interface guarantees compatibility between mock and real hardware driver, as any inconsistency will be detected at compile time. Regarding future changes, extending the real driver should appropriately reflect in the interface.

The main reason of concern with this approach is the introduction of late binding, which inevitably slows down the system in production. However it should be noted that such an indirection is acceptable in most cases.

2) Inheritance based mock replacement Figure 3.5 provides the general principal of inheritance based mock replacement. Basically, the real target driver is directly addressed. Yet as the mock driver inherits from the real target driver, it is possible to switch them according to their respective environment. However it requires that all hardware related methods are identified, at least given protected member access and are declared as virtual. These conditions allow overriding hardware related methods with a mock implementation on host.

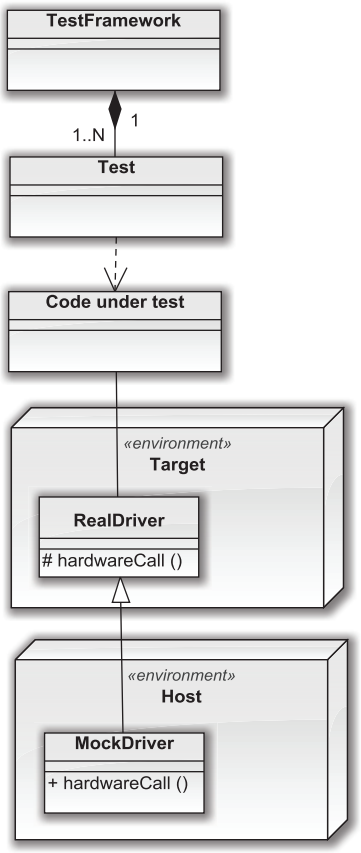

Figure 3.5: UML class diagram of inheritance based mock replacement

However, these issues can be worked around with some macro preprocessing. First, all private members can be adjusted to public access solely for testing purposes. Also the virtual keyword can be removed in the target build, as shown in listing 3.6. Note that guarding the scope of macro definitions with *#undef* is essential for safe usage. Furthermore since these code snippets can be frequently used, injecting them with a file include directive reduces code duplication, which would otherwise obfuscate the business logic.

Listing 3.6: Preprocessor directives to enable inheritance-based mocking without late binding overhead.

```
1  #ifdef TESTONHOST
2  #define private public
3  #else
4  #define virtual
5  #endif
6
7  class RealHardware{
8     private:
9        virtual void hardwareCall();
10 ...
11 };
12
13 #ifdef TESTONHOST
14 #undef private
15 #else
16 #undef virtual
17 #endif
```

Inheritance-based mock introduction is more about managing testability of code than actual testable design. That being said, all overhead of testability can be easily removed in production code. However, ensuring that the macro definitions do not wreck havoc outside the file is fundamental in this approach. Nonetheless, also when dealing with legacy code this approach is preferable. Considering the amount of refactoring, which is necessary to extract the hardware independent interface in the interface-based approach, the adjustments for inheritance-based mock replacement can be introduced without an "unnecessary" layer of indirection.

3) Composition based mock replacement Interface or inheritance-based mock replacement realizes calling the effective methods in a uniform manner. Nevertheless, the correct mock should be installed in the first place, in order to be able to address hardware or mock functionality without overhead of managing dependencies. Therefore a composition based design pattern, called Dependency Injection, allows to manage these dependencies in a flexible manner.

Three different types of Dependency Injection exist. First is constructor injection, in which a member object reference is injected during object construction of the composite object. Listing 3.7 shows a reference design of constructor injection.

Listing 3.7: Constructor injection.

```
1  class ConstructorInjection {
2     public:  ConstructorInjection(HardwareDependency* hw){ m_hw = hw };
3              virtual ~ConstructorInjection();
4     private:    HardwareDependency* m_hw;
5  };
```

Next is setter injection, as shown in listing 3.8, which performs the same operation as constructor injection. Yet, instead of the constructor a setter method is used to register the reference. This introduces the possibility to change the reference at run-time without creating a new composite object. On the other hand, when compared to constructor injection, it requires an additional overhead in test management, namely the call of the setter method itself. Forgetting to do so will lead to incorrect initialized objects under test. Moreover setter injection will introduce additional overhead in the setup of the system in production. However, for instance during a system setup, which is not time-critical, this overhead can be neglected. Yet, considering real-time systems or multiple run-time creation and cleanup of the objects, the overhead becomes critical. Especially when considering resource constrained systems like embedded processors. Therefore a sound advice is to only use setter injection when its flexibility is required and otherwise use constructor injection by default.

Listing 3.8: Setter injection.

```
1 class SetterInjection {
2    public:  SetterInjection();
3             virtual ~SetterInjection();
4             void injectDependency(HardwareDependency* hw){ m_hw = hw };
5    private:    HardwareDependency* m_hw;
6 };
```

Finally, interface injection registers the dependency by inheriting from an abstract class, which contains a setter method. Two approaches can be followed with interface injection. On the one hand, a specific class can be made for each type of objects to be injected. On the other hand, a generic interface class can be provided, which allows injecting objects of all types. Though this mechanism will be discussed in chapter 5.

The previous strategies were based on object-oriented features, however when speed considerations are critical embedded software is written in C. The following techniques do not use OO features, yet allow switching unnoticeable, at least in production, between real and mock code.

4) Link-time based mock replacement First is link-time based mock replacement, also known as link-time polymorphism or link-time substitution. The idea is to provide a single interface of functions in a header file and use the linking script or IDE to indicate which implementation file corresponds to it, i.e. the file containing the actual implementation or a similar file containing the mock implementation. Correspondingly the host build will refer to the mock files and the target build to the real implementation files, as visually represented in figure 3.6.

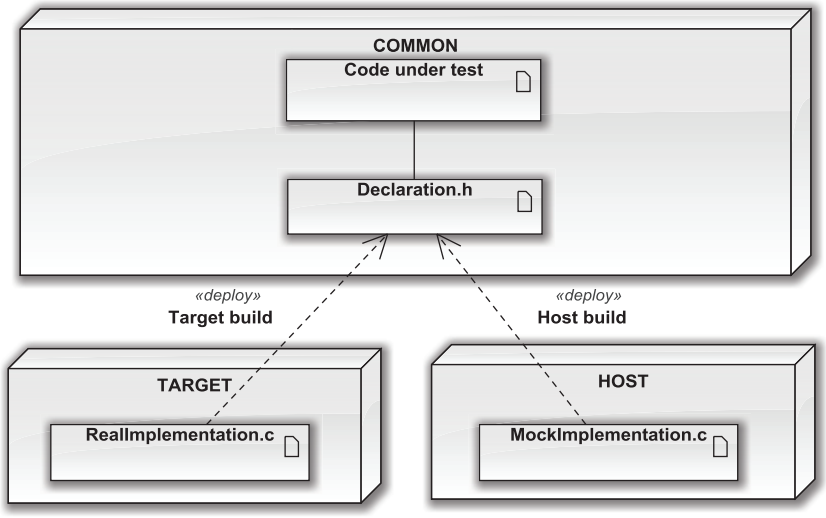

Figure 3.6: Host and target build refer to the mock and real implementation file respectively.

Practically the linker script (or IDE) will refer to three different subfolders. First is the common folder, which contains all platform independent logic as well as header files containing the hardware dependent function declarations. Next is the host folder, which will include the mocks and finally the target folder with the corresponding real implementations. Should a hardware implementation or mock file be missing, the linker will return an error message as a reminder. A practical example of the link-time based configuration is not given, considering the multitude of build systems and IDE's.

5) Macro preprocessed mock replacement While the link-time based macro replacement involves delivering the desired source code files to the linker, the macro preprocessed alternative involves preprocessor directives, which manipulate the source code itself. For instance listing 3.9 provides an almost identical effect to its link-time based alternative.

Listing 3.9: Conditional compilation on a project scope.

```
1 #ifdef TESTONHOST
2 #include "mockdriver.h"
3 #else
4 #include "realdriver.h"
5 #endif
```

However macro replacement allows to intersect inside a source file, as is illustrated by listing 3.10. First, the function call to be mocked is replaced by a new function definition. Also, the testing framework to implement the tests related to the code under test is injected in the file itself.

Listing 3.10: Conditional compilation on a file scope.

```
1  #ifdef TESTONHOST
2  #define functionMock(int arg1, int arg2) function(int arg1, int arg2)
3  void functionMock(int arg1, int arg2) {};
4  #endif
5
6  /* code containing function to be mocked */
7
8  #ifdef TESTONHOST
9  #include "unittestframework.h"
10 int main () {
11 /* run unit tests & report */
12 }
13 #endif
```

Although macros are commonly negatively regarded, the macros shown in the previous two listings are generally safe and will not lead to bugs which are hard to find. However the macro statements will pollute the source code, which leads to less readable and thus less maintainable code. The main advantage of macro preprocessed mock replacement is in dealing with legacy code. Capturing the behavior of legacy code in tests is something that should be done with the least refactoring, because in legacy code, tests are lacking to provide feedback on the safety of the refactoring operations. Using macros effectively allows leaving the production code unchanged, while setting up the necessary tests. Conversely, when developing new applications link-time based mock replacement is preferred, as it does not have any consequences on the production code.

6) Vtable based mock replacement Polymorphism can be obtained in C code by manually building the vtable. However, implementing dynamic dispatch in C is not preferred when comparing it to either the preprocessing or link-time solution. Introducing the vtable in code results in an execution time overhead, which can be critical when considering the typical type of embedded C applications.

Furthermore constructing the vtable in C code is not preferred when the C++ alternative is available. On the one hand, while C++ compilers can do extensive optimization on virtual functions where the actual type can be discovered at compile-time, this cannot be done by a C compiler. On the other hand, there is a manifold overhead in code management to implement the vtable system in C when compared to the native OO solution. In conclusion, the abstraction created by C++ allows to easily forget the overhead introduced by late binding, yet also permits to improve code maintainability.

3.3.3 Process

In order to deal with the slow cycle of uploading embedded program code, executing tests on target and reporting, Test on host is presented as the main solution. In order to do so an assumption is made that any algorithm can be developed and tested in isolation on the host platform. Isolation from hardware-related behavior is critical with the purpose of dynamically delegating the call to the real or mock implementation.

Considering the differences between host and target platform, verification of correspondence between target and host implementation is essential. These differences are:

- Cross-compilation issues, which occur as compilers can generate different machine code from the same source code. Also functions called from libraries for each platform might lead to different results, as there is no guarantee regarding correspondence of both libraries.

- Assumptions on hardware-related behavior. Since the hardware reference is not available on host, the mocks are representing assumptions made on hardware behavior. This requires having an in-depth knowledge of hardware specifications. Furthermore, as the hardware platform for embedded systems can evolve, these specifications are not as solid or verified as is the case with a host system.
- Execution issues concerning the different platforms. These concern the difference in data representation, i.e. little or big endian, word-size, overflow, concurrency memory model, etc., and speed, i.e. memory access times, clocking differences, etc. These issues can only be uncovered when the tests are executed on the target platform.

Test on host is the primary step in the embedded TDD cycle, as shown in figure 3.2. This cycle employs the technique of "dual targeting", which is a combination of Test on host and Test on target. In effect, development in this process is an activity entirely executed according to Test on host, as a reasonable development speed can be achieved. However, in order to cover up for the intrinsic deficiencies of Test on host, Test on target techniques are applied. Specifically, time-intensive activities are executed less frequent, which allows managing the process between development time and verification activities. The embedded TDD cycle proscribes to regularly compile with the target compiler and subsequently solve any cross-compilation issues. Next, automated tests can be ported to the target environment, execute them and solve any problems that arise. Yet, as this is a time-intensive activity it should be executed less frequently. Finally, some manual tests, which are the most labor-intensive, should only be carried out every couple of days.

3.3.4 Code example

In this example a temperature sensor reset method will be developed, which is a one-wire digital temperature sensor. As one-wire implies bidirectional communication on a single wire, the reset command will require that the connected pin direction changes in the process. This is the basis for the following test, as shown in listing 3.11. A problem is encountered since the test is going to run on host. Namely, memory-mapped IO registers are not available and addressing the same memory addresses on host will result in a crashing test. To deal with this problem, an array representing a contiguous chunk of memory can be used. This array has a twofold purpose. On the one hand it encourages the use of relative addressing, which isolates hardware specific memory specifications. On the other hand this array can be addressed by tests to simulate hardware behavior by changing the contents of the array.

Listing 3.11: A test with a mock register to test a temperature sensor initialization.

```
1  TEST(ResetTempSensorTest)
2  {
3     unsigned int IOaddresses [8]; /* IO mapped memory representation on host */
4     unsigned int *IODIRmock;     /* register to mock */
5     IODIRmock = IOaddresses + 7;  /* map it on the desired position in the array */
6     unsigned int pinNumber = 4;
7     *IODIRmock = 0xFF;          /* mock external reset */
8     TemperatureSensor *tempSensor = new TemperatureSensor(IOaddresses, pinNumber);
9     tempSensor->reset();
10    CHECK_EQUAL(*IODIRmock,0xEF); /* test the change of direction */
11 }
```

IODIR is a register, which sets the port direction of the microcontroller. It is part of the general set of IO registers, which are represented on host with the array IOaddresses. With constructor injection, the temperature sensor can be assigned to a specific pin on a port or when the test is executed on host, the mock can be injected that way. After the test is developed, a minimal set of definitions is needed to get through compilation. This listing is omitted, since the definitions can be deducted from the test. Once the compiler has assembled the executable on host, it is run to indicate that the test has a least failed once. Provided that it fails, a minimal implementation can be developed to get a passing test, which has been done in listing 3.12.

Listing 3.12: Temperature sensor constructor injection and reset method implementation.

```
1  TemperatureSensor::TemperatureSensor(unsigned int* port, unsigned int pin)
2  {
3     /* Constructor injection */
4     this->port = port;
5     this->pin = pin;
6  }
7
8  void TemperatureSensor::reset()
9  {
10    unsigned int *IODIR = port + 7;
11    *IODIR |= 1 << pin;
12    *IODIR &= ~(1<<pin);
13 }
```

After running the test again and assuring that it passes, it is time to evaluate if refactoring is in order. In fact, a small amount of duplication has entered between test and driver code. The location of the IODIR register is fixed and could be declared as a global static const variable. Afterwards when more tests have been added to develop a functional reset, a migration to target might be appropriate to verify whether the code effectively runs on target.

This example is an illustration of how TDD influences code quality and low-level design and how Test on host amplifies this effect. As a prerequisite to test on host hardware needs to be loosely coupled. This enables to reuse this code more easily, in case the temperature sensor is placed on a different pin or if the code needs to be migrated to a different platform.

3.4 Remote testing

Test on host in conjunction with Test on target provides a complete development process, in order to successfully apply TDD. Yet, it introduces a significant overhead to maintain two separate builds and to write the hardware mocks. Remote testing is an alternative, which eliminates both of these disadvantages.

Remote testing is based on the principle that tests and code under test do not need to be implemented in the same environment. The main motivation to apply this to embedded software is the observation that TDD requires a significant number of uploads to the target system, i.e. “flashes”.

3.4.1 Implementation

Remote testing is based on the technology of remoting, for instance Remote Procedure Calls (RPC), Remote Method Invocation (RMI), Simple Object Access Protocol (SOAP) or Common Object Re-

quest Broker Architecture (CORBA). Remoting allows executing subroutines in another address space, without the manual intervention of the programmer. When this is applied to TDD for embedded software, remoting allows for tests on host to call the code under test, which is located on the target environment. Subsequently, the results of the subroutine on target are returned to the test on host for evaluation.

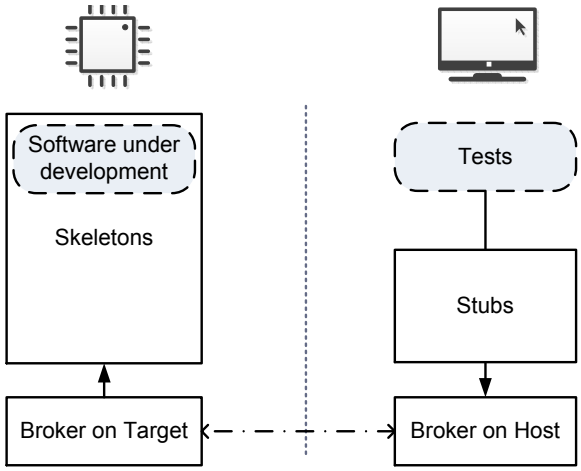

Figure 3.7: Remote testing

Regardless of the specific technology, a broker is required which will setup the necessary infrastructure to support remoting. In homogeneous systems, such as networked computing, the broker on either side is the same. However, because of the specific nature of embedded systems, a fundamental platform difference between the target and the host broker exists.

On the one hand the broker on target has a threefold function. First, it maintains communication between host and target platform on the target side. Next, it contains a list of available subroutines which are remotely addressable. Finally, it keeps a list of references to memory chunks, which were remotely created or are remotely accessible. These chunks are also called skeletons.

On the other hand the broker on host serves a similar, but slightly different function. For one thing it maintains the communication with the target. Also, it tracks the stubs on host, which are interfaces on host corresponding to the skeletons in the target environment. These stubs provide an addressable interface for tests, as if the effective subroutine would be available in the host system. Rather than executing the called function's implementation, a stub merely redirects the call to the target and delivers a return value as should the function have been called locally.

As the testing framework solely exists on the host environment, there is practically no limitation on it. Even the programming language on host can differ completely from the target's programming language. In the spirit of CxxTest[b] a note is made that C++ as a programming language to devise tests might require a larger amount of boilerplate code than strictly necessary. Writing tests in another language is a convenience which can be exploited with Remote testing.

Unfortunately, the use of remoting technology introduces an overhead into software development. Setting up broker infrastructure and ensuring subroutines are remotely accessible requires a couple of additional actions. On target, a subroutine must be extended to support a remote invocation mechanism called marshaling. This mechanism will allow the broker to invoke the subroutine when a call from host "marshals" such an action. Correspondingly on host, an interface must be written which is effectively identical to the interface on target. Invoking the subroutine on host will marshal the request, thus triggering the subroutine on target, barring communication issues between host and target.

[b] CxxTest[2] is a C++ testing framework, which was written in Python.

Some remoting technologies, for instance CORBA, incorporate the use of an Interface Description Language (IDL). An IDL allows defining an interface in a language neutral manner to bridge the gap between otherwise incompatible platforms. On its own the IDL does not provide added value to remoting. However the specifications describing the interfaces are typically used to automatically generate the correct serialization format. Such a format is used between brokers to manage data and calls. As serialization issues concern the low level mechanics of remoting, an IDL provides a high level format, which relieves some burden of the programmer.

3.4.2 Process

The Remote testing development cycle changes the conventional TDD cycle in the first step. When creating a test, the interface of the called subroutine under test must be remotely defined. This results in the creation of a stub on host which makes the defined interface available on the host platform, while the corresponding skeleton on target must also be created. Subsequent steps are straightforward, following the traditional TDD cycle.

1. Create a test
2. Define an interface on host
   (a) Call the subroutine with test values
   (b) Assert the outcome
   (c) Make it compile
   (d) If the subroutine is newly created: add a corresponding skeleton on target
   (e) Run the test, which should result in a failing test
3. Red bar
   (a) Add an implementation to the target code
   (b) Flash to the target
   (c) Run the test, which should result in a passing test
4. Green bar: either refactor or add a new test

Remote testing only provides a means to eliminate one code upload. In order to deal with the rather low return of investment inherent to Remote testing, an adaption to the process is made, which results in a new process called Remote prototyping.

3.4.3 Code example

TDD requires to write a test first and the Remote testing strategy dictates to do this on host. As shown in listing 3.13, the RemoteTemperatureSensorTest addresses the methods as if the temperature sensor is available in the test execution environment.

Listing 3.13: A typical Remote testing cycle starts with a writing a test on host.

```
1  /*Host*/
2  static Broker broker = new Broker(); /* Initialization of Broker on host*/
3
4  TEST(RemoteTemperatureSensorTest)
5  {
6     TemperatureSensor* TempSensor = new TemperatureSensor(port, pin);
7     TempSensor->reset();
8     CHECK_CLOSE(TempSensor->read(), 42, 10);
9     delete TempSensor;
10 }
```

Next, in order to get the test compiling, a stub must also be defined on host. For this example, the remote infrastructure code is given in listing 3.14. However many remoting technologies permit to automatically generate stubs and skeletons. Moreover, the code related to the remoting process is based upon a very simple broker implementation. Other remoting related code will have a different syntax depending on the library or framework used, yet the principles remain the same.

Listing 3.14: Stub implementation example.

```
1  /*Host*/
2  class TemperatureSensor : public ITemperatureSensor
3  {
4     public:
5        TemperatureSensor(Broker* broker);
6        ~TemperatureSensor();
7        void reset();
8        int read();
9  };
10
11 void TemperatureSensor::reset()
12 {
13    broker.call(id, "reset");
14 }
15
16 int TemperatureSensor::read()
17 {
18    broker.call(id, "read");
19    return broker.intReturn();
20 }
```

The constructor of the stub will keep a reference to the host broker and it will also keep track of the ID received by the target broker. Furthermore each public method of the TemperatureSensor interface, which needs to be remotely available, must also be implemented with a stub implementation. In order to keep track of consistency between interfaces on host and on target, it is possible to define a single abstract class and consequently inherit from it. The stub implementation merely calls and returns, respectively the method and its return value.

Listing 3.15: Skeleton definition example.

```
1  /*Target*/
2  class TemperatureSensor : public IBroker, public ITemperatureSensor
3  {
4     public:
5        TemperatureSensor(int port, int pin);
6        ~TemperatureSensor();
7        void reset();
8        int read();
9
10       string invoke(const char* functionAndParameters);
11 };
```

The TemperatureSensor class on target contains the expected constructor, destructor and called methods, yet also has an invoke method. This method is essential for remote addressing of other methods. The skeleton inherits of the IBroker abstract class, whose interface is provided in listing 3.16.

Listing 3.16: IBroker interface.

```
1  /*Target*/
2  class IBroker
3  {
4     public:
5        int ib_ID;
6        virtual string invoke(const char*) = 0;
7  };
```

Inheriting from the IBroker abstract class provides the skeleton with an ID, which binds instances of skeletons on target and stubs on host with each other. Moreover the virtual invoke method ensures an invoke is provided in every skeleton. While the broker on target will automatically assign an ID, the invoke methods must be implemented.

The final part of setting up the remote infrastructure is registration of the remotely addressable methods in the invoke method of the skeleton. Listing 3.17 applies this to the TemperatureSensor class.

Listing 3.17: Registration of the remotely addressable methods in the invoke method.

```
1  /*Target*/
2  TemperatureSensor::invoke(const char* string)
3  {
4     string result = "";
5     if(ib_parsePrototype(functionName))
6     {
7        if(ib_functionName.compare("reset")==0)
8        {  reset();
9           return ""; }
10       if(ib_functionName.compare("read")==0)
11       {  result = parseReturnInt(read());
12            return result; }
13    }
14 }
```

In the invoke method a parser is called, which has split the stream between the brokers in a return value type, function name and list of arguments. Registration of the methods is in fact a comparison of the function name and dispatching of the required function. If a return value is required the value itself must be parsed again to be sent back to host.

In summary,

1. Write a test
2. Generate or manually write the stub
3. Write the skeleton without an implementation
4. Generate or manually register the remote addressable functions in the invoke method
5. Upload to target
6. If the remote infrastructure is set up correctly, a failing test should be the result

Now, the normal TDD cycle can be resumed.

3.4.4 Remote prototyping

Principally Remote prototyping involves developing code on the host platform, while specific hardware calls can be delegated towards the respective code on target. Addressing the hardware-related subroutines on host, delegating the call to the target and returning values as provided by the subroutine prototype are provided by remoting infrastructure.

Software can be developed on host, as illustrated in figure 3.8, as all hardware functionality is provided in the form of subroutine stubs. These stubs deliver the subroutine definition on the host system while an effective call to the subroutine stub will delegate the call to the target implementation.

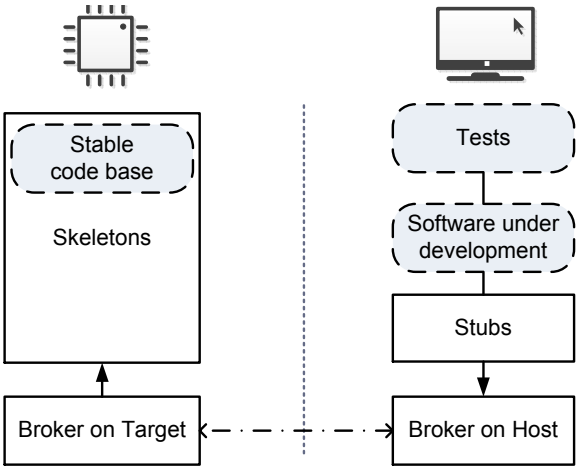

Figure 3.8: Remote prototyping

Remote prototyping is effective under the assumption that software under development is evolving, but once the software has been thoroughly tested, a stable state is reached. As soon as this is the case the code base can be instrumented to be remotely addressable. Subsequently it is programmed into the target system and thoroughly tested again to detect cross-compilation issues. Once these issues have been solved, the new code on target can be remotely addressed with the aim of continuing development on the host system.

An overview of the remote prototyping process applied to an object oriented implementation, for instance C++, is given in figure 3.9. A fundamental difference exists when all objects can be statically allocated or whether dynamic creation of memory objects is required.

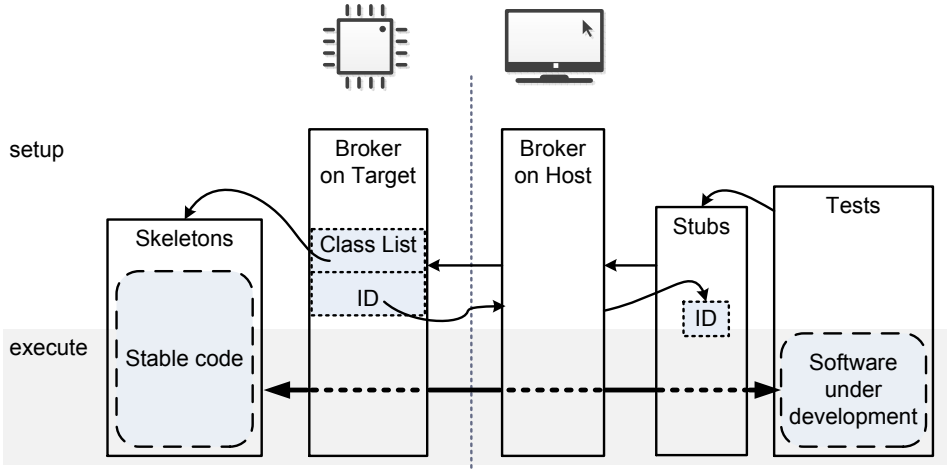

Figure 3.9: Remote prototyping process with dynamically allocated objects

In a configuration in which the target environment can be statically created, setup of the target system can be executed at compile time. The broker system is not involved in constructing the required objects, yet keeps a reference to the statically created objects. Effectively the host system does not need to configure the target system and treats it as a black box. Conversely the process of remote prototyping with dynamic allocation requires additional configuration. Therefore the target system is approached as a glass box system. This incurs an additional overhead for managing the on target components, yet allows dynamically reconfiguring the target system without wasting a program upload

cycle.

The dynamical remote prototyping strategy starts with initializing both the broker as well on target as on host side. Next, a test is executed, which initializes the environment. This involves setting up the desired initial state on the target environment. This is in anticipation of the calls, which the software under development will conduct. For instance, to create an object in the target, the following steps are performed, as illustrated in figure 3.9.

1. The test will call the stub constructor, which provides the same interface as the actual class.
2. The stub delegates the call to the broker on host.
3. The broker on host translates the constructor call in a platform independent command and transmits it to the target broker.
4. The broker on target interprets the command and calls the constructor of the respective skeleton and in the meanwhile assigns an ID to the skeleton reference.
5. This ID is transmitted in an acknowledge message to the broker on host, which assigns the ID to the stub object.

After test setup, the test is effectively executed. Any calls to the hardware are dealt with by the stub object, which are delegated to the effective code on target. Likewise, any return values are delivered to the stub. Optionally another test run can be done without rebooting the target system. A cleanup phase is in order after each test has executed, otherwise the embedded system would eventually run out of memory. Deleting objects on target is as transparent as on host, with the addition that the stub must be cleaned up as well.

Remote prototyping deals with certain constraints inherent to embedded systems. However, some issues can be encountered when implementing and using the Remoting infrastructure.

Embedded constraints The impact, especially considering constrained memory footprint and processing power, of the remoting infrastructure on the embedded system is minimal. Of course it introduces some overhead to systems which do not need to incorporate the infrastructure for application needs. On the other hand remote prototyping enables conducting unit tests with a real target reference. Porting a unit test framework and running the tests in target memory as an alternative will introduce a larger overhead than the remoting infrastructure and lead to unacceptable delays in an iterative development process.

Next, the embedded infrastructure does not always provide all conventional communication peripherals, for instance Ethernet, which could limit remote prototyping applicability. However, if an IDL is used, the effective communication layer is abstracted. Moreover, the minimal specifications needed to setup remote prototyping are limited as throughput is small and no timing constraints need to be met.

Finally, remote prototyping requires that hardware and a minimalistic hardware interfacing is available. This could be an issue when hardware still needs to be developed. Furthermore hardware could be unavailable or deploying code still under development might be potentially dangerous. Lastly, a minimalistic software interface wrapping hardware interaction and implementing the remoting infrastructure is needed to enable remote prototyping. This implies that it is impossible to develop all

firmware according to this principle.

Issues The encountered issues when implementing and using remote prototyping can be classified in three types. First are cross-platform issues related to the heterogeneous architecture. A second concern arises when dynamic memory allocation on the target side is considered. Thirdly, translation of function calls to common architectural independent commands introduces additional issues.

Differences between host and target platform can lead to erratic behavior, such as unexpected overflows or data misrepresentation. These issues could be introduced such as word size, endian ordering or floating point representations. However, most test cases will quickly detect any data misrepresentation issues. Likewise, border problems can be discovered by introducing some boundary condition tests.

Next, on-target memory management is an additional consideration which is a side-effect of remote prototyping. Considering the limited memory available on target and the single instantiation of most driver components, dynamic memory allocation is not desired in embedded software. Yet, remote prototyping requires dynamic memory allocation to allow flexible usage of the target system. This introduces the responsibility to manage memory, namely creation, deletion and avoiding fragmentation. By all means this only affects the development process and unit verification of the system, as in production this flexibility is no longer required.

Finally, timing information between target and host is lost because of the asynchronous communication system, which can be troublesome when dealing with a real-time application. Furthermore to unburden the communication channel, exchanging simple data types are preferred over serializing complex data.

Tests The purpose of Remote prototyping is to introduce a fast feedback cycle in the development of embedded software. Introducing tests can identify execution differences between the host and target platform. In order to do so the code under test needs to be ported from the host system to the target system. By instrumenting code under test, the remote prototyping infrastructure can be reused to execute the tests on host, while delegating the effective calls to the code on target.

3.4.5 Code example

The Remote prototyping example continues on the example of Remote testing (section 3.4.3). Recall that in the previous example a low level driver function TemperatureSensor.read() was the subject of a test and therefore needed to be remotely addressable. In this example the previous functionality is extended with a conversion function to calculate the actual temperature in Celsius. This conversion function is strongly related to the temperature sensor, yet it does not require to be directly involved with the hardware registers. Therefore it is a good example of functionality that can be developed with the Remote prototyping strategy.

As always, a test is a good starting point, for instance the test in listing 3.18.

Listing 3.18: A typical Remote prototyping cycle starts with a writing a test on host.

```
1  /*Host*/
2  static Broker broker = new Broker(); /* Initialization of Broker on host*/
3
4  TEST(RemoteThermometerTest)
5  {
6     Thermometer* thermometer = new Thermometer();
7     /* check if temperature is near room temperature +/- 5 degrees (Celsius)*/
8     CHECK_CLOSE(thermometer.getTemperature(), 20, 5);
9     delete thermometer;
10 }
```

First, the compiler will complain about the non-existent definition of the Thermometer class. Therefore, to satisfy the compiler, see listing 3.19.

Listing 3.19: Only a minimal structure of the class is given.

```
1  /*Host*/
2  class Thermometer
3  {
4     public:
5           Thermometer();
6           virtual ~Thermometer();
7           double getTemperature();
8  };
```

A failing test indicates that the test has at least failed once and allows to proceed to the following step, namely implementation of the code under test (listing 3.20).

Listing 3.20: After the test has failed, a correct implementation of getTemperature() is given.

```
1  /*Host*/
2  Thermometer::Thermometer()
3  {
4     /*tempSensor is a private TemperatureSensor pointer,
5     whose definition has been omitted for brevity*/
6     this->tempSensor = new TemperatureSensor(port, pin);
7  }
8
9  double Thermometer::getTemperature()
10 {
11    tempSensor->reset();
12    int magnitude = tempSensor->read();
13    int sign = tempSensor->read();
14    /* conversion details are omitted */
15    return currentTemperature;
16 }
```

This results in a green bar, which indicates a passing test, thus concluding this example. Effectively this example demonstrates the value of Remote prototyping. All development was on host, even though there were calls on hardware related functions[c] , which is not available on the host system. Most importantly there was no more code needed to run the test on the host system. Granted that work was already done in the Remote testing example, but once the remoting infrastructure is available, developing embedded software becomes as easy as developing PC applications.

3.5 Overview

Test on target, Test on host, Remote testing and Remote prototyping have been defined as strategies to develop in a TDD fashion for embedded. All of these strategies have advantages and disadvantages when a comparison between them is made. Furthermore because of the disadvantages, these strategies excel in a particular embedded environment. In this section a comparison is made between each strategy and an overview is given of how development in a project can be composed of combinations of these strategies.

The baseline of this comparison is Test on target. Namely for the particular reason that when the number of code uploads to target is the only consideration, Test on target is the worst strategy to choose. It is possible to demonstrate this when the classical TDD cycle is considered, as in figure 3.10.

[c] The hardware function calls in example 3.20 are: TemperatureSensor constructor call, TemperatureSensor.read() and TemperatureSensor.reset().

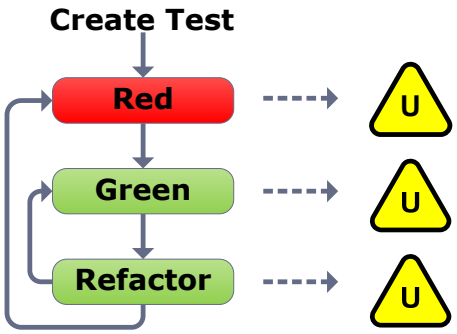

Figure 3.10: Uploads to target when Test on target is considered.

When TDD is strictly applied in Test on target, every step will require a code upload to target. Considering the iterative nature of TDD, each step will be frequently run through. Moreover since a target upload is a very slow act, this will deteriorate the development cycle to a grinding halt. Thus reducing the number of uploads is critical in order to successfully apply TDD for embedded software.

Nonetheless Test on target still has its merit. For instance low level driver code, which closely interacts with the hardware of the embedded system is too convoluted for Test on host mocking and downright impossible for the Remoting strategies.

Remote testing When considering the effect of Remote testing on TDD for embedded software, the following observation can be made. At a minimum with Test on target, each test will require two uploads, i.e. one to prove the test is effectively failing and a second one, which contains the implementation to make the test pass. Note that this is under the assumption that the correct code is immediately implemented and no refactorings are needed. If it takes multiple tries to find the correct implementation or when refactoring the number of flashes rises.

In order to decrease the number of required uploads, tests can be implemented in the host environment, i.e. Remote testing. Effectively this reduces the number of flashes by the number of tests per subroutine minus one. One is subtracted because a new subroutine will require to flash the empty skeleton to the target. Therefore the benefit of remote testing as a way to apply TDD to embedded software is limited, as demonstrated in table 3.1. The ideal case is when a test passes after the first implementation is tried.

	# Tests : # Code uploads : # New remote subroutines	Remote testing
Worst case	1 : 1 : 1	0%
2 TDD cycles (ideal)	2 : 2 : 1	25%
3 TDD cycles (ideal)	3 : 3 : 1	33%
X TDD cycles (ideal)	X : X : 1	Max = 49,99...%
General case	T : C : R	$\frac{T-R}{T+C} * 100\%$

Table 3.1: Remote testing benefit.

Consider that tests have a relative low complexity when compared to production code. This observation implies that a test is less likely to change than the effective code under test, which indicates

an effective reduction of the benefits of remote testing. The possibility of changing code to reach green bar, namely during the implementation or refactoring phase, is higher than the (re)definition of the tests. Effectively this will reduce the ratio of tests versus code under test requiring an update of code on target. When only the number of uploads is taken into account, Remote testing will never be harmful to the development process. Yet considering the higher complexity of production code and refactoring, which mostly involves changing code under test, the benefit of Remote testing diminishes rapidly. When other costs are taken into account, this strategy is suboptimal when compared to Test on host. However, as a pure testing strategy, Remote testing might have its merit. Though the application of Remote testing in this context was not further explored.

Remote prototyping As the effectiveness of Remote testing is limited, an improvement to the process is made when code is also developed on host, i.e. Remote prototyping. Remote prototyping only requires a limited number of remote addressable subroutines to start with. Furthermore, once code under development is stable, its public subroutines can be ported and made remote addressable in turn. This is typically when an attempt can be made to integrate newly developed code into the target system. At that moment these subroutines can be addressed by new code on host, which is of course developed according to the Remote prototyping principle.

Where Remote prototyping is concerned, it is possible to imagine a situation which is in fact in complete accordance to Remote testing. Namely when a new remote subroutine is added on target, this will conform to the idea of executing a test on host, while code under test resides on target. However code which is developed on host will reduce the number of uploads, which would normally be expected in a typical Test on target fashion. Namely each action which would otherwise have provoked an additional upload will add to the obtained benefit of Remote prototyping.

Yet the question remains of how the Remote testing part of the process is related to the added benefit of Remote prototyping. A simple model to express their relation is the relative size of the related source code. Namely on the one hand the number of Lines Of Code (LOC) related to remoting infrastructure added to the LOC of subroutines which were not developed according to Remote prototyping, but rather with Remote testing. On the other hand the LOC of code and tests developed on host. These assumptions will ultimately lead to table 3.2.

In table 3.2 the following symbols are used:

T = # tests
C = # of code uploads
R = # of new remote subroutines
C_D= # of uploads related to remote testing
C_H= # of averted uploads on host
LOC_D= LOC related to Remote testing (Remoting infrastructure + traditionally developed code)
LOC_H= LOC developed according to Remote prototyping
$\alpha = \frac{LOC_D}{LOC_{TOTAL}}$, $\beta = \frac{LOC_H}{LOC_{TOTAL}}$

	# Tests	α	Target (C_D : R)	β	Host (C_H)	Remote prototyping
	1	0	/	1	1	50%
	1	0	/	1	2	66%
Max.	1	0	/	1	Y	Max = 99,99...%
	1	0,75	1 : 1	0,25	1	12,5%
Min.	1	1-β	1 : 1	β	1	Min =$\frac{1}{2}\beta$ * 100%
General case	T	α	C_D : R	β	C_H	$\left(\alpha\frac{T-R}{T+C_D}+\beta\frac{C_H}{T+C_H}\right)*100\%$

Table 3.2: Remote prototyping benefit.

Table 3.2 indicates a net improvement of Remote prototyping when compared with Remote testing. Furthermore it also guarantees an improvement when compared to Test on target. Nevertheless it also shows the necessity of developing code and tests on host as the major benefit is obtained when β and C_H are high when they are compared with respectively α and T.

Test on Host Finally, a comparison can be made with the Test on host strategy. When only uploads to target are considered, Test on host provides the best theoretical maximum performance, as it only requires one upload to the target, i.e. the final one. Of course, this is not a realistic practice and definitely contradicts with the incremental aspect of TDD. Typically a verification upload to target is a recurrent irregular task, executed at the discretion of the programmer. Furthermore Test on host and the remoting strategies have another fundamental difference. Namely while setting up remoting infrastructure is only necessary when a certain subroutine needs to be remotely addressable, Test on host requires a mock. Although there are mocking frameworks which reduce the burden of manually writing mocks, it still requires at least some manual adaptation. When the effort to develop and maintain mocks is ignored, a mathematical expression similar to the previous expressions can be composed as shown in table 3.3. However it should be noted, that this expression does not consider the most important metric for Test on host and is therefore irrelevant[d] .

	# Tests (T)	# Code uploads (C)	# Verification uploads (U)	Test on host
	1	1	1	50%
Min.	1	C	U	Min = 0,0...1%
Max.	T	C	1	Max = 99,99...%
General case	T	C	U	$\left(1-\frac{U}{T+C}\right)*100\%$

Table 3.3: Test on host benefit when only target uploads are considered

[d] This expression is included nevertheless, for the sake of completeness.

In comparison In the previous sections, the only metric which was considered was the number of code uploads. Although this is an important metric to determine which strategy is more effective, there are also other metrics to consider.

First metric is the limited resources of the target system, namely memory footprint and processing power. On the one hand when considering the test on target case, tests and a testing framework will add to the required memory footprint of the program. While on the other hand, the processing power of the target system is also limited, so a great number of tests on target will slow down the execution of the test suite. Another metric to consider are the hardware dependencies, namely how much effort does it require to write tests (and mocks) for hardware-related code? Finally, what is the development overhead required to enable each strategy. For Test on target this is the porting of the testing framework, while Test on host requires the development and maintenance of hardware mocks and finally Remote prototyping requires Remoting infrastructure.

Table 3.4 provides a qualitative overview of the three strategies compared to each other when these four metrics are considered.

	Test on target	Test on host	Remote prototyping
Slow upload	- - -	+++	++
	Test & Program	Test & Program	Broker
	on target	on host	on target
Restricted resources	- - -	+++	+/-
	Target memory	Host memory	Host memory
	& processing power	& processing power	& target processing power
Hardware dependencies	+++	- - -	+/-
	Real hardware	Mock hardware	Intermediate format
Overhead	+	- - -	- -
	Test framework	Mocks	Remoting infrastructure

Table 3.4: Test on target, Test on host and Remote prototyping in comparison

The overview in table 3.4 is flawed as it does not specify the embedded system properties. Yet the range of embedded systems is too extensive to include this information into a decision matrix. For instance, for a large number of embedded systems, resources are not an issue. Therefore Test on target becomes much more preferable. Another example is the case where remoting infrastructure is already available and applying Remote prototyping does not have any overhead at all. Likewise when an application is developed on embedded Linux, one can develop the application on a PC Linux system with only minimal mocking needed, making Test on host the ideal choice. Moreover in this overview no consideration is given to legacy code, yet the incorporation of legacy code will prohibit the use of the Test on host strategy.

When deciding which strategy is preferable, no definite answer can be given. In general, Test on target is less preferred than Test on host and Remote prototyping, while Remote prototyping is strictly better than Remote testing. Yet beyond these statements all comparisons are case-specific. For instance when comparing Test on host versus Remote prototyping, it is impossible to make a sound

decision without considering the embedded target system and the availability of drivers, application software, etc. Rather this matrix is a general guide in the decision process.

3.6 Further reading

The embedded TDD cycle is extensively described by Grenning[33, 34, 35, 36]. Meszaros[47] identifies six different types of mocks or using his nomenclature, test doubles.

- Test dummy: has no actual implementation, but is used as a placeholder to satisfy the linker.
- Test stub: returns a constant value when it is called.
- Test spy: records the parameters passed to it and asserts them. It might optionally return a value.
- Mock object: records the sequence of called methods and asserts on basis of that sequence.
- Fake object: is a partial implementation of the real object.
- Exploding fake: makes the test fail when called.

Fowler[26] gave the three principles of Dependency Injection their respective name. Link-time polymorphism was first identified by Koss and Langr[41]. A reference design and implementation of manually building a vtable in C is provided in chapter 11 of Grenning's book on TDD for embedded C [36].

The CORBA specification[10] is an international standard maintained by the Object Management Group (OMG)[7], a concise overview is given by McHale[46]. CORBA has two embedded profiles, namely CORBA/e Compact and CORBA/e Micro[11]. The first guarantees deterministic real-time execution on high-end embedded platforms, while the latter has been further reduced for resource constrained embedded platforms. LDRA provides a tool, which implements the idea of Remote testing[5]. In a case study on remote testing and prototyping, the mbed platform was used with their RPC library[13].

Chapter 4

Legacy code

> "Modules should be both open and closed."
>
> Bertrand Meyer[48]

The following chapters provide various related topics to Test-Driven Development for embedded software. First, the topic of dealing with legacy code is addressed.

Applying TDD to embedded software development has already been extensively dealt with. However these techniques only described the process starting with a clean slate. In practice, professionally developed code rarely starts anew. Legacy code has many definitions, but in the TDD context it generally means: code that was not developed according to TDD. A less strict definition is code which is not accompanied with a test suite.

The main problem of applying TDD with a legacy code base is the problem to start including tests. Code not developed with unit testing in mind is less testable, namely it likely does not provide clean interfaces of small pieces of code. Rather it is more probable to be convoluted and interweaving a lot of code, which could be separated. Of course legacy code can be refactored to make it more testable, but there are no tests which can indicate whether refactoring did not break any functionality. So, legacy code is best left untouched unless tests can be added to apply refactorings safely. This is a typical circular cause and consequence problem[a], which leads to leaving legacy code mostly untouched.

In order to break the circular reference, tests can be introduced in code seams. The definition of a seam in code is a place in code where it is possible to introduce a test without changing the code itself. There are three types of seams, namely preprocessing, link and object seams. The first is using preprocessor directives, similar to those in listing 3.9 and 3.10. The second is similar to the concept of link-time base mock replacement as described in section 3.3.2. Object seams mostly involve overriding the desired method with another method provided by a test (sub)class, as demonstrated in the inheritance based mock replacement figure 3.5. Seams are closely related to mock replacement techniques, but there is a subtle difference. On the one hand seams are defined as enabling points in code and its three types are merely referring to the time in the software process, namely during preprocessing, linking and run-time[b] . On the other hand mock replacement techniques are code additions which are introduced at the seam.

As mock replacement techniques have already been thoroughly discussed in section 3.3.2, no further elaboration is given on the details of the particular techniques. Therefore the rest of this section deals with the particularities of legacy code in the Test on host/target and remoting strategies. In conclusion an experimental strategy to deal with legacy code is described.

[a] Also known as the chicken or egg problem.

[b] Through dynamic binding.

4.1 Test on target - host

First the principle of dealing with legacy code in Test on host is discussed. However to start it is assumed that no host build is available and that porting a legacy code base to host is not a simple operation. In that case dealing with legacy code starts with adding tests on target to capture and preserve the legacy code's behavior. These tests run on target in order to get meaningful feedback on the state of the legacy system. Ultimately, large refactorings or adding behavior to legacy software will be unmanageable, while developing according to *Test on target*. Conversely, porting the system to host first, introduces the risk of misinterpreting cross-platform issues as detecting an effective on-target bug.

Adding tests prior to refactoring, which enables developing new code, gives an assurance that changing does not break the code. By initiating this cycle on an ad hoc basis, attention is immediately shifted to the regions of code which are likely to change the most in the near future. This leads to the cycle shown in figure 4.1.

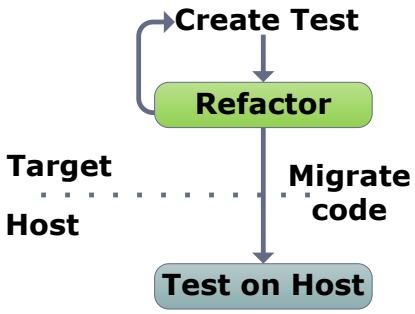

Figure 4.1: Refactoring and migrating legacy software to apply Test on host

In effect, software proven to be reliable and stable in the past is kept unchanged. Still, each refactoring should lead to a state of code, which allows for introducing more tests. Migrating a part of the legacy system to host is only justified to conduct major refactoring operations or to extend its functionality. Once the (sub)system is migrated, *Test on host* is the preferred strategy of developing.

4.2 Remote testing - prototyping

An alternative to gradual migration of legacy code to host is Remote testing. Actually, the relevance of Remote testing is largely attributed to its application in dealing with legacy code. The idea resembles to the conventional Remote testing cycle, yet it starts with instrumenting an existing subroutine with remoting infrastructure. This allows devising a test for the subroutine on host, which in turn enables a safe refactoring phase. Furthermore the principle of remote prototyping can also be applied in a gradual manner, where calls related to legacy software can be marshalled by the remoting infrastructure. A visual representation of this process is shown in figure 4.2.

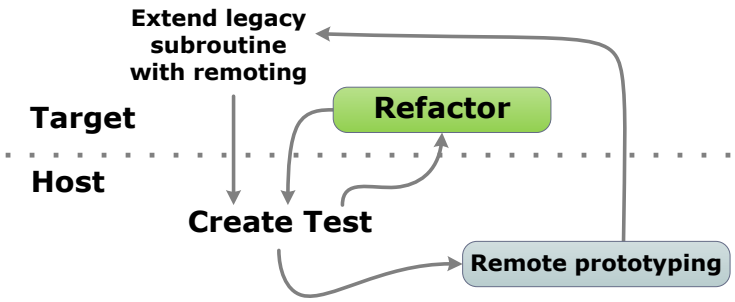

Figure 4.2: Adding remote infrastructure to legacy code to apply Remote prototyping

The resemblance of both practices in dealing with legacy code is striking. The principle idea is to adapt either Test on host in the first case or Remote prototyping in the latter as soon as possible. When evaluating both processes, the time of adopting the desired strategy is a key metric. As sooner this adaption takes place, the more effective the strategy. In this respect, Remote prototyping is superior to Test on host when dealing with legacy code. When the ideal case is considered, Remote prototyping can be immediately applied, whereas Test on host will always require target code refactoring. When a real legacy system is considered this contrast between both strategies is even more substantial. Namely, Remote prototyping can be applied to a single subroutine, while Test on host will require more effort. Either a subroutine is completely isolated on host with mocks or a substantial part of the code needs to be migrated to host.

4.3 Further reading

Feathers [23] wrote the seminal book on the subject of legacy code, which introduces the definition of seams and provides an overview of refactorings, similar to Fowler[25], but applied to legacy code. A paper by Shihab ea.[54] describes a system to prioritize the addition of unit tests in legacy software systems. In their conclusion they state that modification frequency and fix frequency, in the respective code history, along with function size are the most important metrics to write unit tests.

Chapter 5

Patterns

> "Program to an interface, not an implementation"
>
> Erich Gamma ea.[29]

The following sections deal with patterns in general related to Test-Driven Development for embedded software.

5.1 3-tier TDD

In dealing with TDD for embedded software, three levels of difficulty to develop according to TDD are distinguished. Each of these levels imply their specific problems with each TDD4ES strategy. A general distinction is made between hardware-specific, hardware-aware and hardware independent code.

A fine example of hardware independent code is a date/time library, which is generally available in every programming language. Other examples include code to compose and traverse data structures, state machine logic, etc. Regardless of the level of abstraction these parts of code could be principally reused on any platform.

Next is hardware-aware code, which is a high level abstraction of target-specific hardware components. A typical example of a hardware-aware driver is a Temperature sensor module. This does not specify which kind of Temperature sensor is used. It might as well concern a digital or analog temperature sensor. Yet it would not be surprising to expect some sort of getTemperature subroutine. Hardware-aware code will typically offer a high level interface to a hardware component, yet it only presents an abstraction of the component itself, which allows changing the underlying implementation.

Finally hardware-specific code is the code which interacts with the target-specific registers. It is low level driver code, which is dependent on the platform, namely register size, endianness, addressing the specific ports, etc. It fetches and stores data from the registers and delivers or receives data in a human-readable type, for instance string or int. An example of a hardware-specific driver is a 1-Wire driver, which implements the protocol and allows setting or getting a single byte from 1-Wire.

When developing these types of code, it is noted that the difficulty to apply TDD increases, as shown in figure 5.1.

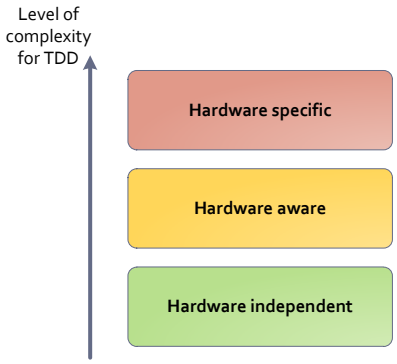

Figure 5.1: Three tiers of TDD for embedded software

5.1.1 Hardware independent code

Typically hardware independent code is the easiest to develop with TDD. Because barring some cross-compilation issues, it can be tested and developed on host (either as Test on host or Remote prototyping). Typical issues that arise when porting hardware independent code to target are:

- Type-size related, like rounding errors or unexpected overflows. Although most of these problems are typically dealt with by redefining all types to a common size across the platforms, it should still be noted that unit tests on host will not detect any anomalies. This is the reason to run tests for hardware independent code.
- Associated with the execution environment. For instance, execution on target might miss deadlines or perform strangely after an incorrect context switch. The target environment is not likely to have the same operating system, provided it has an OS, as the host environment.
- Differences in compiler optimizations. Compilers might have a different effect on the same code, especially when optimizations are considered. Also problems with the volatile keyword can be considered in this category. Running the compiler on host with low and high optimization might catch some additional errors.

Most importantly, developing hardware independent code according to TDD, requires no additional considerations for each of the strategies. Concerning Test on target, the remark remains that all development according to this strategy is painstakingly slow. Furthermore hardware independent code does not impose any limitations on either Remote prototyping or Test on host, so there should be no reason to develop according to Test on target.

5.1.2 Hardware-aware code

By definition hardware-aware code is involved with some part of hardware-specific code. That is to say, it provides a high level interface of low level components, which it conveniently hides from direct addressing. Regardless of the object oriented philosophy or TDD, developing according to this principle provides a loose coupling between control logic, i.e. the hardware independent code and the effective drivers. Therefore changing a driver will not result in a cascade of code changes, rather limiting its effect to the lower tiers. Furthermore it allows calling an abstract interface, which improves code readability and maintainability.

Developing hardware-aware code on host will require a small investment when compared to a traditional development method, because hardware-aware code will typically call a low level driver, which is not available on host. However the benefits of TDD compensate for this investment. The particular investment depends on the strategy used.

On the one hand when developing according to Test on host this investment will be a mock low level driver. The complexity of this mock depends on the expected behavior of the driver. This particular approach has two distinct advantages. First, it allows intercepting expected non-deterministic behavior of the driver, which would otherwise complicate the test. For instance hardware-aware temperature sensor code might effectively call a digital temperature sensor, as shown in listing 5.1. Yet the value it will receive will be dependent on the measured room temperature, which might not be stable. An example implementation is given in listing 5.2.

Listing 5.1: Non-deterministic test with real driver

```
1 /* Non-deterministic test */
2 TEST(getTemperatureTest)
3 {
4    TemperatureSensorDriver realTempSensor;
5    Thermometer t = new Thermometer(realTempSensor);
6    /* check if temperature is near room temperature +/- 5 degrees (Celsius)*/
7    CHECK_CLOSE(t.getTemperature(), 20, 5);
8 }
```

Listing 5.2: Hardware-aware class under test

```
1  class Thermometer
2  {
3     public:
4           Thermometer(TemperatureSensor tempSensor);
5           virtual ~Thermometer();
6           double getTemperature();
7     private:
8           TemperatureSensor tempSensor;
9  };
10
11 Thermometer::Thermometer(TemperatureSensor tempSensor)
12 {
13    /*Constructor injection*/
14    this->tempSensor = tempSensor;
15 }
16
17 double Thermometer::getTemperature()
18 {
19    int rawValue = tempSensor.read()/*Call low level driver*/
20    /* ...
21       Code under test:
22       rawValue conversion to temperatureInCelsius*/
23    return temperatureInCelsius;
24 }
```

However when a mock is called the returned value can be fixed, as shown in listing 5.3.

Listing 5.3: Deterministic test with mock driver

```
1  TEST(getTemperatureTest)
2  {
3     TemperatureSensorMock mockTempSensor;
4     Thermometer t = new Thermometer(mockTempSensor);
5     /* check if the temperature conversion is correct*/
6     CHECK(t.getTemperature(), 20.5);
7  }
8
9  class TemperatureSensorMock : public TemperatureSensor
10 {
11    public:
12          TemperatureSensorMock();
13          virtual ~TemperatureSensorMock();
14          int read();
15 };
16
17 int TemperatureSensorMock::read()
18 {
19    return 0x147F;
20 }
```

The previous code example also provided an indication of the second advantage of using mocks to isolate hardware-aware code for testing purposes. Namely, a consequence of the three-tier architecture is that unit tests for hardware-aware code will typically test from the hardware independent tier. This has two reasons. On the one hand a unit test typically approaches the unit under test as a black box. On the other hand, implementation details of hardware-aware and hardware-specific code are encapsulated, which means only the public interface is available for testing purposes. In order to deal with unit test limitations, breaking encapsulation for testing is not considered as an option. Because it is not only considered as a harmful practice, but is also superfluous as mocks enable testing the man-in-the-middle, also known as hardware-aware code. An overview of this pattern is shown in figure 5.2.

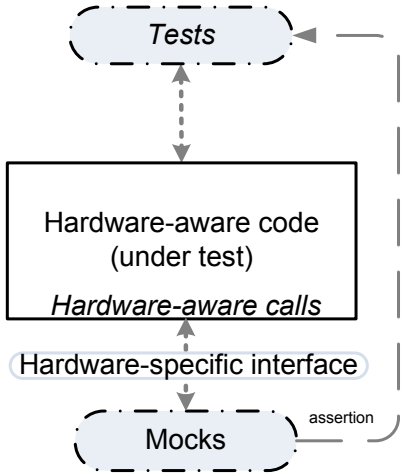

Figure 5.2: Isolating the hardware-aware tier with a mock hardware component.

Fundamental to the idea of using mocks is that the actual assertion logic is provided in the mock instead of in the test. Whereas tests can test the publicly available subroutines, a mock, which is put into the lower tier can assert some internals of the code under test, which would otherwise be inaccessible. Listing 5.4 provides an example.

Listing 5.4: Glass-box testing hardware-aware implementation with a mock

```
1  TEST(TestTempSensorCalls)
2  {
3     TemperatureSensorMock* mockTempSensor = new TemperatureSensorMock();
4     mockTempSensor->expectedCalls(2);  /*prepare the mock what to expect*/
5     Thermometer* thermometer = new Thermometer(mockTempSensor);
6
7     /*method under test which will in turn call the mock methods*/
8     thermometer->getTemperature();
9     CHECK(mockTempSensor->verify());   /*verify the mock*/
10 }
11
12 class Thermometer {
13    public:
14       Thermometer(TemperatureSensor* tempSensor);
15       virtual ~Thermometer();
16       double getTemperature();
17    private:
18       TemperatureSensor* tempSensor;
19 };
20
21 Thermometer::Thermometer()
22 {
23    this->tempSensor = tempSensor;
24 }
25
26 double Thermometer::getTemperature()
27 {
28    tempSensor->reset();
29    tempSensor->write();
30    return tempSensor->read();
31 }
32
33 class TemperatureSensorMock : public TemperatureSensor, public Mock
34 {
35    public:
36          TemperatureSensorMock();
37          virtual ~TemperatureSensorMock();
38          double Read(){ this->record(0); }; /* sets the expected sequence */
39          void Write(){ this->record(1); };
40          void Reset(){ this->record(2); };
41 };
```

In the test of the example 5.4 a mock object is created and instructed what to expect. In this case a sequence of calls is expected, which will be indicated with 2, 1 and 0. Only if the entire sequence is called in that order the test will pass. This is the most used application of mock objects, but it is also possible to register a minimal number of calls on the same subroutine, or even time how long it takes for a subroutine to execute. Once the mock is prepared it is injected with constructor injection in the hardware-specific tier.

Next, the method under test is executed, but rather than check its result, the verify method of the mock is called. This method will return whether or not the sequence of calls was executed in the right order. Note that the mock receives both the interface of the hardware component, in this case the

TemperatureSensor class, and inherits the implementation of the Mock class. The respective hardware component type is inherited so it can be called as if it were a real driver. Whereas the Mock class provides boilerplate code to record and verify calls upon the mock driver. Although multiple inheritance is used in this example, it is by no means a prerequisite to set up mocks. Furthermore as long as the driver interface is an abstract class, any multiple inheritance related diamond problems are avoided.

The other preferred approach to develop hardware-aware code is Remote prototyping. As a strategy Remote prototyping is optimized to deal with hardware-aware code. Namely, developing a part of code which is loosely coupled to a hardware driver only requires the hardware driver functions to be remotely addressable. Once this condition is fulfilled it enables developing the rest of the code on host, without the need of developing mocks.

Yet when considering testing hardware-aware code as a black box, the addition of mocks allowed to test from a bottom-up approach. As Remote prototyping does not require including mocks, it appears to be limited to the typical top-down testing style. To make it worse, injecting mocks with Remote prototyping is a convoluted process, which is not recommended.

Nevertheless mocks, or at least similar functionality, can be introduced in a typical Remote prototyping process. Instead of injecting a mock, the respective stub can be enhanced with the aforementioned assertion logic. This creates a mock/stub hybrid, which one the one hand delegates calls to target and on the other hand records and validates the calls from the code under test. Figure 5.3 presents this mock/stub hybrid in the context of remote prototyping hardware-aware code.

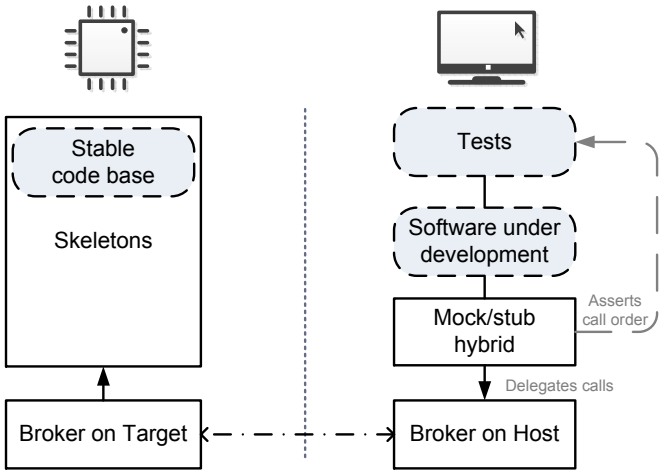

Figure 5.3: Remote prototyping with a mock/stub hybrid, which can assert the call order of the software under test.

A mock/stub hybrid allows to execute a double assertion, namely whether the value is correct and the assertion logic provided by the mock part. Furthermore it can be extended with more advanced mocking capabilities, like raising events, returning fixed data, etc. This counteracts the principle of Remote prototyping of calling the actual code on target, but allows introducing determinism in an otherwise non-deterministic process. For instance, to explore the failure path of a sensor reading in a deterministic way it would be too time-consuming to execute actual sensor readings until a failure has been invoked. Therefore it is easier to mock the failure, which guarantees a failure every time the test is executed.

Listing 5.5 provides an example of a mock/stub hybrid. In this example the read method of a temperature sensor is called by the broker, then it is checked whether the call was successful. In the positive case the call is registered and a fixed value is returned. Otherwise an error value is returned, which ensures the test will fail.

Listing 5.5: Mock/stub hybrid implementation example.

```
1  /*Host*/
2  int TemperatureSensor::read()
3  {
4     broker.call(id, "read");
5     if (broker.intReturn() != -1)
6     {
7        this->record(0);
8        return 40;
9     }
10    return -1;
11 }
```

5.1.3 Hardware-specific code

Hardware-specific code is the most difficult to develop with TDD, as test automation of code which is convoluted with hardware is not easily done. When considering the strategies Test on host and Remote prototyping, each of these has its specific issues. On the one hand, Test on host relies on mocks to obtain hardware abstraction. Although it can be accomplished for hardware-specific code, as demonstrated in listing , developing strictly according to this strategy can be a very time absorbing activity. This would lead to a diminishing return of investment and could downright turn into a loss when compared to traditional development methods. Furthermore as hardware-specific code is the least portable, setting up tests with special directives for either platform could be an answer. However these usually litter the code and are only a suboptimal solution.

Optimally, the amount of hardware-specific code is reduced to a minimum and isolated as much as possible to be called by hardware-aware code. The main idea concerning hardware-specific code development is to develop low-level drivers with a traditional method and test this code afterwards. For both Test on host and Remote prototyping this results in a different development cycle.

Test on host allows the concurrent development of hardware independent and specific code, as shown in figure 5.4. As previously indicated, hardware independent code naturally lends to test first development, while hardware-specific driver code can be tested afterwards. Once a minimal interface and implementation of hardware-specific code is available, hardware-aware code development can be started. Hardware mocks resembling hardware-specific behavior are used to decouple the code from target and enable running the tests on host. Once both hardware-specific and hardware-aware code of the driver has reached a stable state, the hardware-aware part can be migrated to target and instead of the mock the real hardware-specific code can be used. At that moment hardware independent code can be integrated with mocks which provide the hardware-aware interface. Finally all code can be combined on the target system, to perform the system tests.

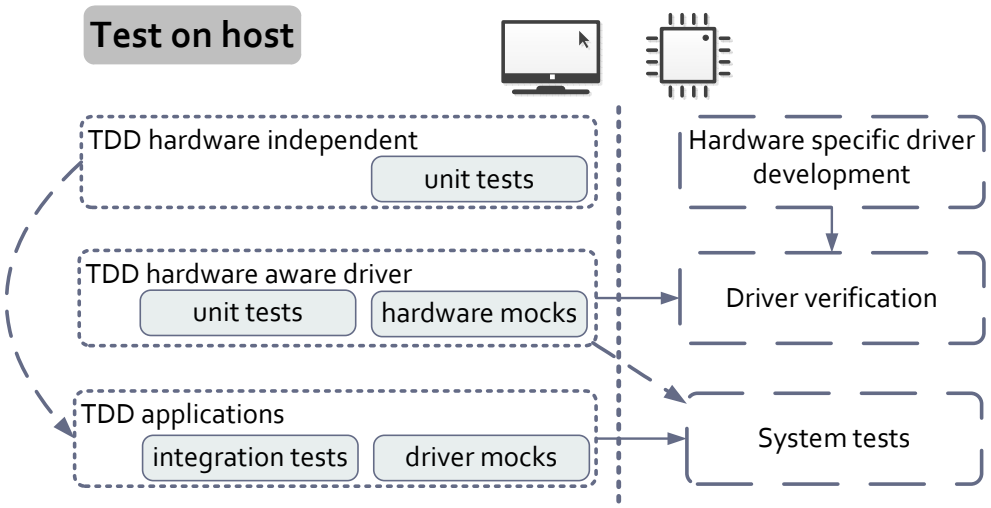

Figure 5.4: 3-tier development process with Test on host

On the other hand the Remote prototyping process starts the same, but differs from Test on host in the later steps. An attempt could be made to mock hardware internals to enable hardware-specific development with Test on Host. However, this is not possible for Remote prototyping as a minimal remote addressable function must be available on target. This naturally leads to a more traditional fashion of hardware-specific code development, yet testing afterwards might as well be according to the principles of Remote testing.

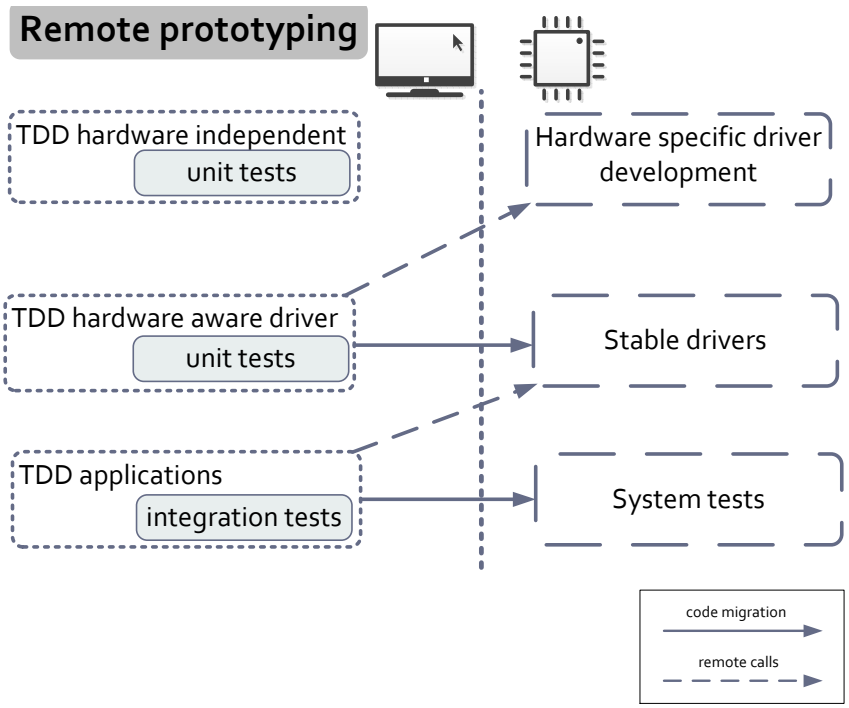

Figure 5.5: 3-tier development process with Remote prototyping

Nevertheless, once a minimal function is available it becomes possible to develop using the Remote prototyping strategy. As indicated in figure 5.5, hardware-aware driver development uses remote calls

to the hardware-specific drivers. The infrastructure for these calls is already present, should the hardware-specific drivers have been tested with Remote testing. Once a stable state has been reached a migration of hardware-aware code is in order to be incorporated in the remote addressable target system. Finally, the applications can be developed in the same fashion. A final migration allows performing system tests on target.

5.2 Testing patterns

As demonstrated in earlier examples, Dependency Injection is a commonly used design pattern for testing purposes. Other patterns can be useful in a testing context. For instance, two classic design patterns prove to be useful in dealing with testability issues raised by the embedded cross-platform development environment.

On the one hand is the Adapter pattern, also known as Wrapper, which is used in two situations. First, it is applied as a class adapter in remoting or mocking instrumentation. In this case unrelated testing concerns need to be provided in a uniform way to the classes under test. Moreover an object adapter can be used to address specific mocked methods when an inheritance based mock principle is used.

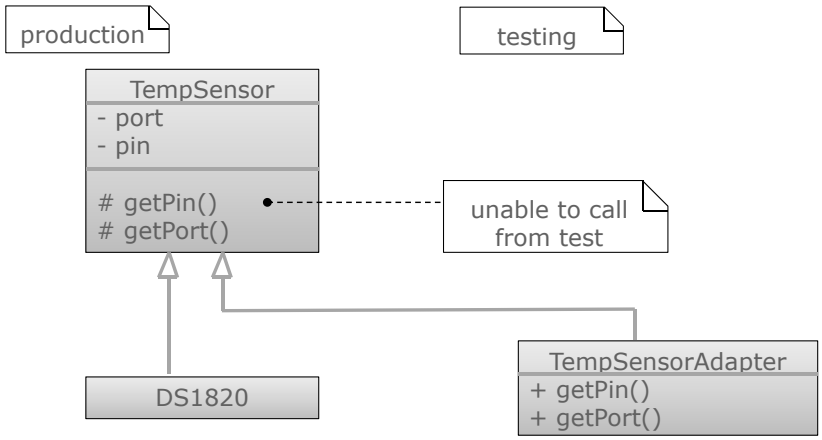

Figure 5.6: A class diagram of an object adapter to solve an access modifier issue.

In figure 5.6, part of a class diagram for a test case is shown in which the particular access modifiers of members of the code under test present problems. Accessing the private methods of a module for testing purposes is a typical case of glass-box testing, yet this is avoided in TDD. However, it is reasonable to exercise a bit more control to deal with the non-deterministic nature of hardware related functions. Nevertheless, accessing specific internal methods leads to brittle tests. Namely, should the specific internal member change afterwards this will typically lead to a broken test, which incurs an additional overhead to fix the test in question. So, a rule of thumb in applying this particular variation of the adapter pattern is to use it only once the code under test has reached a certain stable state.

Another design pattern, which can be used to allow embedded software testing, is the State pattern. This pattern allows an object to change one of its methods if the context state changes. In fact the concrete instance of the object in question is maintained by the context, as this pattern relies on polymorphism. The State pattern is used for the same reason as Dependency Injection, but a fundamental difference exists. Whereas Dependency Injection allows setting a desired composition of objects from the perspective of a test, the State pattern incorporates this capability in production code. Namely, instead of a testing feature, it becomes a feature in production code, which can be exploited for testing purposes.

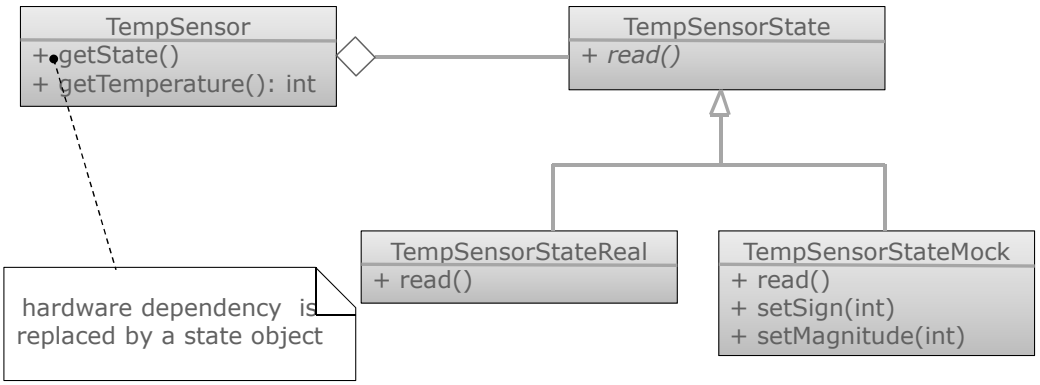

Figure 5.7: A class diagram of the state pattern, which allows more flexibility in testing at the cost of performance.

An example is given in figure 5.7. TempSensor will call the desired read, either the mock or real method, dependent on its state.

5.3 Embedded patterns

This section describes patterns, which provide the solution for specific embedded issues related to TDD. The 3-tier TDD pattern is in fact a specific adaption of the Layered pattern, which proscribes to separate code into a hierarchical organization based on its level of abstraction.

However, embedded issues regarding TDD encompass more than organizational problems. Another important aspect of embedded is its precarious memory management. The memory problems have a twofold classification. Either these problems are related to the limited size of embedded memory. Otherwise embedded programs are expected to run indefinitely or practically whenever the hardware fails. This means that no memory fragmentation is allowed, especially when spare memory is limited. A typical solution to this problem is the Static Allocation pattern, which proscribes to allocate all memory at start-up should the worst case scenario of needed memory fit into the available memory. This solution only suits a subset of embedded programs, mostly embedded control programs which only need a fixed history of events.

Considering the limited memory available on target and the single instantiation of most driver components, dynamic memory allocation is not desired in embedded software. Yet, TDD requires dynamic memory allocation to allow flexible tests on the target system. This introduces the responsibility to manage memory, namely creation, deletion and avoiding fragmentation. By all means this only affects the development process and unit verification of the system, as in production this flexibility is no longer required. However, patterns like Dependency Injection and strategies like Test on host and Remote prototyping will typically influence software design in a drastic measure, that static allocation for production code is no longer an option.

In this case two embedded memory management patterns can be used to deal with this issue.

1. Fixed Sized Buffer deals with memory fragmentation by dynamically allocating memory chunks of a fixed set of sizes. If the required memory chunk sizes are uniformly distributed half of the memory might be wasted when applying this pattern. However this pattern will ensure a program will never crash because of memory fragmentation issues.

2. Garbage Compactor is a garbage collector which solves memory fragmentation problems. Instead of a regular mark and sweep action, the garbage compactor will copy all live objects to another heap. This action can be done atomically, however it requires twice the memory space and also a way to update pointer references while the system is running. As the objects are reallocated,

these can be put in a contiguous region of memory, which solves the fragmentation problem. The major complications with this pattern are the introduction of non-determinism by the compactor interruptions and the computational intensive operation it uses to copy all live objects.

5.4 Further reading

An alternative for 3-tier TDD is the MCH-pattern by Karlesky ea.[24, 39, 40], which is shown in figure 5.8. This pattern is a translation of the MVC pattern[18] to embedded software. It consists of a Model, which presents the internal state of hardware. Next is the Hardware, which presents the drivers. Finally the Conductor contains the control logic, which gets or sets the state of the Model and sends command or receives triggers from the Hardware. As this system is decoupled it is possible to replace each component with a mock for testing purposes.

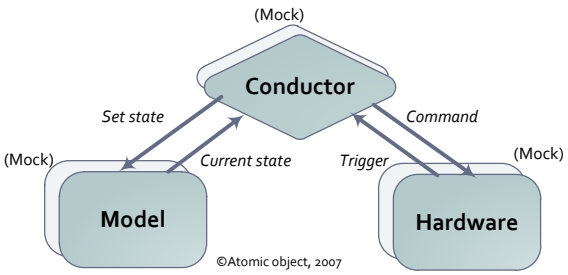

Figure 5.8: MCH pattern

Gamma ea.[29] wrote the first book on design patterns, which is a selection of the most commonly used design patterns, such as Adapter and State. Douglass[22] describes real-time and embedded related patterns, for instance the Layered, Static Allocation, Fixed Sized Buffer and Garbage Compactor patterns. Beck[14] concludes his seminal book on TDD with a number of patterns, which describe practices related to TDD in general. Meszaros[47] provides a pattern language, i.e. strategy patterns, design patterns and coding idioms based on xUnit testing automation.

Chapter 6

In conclusion

> "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."
>
> Edsger Dijkstra[21]

This manual is the transcript of a research project of two years. In this time there were some topics which were identified, but could not have been properly addressed due to time limitations. Section 6.1 provides an overview of these topics. The subsequent section 6.2 gives a synopsis of the most important guidelines to conduct Test-Driven Development for embedded software. Finally a summary of contributions is given in section 6.3, which feature predated or early forms of the concepts in this manual.

6.1 Future work

1) Hardware mocking As briefly indicated in section 2.4 mocks could be partially automatically generated by a mocking framework, which is complementary to a testing framework. No further elaboration is given on the subject, but since hardware mocks are extensively used in the Test on host strategy, a part of the work could be lifted from the programmer.

On the other hand, other techniques to mock hardware could be considered, for instance simulators, emulators or complex hardware models. At first glance these techniques deem to be too complex to be incorporated in a TDD-style of development. However, comprehensive hardware simulators might prove their worth in other stages of software development, namely during integration or specification.

2) Related development strategies Test-Driven Development is a fundamental practice in Agile or eXtreme Programming methodologies. Yet, similar practices exist based on the same principles of early testing. For instance, Behavior-Driven Development (BDD) is an iterative practice where customers can define features in the form of executable scenarios. These scenarios are coupled to the implementation. In turn this can be executed indicating whether the desired functionality has been implemented. BDD for embedded has some very specific issues, since functionality or features in embedded systems is mostly a combination of hardware and software.

3) Code instrumentation for the Remote strategies Developing stubs and skeletons for the Remote strategies requires a considerable effort. However, the boilerplate code which is needed for the remoting infrastructure, could be generated automatically once the interface has been defined on host.

4) Quantitative evaluation of development strategies In an effort to compare the development strategies a qualitative evaluation model has been developed (section 3.5). This model allows conducting quantitative case studies in a uniform and standardized manner. Since the model is a simplified representation of the actual process, it must be validated first. For instance by a number of quantitative case studies, it could be indicated that the model is correct. These would also allow further refinement of the model, so it incorporates additional parameters. Finally an attempt could be made to generalize the models to the development of other types of software.

5) Testing TDD is strongly involved with testing, however as a testing strategy it does not suffice. Real-time execution of embedded software is in some cases a fundamental property of the embedded system. However it is impossible to test this feature while developing, as premature optimization leads to a degenerative development process. Nevertheless another practice from Agile is fundamental in the development process. Continuous Integration (CI) is a process which advocates building a working system as much as possible. Running a suite of automated unit, integration and acceptance tests overnight indicates potential problems. Adding real-time specification tests in a nightly build might be able to detect some issues. However, considering the case of premature optimization, a certain reservation towards the value of these tests on a low level must be regarded.

Testing concurrent software is another issue which cannot be covered by tests devised in a TDD process. As multi-core processors are getting incorporated in embedded systems, these issues will become more important.

6.2 Conclusion

Test-Driven Development has proven to be a viable alternative to traditional development, even for embedded software. Yet a number of considerations have to be made. Most importantly, TDD is a fast cycle, yet embedded software uploads are inherently slow. To deal with this, as shown in the strategies, it is of fundamental importance to develop as much as possible on host. Therefore Remote prototyping or Test on host is preferred. Choosing between the former and the latter is entirely dependent on the target embedded system, tool availability and personal preference. Once the overhead of one of these strategies could be greatly reduced the balance may shift in favor of one or the other. Yet, at the moment of writing, Test on host is the most popular. However Remote prototyping might present a worthy alternative.

Besides Remote testing and prototyping, the main contribution of this manual and the research it describes is 3-tier TDD. This pattern allows isolating hardware and non-deterministic behavior, which are both prerequisites for test automation. This pattern presents a main guideline, which is not only applicable to development with TDD, but generally relevant for all embedded software development. Namely, minimizing the hardware-specific layer improves a modular design, loosely coupled to the hardware system. Such a design is more testable, thus its quality can be assured. Furthermore the software components could be reused over different hardware platforms. This is not only a benefit in the long run, when hardware platform upgrades are to be expected. Moreover, it will help the hardware and software integration phase. In this phase unexpected differences in hardware specifications can be more easily solved in changing the software component. Automated test suites will ensure that changing hardware-specific code to fit the integration does not break any higher-tier functionality. Or at least it will be detected by a red bar.

6.3 Summary of contributions

- J. Boydens. Tdd4es: Testgedreven ontwikkeling van embedded software. Presented at Academia-to-Business forum, DSPValley, 2009.

- J. Boydens. Test driven development: About turning your requirements into executable code. Presented at Technology Seminar: Model Driven vs Test Driven Development for embedded software, 2009.
- J. Boydens. Tdd4es: Test-driven development for embedded software. Presented at Brainmass, Innovatiecentrum West-Vlaanderen, 2009.
- J. Boydens, P. Cordemans and E. Steegmans, Test-Driven Development of Embedded Software. Proceedings of the Fourth European Conference on the Use of Modern Information and Communication Technologies (De Strycker, L., ed.), pp. 117-128, 2010.
- J. Boydens and P. Cordemans. Embedded Software Development Driven by Tests. DSP Valley Newsletter, issue 3, volume 11, pp. 6 - 7, June - July 2010.
- S. Van Landschoot, P. Cordemans and J. Boydens. TDD4ES: Test-Driven Development for Embedded Software. Presented at Academia-to-Business forum, DSPValley, 2010.
- P. Cordemans, S. Van Landschoot and J. Boydens. Migrating from debugging to testing embedded software. Proceedings of the Ninth International Conference and Workshop on Ambient Intelligence and Embedded Systems, 2010.
- P. Cordemans, J. Boydens and S. Van Landschoot. Embedded Test-Driven Development Strategies. Proceedings of the Nineteenth International Scientific and Applied Science conference: Electronics-ET, 2010.
- P. Cordemans, S. Van Landschoot and J. Boydens. Test-Driven Development in the Embedded World. Proceedings of the First Belgium Testing Days conference, 2011.
- P. Cordemans. Test-Driven Development of software for embedded systems. Presented at Onderzoekstreffen, KATHO-KHBO, 2011.
- P. Cordemans, S. Van Landschoot and J. Boydens. Embedded Software Development by means of Remote Prototyping. Proceedings of the Twentieth International Scientific and Applied Science conference: Electronics-ET, 2011.
- J. Boydens. Test-Driven Development in an Embedded Software Environment. Presented at Technology Seminar: Embedded Source Code Quality Control, 2011.

Glossary

This glossary presents the definitions of common terms, regarding technology, in the TDD4ES project. When applicable, a reference is provided to a standard literary article.

A

Acceptance test: Validation test with respect to user needs, requirements, and business processes conducted to determine whether or not to accept the system. [31]

Agile: Software development methodology, based upon four values and twelve principles as described in the Agile Manifesto. [9]

B

Black-box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system. [31]

Broker: A mechanism for invoking operations on a procedure in a remote process. [46]

C

Co-design: Process of enabling concurrent development of hardware and software of an embedded system. [57]

Continuous Integration (CI): Software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. [27]

Co-verification: Process of verifying that embedded system software runs correctly on the hardware design before the design is committed for fabrication. [42]

D

Design pattern: Description of a customizable solution to solve a general design problem in a particular context. [29]

Development board: Printed Circuit Board containing a programmable chip with generic common peripherals.

Driver: Software component directly interfacing with and controlling a hardware peripheral. [42]

E

Embedded system: Combination of hardware and software designed to perform one or a limited set of dedicated functions.

H

Host: Desktop environment to develop embedded software, by means of (partial) simulation of the embedded system. [17]

I

Integration test: Test to expose defects in the interfaces and interaction between integrated components. [31]

Invariant: A condition, which must be satisfied before, during and after the execution of a subroutine. [48]

L

Legacy code: Source code, in which the functionality is desired for continued use, but changes to that source code are hard or next to impossible, because it is a legacy from an older project, third party or older technology.

M

Mock: A fake component in the system that decides whether the unit test has passed or failed. It does so by verifying whether the component under test interacted as expected with the fake component. [51]

Model (ambiguous): (1) Formal: A system description at a higher level of abstraction. [42]

Model (ambiguous): (2) MCH pattern: Representation of the internal state of the hardware. [39]

P

Postcondition: A condition, which must satisfied after the execution of a subroutine. [48]

Precondition: A condition, which must be satisfied before the execution of a subroutine. [48]

Prototype: Experimental version of the hardware of an embedded system. [17]

R

Refactor: Process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. [25]

Regression test: Test to ensure that defects have not been introduced or uncovered in unchanged areas of the software as a result of the changes made. [31]

Remoting: Inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space without the programmer explicitly coding the details for this remote interaction. [12]

S

Scrum: Incremental and iterative agile software development process. [19]

Seam: Place where you can alter the behavior in your program without editing in that place. [23]

Sprint: Thirty-day iteration, resulting in a potentially shippable product increment. [19]

Stub (ambiguous): Replacement of the behavior of a component on which the component under test depends. [31] In this manual we refer to this stub definition as a mock.

Stub (ambiguous): Interface of a component, which uses an inter-process communication mechanism to transmit the call to a broker.[46]

T

Target: The embedded system. [17]

Test-Driven Development (TDD): Write a failing automated test before changing any code. [15]

U

Unit test: Test of a minimal software item that can be tested in isolation. [31]

User story: Description of functionality that is valuable to a user. The details of these stories are covered by tests that can be used to determine whether a story is complete. [19]

V

V-Model: A framework to describe the linear software development life cycle activities from requirements specifications to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development life cycle. [31]

W

Waterfall model: Sequential software development process, where software is first fully designed, then built, tested and finally made ready as a final product. [17]

White-box testing: Testing based on an analysis of the internal structure of a component or system. [31]

X

eXtreme Programming (XP): Discipline of the business of software development that focuses the whole team on common, reachable goals. It defines thirteen primary practices are based upon the values of communication, simplicity, feedback, courage and respect. [15]

Bibliography

[1] Cpputest, http://www.cpputest.org/.

[2] Cxxtest, http://cxxtest.tigris.org/.

[3] Embedded unit testing framework for embedded c, http://embunit.sourceforge.net/embunit/ch01.html.

[4] Googletest, google c++ testing framework, http://code.google.com/p/googletest/.

[5] Host / target testing with the ldra tool suite, http://www.ldra.com/host_trg.asp.

[6] Jtn002 - minunit – a minimal unit testing framework for c, http://www.jera.com/techinfo/jtns/jtn002.html.

[7] Object management group, http://www.omg.org/.

[8] Unity - test framework for c, http://sourceforge.net/apps/trac/unity/wiki.

[9] Manifesto for agile software development, 2001.

[10] Common object request broker architecture (corba) specification, version 3.1, 2008.

[11] Corba/e: industry-standard middleware for distributed real-time and embedded computing, http://www.corba.org/corba-e/corba-e_flyer_v2.pdf, 2008.

[12] Remote procedure call protocol specification version 2, 2009.

[13] Mbed micrcontroller cookbook: Interfacing using rpc, http://mbed.org/cookbook/interfacing-using-rpc, 2011.

[14] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[15] K. Beck and C. Andres. *Extreme Programming Explained*. Addison-Wesley, 2005.

[16] B.W. Boehm. *Software Engineering Economics (Prentice-Hall Advances in Computing Science& Technology Series)*. Prentice Hall PTR, October 1981.

[17] B. Broekman and E. Notenboom. *Testing Embedded Software*. Pearson Education Limited, 2003.

[18] S. Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html, 1992.

[19] M. Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004.

[20] T. DeMarco. *Peopleware: Productive Projects and Teams*. Dorset House, 1999.

[21] E. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[22] B. Powel Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, 2003.

[23] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, 2005.

[24] M. Fletcher, W. Bereza, M. Karlesky, and G. Williams. Evolving into embedded development. In *Agile 2007*, 2007.

[25] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[26] M. Fowler. Inversion of control containers and the dependency injection pattern, http://martinfowler.com/articles/injection.html, 2004.

[27] M. Fowler. Continuous integration. Technical report, ThoughtWorks, 2006.

[28] M. Fowler. Testcancer, http://martinfowler.com/bliki/testcancer.html, 2007.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[30] B. George and L. Williams. A structured experiment of test-driven development. In *Information and Software Technology*, 2004.

[31] D. Graham, E. Van Veenendaal, I. Evans, and R. Black. *Foundations of Software Testing*. Cengage Learning EMEA, 2008.

[32] B. Greene. Using agile testing methods to validate firmware. *Agile Alliance Newsletter*, 4:79–81, 2004.

[33] J. Grenning. Test-driven development for embedded c++ programmers. Technical report, Renaisance Software Consulting, 2002.

[34] J. Grenning. Progress before hardware. *Agile Alliance Newsletter*, 4:74–79, 2004.

[35] J. Grenning. Test driven development for embedded software. In *ESC 241*, 2007.

[36] J. Grenning. *Test-Driven Development for Embedded C*. The Pragmatic Bookshelf, 2011.

[37] P. Hamill. *Unit Test Frameworks*. O'Reilly, 2004.

[38] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 2000.

[39] M. Karlesky, W. Bereza, and C. Erickson. Effective test driven development for embedded software. In *IEEE 2006 Electo/Information Technology Conference*, 2006.

[40] M. Karlesky, W. Bereza, G. Williams, and M. Fletcher. Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *ESC 413*, 2007.

[41] R. Koss and J. Langr. Test driven development in c. In *C/C++ Users Journal*, 2002.

[42] J. Labrosse, J. Ganssle, R. Oshana, C. Walls, K. Curtis, J. Andrews, D. Katz, R. Gentile, K. Hyder, and B. Perrin. *Embedded Software: know it all*. Elsevier, 2008.

[43] N. Llopis. Games from within: Exploring the c++ unit testing framework jungle, http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle.

[44] N. Llopis and C. Nicholson. Unittest++, http://unittest-cpp.sourceforge.net/.

[45] R. Martin. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2008.

[46] C. McHale. *CORBA explained simply, http://www.ciaranmchale.com/corba-explained-simply/*. www.CiaranMcHale.com, 2007.

[47] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[48] B. Meyer. *Object-Oriented software construction second edition*. Prentice Hall PTR, 1997.

[49] R. Mitchell and J. McKim. *Design by Contract, by Example*. Addison-Wesley, 2002.

[50] N. Nagappan, M. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13:289–302, 2008.

[51] R. Osherove. *The Art of Unit Testing*. Manning, 2009.

[52] N. Van Schooenderwoert. Embedded extreme programming: An experience report. In *ESC Boston*, 2004.

[53] N. Van Schooenderwoert. Embedded agile: A case study in numbers. In *Embedded Systems Conference Boston*, 2006.

[54] E. Shihab, Z. Jiang, B. Adams, and A. Hassan. Prioritizing the creation of unit tests in legacy software systems. In *Softw. Pract. Exper. 2011*. Wiley Online Library, 2011.

[55] J. Shore and S. Warden. *The Art of Agile Development*. O'Reilly, 2007.

[56] M. Siniaalto. Test driven development: empirical body of evidence. Technical report, ITEA, 2006.

[57] F. Vahid and T. Givargis. *Embedded system design: A Unified Hardware/Software Introduction*. John Wiley & Sons, 2002.