Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Software Test Automation for IoT Applications – Design and Implementation of a Test Framework for the Digital Concepts Gateway

Aanal Raj Basaula

Course of Study:	Infotech
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dipl.-Inf. Claus Engelhardt, Jonas Fritzsch, M.Sc.
Commenced:	November 2, 2018
Completed:	May 2, 2019

Abstract

The popularity of Smart Devices are increasing everyday, with new devices designed and developed to help people with their daily tasks. Smart Home is one such example of amalgamation of devices where different sensors and actuators work together to help users perform day-to-day activities. This wide-spread adoption of such devices necessitates quality and reliability. This work focuses in the field of IoT devices for methods and measures that are available to test such devices. It presents Digital Concepts gateway as a case study and explores the findings of the special needs of testing within the context of this project. A test framework is proposed to tackle the discovered requirements of testing. An automated test environment is designed and implemented to further improve the quality of the software and an evaluation based on the effort required to maintain and execute tests is used as a decision making tool for Digital Concepts.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CoAP Constrained Application Protocol. 19

CPS Cyber-Physical-Systems. 15

DC Digital Concepts. 18

DCGW Digital Concepts Gateway. 16

HAP HomeKit Accessory Protocol. 29

IoT Internet of Things. 15

REST Representational State Transfer. 22

SDLC Software Development Life Cycle. 15

SOA Service Oriented Architecture. 22

SRP Secure Remote Password. 29

SUT System Under Test. 24

SWEBOK Software Engineering Body of Knowledge. 23

TMM Test Maturity Model. 33

TMS Ticket Management System. 38

1 Introduction

Software Testing as defined by IEEE is, “An activity in which a system or component is executed under specified conditions, the results are observed or recorded and an evaluation is made of some aspect of the system or component”. Software Testing is and has been a major phase in the Software Development Life Cycle (SDLC) and has stayed at around 50% of the total cost as well as the total time required in any SDLC since decades [SBM12]. It was initially applied as “acceptance testing” for quality assurance at the end of the life cycle. But since then due to the rise in importance of early software defect detection has given birth to other types of testing such as unit and integration testing.

Software Testing is an effort intensive task in terms of cost and time required, but, contributes to the overall quality and reliability by verifying that the software operates as expected and by detecting as much defects as possible [SBM12]. To execute tests faster and without extensive human effort, the concept of test automation was introduced at around 1990 [GE17]. It aims at execution of test cases in a uniform manner without human interaction to avoid human error and to deliver test results faster. Through the years we have learned that for pure software solutions, test automation has a high initial setup cost (around 65% more than manual testing) but speeds up the test execution time by a factor close to 18.7 [Gal17]. This has proven to be beneficial for many companies to maintain quality of their software solutions.

1.1 Motivation

Software was previously considered to be an abstract information processing tool, which is a view that is changing mainly due to the growing popularity of Cyber-Physical-Systems (CPS) [SW18]. CPS can be defined as software controlled physical processes and can range from a simple household appliance controlled by an embedded chip to a complete manufacturing process supervised by a computer. Internet of Things (IoT) is defined by ITU-T as, “A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies” [Int12]. Serpanos and Wolf in [SW18] mention that CPS was one of the three major technologies that contributed to the rise in IoT. Some examples of safety critical CPS include self-driving cars, door locks for smart homes. These examples alone emphasize the importance of thorough testing of equipments before public availability.

Testing of IoT Systems is an involved process as compared to pure software solutions, [RWBO15] mainly due to the reasons mentioned below.

First, the limitation of resource for the deployment of complete test suites and high degree of dependence with hardware [RWBO15].

A **Second** factor is that verification of IoT devices using a test-bed has only the possibility of testing a fixed number of devices as planned by the tester, which becomes a challenge in terms of maintenance and scalability when the number of devices and heterogeneous functionality increases. Contrary to the previous approach of using a test-bed, the production testing by deploying the system in a real physical environment is dangerous and should be avoided [Ree16].

Third, most of the failure scenarios occur over an extended period of time, maybe years or more. Among these scenarios many are not widespread through out the target environment. As an example, some places may have higher levels of humidity than the rest of the deployment zone. This requires that the tests are emulated in a shortened period of time to cover as much of the scenarios to find bugs within the software [Ree16].

As mentioned by Reetz in [Ree16], to be able to emulate various conditions during the lifetime of an IoT device, well defined tools and techniques need to be applied. But these tools and techniques are not readily available for IoT testing despite the increasing market adoption and higher risk of physical damage.

Digital Concepts Gateway (DCGW) is one such IoT device which is gaining popularity in the market of around 150 gateways per month. It is a Smart Home appliance and helps connect EnOcean Smart Devices to various Smart Home services such as, Apple HomeKit or Google Home. At present, most of the IoT systems are a controlled environment differentiating between manufacturers and are designed to solve a specific problem while meeting certain criteria and are tested manually [TRMB12]. But DCGW is a service rather than just a solution.

It can be used by any third-party applications to control EnOcean devices via the exposed API. Within this context, the problem still exists that the hardware and software for the DCGW should be tested thoroughly. It is still to be explored whether pure software-based test techniques can be utilized for testing IoT systems and whether the complexity in testing IoT systems can be reduced.

1.2 Research Objectives and Questions

State of the art for software testing is getting sophisticated with novel ideas being presented by top minds all over the globe, but IoT-based service testing requires a domain specific test paradigm [Ree16]. The main goal of this thesis is to survey the possibilities of testing in the IoT subdomain, including the relevant test metrics, the types and methods of testing by considering DCGW as a case study subject. To check whether the testing can be performed without physical actuations to fortify software quality and to evaluate the possibility of emulations to stimulate boundary scenarios.

Test automation is one of the novel approaches that has been discovered to be advantageous for purely software-based systems. It has been observed that when the number of test runs in general are greater than 2 times, then only is test automation advantageous in terms of cost and time. It has also been duly noted that in a normal IT project, the average number of regression tests during development stage is 4 [Gal17]. This clearly shows that test automation reduces the cost and time required for testing software despite the high initial setup cost. Automation of tests for IoT is difficult due to the dependence on physical systems. The thesis aims to check for possibilities of test automation within the scope of DCGW and to validate the proposed testing methods using a test automation framework.

The Thesis expects to answer the below mentioned questions in a systematic approach:

1. What are the peculiarities of Software Testing in the context of developing IoT applications? What are the most relevant test types, methods and techniques?
2. To which degree can test execution be automated for IoT devices? What are appropriate techniques and tools for this purpose, in particular for the Digital Concepts Gateway?
3. Can quality and efficiency of testing be increased by using automated test procedures? Is test automation a feasible option for DCGW from an economic perspective?

1.3 Research Methodology

The thesis consists of two broad sections, theoretical and practical. The theoretical part focuses on information related to basic topics on software testing, test automation or IoT. Multiple research papers have been evaluated to synthesize the theory presented [CH11] and as well to gather information of possible test techniques and methods used for IoT. The practical part employs results gathered from various research papers for DCGW testing to provide a guideline for test techniques and evaluation as part of the Design Science Paradigm [HMPR04].

Methodology 1: Existing works on the topics such as software testing, test automation and IoT testing is gathered to familiarize on the basics of testing. It is then followed by literature review [CH11] to evaluate the peculiarities of testing in the IoT domain and any techniques that can be reused from pure software-based system testing.

- Collection of various published contents to determine existing testing techniques for software systems and to check the relevance of these techniques in terms of testing for IoT
- Determination of initial state of DCGW testing by conducting interviews [KC07] with team members and management
- Evaluation of various metrics that are relevant for testing and quality management of IoT devices in consideration to the requirement and available technology at Digital Concepts GmBH. Evaluating the existing testing techniques for possibility of re-utilization and applicability of automation
- Formulation of relevant test types, testing methods considering the product and company requirements. Determination of automation possibilities for the DCGW

Methodology 2: Initial conceptualization of manual tests followed by automation of relevant tests by generating an artefact [PRTV12] of the Test framework. Clarification of the second research question by determining which tests are possible to be automated and which tests are not. Prototyping [HMPR04] is used as a proof of concept for the test automation framework.

- Implementation of the test framework as method verification of the hypothesis that testing IoT devices is possible without physical interactions
- Validation of test automation possibilities of IoT devices within the scope of DCGW

Methodology 3: Evaluation of the Framework as a Case Study [PRTV12] based on different Illustrative Scenarios using empirical estimations to verify the quality and efficiency of testing using automated methods. Interviews of Testers and developers to compile the nature of Development process and the properties of Manual Testing.

- Definition of a guideline for evaluation of the two test processes
- Generation of a model for effort required while performing manual or automated testing. Estimation of effort in different scenarios for both manual and automated testing
- Evaluation of quality of the product and efficiency of manual testing procedures and compare it to generated prototype

1.4 Thesis Organization

The next chapters of this thesis try to answer the research questions in much detail as possible. The thesis is organized in five chapters covering different parts:

Chapter 2 examines the various concepts in software testing, IoT and any other relevant domains. It presents a generalized IoT architecture that is currently adopted widely in industry and proceeds with a basic background information on software quality and test automation. It tries to summarize the knowledge gathered from multiple different existing papers to answer the first research question.

Chapter 3 on the other hand, focuses on the current state of testing for DCGW. The chapter provides a basic idea about the architecture of the gateway and what is currently being performed to maintain the quality of the product. It further continues to explore the major reasons which necessitate a change in the testing process.

Chapter 4 proposes a test plan that tries to overcome the problems of testing at Digital Concepts (DC). A test architecture is proposed which allows automated as well as manual testing and describes in depth how various specific levels of testing can be performed using this framework.

Chapter 5 is a practical description of the design and development of a prototype allowing automation. It tries to justify the decisions taken and specific implementations made to develop an automated framework from the proposed test architecture.

Chapter 6 provides a general guideline on evaluation of the manual testing and automated testing. It estimates based on different scenarios the effort required for manual testing and automated testing to provide an idea on the benefit of choosing one or the other.

2 Background and Related Work

This chapter focuses on the reviewed works related to software and IoT devices testing. It provides general information on technology stacks that are related to DCGW and points to further references that can be utilized to gather in depth knowledge.

2.1 State of the Art

Testing is an integral part of SDLC and has hence seen many innovative works in academia. A lot of effort has been already put into software testing in topics such as test selection criteria, test oracles, test execution, analysis of test result, etc. Test selection criteria in its early years saw many ideas regarding selection based on code (white box testing), such as statement coverage criteria, branch coverage criteria and path coverage criteria [Ber03] which later on evolved into selection based on specification with the most widely accepted technique being the black box testing [Gal17]. Recent works in software testing suggest the increasing interest towards prioritizing tests using history-based and requirements-based techniques.

Even though software testing has seen many valuable inputs throughout the years, a very limited number of these works relate to the field of IoT. A few related works include protocol testing by PROBE-IT project, interoperability testing of Constrained Application Protocol (CoAP), a few application testing approaches by Diaz et al and model-based testing for IoT service. The dependency on sensors and actuators increases the complexity for testing IoT devices, as a result the current approach is to test the device in real world, or in a test-bed with physical components specifically designed for the purpose [Ree16].

The above-mentioned topics in software testing are at present only used in academia and have not been viewed equally in the software industry [DKPM12], but test automation proves to be widely accepted. It is being actively pursued by many software companies and is also a field of high research interest. This holds true mainly due to the fact that test automation optimizes cost and reduces human effort [Gal17]. Previously test automation meant only the execution of tests without human effort, but now this mindset has shifted to other phases of software testing as well with every research in automatic test case selection or automated test result analysis.

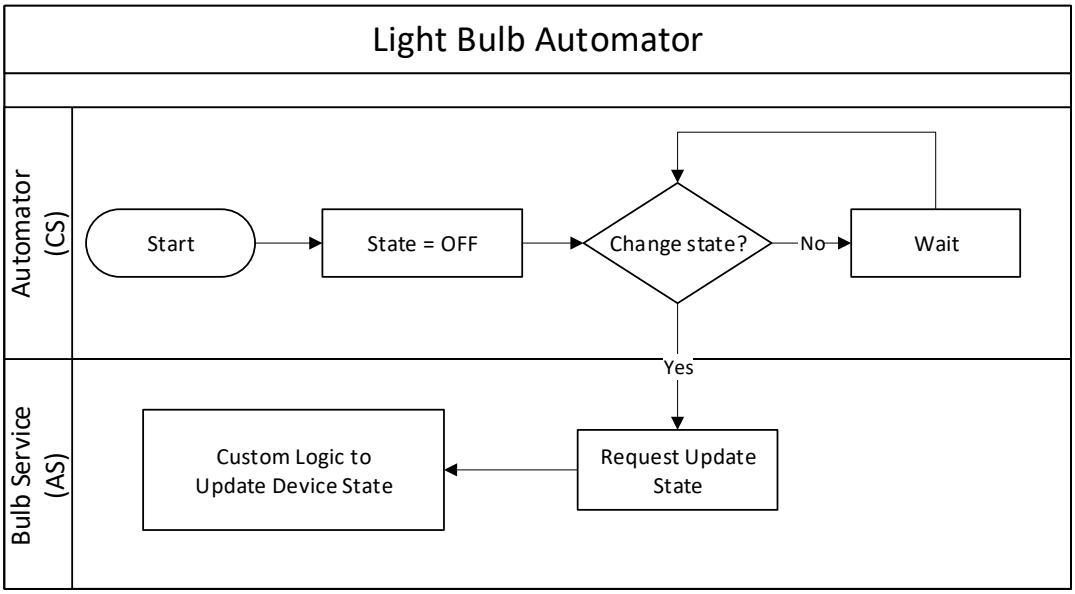


Figure 2.1: Example IoT Service

2.2 Internet of Things

IoT is a general concept for a network of heterogeneous devices able to sense and collect data from around the world and share this data across the network [SR16]. The data gathered can then be used for many common purposes, such as making human livelihood easier and safer and to control the adverse human impact on the environment. IoT is an offspring of various other previous research works [SW18]:

1. **Pervasive Computing**: Providing embedded computational capability to end devices allowing them to efficiently perform tasks and help end user accomplish jobs with ease. Smart refrigerators can be considered as an example of Pervasive Computing, they featured a built in computer allowing users to enter information about the content of their refrigerator for menu planning.
2. **Sensor Networks**: Mainly focuses on data collection at low data rates. Collected data is sent to aggregation points and sent further to a server for analysis. Sensor Networks did not consider on processing within the network.
3. **Embedded Computing**: Designed as either stand-alone devices or tightly coupled networks. Consumer electronics is one of the most prevalent examples of Embedded Computing.

IoT Systems [SW18] are systems that are designed for a finite set of applications rather than being a collection of Internet enabled devices. The DCGW is a part of such IoT system used only for Smart Home applications. Despite the restriction on the application, it still takes into account the dynamics of physical systems that is possible with finite set of devices. As an example, for DC,

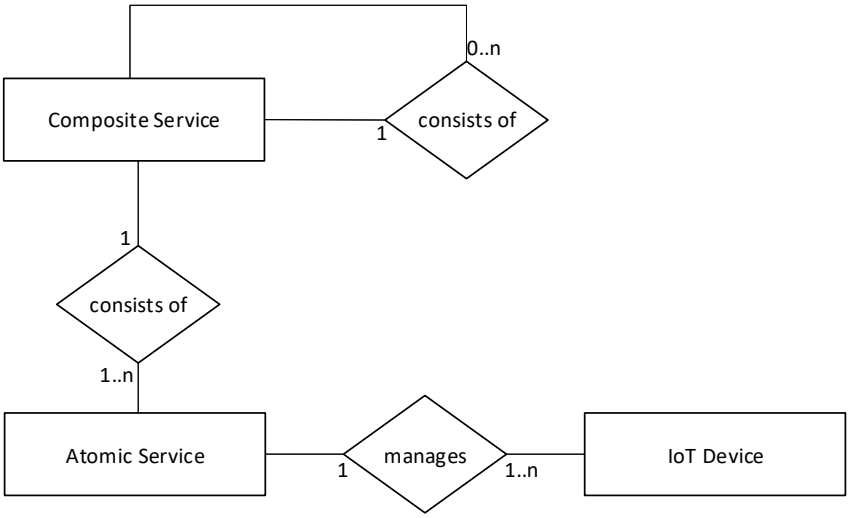


Figure 2.2: Relationship between various types of IoT services

there are only a few category of devices available such as temperature sensors, or smart switches. These categories of sensors can be produced by multiple different vendors and a single client could decide on using multiple instances of the same device or a combination of various devices.

2.2.1 Relevant IoT Systems Technologies

This subsection focuses on various Technologies that are relevant to Digital Concepts Gateways.

EnOcean

EnOcean is a radio technology used for smart home applications and building automation. It focuses on ultra low energy consumption thus allowing harvested energy to operate the devices. The EnOcean community provides its own radio protocol to promote energy saving. The protocol can be compared to the OSI stack with only physical, data link and network layers defined. The most noticeable thing about this technology is that the communication is master/slave configuration.

An EnOcean device can be either uni-directional (only transmit) or bi-directional. The uni-directional devices are mostly sensors and only send data out in certain intervals or in cases of events, whereas actuators listen for requests and send status updates in necessary cases. The sensors are capable of listening to radio signals, but do not use this feature to save energy. They only listen to the radio signals during pairing of the devices and after the pairing process is complete the radio unit is switched off.

The master is responsible for all the slaves connected to it and should listen on the network without using the energy saving mode. The slaves on the other hand are energy restricted and to save power, they may switch off the radio units and postpone communication to a later time. The slaves have to

mandatorily update the master with the current status; thus, many slaves accumulate data/ responses and optimize the open communication channel during status update to push accumulated data/ response. Thus, a single master - slave communication may take from few seconds to multiple minutes. Please refer to EnOcean Protocol Specification for further information.

Apple HomeKit

HomeKit is a software framework by Apple that lets users set up their iOS device to configure, communicate with, and control smart-home appliances. HomeKit specifies a protocol which can be used by a HomeKit appliance to communicate with an iOS device. The protocol works over TCP and Bluetooth Low Energy and allows gateway devices which make it possible for non HomeKit enabled devices to be added into HomeKit. This feature requires that the gateway conforms to the protocol specification for every device type such as light bulbs, switches, etc.

2.2.2 IoT based Service Concept

Service Oriented Architecture (SOA) is a software design paradigm that focuses on separation of varying concerns. This enables larger logic to be constructed using smaller logic blocks. A single block of logic is termed as a service. The SOA paradigm considers construction of complex systems using composition available services. As seen in the example figure 2.1, the automator depends on the bulb service which provides the implementation for changing the state of a light bulb.

SOA allows loosely coupled logic entities; this isolates the service requester from the provider and therefore has no information about the implementation of the service provider. The requester only knows about the implemented interface which shows its capabilities. This type of architecture is readily available in the Internet and has been proven beneficial by allowing efficient parallel development of services, reusability of developed services and simplified maintenance. [Sim06]

On the other hand, research in IoT is in its early stages, therefore standardized architecture approaches are not available. The reference IoT model described by the ITU-T [Int12] and the architectures proposed by various manufacturers [Ree16] follow a generic layered design. It consists of different tiers which are connected by service interfaces. These interfaces abstract the lower layers which keeps the problem simple at a higher level. The resource-constrained objects such as sensors or actuators are abstracted as seen in figure 2.3 by a service interface called the gateway. It hides the implementation and technology used by the resource-constrained objects. White boxes represented in the figure 2.3 are managed resources whereas the grey boxes are the interfaces between objects.

IoT based services utilise interfaces such as CoAP or Representational State Transfer (REST) [Ree16] to encapsulate IoT resources for enhanced usability. Software components that provide information about physical entities or enable the control of devices are termed as resources. These services can be categorized as:

1. **Atomic Service (AS)**: A service which accesses IoT resources via own individual interfaces and radio technologies. These services do not depend on any other services and contain enough logic to operate the physical devices. They also provide standardized interfaces which enables access to IoT devices via the abstracted resources.

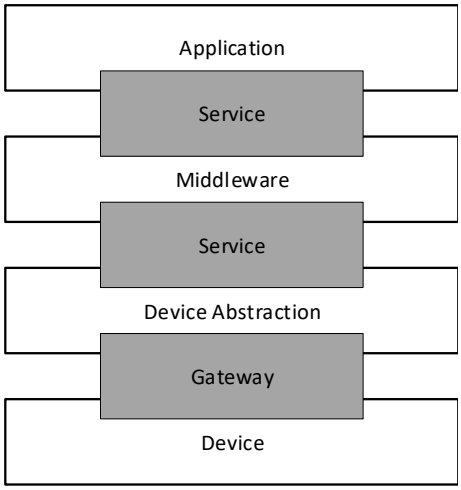


Figure 2.3: Architecture of IoT Applications. Adapted from [Ree16]

2. **Composite Service (CS)**: Services which enable a business process and are composed of various services; AS or CS. Composite Services may utilize other Composite services or the Atomic Services to interact with the physical world. They may as well provide an intricate service to consumers which builds on top of the consumed simpler services.

The DCGW uses EnOcean radio technology to listen to sensors and to operate actuators. An API is provided to interested third-party services to operate these devices. Comparing it with the concept of IoT based services, the gateway can be considered as an Atomic Service and the EnOcean Sensors and Actuators can be considered as the IoT Resources. Third party services that interact with the gateway to operate the EnOcean devices and enable certain business logic are analogous to the Composite Services.

Figure 2.2 shows the relationship between IoT AS, CS and IoT resources as discussed earlier. Any IoT system should contain IoT Resources and should at least have one AS to communicate or control the IoT Resources. It may in addition also contain CS which utilizes the Atomic services or Composite Services to access the IoT resources.

2.3 Software Testing

Testing for a purely software based solution has been started as early as the 1950s and has been improved throughout the years using various testing concepts and techniques [Luo01]. Initially testing was introduced as “Acceptance Testing”; performed at the end of the development cycle right before handing over the software to the customer.

Since then we have made progress and the comprehensive collection of accepted techniques, concepts and knowledge has been gathered in a guide called the Software Engineering Body of Knowledge (SWEBOK) [Ma13]. There have been many publications regarding software testing, focusing on the various concepts, processes and related activities. A similar source of information has not been available for IoT software testing, partly because IoT is a new field and many standardization activities have not been performed in the field [Ree16].

2.3.1 Terminology

The common terminologies for software testing as described in [Ma13] remain tends to stay the same in the context of DCGW testing. These terms are summarized in short below:

Fault vs. Failure

In software testing, a malfunction or an unintended behaviour is termed as a failure. Any malfunction has an underlying cause; this is called the fault [Ma13]. Error on the other hand can be defined as the human mistake that led to the code fault.

Testing for Defect Discovery

A test is termed as successful if it causes the System Under Test (SUT) to fail [SBM12], [Ma13]. This as mentioned by Myers is a necessary mentality to uncover faults within the code.

The Oracle Problem

An oracle is an agent, either human or mechanical and its task is to decide whether the program behaved correct or not. It provides a verdict to the test, either a “Pass” or a “Fail”. Automation of oracles can be difficult and expensive. [Ma13]

Test Adequacy Criteria

It is difficult to describe when a testing process should be concluded.There exists no definitive point that can provide information that the testing uncovers enough defects. As an alternative a predicate is defined which returns a true when a certain target has been met while performing testing. This is called the Test Adequacy Criteria.

2.3.2 Levels of Testing

The various levels of software testing include unit, integration and system test levels. Unit tests target a small section of the software in isolation without considering or by mocking and driving other components. A number of these small tests in the best case cover the complete code base. The expected behaviour of the code block is taken as a reference while writing these tests. Integration Tests verify the integration of two or more components. Creating integration tests include the use of the contract between the combining modules, this asserts that both the components actually respect the contract and they are said to be integrated. System testing on the other hand is performed on the complete product and may not exercise the absolute code base. But system test is not possible without a defined or measurable objective of the product. System testing is useful in also testing non-functional characteristics such as performance, reliability, usability. [Moo11]

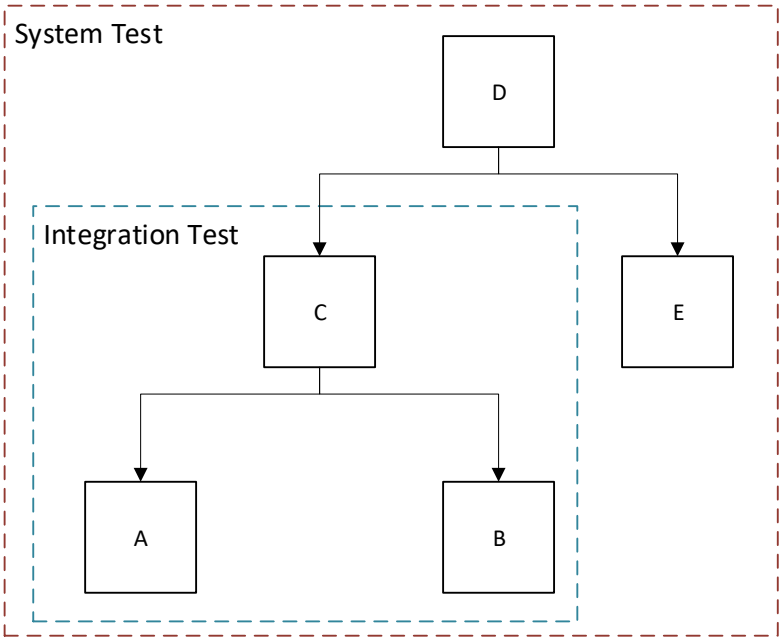


Figure 2.4: Levels of Testing

Figure 2.4 can be considered as an example, the complete software can be represented using sub modules A, B, C, D and E where C utilizes the modules A and B whereas D is composed of C and E making the complete system. Unit testing tests only A, B, C D and E without considering the other modules it depends on. Whereas Integration testing tests two or more interacting modules at once, example C with A or C with B or C with A and B. System test covers the whole test including all the modules and tests on the overall expected behaviour of the whole system.

In comparison to the above levels of testing which has been defined for a complete software system, it is insufficient for Digital Concepts. The gateway as well as third-party services are complete software systems which can be divided into the levels defined above, but the gateway itself is a small part of a bigger system with interacting third-party services, thus giving rise to other higher levels of testing.

2.3.3 Test Objectives

The overall goal of testing is to validate that a software performs as expected. Sandler et al. adds that tests should be focused on breaking the software to detect possible faults in full measure. Once the goal has been defined, it is necessary to decide on a predicate which defines whether the test was satisfied or not is necessary to conclude the testing process. This predicate for a pair <program, test suite> is called the Test Adequacy criteria which when true reflects that the Testing was reliable and that it can be stopped. In case it is false, the Testing should continue until the predicate has been satisfied. This predicate is dependent on the objective of the Test. Some of the test objectives relevant to Digital Concepts are described below:

- **Conformance testing**: Verifying that the system behaviour conforms to the provided specification. The percentage of specification covered can be used as a predicate. The test suite is considered to be adequate if it validates that the software conforms to at least a certain predefined percentage of the specification. This is one of the important goals of testing at Digital Concepts, since the gateway should conform to specifications as defined by EnOcean, Apple and many other parties.
- **Performance testing**: Asserts that the system meets expectations relevant to the performance of the software. Performance can be considered relative to the usage for example, resource utilization such as RAM usage in a memory constrained device, number of packets processed per minute or any criteria that helps describe system performance. Since DCGW is an embedded device with constrained resources such as RAM and storage space, the performance of the gateway in worst case scenario should be measured and it should be maintained that the gateway does not crash in any circumstances.
- **Installation testing**: Validate that the overall installation of the software is running as expected. This is generally important when performing over the air updates to remote gateway nodes. If the updates as well as the update process is not tested before release, the remote gateways can be be placed out of commission which is not a favourable scenario, neither for the customer nor the company.
- **Regression testing**: Verifies that modifications done on the codebase has not changed the overall behaviour of the system. In which case, the test can be considered as complete, if it has been observed that there has been no regressions.
- **Back-to-Back testing**: Verifying that two seemingly equivalent systems provide the same result for the same inputs and conditions.

2.3.4 Testing Strategy

Strategy as defined by Oxford Dictionary is, “A plan of action designed to achieve a long term or overall aim”. Similarly, testing strategy can be described as the overall plan to test the software system. Some of the major strategies include:

- Black Box Testing
- White Box Testing
- Big Bang Testing
- Incremental Testing
- Active Testing
- Passive Testing

Some of the above-mentioned testing strategies are conflicting and cannot be used for the same test, for example black box and white box testing, big bang and incremental testing.

Test classification according to requirements	White box testing	Black box testing
Correctness test	+	+
User Manual test		+
Availability test		+
Reliability test		+
Stress test		+
Software System Security test		+
Usability test		+

Table 2.1: Possible tests using white box vs. black box

Black Box Testing vs. White Box Testing

Black box testing exercises a software without considering the underlying structure of the code. It provides input to the code being tested and validates the output. It does not concern with the code path taken to result in the answer. White box testing on the other hand examines the internal calculation paths in order to identify bugs. The test cases are designed in such a way that the internal code structure is exercised and in general cases tests every statement in the source. Table 2.2 shows a comparison in terms of advantages and disadvantages of choosing white box testing and black box testing.

According to Galin in [Gal17], table 2.1 where white box and black box testing methods are helpful, especially in performing certain types of tests.

Big Bang Testing vs. Incremental Testing

Testing the software as a whole completed package is called “big bang” testing, whereas incremental testing is testing the software in small pieces also called units and then to groups of these smaller units and finally the software as a whole.

Big bang testing has many drawbacks as compared to incremental testing. One of the most important being that the identification of error when a test fails is difficult. Testing the whole software as a completed package, means that a failed test determines only that the software does not perform as it should, it does not point out where the error lies. The defect could lie in the topmost package or even in other depending modules or units.

Incremental testing solves this by individually testing all units separately and the interaction between each module and the whole software package. This helps pinpoint the location of the fault. Due to this reason “big bang” testing has higher cost demand for bug finding and fixing. It also results in a difficulty in planning the bug find, bug fix time requirement [Gal17]. The main disadvantage of Incremental Testing is the additional programming required for preparation of unit and integration testing. They require special entities (discussed later) called the drivers or mocks to be developed for testing the software in isolation.

	White box testing	Black box testing
Advantages	• Checks the correctness of the code paths and is able to check whether algorithms were correctly defined and coded • It is capable of testing quality of coding work	• Relatively lower resources required to perform black box testing • Test automation is easier • Can be used to test all classes of tests
Disadvantages	• Higher resource utilization • Cannot test software in terms of reliability and cannot verify whether all the specification was implemented or not	• Inability to examine the software by checking the code statement by statement • Defects can be hidden due to coincidental errors • Absence of control of line coverage • Impossible to test coding standards

Table 2.2: Advantages and disadvantages of white box and black box testing

Order of Incremental Testing

Incremental testing can still be performed using two different approaches; the bottom-up approach and the top-down approach. In the bottom-up strategy each singular unit is tested first whereas the system as a whole is verified in the end. Top-down approach the system as a whole is tested first and the single units are tested in the end. For the top-down approach, the topmost system component is tested using by using mocks or stubs instead of the actual underlying components. In the bottom-up approach, the units are tested by creating drivers which are fake representations of the higher-level components. The figure 2.4 can be used as an example, where the bottom up approach tests the individual modules A, B and E first and moves up, but the top-down approach tests the module D first and moves downwards.

Active Testing vs. Passive Testing

Active testing is an approach which involves the active stimulation of the SUT. The test execution service provides a carefully selected input and observes the output as generated by the system. It then compares the generated output to the predefined expected result / behaviour. Passive testing

	Active testing	Passive testing
Advantages	• Special input conditions to the system can be achieved easily since the input values to the software under test are provided by the test engine • Testing is faster	• Allows testing of physical media, such as communication channels, physical actuations, etc • Does not invade the system under test and observes special conditions occurring unintentionally
Disadvantages	• Unexpected conditions are not observed since the input conditions are selected by the tester.	• Cannot stimulate the SUT, thus rare conditions may never occur or may take very long time

Table 2.3: Advantages and disadvantages of active and passive testing

on the other hand does not excite the system under test, but monitors it for an extended period of time [CVB+13]. As seen in 2.5 the test system sniffs the ongoing communication and observes the conditions that occur and the output. It then validates whether the result was as expected or not based on the conditions. Table 2.3 shows in brief the various advantages and disadvantages of using active and passive testing.

It is possible for DC to perform passive testing by monitoring all the packets in the test or debug environment. The test system can sniff the EnOcean packets and map the requests to the responses and validate whether the request or response was correct or not. This however gets complicated in Apple HomeKit, specifically because of the encryption mechanism provided by HomeKit.

The communication in HomeKit Accessory Protocol (HAP) uses the Secure Remote Password (SRP) protocol. The algorithm used within SRP protocol allows the server and the client to generate secure tokens without sharing the token via the network. This property of the HAP makes it difficult to sniff the packets and to verify correct network behaviour. A known disadvantage of this method is that it is next to impossible to test boundary cases since the occurrence is rare.

2.3.5 Testing Techniques

In most cases the goal of testing is to break the software in order to make it resilient to malfunctions during extended operations. The tester uses multiple different techniques to come up with test cases which exercises the software. Some of the known categories of techniques are described in short below:

- **Experience-Based**: The test cases are generated based on the intuition and experience of the tester. It is useful for exploratory testing of certain situations that are not easily generated by formalized techniques [Ma13]. An obvious situation that arises while using this technique is the variability of test quality from tester to tester. A known advantage is that the tests are adaptive, in the sense that the tester can dive deeper into situations that cause unexpected behaviour [Moo11].
- **Specification-Based**: The techniques used within this category base themselves in the specification. Some of the techniques include:
  - *Equivalence partitioning*: The domain of input values is divided into subsets that are considered to be equivalent for the property being validated. The partitions are initially created after which one sample value is taken for each partition and the software is tested using these sample input values.
  - *Boundary value analysis*: Test cases are chosen from the boundaries of the input domain on the reasoning that extreme values may generate faulty behaviour.
  - *Random testing*: The tests are generated purely randomly by generating random input from the input domain. The major idea of this technique is to verify the robustness of the system to random input. It is sometimes used as a measure to forecast reliability after deployment [Moo11]. This testing technique can be useful due to the open nature of over the air communication. Many bits of the packet may be randomly affected by the channel errors, which may cause error situations in the gateway.
- **Code-Based**: In contrast to the specification-based testing, the test cases here are generated using the code as a reference. The two techniques available include:
  - *Code flow testing*: This technique uses the code flow to generate test cases, such as using the number of statements executed as a reference to cover the wider code base.
  - *Data flow testing*: This technique uses the data flow within a program to create test cases. In general, it traces the definition and usage of the variables.
- **Nature of application based**: These testing techniques cannot be applied to any application but are only specific to the nature of the application. For instance, API testing, webpage testing, protocol testing, service-oriented software testing is one of the few techniques that can be reapplied for Digital Concepts.

There are many different available techniques that can be used to generate Test Cases and there is no defined rule on how to choose a single technique to generate these test cases. The tester should combine multiple testing techniques to come up with test cases that thoroughly exercises the system under test [Moo11]. An example could be the specification-based and code-based testing. These techniques are complimentary to each other and could be combined together to cover more code base while considering the specification to derive correct behaviour rather than the one defined by the code.

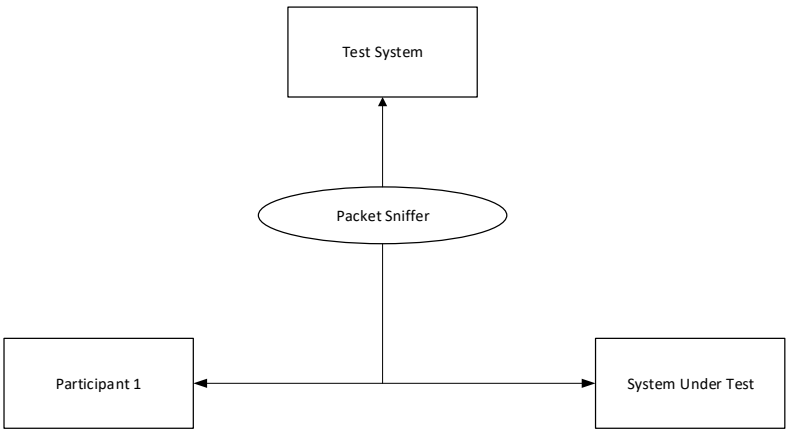


Figure 2.5: Passive Testing Strategy. Adapted from [CVB+13]

2.3.6 Test Oracle

A test activity includes executing a software in the target system and observing the output in comparison to the provided input to the system. A test can only be concluded after an additional mechanism to validate the correctness of the program has been performed using the observed data. Test oracle is an entity that determines whether the executed software was correct or not. It can be defined as a partial function from a test activity sequence to a boolean value, whereas a probabilistic test oracle on the other hand is an oracle which maps test activity sequence to an interval [0, 1] ϵ $\mathbb{R}$ [MHS+14]. A test oracle may be defined for all possible input values or just for a small set of input. There are different major categories of Test Oracles [MHS+14]:

- **Specified test oracles**: Since specifications have an important role on software development, they can be used to create a test oracle. There have been many researches performed over the past 30 years and many methods and formalisms developed. They fall into four major categories: model-based specification languages, state transition systems, assertions and contracts, and algebraic specifications. As the names suggest, model-based languages define models that describe the actual behaviour of the system. Algebraic specifications on the other hand, define algebraic expressions which map the output of the program to the provided input. This can be then used to validate the correctness of the program.
- **Derived test oracles**: A test oracle that has been derived from multiple artefacts such as system executions, documentation or even other versions of the program. This is generally used when no specified test oracle is available. This is a highly useful oracle in the context of Digital Concepts because the current state of the devices has been tested thoroughly and therefore an automated system could verify the outcome of the previous version to the new one. This would at least provide a hint that there has been a discrepancy between the versions inciting further manual testing in the scenario.
- **Implicit test oracles**: An implicit test oracle relies on general, implicit knowledge to distinguish between a system's correct and incorrect behaviour. This includes known facts such as "buffer overflows are errors" or "Array index out of bounds are errors". These oracles do not

need specific knowledge of the domain or any form of specification to be implemented. It is highly system specific; a condition may be abnormal for a system but normal or acceptable for another.

- **Human test oracles**: When no artefacts are present that allow the implementation of automated oracles as mentioned above, Human test oracles is used. As referred to by the name, humans decide whether a test passed or failed. Much effort is placed into reducing the cost of human test oracle by automating as much as possible, specially by finding ways to evaluate test outcomes faster or to write test oracles faster.

2.4 Software Quality

Quality is an ambiguous term which gives a notion about how well the product under consideration functions. It is not a single idea but rather a multidimensional idea and differs when viewed from different perspectives. It can be considered as a multidimensional concept which includes the interest and viewpoint of the person as well as the quality attributes of the product. The relevant views in software quality include [Nai]:

- **Transcendental view**: Quality as a recognizable but difficult to define concept. This is not specific to software quality alone but is also applicable in other fields.
- **User view**: Quality as a fitness for the defined purpose. The key question involved in this is "Does the product satisfy user needs and expectations?".
- **Manufacturing view**: Quality as a conformance to the specification. The quality of level is determined by the extent to which the product meets its specification.
- **Product view**: Internal quality of the product defines the external quality of it as well.
- **Value based view**: Quality in this perspective depends on the amount a customer is willing to pay for it.

A product improvement can only be achieved if the quality can be quantitatively measured [Kan14]. For Digital Concepts, as a manufacturer, it is necessary to have a manufacturing view of quality, measuring the specifications that the gateway conforms to and other details. However, it is also equally necessary to have a user view in terms of usability and reliability.

2.4.1 Methods of Quality Analysis

Testing plays an important part in improving and assessing the quality of the software product. The test - find defect - fix cycle during the development process, improves the overall quality of the product whereas to assess the quality of the complete product system level tests provide the required information. In a broad categorization attempt, the software quality assessment activities can be named:

- **Static analysis**: As the name suggests, the assessment activities is based on the examination of a number of documents, namely requirement documents, source code and other documents without running the product itself. This may include processes such as code reviews, using static code analysis tools, etc.

- **Dynamic analysis**: The product, in our case a Software is executed in order to expose program failures. The program is run typically in a controlled environment and the behaviour is verified to the expected result.

2.4.2 Test Metrics

The phrase as mentioned by Peter Drucker "if you can't measure it, you can't improve it" shines a light on the fact that measurements should be performed in a formalized method to be able to get a meaningful result and to improve it. A measurement in terms of gut feeling is not standard and cannot be quantified, this does not allow proper improvement of a product or process. A quantified metric on the other hand does not vary from person to person and shows whether a product or process has improved or deteriorated since a previous state. Test metrics can fall under three categories depending on what they measure [Kan14]:

- **Product metrics**: As the name describes, they describe characteristics of the product. Depending upon the view they may describe different properties such as professional view describes the product using number of defects per lines of code or mean time to failure of the system or even availability of the system. On the other hand, the user view defines the product using number of problems that the customer faced, satisfaction of the product among users which is generally obtained by performing a customer survey with 5-point scale.
- **Process metrics**: They are useful for describing the software development and maintenance life-cycles. The major goal of these metrics is to understand the processes used in development. Some examples include mean bug response time, mean bug fix effort.
- **Project metrics**: Describes the project categories such as cost of the project, productivity of the developers in a project.

These test measures do not depend on the specific nature of the application, whereas the maturity of testing and business process [DS12]. Furthermore, test metrics can be classified into **base metrics** and **calculated metrics** depending upon the source of the metric. Base metric are the metrics gathered from the source whereas calculated metrics are computed using the base metrics.

2.4.3 Test Maturity Model

Test Maturity Model (TMM) is a set of maturity levels through which an organization is able to progress in order to achieve better testing process [BSC96]. Every maturity level is composed of multiple sub goals that the company should strive for to be able to achieve the respective test maturity level. There are namely 5 levels of maturity:

- Initial
- Phase Definition
- Integration
- Management and Measurement
- Optimization/Defect Prevention and Quality Control

Each maturity level is a type of a control gate, an organization needs to achieve the goals of the current level to be able to quality as the defined maturity level [DS12].

2.5 Test Automation

Test automation in its simple form is the process of executing tests automatically without human effort. It helps decrease the overall production cost by reducing the effort required while performing manual tests. Test automation can also be applied in various other steps of testing such as test case generation where the test framework automatically generates test cases depending upon the defined model.

Test automation should not be mistaken with testing. They are different, in a sense that testing is a skill, which is the ability to write meaningful tests. Test automation on the other hand is generally understood as execution of tests and its reporting using machines. Whether a test is automated or performed manually does not affect its effectiveness or how exemplary it is. [GF99] Automating a test only affects how economic and evolvable it is. Implementation of a test automation system is a difficult task which takes a lot of human effort, which is then later compensated by the fact that consequent test executions do not take a lot of human effort and can use computational power at off hours when the machines generally stay idle.

Due to this, it is generally noted as a tool to improve the quality of the software with efficiency and speed. Hence, relating to only quality and efficiency, but this is not true. Test automation relates to many aspects other than just quality and productivity [Axe18], such as:

- Architecture of the product
- Business process
- Organizational structure
- Even the culture within a company

Which leads to the fact that there exists no general rule for test automation and it needs to be adapted to different organizational needs since different organizations have different types of projects and different types of processes [Axe18]. Therefore, a test automation system for Digital Concepts should be designed according to the product (DCGW), the business process and the organizational structure.

2.5.1 Motivation for Test Automation

Test automation implementation is in general motivated towards reducing manual testing times to shorten the software release cycles. However, this goal is difficult to achieve due to the fact that automated testing cannot be simply used to validate the expected result [Axe18]. In manual testing, a human performs the test who has clear understanding of the software system and can decide whether the expected result was achieved or not, but for a machine, this intuition is not present.

Another good motivation for test automation would be to achieve constant costs during addition of new features. Adding of a new feature increases the complexity of the software which results in an increased cost. Apart from that, the test automation can be utilized to make the running cost of regression test suite which is ever-growing negligible [Axe18].

2.5.2 Manual vs. Automated Testing

The automated tests are executed using the same input and same precision through various executions which cannot be guaranteed using a manual test. The execution of tests by a machine allows skilled testers to [GF99] place effort in other tasks such as generating test cases, maintenance of available test cases. The basic process of test automation includes:

- **Step 1**: Generate test cases for execution: includes generation of test steps, inputs to provide and the expected output
- **Step 2**: Execute the test: exercise the system under test using the provided input and environment setup.
- **Step 3**: Observe the generated output and validate against the expected output.

Manual testing is generally performed by testers who have knowledge of the system and have the ability to decide whether the system behaved in a correct way or not. This is only possible by a system if the Test is precise. However, if the test is precise, it is not maintainable since a minor change in the system behaviour would nullify the precise test case. Since automated tests need to be precise, the test should be as short as possible by testing only a single feature, but manual testing, on the other hand, should focus on testing multiple things on the way since it is time efficient.

2.5.3 Record and Playback

Record and playback is the simplest form of test automation, where the job of the manual tester is automated. This is achieved by recording the manual tester's interaction with the UI or recording network traffic or some other data that indirectly reflects the actions. Theoretically, the recorded interactions are played back automatically and the results are validated.

The recording of UI interactions is possible and easily achievable in this method, but the validation of results is difficult. As an example, the validation for UI testing can be done by comparing the screenshot of the software after replay to the one obtained after manual testing. However, this is difficult due to the resolution differences, the other details that change over time, such as time printed on the UI itself. This could result in false positives.

3 Software Testing for Digital Concepts Gateway

Digital Concepts Gateway can be divided into different layers, an architecture diagram can be seen in Figure 3.1. A core service provides a RESTful API to be consumed by other services (named as connectors) which enables them to operate the EnOcean devices. Some of the major parts of the gateway include:

- **Core**: The Atomic Service that is hardware aware and is the only service capable of managing IoT resources. It contains logic for communication with EnOcean devices and provides a REST API for control of these EnOcean devices.
- **Configuration service**: It is a web application which can be accessed throughout the local intranet and allows the gateway to be configured for personalized requirements. It provides features for learning in of new devices along with some important functionality like turning on or turning off specific services.
- **Connectors**: Apart from the basic services, it runs a special program called the Connector (as per Digital Concepts) which utilizes the core service to communicate with the EnOcean world and provides additional business value to the gateway.

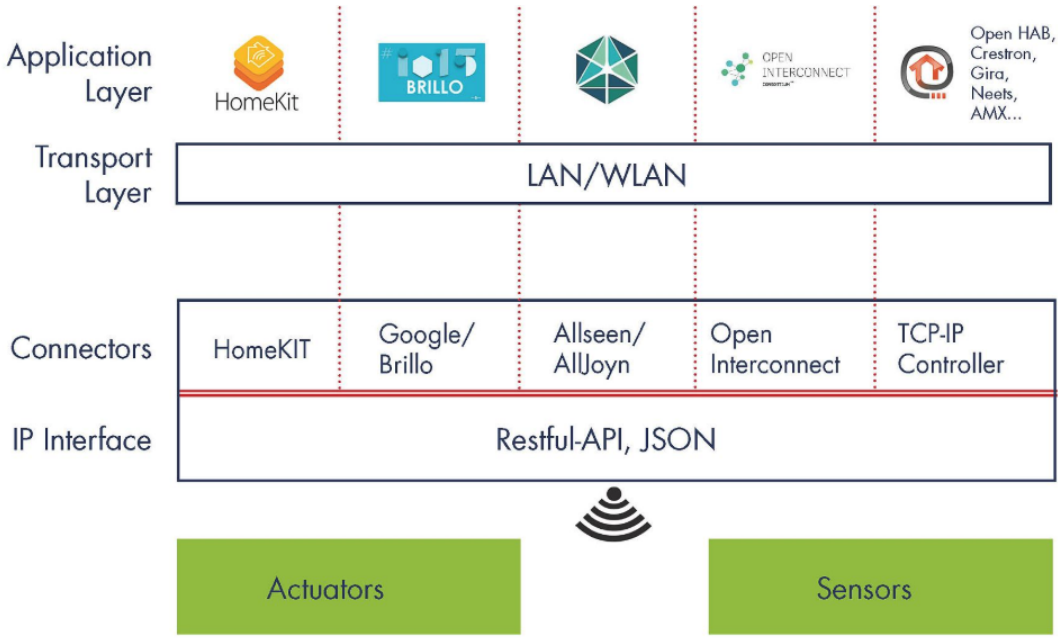


Figure 3.1: Digital Concepts Gateway Architecture

	Developer	**Tester**
Level of testing	System test	System test
Objectives	Functionality testing	Beta and Usability testing
Test environment	IoT test bed	IoT test bed
Testing technique	Specification- and code-based	Experience-based
Testing strategy	White box testing	Black box testing

Table 3.1: Current test phases and their properties

3.1 Software Development Workflow

The information on software development process for DC was gathered by observing the development process as a part of the team, interviewing the team members and from the Ticket Management System (TMS) for Digital Concepts Gateway project. The workflow as defined in the TMS can be seen in the figure 3.2.

According to the observations, the software development at DC follows agile method, which starts first from the creation of the ticket in the TMS. An open ticket is assigned to the developers and depending upon the type of ticket (whether bug report or feature request), the developer writes additional feature or fixes the bug. The developer initially manually tests the feature or the fix and after certain degree of confidence commits the code.

All the changes are accumulated throughout the day and an automatic nightly build compiles all the changes into a new image ready to be flashed into the system. The system tester takes a system image with all the completed tasks at the end of the sprint. The tester then performs the system tests on the gateway, if the tests pass the ticket is closed. But if the tests fail, the tickets are placed back in to the development pipeline.

3.1.1 The Testing Sub Process

The testing takes place in two stages of the development process, independently performed within the life cycle by developer and tester. The observed specifics of each testing process can be seen in the table 3.1. As seen in the table 3.1, there is not much difference between the tester as well as the developer when performing tests. The only difference is that the tester has no access to the source code and thus uses only the specification for verification and treats the system as a black box, whereas the tester performs tests in full exposure to the code.

3.1.2 Test Environment and Tooling

The basic environment setup for testing includes a gateway, also termed as the SUT. An array of varying standard EnOcean IoT devices, an Apple device (specifically an iPhone) or a REST client depending upon the test. Note that Apple HomeKit implementation is a special flavour of the gateway that allows Apple devices to communicate with the EnOcean devices. The tester uses

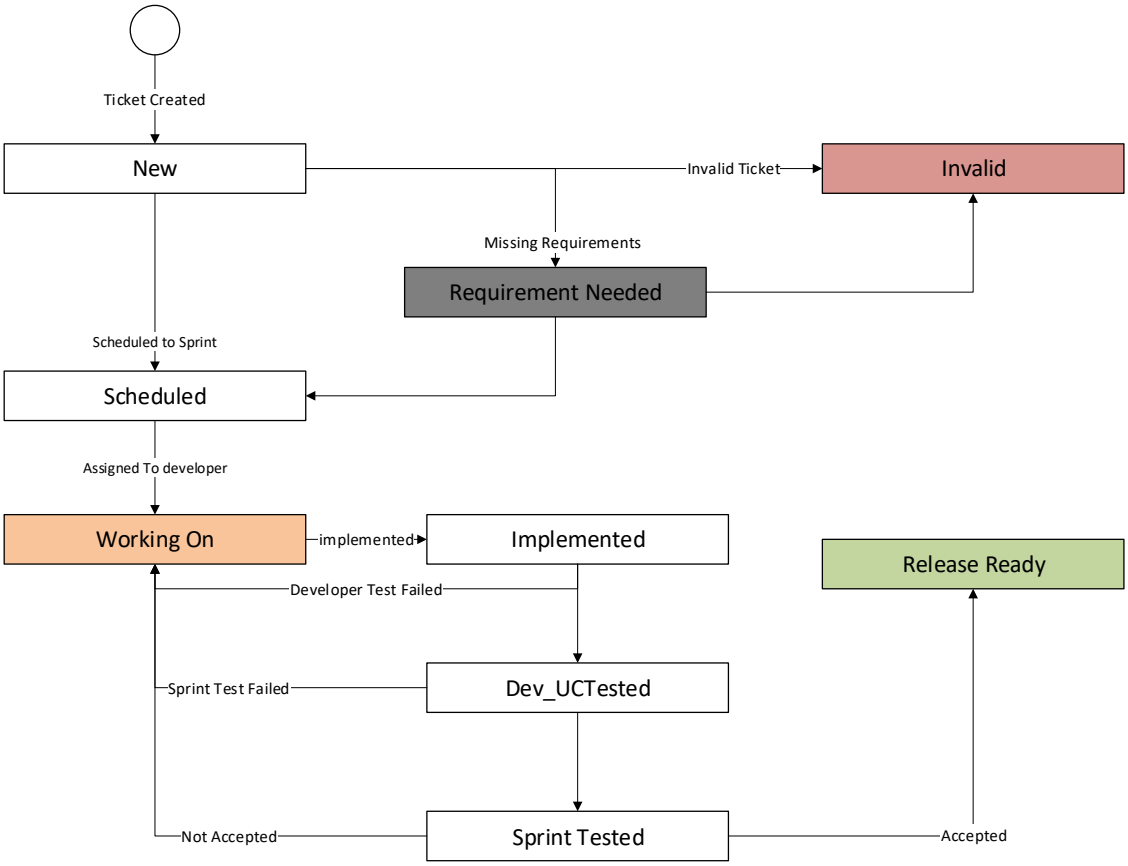


Figure 3.2: Digital Concepts workflow based on Ticket Management System

this special implementation of the gateway as a platform for testing the various system features. At times, using the raw API when the Apple HomeKit does not cover the test case. In some situations such as testing over the internet, a web services provided by third-party services are as well used.

The tester sets up the required test bed using the gateway and IoT devices, pairs the devices with the gateway or in EnOcean terms, learns in the devices, then starts testing the system by pairing the gateway with the HomeKit Application. An example test case from the tester during an interview was to pair the gateway with the Apple device. Then check whether all devices paired with the gateway show up on the iPhone HomeKit application. While performing these interactions on an Apple device, the tester tailed the system log file and tracked any an exception or suspicious messages logged . If there is any error messages, these messages along with the steps to reproduce are added to a bug report.

Table 3.2: Current Test Phases and their Properties

Degree of satisfaction	Criteria
VERY HIGH	90% or more satisfaction
HIGH	70% - 89%
MEDIUM	50% - 69%
LOW	30% - 49%
VERY LOW	below 30%

3.2 Test Process Maturity Assessment

The maturity of the testing process was assessed to evaluate the current state of the testing process. This assessment was utilized to base further improvements on the initial Digital Concepts testing methods. The evaluation was done in part using questionnaires as a method of assimilating the view within the company and secondary observations based on process workflow and available documentation.

3.2.1 Method of Evaluation

The TMM defines multiple levels and goals (or markers), which is an evaluation based on the degree of satisfaction. This degree of satisfaction on a defined goal is obtained using questionnaires from the members of the company. The questions were created based on the individual test maturity level and the different goals that are under this level. An additional field apart from Yes/No was provided along in the questionnaire to be able to assess the level of understanding and as well to remove any inconsistent answers. The questionnaire sample can be found in appendix Appendix A.

To avoid a biased view of the questionnaire, an external evaluation was as well performed and the overall degree of satisfaction was weighed giving equal preference. This uniform inclination considers the fact that, the external may not have gained enough knowledge about all the details of the company, whereas the company may have a positive view of the process. It was as well observed that various ranks within the company had limited knowledge on Testing.

The recorded data was then statistically evaluated and organized according to the various levels of testing and its sub groups. The percentages of satisfaction were converted into a degree of satisfactions [ARV+13] according to the criteria as shown in table 3.2.

3.2.2 Test Maturity Level

The TMM was evaluated and the obtained result was used to plan the further improvements necessary for Digital Concepts, starting with the a different organization of the test plan. Improved test techniques were introduced and a set of metrics were defined to be able to measure the overall effect of the new process. Each maturity level should be considered as a checkpoint [DS12] and therefore it is proposed that Digital Concepts should take the Maturity Model as a roadmap for further improvement activities.

3.2.3 Metrics for Measuring Framework Improvement

Based on the test process maturity of Digital Concepts, we can define a few metrics to measure and evaluate the state of testing. These metrics later can be used as a basis of improvement on the testing process.

Effort required for Testing

The measurement of the effort requirement to test the Digital Concepts Gateway. This can be measured using the total number of human hours required to perform a complete test of the Digital Concepts Gateway.

Number of Bugs Captured to the Number of Bugs Reported

A simple calculated metric based on the number of bugs captured by the testing process and the number of bugs reported by the users. This provides an overview on how good the testing process is in order to capture bugs before release since the bugs reported by the users shows that the testing process is not effective.

EnOcean Profiles Tested to the Number of EnOcean Profiles Implemented

This metric provides the number of EnOcean profiles tested to the number of EnOcean profiles that are currently implemented in the system. The decrease in the coverage suggests that the core might require more testing.

Specification Coverage

This metric provides a feedback on the number of specifications covered as provided by the third-party vendors such as Apple HomeKit or IBM Watson.

3.3 Major Problems and Requirements in Testing

The testing of Digital Concepts Gateways presents problems that are particular to the application. A discussion with developers and testers provided initial feedback on the major problems of testing. These problems leads to a set of requirements for the test framework which have been described in this section.

3.3.1 Problems

- Manufacturer has little or no influence on the structure of the physical network, which makes it important that the test bed to be changeable with varying devices and conditions.
- End to end testing is difficult due to the necessity of user interactions in third-party services which may be hardware as well as software and as well requires observation of result in different active participants.
- The result of a test could be available not only in the direct interactions but secondary information sources should be analysed as well for possible failures.
- Extend time required due to the EnOcean protocol where the IoT devices may send out response in hours as part of energy conservation.
- A complete test suite cannot be loaded into the Embedded System.

3.3.2 Requirements

- **Scalability**: There are a lot of devices available in the market that are supported by the gateway. Depending upon the user assuming that the user only uses a single instance of these devices to make a smart home system, that would lead us to 2^n possible permutations of device combinations. These combinations cannot be tested within a definite period of time. Let alone the fact that a user can have multiple devices of the same type or model within the smart home system, which would increase the permutations of the possible smart home network. Even though all of these permutations are not tested, the testing environment, techniques and methods should be able to adapt to the changing configuration for scalable and flexible testing [Ree16]. This multiple possible combination of devices makes it highly difficult for a manual tester to effectively test the many devices within a certain time frame. A scalable test framework would try to reduce the effort required for manual testing as much as possible, thus contributing to the two major metrics defined in subsection 3.2.3: Effort required for testing and EnOcean profiles test ratio.
- **Testability of boundary conditions**: There are many conditions that occur in rare cases and to replicate these conditions might be next to impossible or might take very long to achieve physically. The testing framework should first of all make these difficult conditions possible to test and as well make it achievable faster. Testing of boundary conditions allows us to improve the coverage of specifications, which increases the product quality.
- **Possibility of automation**: The testing techniques and methods should allow the possibility for automation, to further increase the speed and evenness of testing. It reduces the effort required for executing tests and helps maintain a steady specification coverage in consecutive releases.
- **Testability of the hardware** should also be a possibility due to the hard-coupled nature if the software with the physical system.

4 Proposed Test Plan

The DCGW architecture presented in Chapter 3 displays the various modules that composes the gateway. Considering this complex structure of the gateway, an incremental testing strategy should be employed with the different levels of testing as shown in the figure 4.1.

The connectors can be considered as a complete software due to the resemblance to a service in within the SOA paradigm and should be as well individually tested to maintain high reliability of the overall service of the gateway. Therefore, two different levels of testing are required:

- **Service level testing**: The testing is done on a service level, where individual services, such as connector, core are tested to provide high quality
- **Gateway level testing**: The testing is done while considering the overall gateway or at least multiple components of the gateway.

The service level testing with internal levels, unit, integration can be considered as a regular pure software testing problem and thus, standard frameworks available for the individual service depending upon the programming language can be chosen. But the system level testing for the services requires extra attention and special techniques to the dependence on physical objects, where the thesis focuses on.

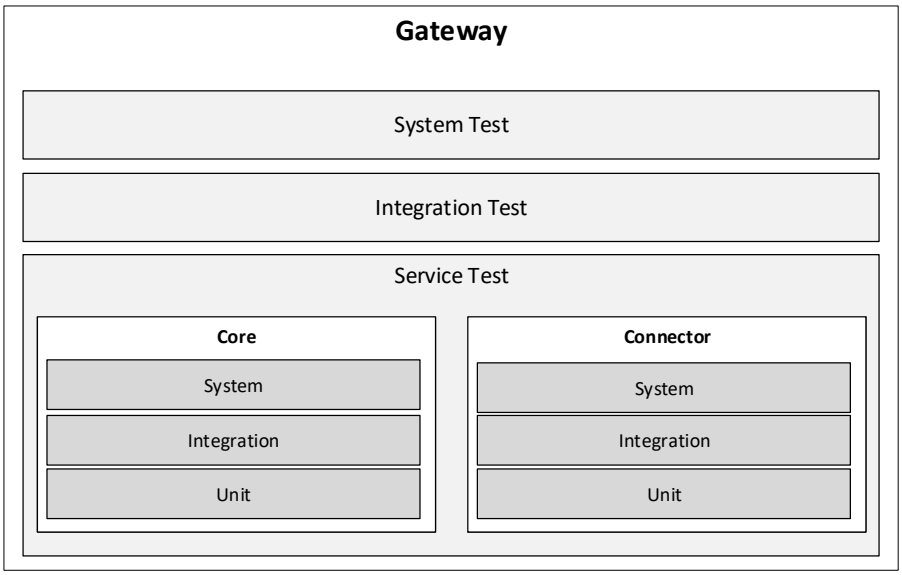


Figure 4.1: Levels of Testing for Digital Concepts Gateway

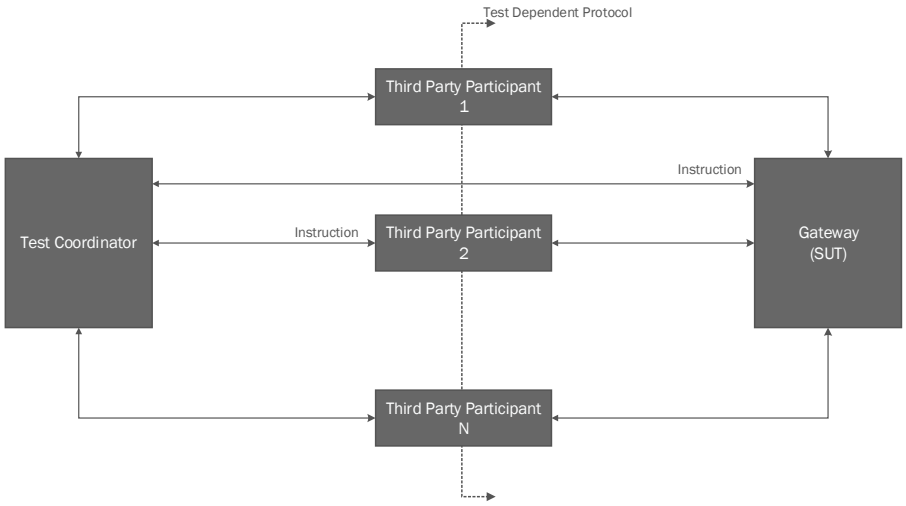


Figure 4.2: Proposed Test Framework for Digital Concepts

According to the list of problems for testing as mentioned in chapter 3.3, the foremost problem is the lack of compute capacity of the embedded system due to which the test suite cannot be implemented or loaded into the embedded system. An additional problem is the environment setting of the gateway. There are multiple different participants involved, IoT devices, smart phones, web services, etc. and most of which contribute to the overall result of the test. Thus, an overall coordination between the all the participants of the test is required to generate test conditions for the gateway, as well as to verify the correct functioning of the software.

4.1 Test Framework

There are various participants that need to be orchestrated in order to create the desired scenario in which the gateway is to be tested. Such a framework has been depicted in the figure 4.2. The defined framework makes use of an entity which acts as the brain of the overall framework called the Test Coordinator. The major task of the Coordinator is to store and execute all the test cases. This solves our first problem for test deployment; the test suite cannot be deployed in the Embedded System. The added advantage of this structure is that it allows automation. The Test Coordinator acts as a human tester. When provided with proper test steps and expectations, it performs these steps and validates whether the test was successful or not.

On the other hand, the participants are the aforementioned third-party services which are not accessible to Digital Concepts as a framework. This leads us to the second problem; Digital Concepts does not have full access to the implementation of different participants within the environment, such as the HomeKit Controller for automated system stimulations. The HomeKit Controller within the framework is an Apple device which pairs with a gateway and controls the accessories exposed by the gateway. The HomeKit in turn responds to the interactions made by the user on the app UI, which makes automation a difficult task.

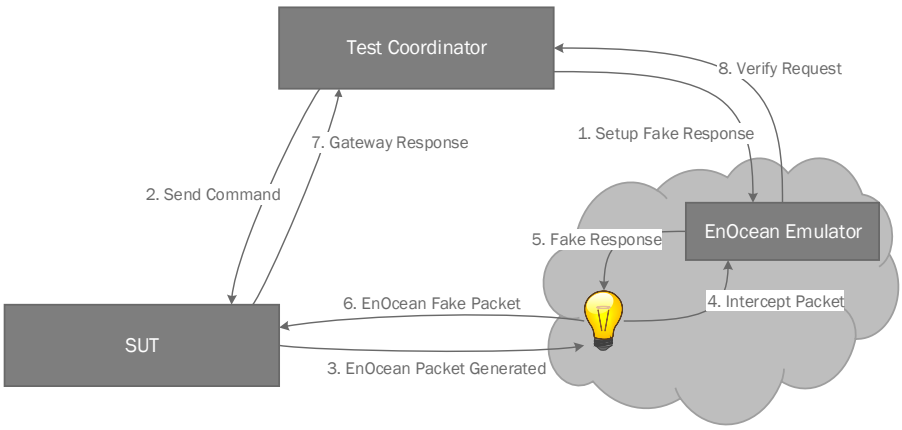


Figure 4.3: Conceptual EnOcean emulation of a Smart Device

4.1.1 Emulators for Automation Support

The initial and obvious approach for test automation would be to record and replay the UI interactions performed by the tester. But this highly depends on the UI of the software and in case of changes in the UI, the chances are high that most of the Test cases would fail and need fixing, which leads us to the development of emulators to support automation. The major idea behind these emulators is the possibility to replicate third party software behaviour.

Emulators do not contain logic to perform any self-initiated interactions, but have just enough to replicate the behaviour in terms of data transfer and communication. The Test Coordinator sends commands to an emulator to be performed. After the emulator has performed the action, it responds to the coordinator with a response, regarding the state of the action.

4.1.2 EnOcean Device Emulators

The third problem for testing of the Digital Concepts Gateway is the lack of influence on the structure of the physical network and thus the need for testing of the combination of devices in a scalable manner. A physical test bed would require adjustment in every test case or would be unmanageable in case that the number of devices increases. With this respect, [Ree16] proposes model based technique to test the IoT software, but this method does not test the Radio Communication of Digital Concepts gateways. For this purpose, EnOcean devices are modelled by an emulator which can be controlled by the Test Coordinator. The emulated device is used to validate the communication from the gateway and the device.

The figure 4.3 shows a step by step interaction on the usage of EnOcean Emulator to fake the EnOcean Device (a bulb in this situation). The Test Coordinator first configures the EnOcean emulator to respond a fake message when certain criteria have been met. It then asks the gateway to perform an action. The gateway sends out an EnOcean packet and notifies the Coordinator in case it receives any packet. The EnOcean emulator intercepts the radio packet and generates the fake response as determined previously by the Test Coordinator. The Test Coordinator then verifies the result using the notification the SUT provides as well as the EnOcean emulator provides.

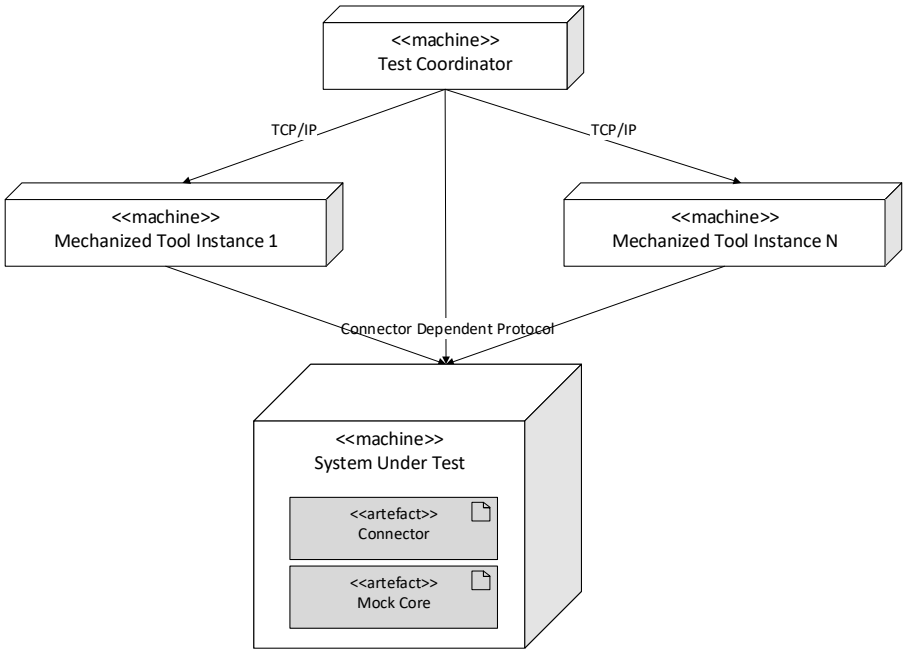


Figure 4.4: A proposed environment setup for Connector system testing

4.1.3 Hardware Testing

The testing of hardware seems to remain a difficult topic in terms of automation, but is not impossible. The testing of whether a hardware functions is possible by set defining a test bed where multiple sensors can be attached to actuators and vice-versa to validate correctness. But, the major problem with this method is the lack of scalability and the limited possibility to test for boundary conditions, which is one of the other requirements. If a test bed has been fixed, then it is not easily scalable to added devices. The gateway compatibility is initially tested by the developer manually as part of the debugging activity. Thus, a scalable testing framework along with time-to-time complete manual testing should be sufficient to verify the proper functioning of the hardware.

4.2 Test Environment Configuration for different Levels of Testing

For different levels of testing, a different setup for the environment should be utilized to allow individual, as well as holistic validation of the gateway software.

4.2.1 System Level Connector Testing

The testing of the system level should be conducted by isolating the Connector logic to the depending logic. The deployment diagram in figure 4.4 depicts the Test Environment for testing of the Connector where the Test Coordinator is deployed in a separate machine since there exists only one instance of the Test Coordinator whereas a multiple number of emulator instances depending upon the Test.

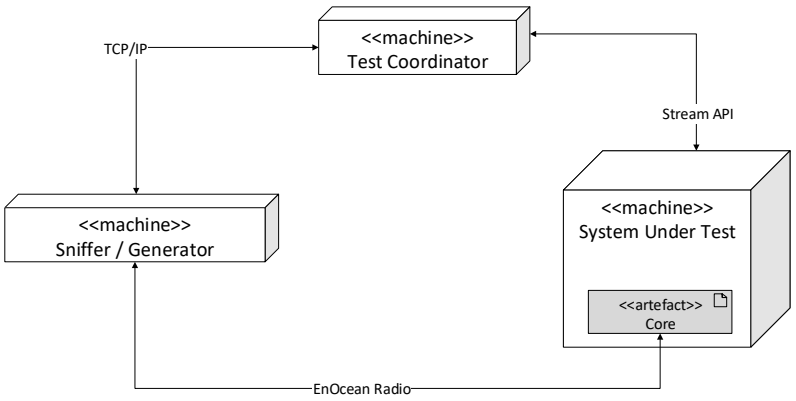


Figure 4.5: A proposed environment setup for Core system testing

These emulators could be deployed in a Docker Container or VM or even a specialized gateway, depending upon the test case. The gateway contains the to-be-tested Connector, as well as a mock core service which the Test Coordinator can control.

The coordinator is capable of controlling the responses sent by the mock core and the input provided to the Connector directly or via the use of emulators if required. This allows the monitoring of the behaviour of the Connector on directions.

4.2.2 System Level Core Test

The Core logic has no dependencies and is stimulated via the REST API or the EnOcean radio communication. Any interaction on one side produces a result on the other side, thus the following deployment model should be used in order to be able to monitor the system.

The Test Coordinator communicates with a secondary gateway which has been configured to be able to sniff EnOcean packets as well as generating fake packets. The Sniffer / Generator gateway behaves as a fake EnOcean device and reports any EnOcean traffic to the Test Coordinator.

4.3 Metrics for Product and Process Quality

The metrics defined in Section 3.2.3 are crude and have been provided as a method of evaluation between the existing process to the proposed test framework. Since the target is to automate most of the tests, a new set of metrics can be obtained and are necessary to further improve testing.

Company Objectives:

- Improve the product quality by reducing the number of total bugs in the source after release.
- Improve the product quality by improving the downtime of the system
- Measure the number of Defects introduced while adding new features
- Improve the time to fix Bugs and measure the effort required for planning purposes

4.3.1 Product Metrics

Problems per User Month

The problems per user month is an indicator for the defects that has affected the user over a month's period. This can be a pointer to the sense of quality as observed by the user. This is a measure of the customer perspective of the product and is directly linked to the number of defects available after release. A lower number can be used as a hint on the customer level of satisfaction.

$$problems\ per\ user\ month = \frac{number\ of\ reported\ problems}{number\ of\ active\ gateways\ in\ the\ month}$$

Defect Density per shipped or changed code

This is a derived metric from the number of bugs found and the code size of the software. This can be utilized in checking the quality of the software, with the number of defects found or reported along in terms of the code size of the software. The added benefit of this metric is that it allows for future references the defect density for new projects and helps in the planning of testing resources.

This metric helps measure and improve the third and fourth objective of Digital Concepts; reduce defects in added features and effort planning. It provides a an idea on the size of change of the software and the total defects that can be expected in such a change. This allows planning of testing effort for such changes.

$$defect\ density = \frac{number\ of\ defects\ reported\ or\ detected}{total\ lines\ of\ code}$$

Mean Time to Failure or Continuous Hours of Operation

The mean time to failure is a simple metric to test that the gateway operates for a long period of time and without problems. The higher the number represents the stability of the system over time. This is a crucial non-functional property for IoT devices since they should be in operating condition for extended periods of time.

$$mean\ time\ to\ failure = \frac{number\ of\ failures}{total\ hours\ of\ operation}$$

4.3.2 Process Metrics

Code/Specification Covered

This shows the total percentage to code or the specification covered depending upon the test being considered. The quality of unit testing can be represented using code covered, whereas the overall quality of system testing can be maintained using the specification coverage.

This metric serves as the overall quality of the testing process and helps us understand the above product metrics better. As an example scenario, the defect density decreases, this does not directly mean that the product quality is better. We should then consider checking the code coverage or specification coverage for the quality of testing being performed. If this metric shows a lowered quality of testing, then the defect density metric is deemed to be less reliable and therefore more testing effort should be provided.

$$code\ coverage = \frac{lines\ of\ code\ executed}{total\ lines\ of\ code}$$

$$specification\ coverage = \frac{specifications\ covered}{total\ number\ of\ specifications}$$

Mean Bug Response Time

The mean bug response time shows the average time taken starting when a bug was reported to the point where a developer takes a bug for fixing. This can be a good pointer to resource allocation on testing and development.

Mean Bug Fix Effort

This is an administrative metric useful to visualizing the mean effort placed in fixing a bug and can be useful for resource optimizations and test planning.

Duplicate Bug Reports

A special metric required at DC for that represents the number of bugs that have been duplicated. The reason behind this is a history on multiple bug tickets on the same issue and multiple developers devoting time to solve the same problem.

Test Automation Status

This metric can be used to represent the status of automation at current state of testing. This can be calculated using the number of tests that are automatable and that have been automated.

$$percentage\ automated = \frac{number\ of\ automated\ tests}{total\ number\ of\ tests\ automatable}$$

5 Design of a Test Automation Framework

The implementation of the Test Framework as proposed in chapter 4 begins with the general organization of the Test Environment and the deployment of artefacts in specific places. The deployment diagram presented in Figure 5.1 provides an overview of the type of machine used. The Test Coordinator is deployed in a VM along with a MySQL database and the Test Coordinator artefact. At present only two different types of Emulators have been reviewed, the HomeKit Client Emulator and the EnOcean sniffer. The HomeKit Client can be deployed in a virtual machine whereas the EnOcean sniffer requires a special version of the Gateway to allow sniffing of the packets over the air. The HomeKit emulator should be deployed in a VM to address many Apple specifications, e.g. The accessory should be able to pair with at least 16 Apple devices. To test this specification, a multiple number of VMs can be spun up and in other scenarios, a small number of VMs can be used to perform the Test. This allows reduction of dedicated hardware for just a few Test Cases. The test packets are as well directly deployed on the gateway. Emulations were difficult to achieve due to the lack of hardware in a virtual machine.

A data flow diagram for the Test Coordinator was created in relation to the use cases of the Test Framework, of which a high-level Data Flow Diagram can be observed in Figure 5.2. The major processes that takes place within the coordinator is the, storage of requests, execution of tests, storage of execution reports and finally generation and storage of test results. As mentioned in Chapter 3, in one of the major problems, the exercising of the software may not provide complete

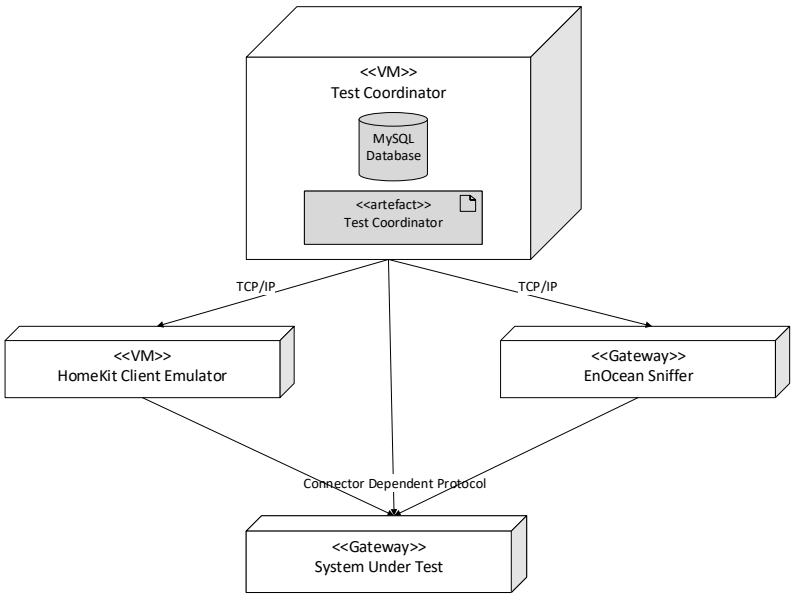


Figure 5.1: Digital Concepts Test Environment deployment diagram

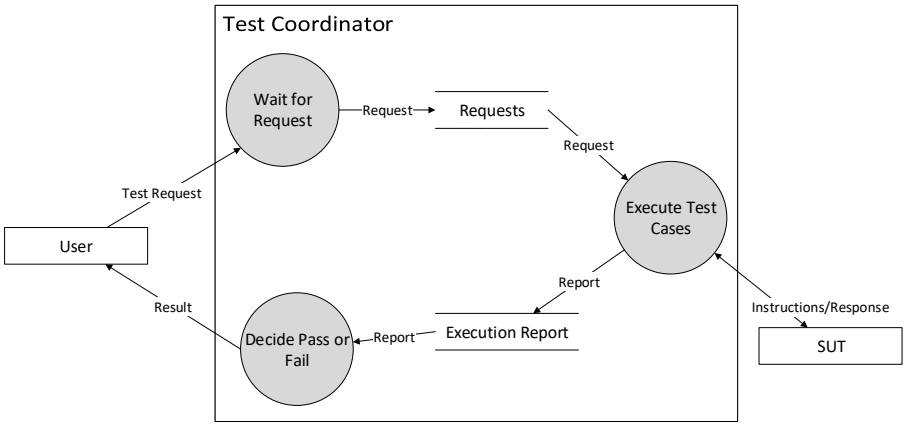


Figure 5.2: Test Coordinator level 0 data flow diagram

information on the correctness of the software. Thus, the test report contains a part of the complete information required for decision making. This separation allows the oracle to consider multiple different sources before giving verdict on the Test Case.

5.1 Database Design

Based on the data flow diagram in figure 5.2 (complete DFD see in appendix), the Test Coordinator is responsible for maintaining the test requests and the results of these test executions. To support this, an initial database schema has been defined, which can be seen in figure 5.3. The provided schema is a summarized version, for a complete diagram please refer to the appendix B. The test cases are organized as part of test suites. A single test case can be a part of multiple test suites. A feature to be implemented is the hierarchical test suite structure, allowing test suites to be included within other test suites.

The test cases are only organized in the database, whereas the actual test case is stored within a file in the system storage. There are many other tables apart from the test suite and the test case table, such as test schedules and the test details table. The test schedule table stores the tests that were requested whereas the test detail stores the information on which test suite to be executed and what the result of the test execution was. A test result on the other hand is divided into, individual test cases result and a single test case result is further sub divided into single step result from within the test case.

5.2 Test Suite and Test Cases

A tester is allowed to create test cases, test suites and assign test cases to different test suites. There are mainly two different types of test suites available, first native test suite and second imported test suite. Imported test suite is a test suite that has been imported from another testing framework, for example Katalon Studio. Katalon Studio is a web services testing framework which can be used to test web UI as well as the REST API. The execution of the native test suites is performed by the test

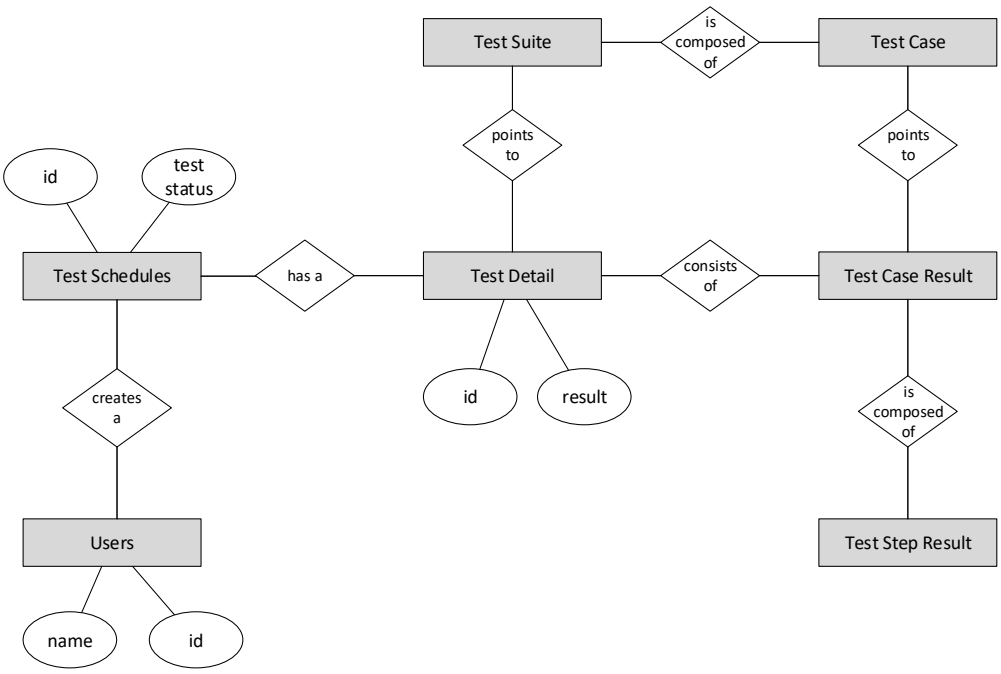


Figure 5.3: Simplified deployed database diagram for test management

framework implementation described in this chapter, whereas the imported test suites are executed by the respective test framework. The results of these test frameworks are later parsed and updated into the system.

5.2.1 Test Case

A test case provides clear instructions to the test framework on the steps to be performed, the coordination of multiple participants and as well the result to be expected from each individual test step. The test cases at present is written by a human tester with high degree of knowledge of the system. An example test case is provided in the listing 5.1.

The test case provides three specific phases, the setup, the execute and the teardown. The setup can be used to setup the gateway and other participants to a specific configuration, the execute phase is where the actual testing is done and the teardown phase is used to clean up any actions performed during the testing to reset the state. As seen in the listing, the test case is written in YAML format. An alternative to the YAML format was the JSON format, but YAML was chosen instead of JSON since it allowed a referencing mechanism. This referencing mechanism can be seen in the listing above, where the first step is referenced by the second step and makes changes to the previously defined object. This allows compact definition of the test case. Further information on writing Test Cases and validations is available in Appendix B.

Listing 5.1 Example Test Case written in YAML

```
--- # Test Case
name: Nodered Service Start Test
description: >
Node red service should be successfully started by the script as a docker container.
expected: >
The docker service should be successfully started.
testCaseId: nodered_start
criticality: NORMAL
phases:
  - name: setup
    steps:
  - name: execute
    steps:
      - &exec_gw
        instruction: exec
        target: $gw
        params:
          - '/opt/dcgw/scripts/nodeRed'
          - '1880'
          - 'admin'
        expect:
          - result:
            code:
              - validator: equals
                value: '0'
      - <<: *exec_gw
        params:
          - 'cat'
          - '/opt/dcgw/config/node-red.config'
        expect:
          - result:
            stdout:
              - validator: contains
                value: 'PORT = 1880'
  - name: teardown
    steps:
```

5.3 Test Coordinator

The Test Coordinator is the main component of the test framework and is responsible for executing the tests and validating the result of the test. The Coordinator maintains all the test cases with the database mentioned above and in case a test request is received, it starts executing the relevant test cases to the test request.

The Test Coordinator is composed of three major components, namely the Request Monitor, the Test Executor and The Oracle. As seen in the Figure 5.4, the when a Test Request is submitted, the submitter provides a request information, containing which test suite should be executed.

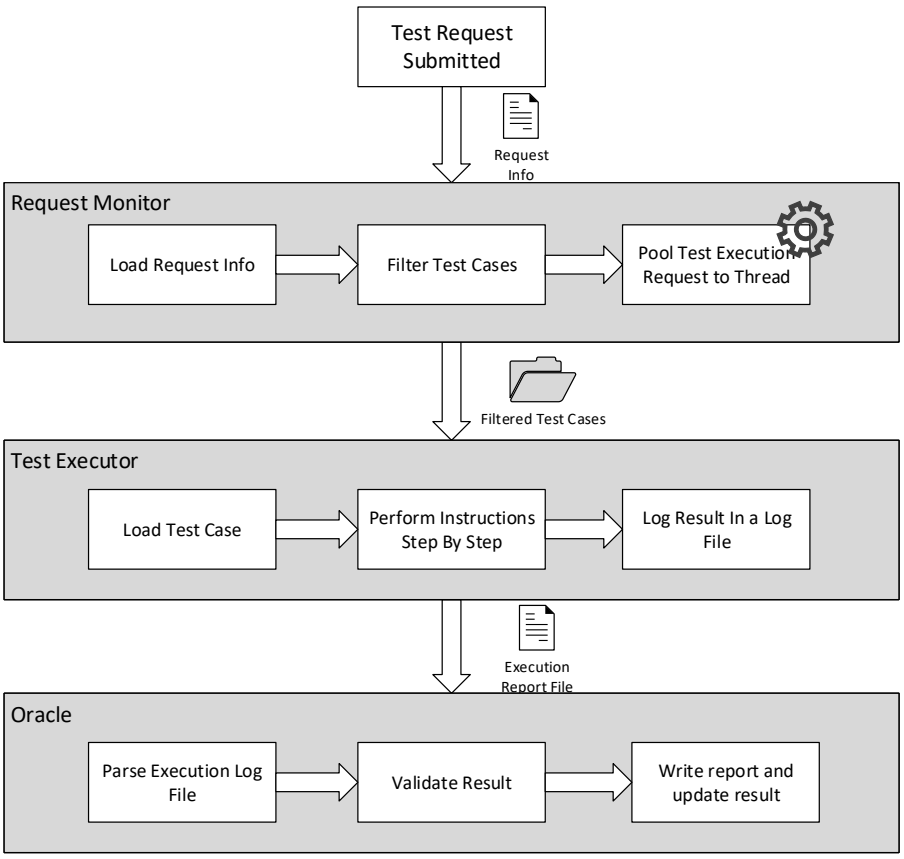


Figure 5.4: Process flow for Test Coordinator along with artefacts shared among components

5.3.1 Request Monitor

The test request information provided by the submitter is parsed by the Request Monitor. The Request Monitor after parsing the request information, filters the required test cases and prepares them for execution. The Request Monitor then after, instantiates the executor process. The Request Monitor contains, a thread pool which is used to limit the number of Executor instantiations. This allows the queuing of test requests depending upon the available Test Execution Environments. As shown in the flow chart in figure 5.5, the Request Monitor checks if the request was made for an imported test suite, in which case the Imported Test Executor is instantiated without any filtering of the test cases. The filtering is done by the Imported Test Executor.

5.3.2 Test Executor

The Test Executor, or the native Test Executor logic can be observed in the flowchart in figure 5.6. The test executor first loads all the YAML files within the requested directory. The YAML files are then parsed and every phase of the Test Case executed in sequence. The Test Executor was implemented in python; therefore, a dictionary object was pre-populated (see listing 5.2 for sample) with the mapping of the method to be executed was mapped to the instruction as the key. In case

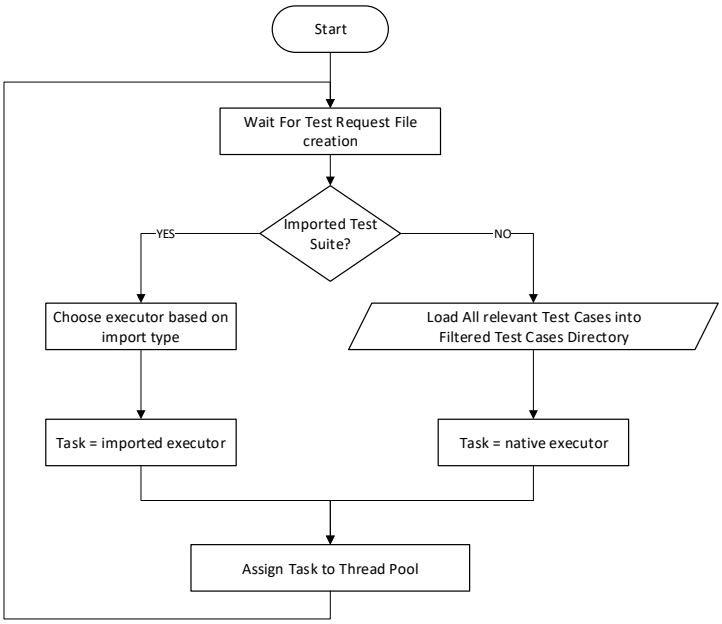


Figure 5.5: Flowchart for Request Monitoring

Listing 5.2 Test Executor Instruction to Method mapping for execution of commands

```
# command mappings to python methods
mappings = {
'exec': instructions.execute,
'wait': instructions.wait,
'connect': instructions.connect,
'disconnect': instructions.disconnect,
'reset mock core': instructions.reset_mock_core
}
```

there was no mapping available, the Invalid instruction information was logged and further steps were still processed. The Test Executor logs the execution information into a separate file for every test case executed.

The execution information is a simple log file, where the time of logging along with one-line message is written. The message cannot contain new line in between the text, and no binary message. The binary message should be converted to a human readable format, either base64 or hex. Since the decision on the outcome of the test case is performed in a later stage, the executor needs to provide result on the log file for the oracle to process. Therefore, a regex ‘Result: <name>: <ascii formatted result>‘ is used to log any result that should be considered during verification of the execution log.

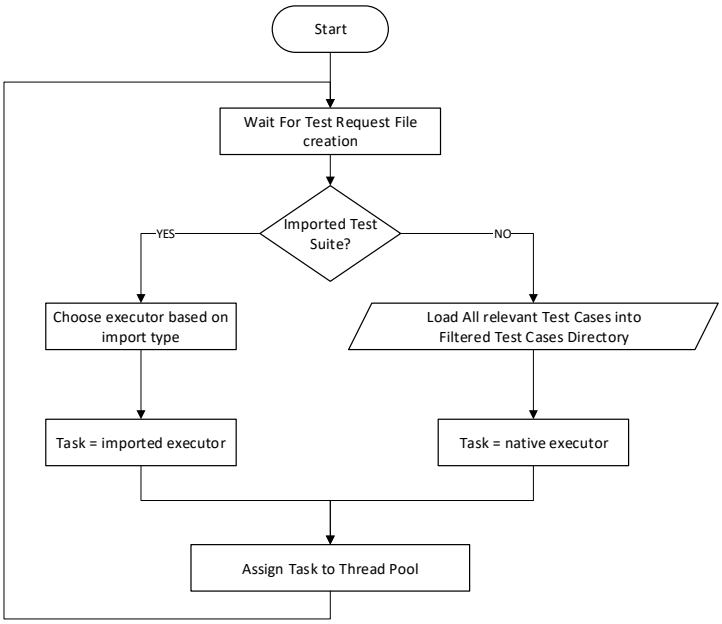


Figure 5.6: Flowchart for Request Monitoring

After the execution of a test case has been completed, it sends the information of the execution to the oracle for further processing, whereas continues executing other test cases until all the test cases in the provided directory have been completely execution. In any case of execution error, the result of the test case is reported as failed and the cause notified, but the execution still continues.

5.3.3 The Oracle

One of the biggest problems in automated testing is determining whether the test passed or failed. The oracle problem is a well-known problem for test automation and quite a few categories of oracles have been discussed already in Chapter 2. The major problem at Digital Concepts is the lack of proper specification document that can be used to automate the conclusion of the test result. Thus, at present, a human oracle approach has been utilized. The developer of the test case writes the test case, along with the expected values or behaviour. An example is shown in listing 5.3 where the user specifies the action to be performed, which in this case is the verify mock core, and expect that there was a request where the request method was GET and the path was '/system/info'.

The developer of the test case can also choose to use scripts for validation purposes for complex behaviour. For further information on test case expectations and test cases in general, please refer to the appendix.

Listing 5.3 Example validation of results provided in test case

```
instruction: mock core verify
target: '$mock-gw-'core
expect:
  - result:
      requests:
        - path: '[0].method'
          value: 'GET'
        - path: '[0].uri'
          value: '/system/info'
```

5.3.4 Emulators

The test coordinator, sends instructions to the emulators and the emulator simply performs them. They do not contain more logic than simple execution of commands based on requests. For the implementation of such protocol, multiple choices were possible, one of them included the use of RPC, so that the coordinator could call a remote procedure on the emulator, and the result of the function call would be transferred back to the caller; the coordinator. An alternative to that was the use of socket streams. The advantage of the socket stream was the possibility of returning not only a single result, but as well the execution logs as well as multiple levels of logs. This has been utilized in the HomeKit emulator to allow coarsely granulated commands with multiple results for verification. Further details on the socket communication can be found in the appendix.

5.3.5 HomeKit Client Emulator

The home kit client emulator replicates the network behaviour of an Apple device. Coarse-grained commands are available for the tool, that the tester can utilize to stimulate the system. The HomeKit client uses a modified version of the SRP protocol with mFI verification. The mFI verification is optional, in a sense that the mFI verification is useful only when the apple device wants to validate whether the accessory it is connecting to is verified apple accessory or not. The pairing process is highly complicated and impossible without access to a valid mFI chip. Therefore, a Gateway without the mFI chip is used to make the pairing possible using a software. The mFI chip functionality can alone be tested manually.

5.3.6 Mock Core

The mock core is a special implementation of the Core done in python using the Flask framework. Mock responses can be setup into the core using a HTTP Method and URL match string. The mocked response and the URL match is stored in memory and when a request arrives matching the parameters, the response is invoked. The response status, headers and body can be setup while setting up the mock. Since the Mock Core, as well supports a streaming API which needs to be mocked. The code snippet for this can be seen in listing 5.4. Since the core is a REST API, the mock handling behaviour is actually managed by using REST endpoints outside the domain of the Gateway API.

Listing 5.4 Flask implementation of a stream API mock

```
@app.route('/devices/stream', methods=['GET'])
def stream_api():
  save_request_info()
  mapping_key = generate_mapping_key('/devices/stream', 'GET')

  # the generator function which creates the stream response and allows parallelism in flask
  def generate():
    if mapping_key in mock_mappings:
      body = json.dumps(mock_mappings[mapping_key]['body'])
      yield str(len(body) + 2) + '\n'
      yield body
      yield '\n\n'
      if 'code' in mock_mappings[mapping_key] and \
        not mock_mappings[mapping_key]['code'] == 200:
        return
    while True:
      stream_out_event.wait()
      body = json.dumps(mock_mappings[mapping_key]['body'])
      yield str(len(body) + 2) + '\n'
      yield body
      yield '\n\n'
      stream_out_event.clear()

  status = 200
  if mapping_key in mock_mappings and 'code' in mock_mappings[mapping_key]:
  status = mock_mappings[mapping_key]['code']

  return Response(stream_with_context(generate()), status=status)

@app.route("/mock/stream/notify", methods=['POST'])
def stream_notify():
  """
  Notify the devices connected in the stream with the information posted
  :return: Response
  """
  mapping_key = generate_mapping_key('/devices/stream', 'GET')
  (request_info, response_info) = parse_mock_request(request.json)

  mock_mappings_lock.acquire()
  mock_mappings[mapping_key] = response_info
  mock_mappings_lock.release()

  stream_out_event.set()
  return make_response('', 200)
```

6 Evaluation

This chapter compares the initial DC testing with the new Test Framework in order to determine the improvements achieved or possible. The evaluation is based on two broad aspects; the effort required for testing and the quality and efficiency of testing. Since it was determined in Chapter 3 that the Digital Concepts initial testing method that only a limited number of metrics can be utilized to evaluate both the testing methods. As human effort is directly linked with the cost, we first look into the test effort aspect, where we define the areas which require effort for both Manual Testing and Automated Testing. After this, we move into the area of test quality and efficiency. Throughout this chapter, Method A refers to the Automated Testing, whereas the Method M refers to the Manual Testing.

6.1 Human Effort for Testing

Human Effort is an indirect indicator of the cost incurred to perform a certain task. A measurement in terms of man-hours can be utilized to compare the Method A to Method M. The overall Effort can be described through a simple model;

$$E(m,n) = S(m) + n \cdot X(m) \tag{6.1}$$

where, m is the total number of tests run in one iteration, n is the number of iterations performed, $S(m)$ is the total effort required for the setup of a Test Environment with m test cases and $X(m)$ is the total Execution Effort for m tests for 1 iteration. Furthermore, we can define;

$$S(m) = s + T(m) \tag{6.2a}$$

$$X(m) = m \cdot x \tag{6.2b}$$

where, s is the effort for setup of the environment, $T(m)$ is the setup effort of m test cases, x is the effort required to run a single test.

This model as defined in equation 6.1 assumes that there are no changes in the specifications, but in fact they keep on changing. An interview was conducted with the team members who work in close relation with specifications created by Apple, EnOcean, or other parties in order to assess the general frequency and kind of changes that occur in the specification and how it generally affects the source code. Thus, a slight modification to the model is necessary. This change can be accounted by adding in the effort required for the changes and can be represented as (using 6.1 and 6.2):

$$E(m,n) = s + n \cdot m \cdot x + T(m) + C(m,n) \tag{6.3}$$

where, $C(m,n)$ is the total effort required to adapt to the changed specification with m tests and n repetitions.

Interview with testers and developers provided a general feedback that the specifications change almost 3 times every year or once every 4 months. It was also concluded that a testing is performed every 14 days, after the sprint concludes. Thus, changes occur almost every 8 to 9 test repetitions. Let us assume every 8 iterations for the worst-case scenario. It is also to be noted that the expected life of the project is a minimum 3 years, or 79 iterations.

Digital Concepts Initial Testing Process

The information about the initial testing process was gathered using interviews with Testers. The interview concluded a few peculiarities of the testing process, specially that the test environment is setup before every test execution. Multiple devices are not tested in parallel by a single tester, due to the high amount of cognitive load despite devices taking hours to initiate interactions.

According to the model described in Equation 6.3 the initial testing process can be represented as;

$$E_m(m,n) = s_m + n \cdot m \cdot x_m + T_m(m) + C_m(m,n) \tag{6.4}$$

But from the information obtained from the interviews, we can assume that the $s_m = 0$ but the effort of setting up of the test case can be divided up equally into the effort required per test execution x. Therefore;

$$x_m = \frac{X_m}{n \cdot m}$$

We know from the interview conducted with testers that in average, $X_m = 3$ days, $n = 1$ and $m = 15$ we get; $x_m = 1.6$ hours or 96 minutes. We can assume an error margin of 1 day to account for tester biasness or any unforeseen circumstances. This provides us with two different possible values, i.e. $X_m = 2$ *or* 4, $n = 1$, $m = 15$ and therefore, $x_{m1} = 1.04$ hours or 62 minutes and $x_{m2} = 2.13$ hours or 128 minutes. Where, x_{m1} is the best case scenario and x_{m2} the worst case.

We can also simplify the model since we know that the test cases are created purely on experience basis and generally not a lot of efforts are placed into test case creation. Thus, we have

$$E_m(m,n)(in\ man - hours) = n \cdot (m \cdot x_m) + C_m(m,n) \tag{6.5}$$

In case of no changes, the Effort required for the testing grows linearly with respect to the number of test repetitions. It has been determined that a tester requires a minimum of 8 hours to go through the changes to get acquainted with the new specification and thus, the effort required for handling changes can be considered as;

$$C_m(m,n) = 8 \cdot \left\lfloor \frac{n}{8} \right\rfloor \tag{6.6}$$

The Figure 6.1 graph shows the effort accumulation for initial DC testing effort for 3 years or 79 iterations in terms of man days (8 man-hours) for different number of tests performed at every iteration and with the effort changes every 8 iteration of 8 man-hours. A best case effort estimation and a worst case estimation can be seen based on the values of x_{m1} and x_{m2}.

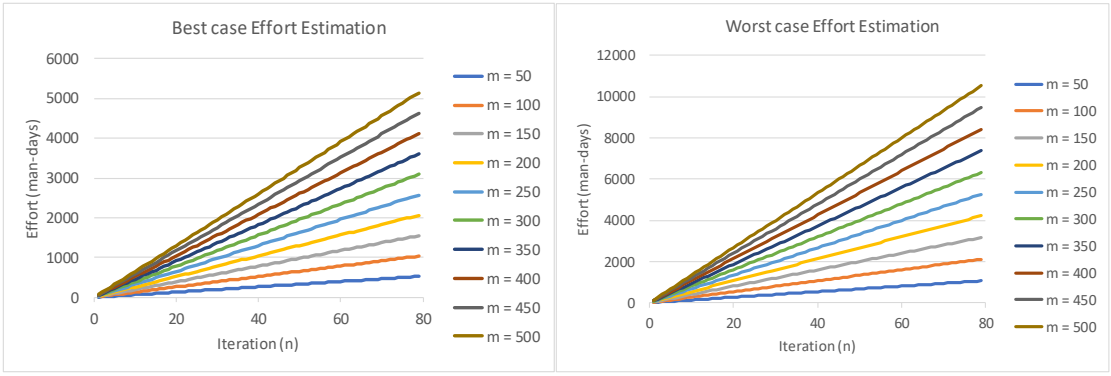


Figure 6.1: Effort Estimation for Manual Testing with m tests per iteration

6.1.1 Testing Process with Automated Framework

To evaluate the automated testing effort, the data was gathered by evaluating the test execution logs as part of the development session along with interviews with the developer of the test cases to assess the additional effort required. Due to the nature of automated testing, the above model can be adjusted a bit stating that the execution times of the machines are 0 ($x_a = 0$). Therefore equation 6.3 can be reduced to:

$$E_a(m,n) = s_a + T_a(m) + C_a(m,n) + M_a(n) \tag{6.7}$$

The total time required to create n test cases can be assessed using the simple model [RS] below:

$$T_a(m) = \sum_{i=1}^{m} t_{min} + k \cdot e^{-m}$$

The minimum threshold is evaluated by measuring the average time an "Expert" level tester takes to create a single test case. The creation time of varying types of tests were measured for the "Expert" and the average was found to be 21 minutes or 0.35 hours. In this scenario the "Expert" tester did not have extended training levels and thus the data obtained should theoretically be greater than the actual average on the long run. Since we plan on tracking the worst case effort requirement for automated testing, this obtained data is highly relevant.

The constant k for the equation can be solved by using the observed value while creating test cases with tester. The tester required 3 hours to create the first test case; plugging in these values to the above learning curve model, we get:

$$\begin{aligned} T_a(m = 1) &= 0.35 + k \cdot e^{-1} \\ 3 &= 0.35 + k \cdot e^{-1} \\ k &= 2.65 \cdot e \end{aligned}$$

Therefore; we can rewrite the $T_a(m)$ as:

$$T_a(m) = \sum_{i=1}^{m} 0.35 + 2.65 \cdot e \cdot e^{-m}$$

Iteration (n)	Number of Lines Changed
1	127
2	125
3	0

Table 6.1: Number of lines changed per iteration for Test Framework maintenance

At present, only 3 out of 4 possible automation systems have been developed within the duration of the Thesis, costing around 960 man-hours. This effort estimate is a worst case estimate. Similarly for the complete automation effort a total of 1280 man-hours would be necessary(or $s_a = 1280$ man-hours). Therefore, the total effort required for automated testing can be modelled by:

$$E_a(m,n) = s_a + C_a(m,n) + M_a(n) + \sum_{i=1}^{m} 0.35 + 2.65 \cdot e \cdot e^{-m}$$

We can observe that the cost of testing stays constant in terms of number of repetitions when not considering the cost of changes or maintenance. But since maintenance effort is inevitable, the effort of maintenance can be drawn from the regular maintenance effort required for the Framework. During the operation period (a three iteration window as seen in Table 6.1), an average of 126 lines of code were changed per iteration without considering iteration 3 with an assumption that there is always maintenance of the software.

Table 6.2 provides the observed time required for a developer to code the different features of the test framework. This data was observed using the TMS, the time taken for a ticket to change from the state "Working On" to "Implemented". On an average, it took around 4 minutes to change 1 line of code. Therefore we can conclude:

$$M_a(n) = 126 \cdot 4 \cdot n = n \cdot 8.4(in\ man\ hours)$$
$$C_a(m,n) = (2 \cdot 1.6 + m \cdot 0.35) \cdot \lfloor n/8 \rfloor$$

$$E_a(m,n) = 1280 + 8.4 \cdot n + C_a(m,n) + \sum_{i=1}^{m} 0.35 + 2.65 \cdot e \cdot e^{-m} \tag{6.8}$$

The Figure 6.2 shows the effort required for different number of test cases when used for 3 years or 79 iterations. It also assumes the effort required for changing tests cases every 9 iterations, for the worst-case scenario, assumes all the test cases have to be redone and $t_m in$ amount of time is required to do them again.

Feature (instruction)	Time required (in mins)	Lines of Code
wait	45	8
execute	50	25
connect	60	19
disconnect	70	12
reset mock core	90	16
mock core	120	42
instruct	145	32
mock core verify	180	19
http	140	34
dump stream	125	72
stop dump stream	65	12
seek stream	70	49
Average	96.67	28.33

Table 6.2: Effort required per feature added

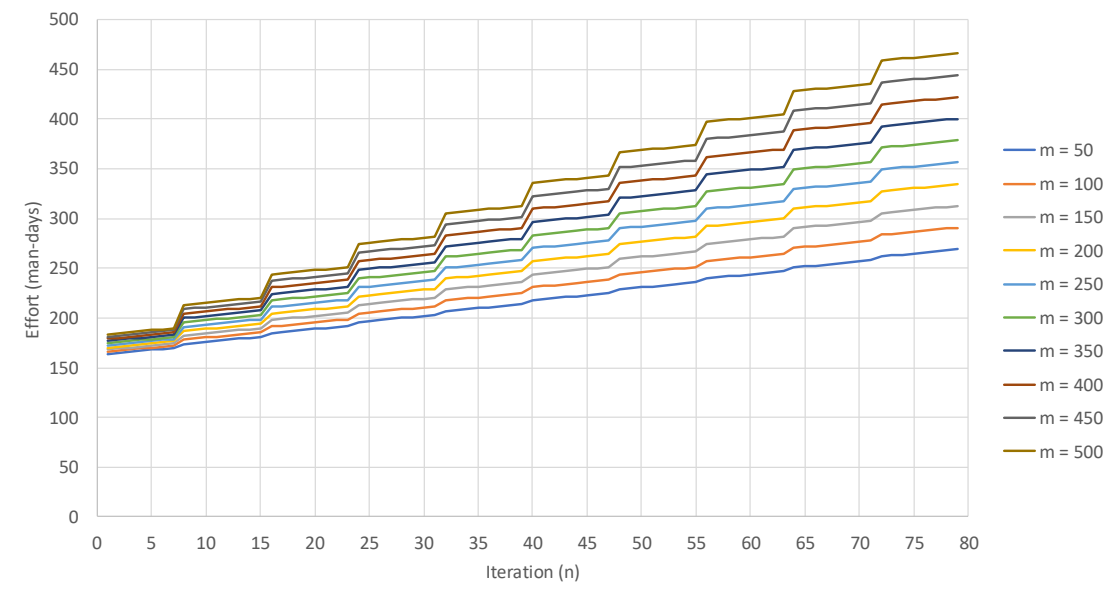


Figure 6.2: Effort Estimation for Automated Testing with m tests per iteration

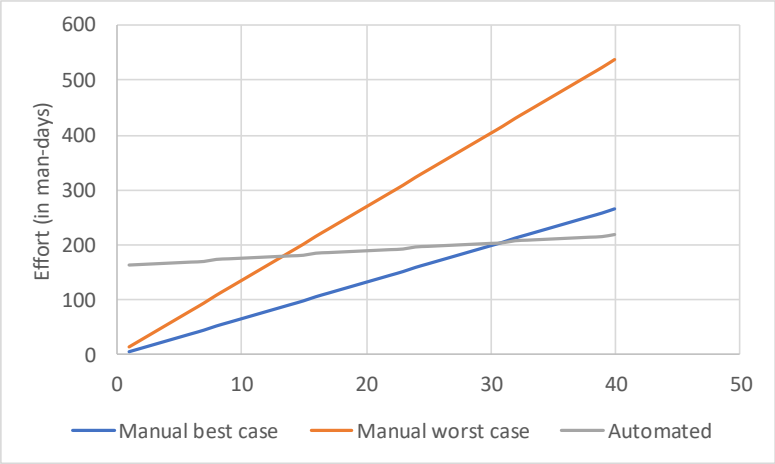


Figure 6.3: Effort Break even for $m = 50$

6.1.2 Break Even Analysis

For a fixed number of Test cases, a breakeven point for Method M and Method A can be determined using equations 6.5 and 6.8:

$$E_m = E_a$$

$$n \cdot (m \cdot 1.6) + C_m(m,n) = 960 + 8.4 \cdot n + C_a(m,n) + \sum_{i=1}^{m} 0.35 + 2.65 \cdot e \cdot e^{-m}$$

The graph in figure 6.3 shows the efforts for automated testing vs the effort for the current digital concepts processes when only 50 test cases are performed in every iteration. It can be observed that at iteration 14, the effort of testing for both methods are almost equal. Therefore, the break-even for Manual vs. Automated Testing for 50 test cases is 14 iterations. In the other figure 6.4, where there are 150 number of test cases performed in each iteration has a break-even of about 5 iterations.

6.2 Test Quality and Efficiency

The quality and efficiency of test can be measured by the number of defects captured or as well using the ratio of defects caught to the defects reported. This ratio shows the number of defects that the customers faced to the number of defects that were caught before release.

6.2.1 Digital Concepts Initial Process

To obtain the state of the defects, the TMS is utilized as the source of truth to obtain various bug reports that was created for the various releases. The ticket system was manually analysed for either a feature request or a Bug fix request. All the tickets were filtered out manually according to the

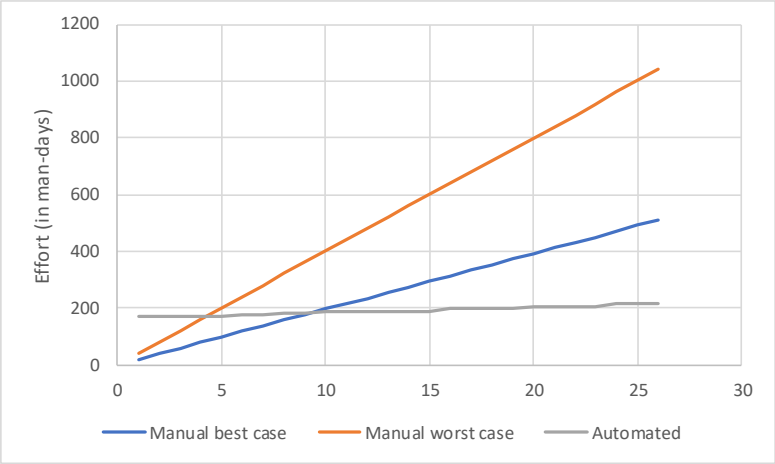


Figure 6.4: Effort Break even for $m = 150$

Number of Tests (m)	Best case break-even (n)	Worst case break-even (n)
50	31	14
100	14	7
150	10	5
200	8	3
250	6	3
300	5	3
350	4	2
400	4	2
450	4	2
500	3	2

Table 6.3: Break even iterations for different number of tests

category and only the bug fix requests were further analysed for information. As a general pattern discovered, the bug fix requests could be separated into an internal report or an external report using the body of the request.

- External Bug reports were added into the ticket with a basic description of the failure, what the "Third Person" had done to cause the error (the reproduction method) and what was expected.
- Internal Bug reports were a bit more direct on what the problem was, pointing to a log file directly as well as to some specification document if available.

The above separation method was applied for the separation of internal bug to the external bugs and these bugs were further sorted out to various software components of the gateway. According to the data gathered, the Digital Concepts Gateway had a total report of 82 defects out of which 16 defects were reported by customers.

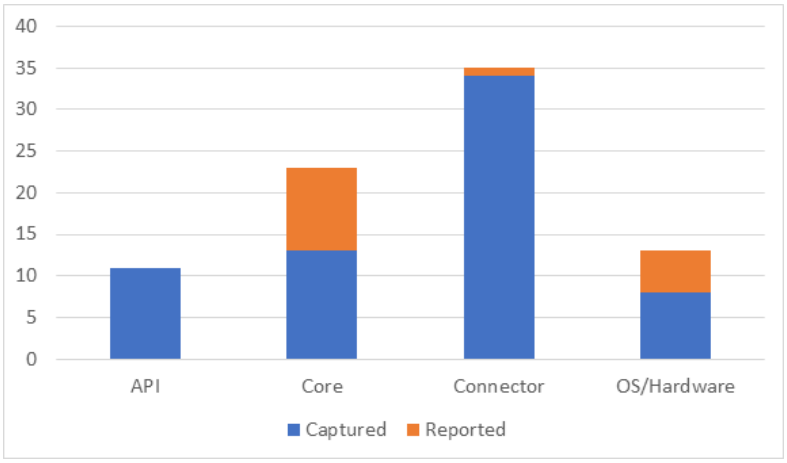


Figure 6.5: Bug detected and captured for Digital Concepts Gateways

$$DCR = \frac{16}{82}\,0.2$$

The bar graph in figure 6.5 shows the distribution of these bugs into separate categories corresponding to the area of concern.

It can be observed that the greatest number of defects was found to be in the Connectors which was about 42% of the total. The Core contained as well around 28% of the total defects of which 44% defects were reported by the customer.

6.2.2 Testing Process with Automated Framework

The Evaluation of the Automated Framework quality and efficiency requires the measurements for an extended period of time regarding the number of bugs detected and as well reported by users. The graph in figure 6.6 shows the current status of the automation framework, with a total of 7 test cases generated by iteration 3. It can be observed that the initial iterations detected a lot of defects; but with further evaluation of the tests, it was concluded that the test environment had the defects and only 1 actual defect was found in 3 iterations. Therefore, it is desirable to have more tests written and more data gathered to determine the exact efficiency of the testing framework.

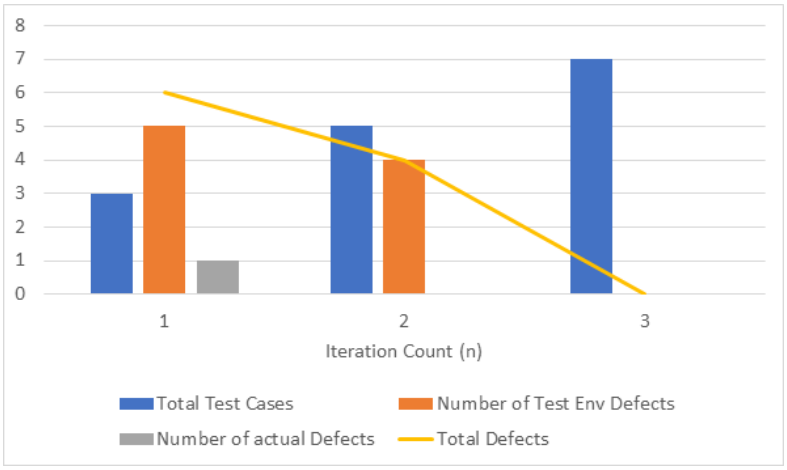


Figure 6.6: Bug captured by automated framework

7 Results and Discussion

A thorough study of Digital Concepts Gateway; architecture, use cases, development and testing process, has helped answer the research questions. First, testing in IoT has its own peculiar properties and therefore needs specialized solution for system testing. Second, test automation can be achieved but a complete hardware test automation requires high setup cost and is inflexible to changes. A right balance between manual and automated support can result in a thorough testing with controlled amount of effort required. The exact degree of automation could not be measured due to limitations on time. The testers required extended periods of time to learn the test system and effectively put it into use. This as well hindered the efficiency evaluation of automated testing with manual testing. Only a few tests were successfully created, therefore a broader code segment could not be tested, revealing almost no defects.

7.1 Testing at Digital Concepts

Testing of IoT devices till present has only been performed using manual test techniques [Ree16]. A physical test bed is generally deployed to test the system which correlates to less available opportunity for performing diverse tests. This statement holds to be true at Digital Concepts; a physical test-bed with a pre-defined set of devices are present, but when a new device is to be tested, the complete test bed has to be changed and adapted. Another peculiar testing necessity is the test bed setup; even though a test bed is pre-defined, it is required to setup the test bed every time before execution.

The interaction to IoT resources through DCGW is possible using the gateway API, either via iOS device, Android device or directly over the cloud. Which means that any interaction that is performed on the IoT devices initiates from a remote event, example a UI interaction on a smart phone. Therefore, any automation attempt should replace these devices or automate them. Another noticeable feature of IoT software is the use of standard UI systems such as HTML for the configuration of the device. These UI components can be test automated using readily available tools.

It is in the nature of IoT applications to save energy. Many devices or protocols designed for IoT focus on energy conservation, therefore a communication or a simple event takes extended periods of time. Temperature sensors are a typical example where the sensor updates its status only in particular intervals. It affects the number of tests that are possible within a fixed time and can be directly related to the quality of the software. Test automation utilizing regular hardware devices does not therefore improve the time required for testing, since the automated test has to wait for these sensors as well. Which points towards the need to emulate devices in order to test cases rapidly.

Automation of hardware testing can be possible by attaching additional sensors to actuators and vice-versa. This is a small part of the test to be performed and for Digital Concepts, a thorough testing of the hardware feature is not necessary. This is performed by the individual manufacturer of the device. For digital concepts, a strict compliance to the specification with minimum device testing is an applicable scenario.

Model based testing of IoT devices seems to be an innovative approach to automated testing, but it can only test the behaviour of the system in terms of software correctness. But there are many situations where a fault in the hardware causes errors in the software. There are cases that the SUT hardware has been manufactured by a vendor and the software written by another. Therefore any test execution should target as much hardware inclusion as possible.

In such a case, it seems to be a feasible solution that automated testing be used for regressions and data communications, whereas manual testing be used to newly added feature. This allows a balanced and thorough testing of the system with a controlled level of quality.

7.2 The Shift towards Test Automation

The proposed framework in Chapter 4 has been implemented as an automated framework. The degree of automation for the new framework could not be evaluated in fine detail using the metric defined in Section 4.3.2, due to the lack of automated test cases. A rough evaluation on the degree of test automation could be made using the broad categories that are automatable. The gateway system can be divided into five parts; Web UI, Core EnOcean Communication, Homekit Connector, Cloud Connector, IoT devices. Among these five parts, only three parts could be automated as of now, the Cloud Connector needs to be further evaluated. Therfore we can estimate a rough 60% automation after all of the test cases have been written.

A comparison using the evaluation provided in Chapter 6, we can conclude that the development of automated testing is a beneficial investment for Digital Concepts. Even for minor testing (only 50 test cases) a break-even point can be achieved within half the expected life-span of the project. Supposing we take 150 test cases which is the number of implemented EnOcean profiles within the gateway. The total work hours required to test these many test cases would be 19.5 working days, which would mean Digital Concepts would require at least two dedicated testers throughout the lifespan of the project; supposing the number of test cases remains constant.

Despite the benefit of using test automation, manual testing is as well required, especially to test the hardware functionality. Manual testing should be performed from time to time or on every added new feature. This would allow Digital Concepts to limit the required manual effort in testing.

7.3 Threat to Validity

The gathered data for the initial testing process of Digital Concepts was solely based on the interviews with a Tester at Digital Concepts who has had long experiences of Manual Testing for Digital Concepts of around more than 2 years. Since no proper recorded data are available, the information obtained from the interviews is the only source of truth and therefore a certain error margin has been used to cope with this scenario. As for automated testing, the data has been gathered using

measurements from the Test Automation System, but it should be noted that the gathered sample set was small and thus could present biasness to the result. Due to this reason, a refinement of the model should be later carried out with better available data.

But it can be noted that for the presented 50 test cases in one iteration is a very less number in practice. It can be safely assumed that there are at least 150 test runs assuming a single test run per implemented EnOcean Profile. where we can see that the break-even point is around ten for the best case scenario and five for the worst case. If we increase the number of tests, three tests per profile, which is a more likely scenario, then we would get a break even point of four iterations on the best case and two iterations for the worst case. This implies that, even with a biased data obtained from a small sample space with high error margin. It is highly likely that the break-even point lies within the range of the project life-time. Rougly comparing the numbers obtained 5000 man-days for manual testing with 450 man-days for automated testing, shows us that manual testing requires around 11 times more effort.

Efficiency and Quality of the Manual Testing was determined using the number of defects captured by the testing process. The same could not be done for automated testing, due to the time requirement for integration of the testing framework into the company process. This resulted in the generation of only 7 test cases in total in 3 iterations, which can be mainly attributed to the lack to resources caused by business needs at the moment and partly to the difficulty in learning the test framework. But it is sure that this framework has provided a documented way of storing test cases which is a positive qualitative property.

The defects detected at present, generally are found using the API or the UI layer, which does not provide a clear idea as to which layer the problem lies in. This is solved by automated testing, since each layer of the software is individually tested and verified incrementally, providing a narrower zone for defect positioning.

Bibliography

[ARV+13] A. F. Araujo, C. L. Rodrigues, A. M. R. Vincenzi, C. G. Camilo-Junior, A. F. Silva. "A Framework for Maturity Assessment in Software Testing for Small and Medium-Sized Enterprises". In: *XI International Conference on Software Engineering Research and Practice – SERP'13* (2013), pp. 225–230 (cit. on p. 40).

[Axe18] A. Axelrod. *Complete Guide to Test Automation*. Berkeley, CA: Apress, Sept. 2018. ISBN: 978-1-4842-3831-8. DOI: 10.1007/978-1-4842-3832-5. URL: http://link.springer.com/10.1007/978-1-4842-3832-5 (cit. on pp. 34, 35).

[Ber03] A. Bertolino. "Software Testing Research and Practice". In: *Abstract State Machines 2003*. 2003, pp. 1–21. ISBN: 3540006249. DOI: 10.1007/3-540-36498-6_1. URL: http://link.springer.com/10.1007/3-540-36498-6%7B%5C_%7D1 (cit. on p. 19).

[BSC96] I. Burnstein, T. Suwanassart, R. Carlson. "Developing a testing maturity model for sofiware test process evaluation". In: *International Test Conference* (1996), pp. 581–589. ISSN: 0010-440X (cit. on p. 33).

[CH11] H. Cooper, L. V. Hedges. "Research Synthesis as a Scientific Process". In: 2011, pp. 1–14. ISBN: 9780871541635 (cit. on p. 17).

[CVB+13] N. Chen, C. Viho, A. Baire, X. Huang, J. Zha. "Ensuring Interoperability for the Internet of Things: Experience with CoAP Protocol Testing". In: *Automatika Journal for Control, Measurement, Electronics, Computing and Communications* (2013). ISSN: 1848-3380. DOI: 10.7305/automatika.54-4.418 (cit. on pp. 29, 31).

[DKPM12] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, M. V. Mantyla. "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey". In: *2012 7th International Workshop on Automation of Software Test (AST)* June (2012), pp. 36–42. DOI: 10.1109/IWAST.2012.6228988. URL: http://ieeexplore.ieee.org/document/6228988/ (cit. on p. 19).

[DS12] F. I. Duncan, a. G. Smeaton. "Assessing and improving software quality in safety critical systems by the application of a SOFTWARE TEST MATURITY MODEL". In: *System Safety, incorporating the Cyber Security Conference 2012, 7th IET International Conference on* (2012), pp. 1–4. DOI: 10.1049/cp.2012.1509 (cit. on pp. 33, 34, 40).

[Gal17] D. Galin. *Software Quality Assurance: Concepts and Practice*. Hoboken, NJ, USA: John Wiley & Sons, Inc., Mar. 2017, pp. 1–680. ISBN: 9781119134527. DOI: 10.1002/9781119134527. URL: http://doi.wiley.com/10.1002/9781119134527 (cit. on pp. 15, 16, 19, 27).

[GE17] V. Garousi, F. Elberzhager. "Test Automation: Not Just for Test Execution". In: *IEEE Software* 34.2 (Mar. 2017), pp. 90–96. ISSN: 0740-7459. DOI: 10.1109/MS.2017.34. URL: http://ieeexplore.ieee.org/document/7888399/ (cit. on p. 15).

[GF99] D. Graham, M. Fewster. *Software Test Automation: Effective Use of Test Execution Tools*. 1999. ISBN: 4418653813. URL: http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:software+test+automation+effective+use+of+test+execuation+tools%7B%5C#%7D0 (cit. on pp. 34, 35).

[HMPR04] Hevner, March, Park, Ram. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), p. 75. ISSN: 02767783. DOI: 10.2307/25148625. URL: https://www.jstor.org/stable/10.2307/25148625 (cit. on p. 17).

[Int12] International Telecommunication Union. "Overview of the Internet of things". In: *Series Y: Global information infrastructure, internet protocol aspects and next-generation networks - Frameworks and functional architecture models* (2012), p. 22. ISSN: 0018-8646. DOI: 10.1109/ESEM.2015.7321184.3. (cit. on pp. 15, 22).

[Kan14] S. H. Kan. *Metrics and models in software quality engineering*. Second Edi. Addison-Wesley, 2014. ISBN: 0133988082 9780133988086 (cit. on pp. 32, 33).

[KC07] B. Kitchenham, S. Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. 2007 (cit. on p. 17).

[Luo01] L. Luo. "Software testing techniques". In: *Institute for software research international Carnegie Mellon University Pittsburgh, PA* 15232.1-19 (2001), p. 19 (cit. on p. 23).

[Ma13] E. Ma. *SWEBOK 3.0 The Guide to the Software Engineering Body of Knowledge*. 2013, p. 369. ISBN: 9780769551661 (cit. on pp. 23, 24, 30).

[MHS+14] P. McMinn, M. Harman, M. Shahbaz, S. Yoo, E. T. Barr. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2014), pp. 507–525. ISSN: 0098-5589. DOI: 10.1109/tse.2014.2372785 (cit. on p. 31).

[Moo11] J. W. Moore. "Knowledge Area: Software Testing". In: *The Road Map to Software Engineering*. IEEE, 2011, pp. 1–15. ISBN: 9780471746256. DOI: 10.1109/9780471746256.ch9. URL: http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=5989679 (cit. on pp. 24, 30).

[Nai] K. Naik. *Software Testing AND QUALITY Theory and Practice*. ISBN: 9780471789116 (cit. on p. 32).

[PRTV12] K. Peffers, M. Rothenberger, T. Tuunanen, R. Vaezi. "Design Science Research Evaluation". In: *Foundations and Trends® in Machine Learning*. Vol. 3. 1. 2012, pp. 398–410. DOI: 10.1007/978-3-642-29863-9_29. URL: https://doi.org/10.1007/978-3-642-29863-9%7B%5C_%7D29%20http://link.springer.com/10.1007/978-3-642-29863-9%7B%5C_%7D29 (cit. on pp. 17, 18).

[Ree16] E. Reetz. "Service testing for the internet of things." PhD thesis. 2016. URL: http://epubs.surrey.ac.uk/810848/ (cit. on pp. 16, 19, 22, 23, 42, 45, 71).

[RS] F. E. Ritter, L. J. Schooler. "Powerlaw of learning for IES&BS 1 The learning curve". In: *Ritter.Ist.Psu.Edu* (), pp. 1–12. URL: http://ritter.ist.psu.edu/papers/ritters01.pdf%7B%5C%7D0Ahttp://www.iesbs.com/ (cit. on p. 63).

[RWBO15] P. Rosenkranz, M. Wählisch, E. Baccelli, L. Ortmann. "A Distributed Test System Architecture for Open-source IoT Software". In: *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems - IoT-Sys '15* (2015), pp. 43–48. DOI: 10.1145/2753476.2753481. URL: http://dl.acm.org/citation.cfm?doid=2753476.2753481 (cit. on p. 15).

[SBM12] C. Sandler, T. Badgett, G. J. Myers. *The Art of Software Testing - 3 RD Edition*. 2012. ISBN: 9781118031964 (cit. on pp. 15, 24, 25).

[Sim06] Y. W. Sim. "JISC DEVELOPMENT PROGRAMMES Project Document Cover Sheet An Overview of Service-Oriented Architecture Project An Overview of Service-Oriented Architecture 1 Introduction". In: July 2005 (2006), pp. 0–12 (cit. on p. 22).

[SR16] V. Sharma, T. Ravi. "A review paper on "IOT" & It's Smart Applications". In: *International Journal of Science, Engineering and Technology Research* 5.2 (2016), pp. 472–476 (cit. on p. 20).

[SW18] D. Serpanos, M. Wolf. *Internet-of-Things (IoT) Systems*. 2018, pp. 1–102. ISBN: 978-3-319-69714-7. DOI: 10.1007/978-3-319-69715-4. URL: http://link.springer.com/10.1007/978-3-319-69715-4 (cit. on pp. 15, 20).

[TRMB12] R. Tönjes, E. Reetz, K. Moessner, P. Barnaghi. "A test-driven approach for life cycle management of internet of things enabled services". In: *Future Network & Mobile Summit (FutureNetw), 2012* (2012), pp. 1–8 (cit. on p. 16).

All links were last followed on April 30, 2019.

A Test Maturity Model Assessment

Role: _____

Question	Yes/No	Notes
Is Testing a defined separate phase after coding?		
Is Testing just a part of debugging activity?		
Is Testing planned and repeatable? If yes, when is it performed in general?		
Is there any basic technique applied while testing? If yes, name few or describe		
Is Testing a Major phase which is provided dedicated time and effort?		
Is there a separate Group for Testing the software apart from the developer?		
Are Basic Tools available for Testing the software? Such as API testing tools? If yes list a few		
Is the Testing Process Monitored? If yes what aspects are monitored?		
Is the staff trained to perform testing activities? If yes what kind of training?		

Are reviews and inspections a part of the process? If yes, when and how are they performed?		
Is project history such as reviews and defect data are stored for later reference?		

B Digital Concepts Test Environment

Anatomy of Test Environment

1. MySQL Database: Used by the test coordinator to keep track of test requests and results
2. Test Coordinator: The component that coordinates the test including execution, validation
3. Apple Client Simulator: A simulator to simulate apple devices to help testing
4. System Under Test: The Machine that is being tested

Test Coordinator

Test Coordinator is the most important component of the test environment, and is responsible for keeping track of the execution and validation of the tests. The Test coordinator can be subdivided into other components such as:

Request Monitor

The request monitor is a component that waits for test requests either from a user or from the build system. The request monitor adds an inode listener on the requests folder (refer working directory structure) for request file creation. After the request has been written on a request file, the request monitor reads the request information and then gathers all the relevant test cases for the executor. At present the request monitor instantiates test executors, but in actuality the test executor should be a single process and should maintain worker threads which run tests.

Test Executor

The test executor runs the tests and nothing more. It takes an argument namely the original created request file. It then takes the filtered test cases one by one and executes them in order. The overall execution is logged in a general execution file, whereas each test case execution is logged into special files where the output of each execution is written to the file. The idea behind the separation of executor and oracle is that a test case data may be available only later and not during execution, such as stream data, or log entries, where the values cannot be guaranteed to be generated exactly in time for a distributed system.

Oracle

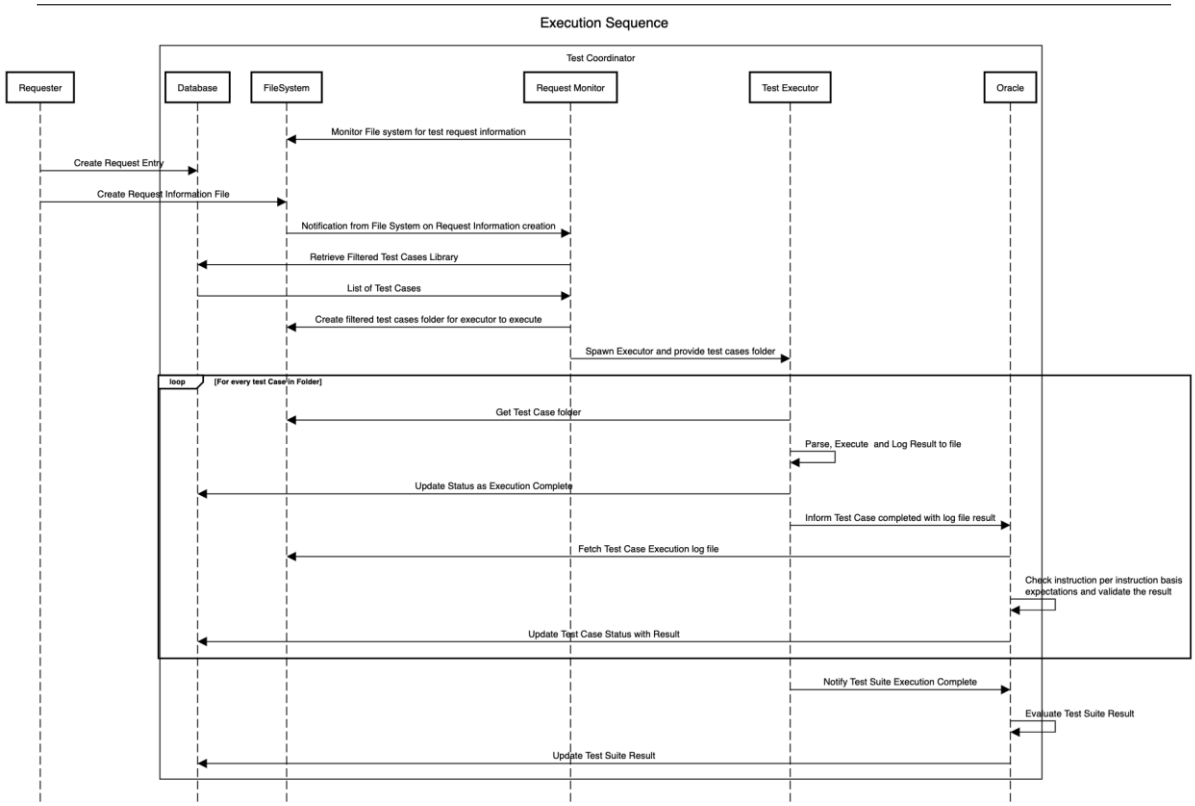
The oracle is the part of the environment which is responsible for validating expectations. The oracle performs the validation based on the values provided by the tester in the test case itself.

Basics of Test Execution

The test execution can be divided into two categories, the local test execution for development purposes whereas the Execution on Server Mode.

Intended Execution Mode

The actual intended mode of operation was running the test coordinator on a server with a sql database to keep record of the test executions as well as the results. The Test Coordinator in this case is the complete machine and comprises of three major components as described above. The Request Monitor, Test Executor and the Oracle. A sequence diagram shown below should explain the execution sequence in detail. In case the diagram is not readable please use the “Execution Sequence.png” on the docs folder.



In Short, the three components work together to create the Test Coordinator. The test executor and the Oracle are independent of each other for a particular reason. The major reason behind this is that the Test executor might not possess complete information of the complete system during execution since the executions take a very short amount of time. Therefore the results that the executor determined are written onto a file immediately and the oracle reads the files later along with other necessary files such as logs or stream dumps to verify whether the execution was successful or not.

Local Test Execution

The Local Execution mode is a wrapper to the actual mode of intended operation, ie. The Execution on Server. This mode allows developers to quickly execute test cases on a local level with their local test environment. This allows faster development cycle for test designers to get feedback from the designed test cases since the tool is new and a longer cycle time would impact the learning rate negatively. To allow local testing, a wrapper script has been created called the local_test_coordinator.py which wraps the working of the actual test coordinator without the need for SQL communication. The reports in this case are written directly to a reports folder. Apart from this, there is no difference to the actual test execution mechanism.

Test Case Structure

Base Test Case Layout

A test case is defined within a yaml file for any kind of testing and is used to unify the test writing procedure. The yaml test case template is as shown below:

```
--- # Test Case
name: Nodered Service Start Test
```

```
description: >
  Node red service should be successfully started by the script as a docker container.
expected: >
  The docker service should be successfully started.
testCaseId: nodered_start
criticality: NORMAL
phases:
  - name: setup
  - name: execute
  - name: teardown
```

Please refer to the table below for the definition of each of the fields

Field Name	Description	Type	Remarks
Name	The name of the Test Case	String	Only for readability (Optional but **Recommended**)
Description	Description of the Test Case	String	Only for readability (Optional but **Recommended**)
Expected	Expected behaviour in the test case	String	Only for readability (Optional but **Recommended**)
testCaseId	A Unique String identifier for the test case	String	**Required**. Needed for now while creating the database entry (Should be removed from here later)
Criticality	The criticality of the test case. Shows how important this test case is.	ENUM	Allowed values: LOW, NORMAL, HIGH. For now, not considered. But should be considered later
Phases	The different phases of the test cases, such as setup, execute and the tear down	List <Phase>	**Required**. Should be only three and in the specific defined order.

Test Phase Layout

There are only three fixed phases within a test case, the setup phase, the execute phase and the teardown phase. The setup phase is executed first and should be used to perform actions that setup the test case. Example test phase template has been provided below:

```
- name:
   steps:
     - instruction:
       target:
       params:
       expect:
```

Field Name	Description	Type	Remarks
Name	The name of the phase	**ENUM**	**Required**. Allowed values: setup, execute, teardown
Steps	The various steps that are included in this phase	**List<Step>**	**Required**.
Step.instruction	The instruction to be performed for a step	**String**	**Required**. Refer to instruction manual
Step.target	The target device where the execution is to be targeted	Host:port or just host depending upon the instruction	**Required**. Special keyword self to refer to the test coordinator
Step.params	The parameters for the instruction.	**List<String>** Note: Defined within single quotes to be sure	**Required**. Refer to instruction manual for available params
Step.expect	The expectations when this step is performed. Refer to expectation manual for available expectations	**Expectation**	**Optional**

Available Instructions Cheat sheet

The following instructions are currently available for use

Instruction	Comment	Parameters	
		Name	Description
wait	Sleeps for a set defined period of time	Time	The amount of time to wait in milliseconds
exec	Execute a certain shell command	Args	The command to execute along with the arguments
Connect	Connect to a socket stream. Useful to connect to the slave, since the slave uses socket stream to communicate		
Disconnect	Disconnects from the previously connected socket stream		
Reset mock core	Resets a core mock.		
Mock core	Specifies a mock response for the core.		
Instruct	Instruct a secondary helper machine such as the homekit client simulator to perform some actions. The actions are totally dependent on the helper machines	Instructions	The list of instructions to be sent to the helper machine.
Mock core verify	Request the core mock for requests that it has received till now		

http	Make a generic http request		
Dump stream	Starts dumping of a stream into a defined file	Output filename	The filename where the output should be dumped
Stop dump stream	Stops the dumping of a stream		
Seek stream	Opens the stream dump if not open and seeks to a defined position by the match field	Dump file	The name of the dumpfile where the stream was dumped into

Instructions in Detail

Wait

The wait instruction can be used to wait for some event example timeouts. Currently it can only wait on time provided in the parameters. But should be later used to wait on events such as notifications or packets.

Property	Description
Target	The target for the instruction. **Note**: self is a reserved special keyword which means the Test Coordinator waits for some event.
Params	At present: - Milliseconds to wait (as string)
Future improvements	- Make test coordinator wait for HAP notifications - Make test coordinator wait until a certain packet is sniffed
Possible Expectations	None
Example Usage	Make the test coordinator wait for 15 seconds instruction: wait target: self params: - '15000'

Execute (exec)

The execute instruction executes a shell command in the target test machine. The use of this command is to test operating system level operations, such as testing whether the system restarts services that are vital to the operation of the device. The execution of the commands makes use of SSH services; therefore, the test machine should have pre-installed SSH public key from the test coordinator to allow the coordinator to gain access to the target test machine. An additional requirement might be that the target test machine be already in the known hosts (required by python). The arguments to the instruction are the commands that is to be executed on the remote machine. The commands can be executables with additional parameters to the executable, such as ls -la or could be a one liner bash script such as ‘if [! -f /opt/dcgw/config/node-red.config]; then echo "False"; else echo "True"; fi’.

Property	Description

Target	The target of the instruction is expected to be either a **hostname or an ip address** if the username is already setup else, **username@hostname**
Params	- Arg0 - Arg1 - …
Future improvements	- Allow execution of commands on the local machine using the self-keyword
Possible Expectations	- result.code = string represented number - result.stdout = String (Standard Output) - result.stderr = String (Standard Error)
Example Usage	Execute nodered script with the parameters ‘set off’ and expect that the output code is 0 instruction: exec target: gw params: - '/opt/dcgw/scripts/nodeRed' - 'set' - 'off' expect: - result: code: ‘0’ - result: stderr: - validator: isEmpty

Connect

The connect method opens a socket stream to a desired participant. The participant should already have the port open and should be able to communicate using the method as defined in **socket communication**. An example of this the homekit client simulator, it accepts commands using the socket stream and performs homekit actions according to the commands. Read more on the Homekit Client Simulator. **NOTE**: Recurring call to this instruction does not create the connection. A connection is created only when none exists.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber** for proper socket communication.
Extra Fields	- None
Params	- None
Future improvements	- Connection to be closed automatically during cleanup of the test case execution
Possible Expectations	- Result.status = (SUCCESS|FAILURE)
Example Usage	Connect to the specific host and port defined by the environment variables

	instruction: connect target: '$slave-host:$slave-port'

Disconnect

The connect method closes the socket stream from a participant that was previously connected.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber** for proper socket communication. And exactly same as the one provided in connect. If the connect method utilizes the environment variable, then this method should also do the same
Extra Fields	- None
Params	- None
Future improvements	- The requirement of exact same target definition
Possible Expectations	- Result.status = (SUCCESS|FAILURE)
Example Usage	Connect to the specific host and port defined by the environment variables instruction: disconnect target: '$slave-host:$slave-port'

Reset Mock Core

The method sends a reset request to the mock core. The reset request, removes all the mock mappings as well as the log of received requests, but does not close any open stream connections present. Please refer to the mock core description for further details on how the mocking has been achieved.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber** since the mock core listens only to a specific port number.
Extra Fields	- None
Params	- None
Future improvements	- None
Possible Expectations	- Result.status = (SUCCESS|FAILURE)
Example Usage	Request the mock core to be reset. instruction: reset mock core target: '$host-port'

Mock Core

The method sends a reset request to the mock core. The reset request, removes all the mock mappings as well as the log of received requests, but does not close any open stream connections present. Please refer to the mock core description for further details on how the mocking has been achieved.

Property	Description

Target	The target of the instruction is expected to be a **hostname:portnumber** since the mock core listens only to a specific port number.
Extra Fields	- Ref: Represents a reference to a Test object that is sent to the mock core. Refer Mock Core Test Objects for further information
Params	- None
Future improvements	- Addition of other fields in the test step to allow direct setting of the mock object. (Not only test object references)
Possible Expectations	- Result.status = (SUCCESS|FAILURE)
Example Usage	Request the mock core to be set using a test object. instruction: reset mock core target: '$host-port' ref:test_object_name

Instruct

Instruct sends instructions to a specific device using a socket stream. A connection to the device should be opened before instruct commands are performed. The instruct command logs anything returned in the socket stream and thus the documentation of the specific target service should be followed when validating results. The instruct object stops when a “Done:” message is returned in the stream. This should message should be a reserved keyword and should be sent only when the instruction has been performed. The instructions sent by the coordinator is as JSON Array, thus the json array should be first parsed to obtain the actual instructions. Please refer to the Socket communication section for further information.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber**. Same as the one used with connect command.
Extra Fields	- Timeout: The amount of time to wait for this instruction to timeout from stream read or write
Params	- Arg0 (json compatible items/ *<test_object_name> to refer to a test object) - Arg1 - …
Future improvements	- Remove the requirement of exact same target definition as in connect
Possible Expectations	- Depends on the target component and as well as the instruction sent to the target.
Example Usage	Instruct the Homekit simulator to connect to a gateway. (Refer Homekit simulator for various possible commands) instruction: instruct target: '$slave-host:$slave-port' timeout: 30000

	params: - 'connect' - '$gw-homekit'

Mock Core Verify

At its current stage the mock core verify only requests the mock core to provide all the requests that it has received since last reset. This information is stored as part of the Result object under requests key. Validations can be made on this request information using the available result object validation methods. Please refer to Expectation validation for further information.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber**. Since the core mock only listens on a specific port number
Extra Fields	- None
Params	- None
Future improvements	- Allow other validations as well, such as Number of connected devices, etc
Possible Expectations	- Result.requests : (JSONArray)
Example Usage	Verify the mock core received the first request as GET /system/info instruction: mock core verify target: '$mock-gw-core' expect: - result: requests: - path: '[0].method' value: 'GET' - path: '[0].uri' value: '/system/info'

http

Make a generic HTTP request to the defined target. Only the HTTP protocol is available at present and cannot be changed. There are two possible ways of determining the input to this instruction, first is to provide a reference test object which contains the fields 'method', 'headers', 'params', 'body' and 'uri'. In actual practice all of the fields may not be used since for example a GET request does not require body as input. The only required parameters are method and the uri.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber**. Since a http server listens on a specific port.
Extra Fields	- Method: The http request verb such as GET, POST (Required) - Headers: The headers to be added to the request as MAP - Body: The body as a string object - Params: The url query parameters as MAP - Uri: The path of the request object. Without the host and port (Required)

	Or, - Ref: A reference to a test object containing the above fields
Params	- None
Future improvements	- Accept the body as a string or a json map - Allow username and password setting in the test itself rather than having to create a
Possible Expectations	- Result.code: Integer represented string - Result.body: Depending upon the request
Example Usage	Perform a GET /devices request and then validate the code was 200 and the body is a json object, with a field devices with the value test. instruction: http target: '$mock-gw-core' method: GET uri: /devices expect: - result: code: 200 body: - path: 'devices' value: 'test'

Stream dump

Connects to the defined target stream and dumps the stream into a file for later parsing and validation. Please refer to stream dumps for detailed information of parsing and validation. For the sniffing of complete network information please refer to the Listening to complete enOcean Packets. This instruction does not allow multiple stream connections

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber**. Since the core mock only listens on a specific port number
Extra Fields	- Username: The stream username - Password: The stream password
Params	- Output filename: The name of the output file
Future improvements	- None till present
Possible Expectations	- None (Use stream validation)
Example Usage	Start the stream dump from the defined network address into the defined file name in the parameter. instruction: stream dump target: '172.28.28.150:8080' username: '$username'

	password: '$password' params: - dumpfile

Stop stream dump

Connects to the defined target stream and dumps the stream into a file for later parsing and validation. Please refer to stream dumps for detailed information of parsing and validation. For the sniffing of complete network information please refer to the Listening to complete enOcean Packets.

Property	Description
Target	The target of the instruction is expected to be a **hostname:portnumber**. Should be exactly the way how it was declared in the stream dump command
Extra Fields	- None
Params	- Output filename: The name of the output file
Future improvements	- None till present
Possible Expectations	- None (Use stream validation)
Example Usage	Stop the stream dump started from the previous stream dump command. instruction: stop stream dump target: '172.28.28.150:8080'

Seek stream

Loads the dumped stream and seeks through the packets. This is used to validate the enOcean Packets that have been dumped during the test execution. For extra details follow section Stream Dump and Analysis

NOTE: Seek stream does not allow scripted expectations.

Property	Description
Target	None
Extra Fields	- match: A list of path value match arguments. (see example)
Params	- filename: The name of the stream dump file
Future improvements	- Allow added seek functionality such as seek without consuming messages - Allow Scripted validation here
Possible Expectations	- found match (JSON path matching)
Example Usage	Seek the stream dumped in the file streamdump until a packet with telegram.deviceId with '019D5E6C' is found and validate whether the telegram had telegramInfo.rorg as 'A7' instruction: seek stream params: - streamdump match: - path: 'telegram.deviceId' value: '019D5E6C'

	expect: - path: 'telegram.telegramInfo.rorg' value: 'A7'

Expectation validation

The expectation validation works on values reported during the execution. There are fixed defined objects that can be verified depending upon the instruction. These objects are accumulated during validation and any expectation on these objects are performed utilizing the provided expectation hints.

A Schema for expectation is as shown below:

```
expect:
 - result:
    object1: <Expected Value>
    object2:
     - path: '<JSON Path>'
       value: <Expected Value>
     - validator: '<Name of Custom Validator>'
       value: <Expected Value>
```

The expectation can contain multiple expectations for result objects such as in the example object1 and object2. And as in object2 multiple expectations can be concatenated to a boolean 'AND' operation. Apart from this the result object can be defined multiple times for cascaded expectations that have separate concerns. There are three different ways to provide expectation hints to the validator:

1. **Direct Expected Value**: Example usage is for the object1 in the schema, where the expected value is directly utilized. This performs an equals comparison to the expected value with the actual obtained value.
2. **JSON path query and validate**: Example usage is for object2 first list item, where the path and the value keys are defined. The path key tells the validator that it should query the path as defined and compare whether it equals with the expected value.
3. **Custom Validator**: The object2 second list item allows the usage of a key called validator. This key defines the validator method as defined within the custom_validators.py and is invoked to validate whether the expected and the actual results match. Please refer to creating a custom validator for further information.

A General expectation for a Mock Core Verify instruction has been provided here:

```
- result:
   requests:
    - path: '[0].method'
      value: 'GET'
    - validator: valid_request
      value: '/system/info'
```

The first expectation hint compares whether the the method value within the index 0 or the json array is equal to GET or not. The second hint invokes the valid_request custom validator and performs a custom validation logic using the actual value as well as the expected value.

How to Create a Custom Validator

A custom validator is a user function that takes two definite parameters, namely expected value and the actual value. This allows a custom logic to be implemented to validate the expected and the actual value. Custom validators are all defined at present in "custom_validators.py" and should respect the following constraints:

1. The function should take exactly two positional parameters: expected value, actual value. in the defined specific order
2. The function should return a tuple after validation:
    a. passed: Did the validation pass
    b. message: If the validation did not pass, what was the cause?

Example Custom Validator implementation can be seen below:

```
def contains(expected_value, actual_value):
    """
    Validates whether the expected value or list is contained within the actual string or list
    :param expected_value: the expected value
    :param actual_value: the expected string or list
    :return: (passed, message)
    """
    result = expected_value in actual_value
    message = None

    if not result:
        # The validation failed, so adding a failure message
        message = 'Could not find {} within {}'.format(expected_value, actual_value)

    return result, message
```

Socket Communication

The socket communication is the defacto method of communication between the test coordinator with any other mechanized tools used for testing of the Gateway example the Homekit Simulator. The tool that is being controlled by the Homekit gateway should open a socket server on a pre-agreed port number and listen for any messages received in the open socket port.

Basic Terminology:

Downstream: The flow from the Test coordinator to the mechanized tools
Upstream: The flow from the mechanized tools to the Test Coordinator

Downstream Communication

The Downstream communication consists of instructions from the Test coordinator and is delivered as a JSONArray. This is to allow transfer of complex test objects which can be parsed and extracted from the messages with ease. An example of the Data Structure is provided below:

```
[
   "instruction",
   "param1",
   {
      "title":"param2",
      "iscomplex":true
   }
]
```

The first index of the JSONArray is always a string and is the instruction keyword which is supposed to be performed by the mechanised tool. The following items in the array depend on the instruction keyword and the tool being targeted. This allows us to pass parameters that are complex and helps pass reusable test components to the tool to be utilised.

The message (JSONArray) is pushed into the network as per the delimited by length of characters technique used in the Gateway Stream API. The number of bytes are transferred first followed by a newline then the total message is transferred as a byte represented string.

NOTE: Multiple Downstream commands are not handled simultaneously. One command is performed and then only the second is handled. It is unwise to send multiple requests concurrently.

Upstream Communication

The upstream communication is only performed as a reaction to the obtained downstream command. The upstream streams the execution log taking place in the mechanized tool for later evaluation by the Test Coordinator. There are a few reserved keywords which should not be used lightly. They are:

- “Done: <command name>” : This denotes that the execution of the command has been completed
- “Result: <name>: <string representation>” : This sends a named result back to the Test Coordinator for later evaluation.

The streamed information should always be a utf-8 string so that the bytes are not falsely interpreted. In case of extra interpretation required, custom validators should be used on the Test coordinator to parse the string and compare it.

Note: The over the network transmission is as described in Downstream communication

Key Point to Remember: Always stream utf-8 encoded strings.

Stream Dump and Analysis

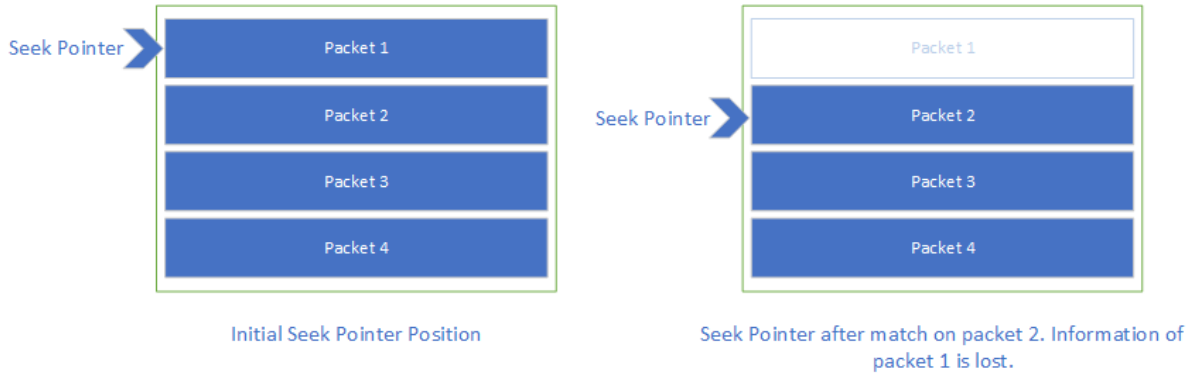
The digital concepts stream API streams all the packets from the devices that have been learned in into the device. This does not allow us to analyze all the packets that were transmitted. Thus a special gateway with the filter packets mode turned off should be used as a preliminary solution. This gateway acts as a sniffer and parses the packets for us. But a fault in the core might cause wrong results which might cause the test to pass even when it should fail. To prevent this, a proper sniffer or tester tool for EnOcean, namely the Dolphin view along with EnOcean USB stick which allows testing of the EnOcean Radio Technology. This allows us to create fake packets originating from different source making the test scalable.

Current State of Stream Dump Analysis

At present the stream dump is performed by the step instruction “stream dump” which takes in a filename as a parameter. The stream dump connects to the target stream using the provided credential and requests for stream delimited by newline. The stream dump is runs as a separate thread, thus the testing can proceed even after stream dump, allowing the Test Coordinator to perform various other steps that would stimulate a Radio packet to be generated. To stop the stream dump, the “stop stream dump” instruction can be used. The results of the stream are stored in a file in a packet per line basis.

Validating the Stream

The stream dump can be validated using the “seek stream” instruction. The stream dump is a file that has each packet written per line. It can be visualized as below:



The packet seek opens the dump file and then starts going through the file on a packet per packet basis as seen in figure above. The individual packets are matched according to the match criteria defined in the instruction and the packets already parsed cannot be matched again. This is under the assumption that the stream has a definite order and the order is to be validated. A second seek stream instruction does not reset the seek. It continues where it was last time.

Test Environment Setup

The test environment can be setup using the following procedure:

Requirements

- Virtual Machine for the Test Coordinator
- Target Test Gateway

Setting up Test Coordinator

- Install MySQL 5.7 (had issues with the new one 8.0) using the general guide as provided here: https://dev.mysql.com/doc/mysql-installation-excerpt/5.7/en/general-installation-issues.html
- Change any settings as per required
- Clone the Test Repository
- Use the provided database model file under “repo/db/db_model.mwb” to create the database structure. MySQL workbench can be used to open the db model file and change the structure. https://www.mysql.com/products/workbench/. It can be used to deploy the

model directly into the database. (This can be done using a personal computer. The VM is not necessary, but make sure to allow external access to the MySQL server before trying it)
- Setup python 3.4+ on the VM
- Change the credentials in the sql.py file to reflect the credential for the newly created user for the test machine
- Run setup.py to create the necessary folder hierarchy in the home directory (creates a dctest folder and subfolders)
- Run the following python files as separate processes

```
python3 oracle.py &
python3 test_case_filter.py &
```

- The first homekit slave can be run on the same VM thus, start the homekit client simulator before hand or can be part of a test case to start the simulator when necessary. (latter method requires added modification in the software)

```
java -jar homekit-client.jar &
```

- Test case requests can be made using request_test.py

(Please refer to python scripts and usages of the various python scripts)

Future improvements

1. Allow Nested referencing of Yaml objects (in test cases as well as in test objects)
2. Clean up folder structure
3. Remove unnecessary file copy, paste and delete as a part of notify
4. Allow multiple files for custom validators
5. Test Coordinator should listen to asynchronous messages on the connected socket stream for notifications ex. homekit notification
6. Make seek stream validation similar to the other validation

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature